About ObjectPAL

See also

ObjectPAL is the object-based, event-driven, visual programming language for Paradox. It is different from traditional procedural languages in many ways. Using ObjectPAL, you place objects (such as buttons and fields) in a form, and then attach code modules (called methods) that execute whenever the object detects an event.

Note: PAL, the programming language for Paradox for DOS, is interpreted, and its code is saved in script files. ObjectPAL is fully compiled, and its compiled code is stored in Windows DLLs along with its source code.

ObjectPAL has two aspects:

- The language itself (its object types, data types, methods, procedures, and constructs)
- The Integrated Development Environment (IDE), including
- The Editor
- The Debugger
- A mechanism for creating and playing ObjectPAL scripts
- The application-delivery facilities

There are six ObjectPAL language categories: data model objects, system data objects, data types, design objects, display managers, and events. Each category is in turn divided into several types.

In addition to these six categories, there are Basic Language Elements that are common to all methods and procedures.

The methods displayed for each type depend on the level of ObjectPAL that you are working in, Beginner or Advanced. (All methods are *available* in both levels; not all methods are *displayed* in Beginner level.) To change the ObjectPAL level in the Developer Preferences dialog box, choose Edit|Developer Preferences.

You can copy and paste the examples into your own code through the Clipboard.

- To copy the entire example to the Clipboard, right-click in the ObjectPAL Example window, and choose Copy.
- To copy a selected part of the example, highlight the block of code you want in the Example window, right-click and choose Copy.
- Place the insertion point in your code where you want to insert the example, then choose Edit|Paste from the Paradox menu.

About programming tasks

See also

Here is a road map to the ObjectPAL language. Before you jump to any of these topics, make sure you read first about Objects, Methods, and Events.

Messages and dialog boxes	Messages and built-in dialog boxes give you a way to interact with a user.
Handling keyboard events	You can trap for any keypress in ObjectPAL, which means you can easily develop hotkeys for your application.
Working with menus	Using ObjectPAL, you can define menus and pop-up menus to display choices to users.
Working with lists	You can use List boxes and Drop-down Edit boxes to let a user choose from a group of items.
Multiform applications	To design applications that use more than one form, you'll need to know how to open a form and control it from another form. Forms can also be opened as dialog boxes.
Working with text files	You can use ObjectPAL to work with text files. Text files are called TextStreams in ObjectPAL.
Using DLLs	With the Uses clause, you can declare and subsequently use functions called from DLLs (dynamic link libraries).
Working with the file system	Using methods in the FileSystem type, you can access and get information about disk files, drives, and directories. ObjectPAL's fileBrowser procedure lets you display the Paradox File Browser.

Messages and dialog boxes

See also

Dialog boxes allow you to interact with Paradox. The System type procedures display dialog boxes, such as **dlgAdd** and **dlgCopy**, begin with the prefix *dlg*. Procedures that display messaging dialog boxes, such as **msgInfo** and **msgQuestion**, begin with the prefix *msg*. The former do not return values; the latter return values such as Yes or No, in mixed upper- and lowercase. You can also design your own dialog boxes.

A dialog box in Paradox is really a form whose properties are set to make it act like a dialog box. To see the Window Style dialog box, create a new form and select Form|Window Style. Here you can add scroll bars and a standard menu, specify whether the dialog box is modal, and more. You can even open a normal form as a dialog box with the **openAsDialog** method. If you open a form from another form, you use the **wait** method to suspend execution in the first form until a **formReturn** method closes the second form.

Dialog boxes can be informational, displaying a simple message; or complex, for instance, prompting the user to enter search criteria for a query. There are many examples of the functionality of dialog boxes that you can explore in the *dlg* and *msg* topics. Also, the **message** procedure makes it easy to display up to six strings in the status bar.

Guide to ObjectPAL

- Chapter 2, "Learning ObjectPAL
– A Tutorial"

Handling keyboard events

See also

Keyboard events occur whenever you enter data at the keyboard and whenever a keystroke autorepeats. Paradox provides two built-in event methods for capturing KeyEvents: **keyChar** and **keyPhysical**.

Whenever you press a key, the event first goes to the form, which processes it if possible, as when F1 is pressed, or dispatches it to the active object. The active object's built-in code then calls the **keyPhysical** method. If the keystroke represents an action to be performed, **keyPhysical** calls the **action** method to perform it. Otherwise, **keyPhysical** calls **keyChar**, which displays the character in a field on the active object. (In edit mode, **keyChar** first locks a field before it inserts a character.) If **keyChar** receives a spacebar press when the active object is a button, it triggers the object's **pushButton** method.

You can use **keyPhysical** to create "hot keys," keystroke equivalents of pressing buttons on a form.

Guide to ObjectPAL

- Chapter 10, "Understanding the event model"
- Chapter 12, "Default behavior of event methods"

Working with menus

See also

Paradox provides three types of menus.

Built-in menus and toolbars are complete and easy to use, and require no ObjectPAL coding. There may be times, however, when you want to disable menu options or add functionality to the default menus. Custom menus let you create menu designs for your applications.

If you decide to create a custom menu for your application, first decide where to put your ObjectPAL code. If, for instance, the menu will be available from the entire form, then you might put the code in the form's **open** method. In a multi-page form, where you want different menus for different parts of the form, you might want to put your code in the page's **arrive** method. Next, decide where you want standard menus, which display across the top of the screen, and popup menus. You can then design the layout for your menus. The ObjectPAL Menu Type provides methods for creating and displaying menus, and for receiving user input.

Guide to ObjectPAL

- Chapter 15, "Working with Menus", and Chapter 2, Lesson 9.

Working with lists

See also

Lists and drop-down lists present a group of options or values from which the user selects. Lists are compound objects with two parts: the field object that you place on the form, and the list object, which contains the data shown in the list. You place a field object on the form and set its DisplayType property, either to List or Drop-down Edit.

In a form, place a field object whose DisplayType property is set to Drop-down Edit or List. Click OK in the Define List dialog box. Choose Tools|Object Tree to view the relationship between the field object and the list object. You attach code, using the Object Explorer, to the field object that affects values displayed in the field, and to the list object that affects values displayed in the list. Use the DataSource property to specify the source table and field whose data appears in the list.

You use a TCursor and its various methods to search the table field for a particular value, to add the value to the table if it does not exist, to edit the value, and more.

There are three List properties that are particularly useful:

list.count shows the number of items in a list. Setting this value to zero clears the list.

list.selection sets the current index pointer in the list.

list.value sets the value of the item currently pointed to by list.selection.

Guide to ObjectPAL

- Chapter 2, "Learning ObjectPAL
– A Tutorial"

Multi-form applications

See also

You may want to create applications that use more than one form or dialog box (which is itself a special kind of form). Use multiple forms sequentially when one task must be completed before another begins. Use multiple forms simultaneously to divide a complex application into functional modules.

Before you create a complex multi-form application, decide how you want its forms and dialog boxes to interact. Dialog boxes are either modal or non-modal. Typically, you use modal dialog boxes in sequential applications; that is, whenever user input is required before the application can continue. Modality prevents the focus from changing to other windows, forms, system menus, and Windows applications. You use non-modal dialog boxes and standard forms in simultaneous applications; that is, whenever you want the user to have simultaneous access to other windows and resources.

How can you tell the difference between the behavior of modal and a non-modal dialog boxes? A modal dialog box cannot be resized. You respond to the dialog box, either by typing in data, clicking a button or closing it, before you can change focus to another object. A password dialog box is usually modal; you either enter a valid password or exit.

A non-modal dialog box can be resized, and allows you to change focus to another window; a text search dialog box that lets you change focus to the underlying document is typical. Both modal and non-modal dialog boxes always rest on top of open forms, and both can be moved on top of the Toolbar.

ObjectPAL provides pre-built dialog boxes; the System Type procedures that you use to display them begin with the prefixes *msg* or *dlg*. You can also design your own dialog boxes. You can even open a standard form as a dialog box with the openAsDialog method, specifying display attributes with WindowStyles constants.

The FormType methods, especially **wait**, **close**, and **formReturn** give you power and flexibility in handling the interaction of multiple forms and dialog boxes in applications.

Guide to ObjectPAL

- Chapter 21, "Displaying Output"
- Chapter 28, "Developing Multi-form Applications"

Working with text files

See also

To work with ANSI text files, you use the TextStream data type. The TextStream type includes all ANSI characters, including such non-printing characters as the carriage return and line feed. To work with formatted text files, which include such attributes as font, alignment, and margins, use the Memo type.

There are TextStream methods for such tasks as opening and closing files, reading one line or one character of text at a time, reading from and writing to disk, and setting the position of the pointer within a file. A TextStream pointer shows the current position in the file counted from the beginning: 1 is the first character, 2 the second, and so on. The TextStream **advMatch** method searches a file for a pattern, similar to the String **advMatch** method.

TextStreams and Strings are related objects that both contain text; only TextStreams, however, can read from and write to files on disk.

Guide to ObjectPAL

Chapter 27, "Working with Text Files"

Using DLLs

See also

DLLs, or Dynamic Link Libraries, store functions in a library external to Paradox and ObjectPAL. Before you call a custom, external routine from ObjectPAL, you first declare it in a **uses** clause at the beginning of your method or procedure, or in the Uses window available from the Paradox Object Explorer.

It is best to use Paradox and ObjectPAL methods and procedures when building applications, because they are efficient and powerful. You create a DLL only when you want to add to or enhance Paradox capabilities. For instance, you may want to call routines in COMMDLG32.DLL to use a Windows common dialog box; or in USER32.EXE to use other Windows functions; or to create a custom DLL for serial port access. If you have an existing application that uses 16-bit DLLs or 16-bit Windows calls, you need to replace the DLLs and Win API calls with 32-bit versions.

Guide to ObjectPAL

Chapter 33, "Using Libraries"

Working with the file system

See also

You use the FileSystem data type to work with disk files, drives, and directories. FileSystem variables provide a handle for access to files. The first step is often to call the **findFirst** method for a FileSystem variable, which checks whether a file or directory exists; if so, **findFirst** initializes the variable with a handle to the file or directory. Then you use the FileSystem methods to work with drives, directories and files.

Paradox provides a File Browser for performing interactive file system tasks. The ObjectPAL System Type provides the fileBrowser procedure, which displays the Paradox FileBrowser.

ObjectPAL language categories

See also

Paradox and ObjectPAL let you create compiled applications from these major components:

Category	Description	Object types
Data model objects	Let you work with data stored in tables	Database, Query, Table, TCursor, SQL
System data objects	Let you store data, but not in tables	DDE, FileSystem, Library, Session, System, TextStream
Data types	The basic ObjectPAL data types	AnyType, Array, Binary, Currency, Date, DateTime, DynArray, Graphic, Logical, LongInt, Memo, Number, OLE, Point, Record, SmallInt, String, Time
Design objects	Let you create the user interface to your application	Menu, PopUpMenu, UIObject Toolbar,
Display managers	Let you control how data is presented to the user	Application, Form, Report, Script, TableView
Event Types	Contain information about actions in Paradox	ActionEvent, ErrorEvent, Event, KeyEvent, MenuEvent, MouseEvent, MoveEvent, StatusEvent, TimerEvent, ValueEvent

With Paradox and ObjectPAL, you build applications incrementally. First, build tables, forms and objects in Paradox, and write custom ObjectPAL methods that alter the default methods. Next, test, debug, and refine the application. Continue to add tables, objects, and methods, and debug them until the application is complete.

Objects (overview)

See also

In Paradox, most of the things you work with are objects the buttons and fields you create using the Toolbar, tables and text files stored on disk, and menus created in code, to name a few. Paradox recognizes two kinds of objects: design objects and data objects. You place design objects, such as buttons, list boxes, and other UIObjects, in forms. Data objects are files, data types, and programming structures.

All Paradox objects have

Properties	attributes such as color, font, line width
Methods	code that defines how the object responds to an event

Most objects (except for OLE controls) you can create or modify using Paradox interactively, you can create or modify using ObjectPAL.

Objects and methods

Creating custom Paradox applications is largely a process of placing objects in forms, and then writing ObjectPAL methods to define how those objects respond to events. Such applications are sometimes called "Hey you, do this" applications. The "Hey you" part (called an event) happens when the user does something to an object for example, points to a button in a form and clicks the mouse. The "do this" part is defined by methods

code that runs when the object handles an event. Some objects can display other objects (such as a lookup list), or chain to another stage in the application (for example, another form, query, or report).

Object types

ObjectPAL objects are grouped by type. ObjectPAL Language Categories contains groups of types and gives you access to help on the methods in each type.

For each method, you'll find syntax, description, and sample code you can copy and paste into your own code through the Clipboard.

Properties

Each type of object has properties, attributes such as focus, value, name and visible, which are appropriate to that type. For instance, buttons have the ButtonType, CenterLabel, and Name properties; box objects have the Color, Frame.Color, and Size properties; and forms have DialogForm, HorizontalScrollBar, and SnapToGrid properties.

Events

Paradox recognizes certain actions or conditions within forms as events. When Paradox detects that an event has occurred, it triggers execution of the method associated with that event. There are different types of events that are appropriate for different types of objects. For instance, the pushButton event is recognized by a pushButton object, but not by a graphic object. There are, however, events that are recognized by most or all objects, such as the timer event, events related to focus, and opening and closing.

To understand how Paradox processes events, you must understand bubbling, the process by which events pass from the target object up through the containership hierarchy. When an event occurs, the target object of the event does not immediately process it. Instead, the event passes up to the form level, where the form determines whether to process the event, to send it back to the target object, or to send it to another object for processing. The object that finally processes the event triggers the method (code) associated with the event.

Events (overview)

See also

Examples of events are:

- Pressing the mouse button
- Releasing the mouse button
- Moving the mouse pointer over an object
- Pressing a key
- Moving the cursor into a field
- Moving the cursor out of a field
- Selecting an item from a menu

Events can happen for other reasons, too. For example, the timer event happens after a certain amount of time passes. You can also generate events from within your own methods.

Using ObjectPAL, you can create methods that define how objects respond to events. All objects have default methods for ObjectPAL events. You don't have to write methods for all the events an object can handle, and an event never goes unrecognized.

Properties (overview)

See also

Objects have properties such as color, pattern, font, and line width.

You can use Paradox to set and change these properties or you can do it with ObjectPAL. Almost everything you can do in Paradox, you can do in ObjectPAL.

For example, the following statements set the color of rectangle *box1* to red, set the font of field *field1* to Times, and make *myCircle* invisible.

```
box1.color = "Red"    ; sets color of box1 to red
field1.font = "Times" ; sets font of field1 to times
myCircle.visible = No
```

List of data types

See also

The ObjectPAL language contains the following data types:

AnyType

Array

Binary

Currency

Date

DateTime

DynArray

Graphic

Logical

LongInt

Memo

Number

OLE

Point

Record

SmallInt

String

Time

Containership

See also

Paradox objects coexist in a hierarchy of containers: When you place a table object, for example, on a page of a form, that page **contains** the table. Forms contain tables, tables contain records, records contain fields, fields can contain buttons, and so on.

An object is contained only if it is completely within the boundaries of the container.

Position in this hierarchy is important because it defines what an object can see of other objects their properties and their variables.

An object cannot see variables in the objects it contains.

An object can see its own variables, as well as the variables in objects that contain it.

To put it another way, if you think of objects as boxes containing smaller boxes, the smallest box has the best view.

Basic language elements (overview)

See also

Basic language elements are the fundamental structural elements of ObjectPAL. Most of these elements are not bound to specific object types; they work for all object types. You can use these elements to assign values, call functions from DLLs, build control structures like **if...then...else...endIf** loops, **while...endWhile** loops, and **switch...case...endSwitch** structures. You can also declare methods, procedures, constants, variables, and data types.

About procedures

See also

There are two kinds of procedures in ObjectPAL: procedures in the ObjectPAL run-time library (RTL) and custom procedures you create. Procedures in the RTL are just like methods except they never explicitly specify an object. A custom procedure resembles procedures in many other programming languages; it is a routine you write yourself and use like a subroutine.

RTL procedures

See also

The procedures in the ObjectPAL run-time library are just like ObjectPAL methods, with one exception: Procedures never specify an object. Any method in any object can call any ObjectPAL procedure, and the procedure will know what to do. For example, the statement

```
close()
```

calls the Form type procedure **quit**, which closes the current form. The System type includes a number of procedures for interacting with users, for example **message**, **msgInfo**, and **msgStop**.

```
msgStop("Alert!", "This file already exists.")
```

The System type also includes the procedures **beep** and **sleep**, and several enumeration procedures for getting and setting the mouse position and shape.

```
method pushButton(var eventInfo Event)
beep()             ; plays the system beep sound
sleep(2000)        ; waits for 2 seconds
beep()
message("Did you hear two beeps?")
                   ; displays a message in the status line
sleep(2000)
enumAllObjectSource("mySource.db")
                   ; creates a table of all methods in this form
endMethod
```

Like ObjectPAL methods, ObjectPAL procedures are associated with object types, and they execute in response to events. It may be helpful to think of ObjectPAL procedures as methods with the object implied.

Custom procedures

See also

Custom procedures in ObjectPAL resemble procedures in many other programming languages. A custom procedure is a routine you write and use like a subroutine.

Custom procedures can be attached to the object itself, or to any object in the containership hierarchy, or to the form itself.

Custom procedures can be included in libraries, but can only be invoked from within the library.

Note: ObjectPAL can call a custom procedure faster than it can a call custom method. The code executes at the same, speed, but ObjectPAL can "find" a procedure faster than it can find a method.

Use a **proc** block to declare a custom procedure. The structure is

```
PROC name (parameterDescription) [return type]
   [CONST section]
   [TYPE section]
   [VAR section]
   [ObjectPAL statements]
ENDPROC
```

You can declare procedures in two places:

- Within a method
- In an object's Proc window

Procedures declared in methods

See also

A procedure declared in a method is private: Its scope is limited to the method in which it is defined.

Here's an example of a custom procedure:

```
proc inc(x SmallInt) SmallInt
   return x+1 ; increments a number
endProc
```

The following example shows how to call that procedure (and another one) from within a method. (In this example, it's the **pushButton** method, but it could be any method.)

```
proc inc(x SmallInt) SmallInt
   return x+1
endProc

proc showMe(x SmallInt)
   msgInfo("myNum = ", x)
endProc

method pushButton(var eventInfo Event)
var
   myNum SmallInt
endVar
   myNum = 3
   showMe(myNum)
   myNum = inc(myNum)
   showMe(myNum)
endMethod
```

Procedures declared in an object's Proc window

See also

A procedure declared in an object's Proc window has the same syntax as a procedure declared in a method, but it has a different scope.

A procedure declared in an object's Proc window is visible to all methods attached to that object, and to all methods in objects contained by that object. So, to make a procedure available to every object in a form, declare it in the form's Proc window.

About methods

See also

A method is code that defines the behavior of an object in response to events. ObjectPAL methods fall into one of three categories:

- Built-in event methods included with every Paradox object
- Methods in the ObjectPAL run-time library
- Custom methods you create

Editing a method

See also

To edit a method for an object, right-click the object, select Object Explorer, and click the Events tab. Select a method from the Object Explorer by double clicking it. The code for the method appears in the Editor window.

To edit the method for a form, right-click the form's title bar and then choose Object Explorer. Click on the Events tabbed page on the right side of the Object Explorer window and double-click the method you want to edit.

You can type the text for a method directly in the Editor, or use the Clipboard to copy, cut, and paste methods and parts of methods from other objects. However, there is no linkage or relationship between the original method and the copied method. Changes made to one are not reflected in the other.

You can also copy a method by copying an object from a design document. When you copy an object, all methods attached to the object are copied as well. However, there is still no linkage after the copy.

Built-in event methods

See also

Every Paradox object comes with built-in event methods (for example, **open**, **close**, and **mouseUp**) for each event it can respond to. The built-in code for these methods specify an object's default behavior in response to a given event. You can add your own code to built-in event methods using the ObjectPAL Editor.

To edit built-in events method for an object, right-click the object and select Object Explorer from its menu. Choose the Events tabbed page on the right side of the Object Explorer window, and select the events:

To edit one method, select the method from the Object Explorer, then press Enter, or right-click the method and choose Edit Event.

To edit more than one method, select them using Shift+click or Ctrl+click, then press Enter. An ObjectPAL Editor window opens for each event selected.

You can type the text for a method directly in the ObjectPAL Editor, or use the Clipboard to copy, cut, and paste methods and parts of methods from other objects.

Methods in the run-time library

See also

The ObjectPAL run-time library (RTL) is a collection of predefined routines. It includes methods you can use to perform a wide range of tasks, from reading and editing data in tables to creating and displaying menus. Each of these methods is associated with an object type; all the methods for working on forms are in the Form type, all the methods for working with text files are in the TextStream type, and so on.

ObjectPAL methods are symmetrical and consistent. Within a type, methods often come in pairs. For example, if a type has an **open** method, you can expect it to have a **close** method, too. If you can read information from an object, you can write to it; if you can get a value, you can set it.

ObjectPAL methods are consistent across types because methods with similar names do similar things. For example, **open** makes an object available for manipulation, whether the object is a table or a text file, and **close** puts it away. The underlying code may differ, but conceptually, the results are the same.

Methods in the run-time library require you to use dot notation to specify an object to operate on.

Custom methods

See also

Custom methods are auxiliary methods you create. They are convenient for making frequently used routines available to several objects.

Custom methods attached to a form are available to all objects in the form. That way, you only have to maintain the code in one place.

To create a custom method, right-click an object, choose Object Explorer, choose the Methods tabbed page on the right side of the Object Explorer window, then choose <New Method>. In the New Method box, type a name for the custom method, then choose OK to open an ObjectPAL Editor window. You can type or paste text into custom methods just as you can for built-in event methods.

After you save a custom method, its name is listed in the Object Explorer. To make changes, choose the name and open an ObjectPAL Editor window, just as you would to edit a built-in event method.

You can copy, cut, and paste an entire object. When you do, all methods attached to the object are copied as well. However, there is no link or relationship between the original method and the copied method. Changes made to one are not reflected in the other.

Methods in other objects

See also

Methods are public: that is, methods attached to an object can be called by other objects. For example, suppose a form contains two boxes: *box1* and *box2*. If *box1* has a method **fred**, *box2* could use dot notation to call it:

```
box1.fred()
```

If you attach a custom method to a form, which is the top level of the containership hierarchy, all objects contained in the form have direct access to that method. For example, if you attach the custom **goNextPage** method to a form, a button on that form could call **goNextPage** like this:

```
method pushButton (var eventInfo Event)
goNextPage() ; this is a custom method attached to the form
endMethod
```

In this example, we didn't have to use dot notation because the **pushButton** method is attached to the button, and the button is contained by the form, so it has direct access to the form's methods.

When you compile this method, ObjectPAL searches other objects for **goNextPage**, so it executes without delay at run time.

Method language structure and syntax

See also

In terms of structure and syntax, ObjectPAL methods resemble traditional programs. Some aspects of this structure are:

- Methods can have parameters (also called arguments).
- Methods are delineated by the **method...endMethod** keywords. You can define an ordered structure of execution because ObjectPAL supports control structures and loops like **while...endWhile**, **if...then...else...endIf**, and **switch...case...endSwitch**.
- As in Pascal and C, you can define procedures to perform one or more tasks. Procedures can receive arguments from and return results to the method that calls them.
- Also as in C, you can freely use whitespace (tabs, spaces, and blank lines). You can choose to indent subordinate method lines, put one or more statements on a line, and append a comment to any method line
- whitespace has no effect on how statements are executed.

Variables

See also

A variable is like a slot where you can temporarily store one item of information.

The value of a variable can be of any ObjectPAL type (also called a data type). It is not necessary to explicitly indicate a data type for variables.

Specifying a variable's data type before using it is called declaring a variable.

The simplest way to give a variable a value is to use the assignment operator (=).

The scope of a variable

See also

The term "scope" means "accessibility." The scope of a variable, that is, the range of objects that have access to it is defined by the objects in which it is declared, and by the containership hierarchy. Objects can access only their own variables and the variables defined in the objects that contain them. Also, the scope of a variable depends on where it is declared:

Within a method

Variables declared within a method are visible only to that method, and are accessible only while that method executes. They are initialized (reset) each time the method executes.

Outside a method

Variables declared in a method window before the keyword **method** are visible only to that method, but are not initialized each time the method executes.

In the Var window

Variables declared in an object's Var window are visible to all methods attached to that object, and to any objects *that* object contains. A variable declared in an object's Var window is attached to the object, and is accessible as long as the object exists in the form and the form is open.

Within the containership hierarchy (compile-time binding)

In programming terms, "binding" a variable is the process of connecting a variable to a data type. The ObjectPAL compiler binds variables when it compiles the source code; there is no run-time binding in ObjectPAL. When the compiler encounters a variable in a statement, it searches the rest of the source code to find out where the variable is declared so it can bind the variable to the declared data type.

Constants

See also

The ObjectPAL language includes many predefined constants. Constants are like variables except they are protected from change when the program runs, enabling the compiler to generate more efficient code.

You can define constants for a single method, or open a Const window to define constants for all the object's methods.

Constants are automatically put into resources, where they can be modified without affecting the source code.

Introduction to scripts

See also

A script consists of code in its own file, not attached to a form. It is an object, and displays on the Desktop as an icon. Use a script when you want to execute code without opening and displaying a form window. You can

- Attach code to one or more built-in event methods.
- Declare variables, constants, data types, custom methods, and procedures.
- Call custom DLLs.

A script does not display in a window, and does not contain any design objects. A script has the built-in event methods **run**, **error**, and **status**. (You must set your ObjectPAL Level to Advanced in the General page of the Developer Preferences dialog box to display **status** in the list of built-in event methods.) You can execute these methods using Paradox interactively, or call them from within an ObjectPAL method or procedure. Like any other object, a script also has windows for declaring variables, constants, procedures, data types, and external routines. You can also declare custom methods.

From a script, you have complete access to the ObjectPAL run-time library, so you can control other objects. For example, you can call other scripts, open and work with tables, forms, and reports, and run queries. You can call methods attached to other objects, and get and set their properties.

A Script type was added to ObjectPAL in version 5.0. It includes methods for creating and manipulating scripts and the code they contain from within an ObjectPAL method or procedure.

Creating a script

See also

Choose File|New|Script to create a script. An ObjectPAL Editor window opens for the Script's built-in **run** method. This is where you type the code. This is a standard ObjectPAL Editor window, so you can edit, check syntax, and debug the **run** method as you would any other object. Keep in mind, though, that whatever you declare is visible only to the script's **run** method.

When you are finished editing, close the window. A dialog box prompts you to enter a name for this script. Enter a name and choose OK to save the script to disk. Like a form, a script can be saved or delivered. Saved scripts can be changed; delivered scripts cannot.

You can also create a script using ObjectPAL. See Script Type for more information.

About adding code to a script

See also

To add code to a script, choose File|Open|Script or File|New|Script. Then attach your code in the Editor window that opens. When you are finished writing code, choose File|Save to save the script (both source code and executable code) to disk. Or, to save only the executable code, choose Program|Deliver.

Using the Object Explorer and ObjectPAL Editor windows, you can add code to a script in the following ways:

- Attach code to the built-in event methods.
- Add custom methods.
- Add custom procedures.
- Declare variables, constants, data types, and external routines.

You can also add code to a script using ObjectPAL. See Script Type for more information.

Attaching code to built-in event methods

See also

Every script has the following built-in event methods: **run**, **error**, and **status**. (You must set your ObjectPAL Level to Advanced to display **status** in the list of built-in event methods.) You can attach code to these built-in event methods as you would with any other object. You do this in the Editor window.

You can also attach code to a script's built-in event methods using ObjectPAL. See Script Type for more information.

Adding custom methods

See also

You can add custom methods to a script using the Object Explorer.

To display the Object Explorer, right-click the Script window and double-click <New Method>, or press Ctrl+Spacebar. Then type in a name for the new custom method, just as you would for any other object, and choose OK to open another Editor window.

For information on adding custom methods and programming using ObjectPAL, see your ObjectPAL documentation.

You can also add custom methods to a script using ObjectPAL. See Script Type for more information.

Adding custom procedures

See also

From a script's Object Explorer, you can choose Proc to open an Editor window where you can declare custom procedures for the script.

For information on adding custom procedures and programming using ObjectPAL, see your ObjectPAL documentation.

You can also add custom procedure to a script using ObjectPAL. See Script Type for more information.

Declaring variables, constants, data types, and external routines

See also

From a script's Object Explorer, you can declare variables, constants, data types, and external routines by choosing Var, Const, Type, or Uses, respectively, to open the appropriate Editor window. Items declared in these windows are global to the script but cannot be accessed by other forms or objects.

To edit a script

See also

You can edit a script in an ObjectPAL Editor window.

To edit a script

1. Choose File|Open|Script and select the script you want to edit.
2. Click Edit the script at the bottom of the Open Script dialog box, then click Open. ObjectPAL opens your script in an Editor window with the built-in event method **run** displayed.
3. Use the ObjectPAL Editor to edit your script as you would a method.

To debug a script

See also

You can debug a script using the ObjectPAL Debugger.

To debug a script

1. Choose File|Open|Script and select the script you want to debug.
2. Click Edit the script at the bottom of the Open Script dialog box, then click Open. The script opens in an Editor window.
3. Set breakpoints and watch points as desired.
4. Run the script.

When ObjectPAL encounters the breakpoint, execution stops and the script opens in a Debugger window. Use the ObjectPAL debugger to debug the script as you would a method.

About playing a script

See also

You can play a script using Paradox interactively or from within a method. In either case, the result is that you execute the script's built-in **run** method.

To play a script interactively

See also

To play a script interactively,

1. Choose File|Open|Script. A dialog box lists available scripts.
2. Choose one of the scripts, choose Run the script at the bottom of the Open Script dialog box, and choose Open. The script's built-in **run** method executes.

To play a script programatically

See also

Use the System type method play to play a script from within a method or procedure. For example:

```
switch
   case theValue = "this" : play("doThis")   ; play script "doThis"
   case theValue = "that" : play("doThat")   ; play script "doThat"
   otherwise              : play("theOther") ; play script "theOther"
endSwitch
```

Delivering a script

See also

Use File|Deliver to deliver a script you have created. When you deliver a script, Paradox removes all the source code. Your code is not lost; it is protected.

If you save the script using File|Save, anyone who uses it can modify the ObjectPAL code, changing your application. Delivery gives you a way to let others use your code, but not change it.

When you choose File|Deliver, Paradox saves a copy of the script with an .SDL extension. You can still change your ObjectPAL code using the script with its .SSL extension, but if you want others to use it safely, give them the delivered script.

Introduction to libraries

See also

A library is a collection of custom methods and procedures. Libraries are useful for storing and maintaining frequently used routines, and for sharing custom methods and variables among several forms. When you choose File|New|Library, Paradox opens the Library window.

When you right-click the Library window and choose Object Explorer, Paradox opens the Object Explorer.

In many ways, working with a library is like working with a form. For example, a library has built-in event methods. You add code to a library just as you do to a form, using the Object Explorer and the ObjectPAL Editor. As with a form, you can open Editor windows to declare custom methods, procedures, variables, constants, data types, and external routines.

However, there are some important differences:

- At run time, a library does not display in a window.
- A library cannot contain design objects; it can contain only code.
- In a Library, statements that use Self do not refer to the Library
- instead, they refer to the object that called the method.
- The scoping rules are different for libraries.

Choose File|New|Library to create an ObjectPAL library.

Library methods

See also

You can use the Library methods in your own code attached to any object, even another library to manipulate a library. See Library Type for a list of the run-time methods.

Calling library methods

See also

To call a method in a library, you must first declare the library in the Uses window of the object doing the calling. For example, suppose you want a button's **pushButton** method to call a custom method from a library. Declare the library in the button's Uses window (or in the Uses window of an object that contains the button), so Paradox knows where to look for the method, and knows what arguments it will take.

Creating a library

See also

To create a new library, choose File|New|Library. Right-click the Library Design Window to open the Object Explorer. Then add your code as you would to any other object.

When you are finished adding code, you can

- Save both the source code and the executable code (to an .LSL file) by choosing File|Save.
- Save only the executable code (to an .LDL file) by choosing File|Deliver.

About adding code to a library

See also

To add code to a library, right-click the Library Design window to open the Object Explorer. Then attach your code as you would with any other object. You can add code to a new or existing library.

When you are finished adding code, you can

- Save both the source code and the executable code (to an .LSL file) by choosing File|Save.
- Save only the executable code (to an .LDL file) by choosing Program|Deliver.

Using the Object Explorer and ObjectPAL Editor windows, you can add code to a library in the following ways:

- Attach code to the built-in event methods.
- Add custom methods.
- Add custom procedures.
- Declare variables, constants, data types, and external routines.

Attaching code to built-in event methods

See also

Every library has the following built-in event methods: **open**, **close**, and **error**. You can attach code to these built-in event methods, as you would with any other object, in the Editor window.

A library's built-in **open** method is called when the library is first opened; **close** is called when the library is being closed; **error** is called when code in the library generates an error. Typically, you will use *open* to initialize global library variables, and use **close** to "tidy up" after using the library. By default, a library's **error** method calls the **error** method of the form that called the library routine.

For information on attaching code to built-in event methods and programming using ObjectPAL, see the *Guide to ObjectPAL*.

Adding custom methods

See also

The custom methods in a library can be called by other methods in the same library, by methods in other forms, and by methods in objects in other forms. This accessibility makes libraries very useful.

To add a custom method, right-click the Library Design window and choose Object Explorer from the menu that appears. Click the Methods tab and double-click <New Method>. Type in a name for the new custom method, then choose OK to open another Editor window.

For information on adding custom methods and programming using ObjectPAL, see the *Guide to ObjectPAL*.

Adding custom procedures

See also

To add a custom procedure, right-click the Library Design window and choose Object Explorer from the menu that appears. In the Object Explorer, click the Methods tab and double-click Proc to open an Editor window where you can declare custom procedures for the library.

Note: Unlike custom methods, which can be called from other forms and other objects, custom procedures can only be called from within the library in which they are declared.

For information on adding custom procedures and programming using ObjectPAL, see the *Guide to ObjectPAL*.

Declaring variables, constants, data types, and external routines

See also

From a library's Object Explorer, you can declare variables, constants, data types, and external routines by choosing Var, Const, Type, or Uses, respectively, to open the appropriate Editor window. Items declared in these windows are global to the library, but are not available to other forms or objects. However, other forms and objects can call library routines that access these variables.

For information on declaring variables, constants, data types, and external routines, and programming using ObjectPAL, see the *Guide to ObjectPAL*.

Debugging called libraries

See also

To debug ObjectPAL code on a form or library that is called from another form or library, you must set a breakpoint both in the ObjectPAL code that calls the second form or library and in the code on the second form or library. While single-stepping through the code that calls the second form or library, step into the method or procedure on the second form or library.

To edit a library

See also

You can edit a library in an ObjectPAL Editor window.

To edit a library,

1. Open the library you want to edit. An empty Library window opens.
2. Right-click the Library window and choose Object Explorer from the menu that appears.
3. On the Methods page, double-click the method you want to edit.

An Editor window opens for the method you selected. Use the ObjectPAL Editor to edit this as you would any method.

To deliver a library

See also

When you deliver a library, Paradox removes all the source code. Your code is not lost; it is protected.

If you save the library using File|Save, anyone who uses it can modify the ObjectPAL code, changing your application. Delivery gives you a way to let others use your code, but not change it.

To deliver a library

- Choose File|Deliver.

Paradox saves a copy of the library with an .LDL extension. You can still change your ObjectPAL code using the library with its .LSL extension, but if you want others to use it safely, give them the delivered library.

For information on developing applications using libraries and programming using ObjectPAL, see the *Guide to ObjectPAL*.

Controlling the scope of a library

See also

The scope of a library refers to its accessibility that is, which objects have access to the library's code. A Library variable follows the same scoping rules as any other ObjectPAL variable. Two things determine a library's scope: where the Library variable is declared and how the library is opened.

Declaring a library variable

See also

A Library variable follows the same scoping rules as any other ObjectPAL variable. A variable declared in a method is available as long as that method is executing. A variable declared in an object's Var window is available to all methods attached to that object, and to all objects that object contains.

To make a library available to all objects in a form for as long as that form is running, declare the Library variable in the form's Var window, and declare the library routines in the form's Uses window.

Using library variables as arguments

See also

You can use a Library variable as an argument in a custom method or custom procedure.

By passing a library as an argument, you can change the behavior of a routine (method or procedure) and still maintain the routine's independence. A routine may use a library and routines from the library, but the caller can determine the function of the routines by just changing the library.

Opening a library

See also

The Library method **open** takes arguments that specify the scope. A library can be opened in either of the following ways:

- Private to the form
  Only the form that opened the library has access to its code.
- Global to the Desktop
  Every form on the Desktop can access the library. This lets several forms access the same custom methods and share the same global variables. By default, a library opens. global to the Desktop.

For two or more forms to share the same library, each form must open the library global to the Desktop, and each form must have a Uses window that declares which library routines to use.

About the Object Explorer

See also

The Object Explorer is your entryway to the ObjectPAL Editor. It also lets you view an object tree for the current form, and, in addition, gives you a developer's interface to properties, which you can change in the explorer.

The two panes share four menus:

- The File menu, for closing the Object Explorer
- The Edit menu containing editing commands and Developer Preferences command.
- The View menu, used to specify what part of the Object Explorer to view, whether to hide the main menu, and whether to temporarily pin the Object Explorer to the Desktop.
- The Help menu, for getting help on using the Object Explorer.

You can view both the object tree and the Object Explorer tabbed right pane (showing the object's methods, properties and events), or you can view them individually. These choices are available on the Object Explorer View menu. You can also adjust the size of the panes by dragging the border between the two.

The object tree displays information for the current document; the tabbed pane displays information for the selected object. If you select another object or document, the contents of the Object Explorer change to reflect the new object or document.

The object tree

The object tree shows the hierarchical relationships among objects in the current form. It works like the Windows Explorer: click on a plus (+) icon to expand that node of the tree; click on a minus (-) icon to collapse it. When fully expanded the object tree shows all objects you've placed in the current form. You can move and copy objects in the object tree using the right-click menu. You can also right-click the objects in the object tree to change their properties.

For information on using the object tree, see About the object tree.

The tabbed pane

The tabbed pane contains separate pages to show what custom methods, events, and properties are attached to an object. It lets you change the properties for an object and open individual Editor windows to edit Methods and Events.

For information on using the tabbed pane, see About the tabbed pane.

To open the Object Explorer

See also

Do one of the following:

- Click the Object Explorer button on the Toolbar.
- Choose Tools|Object Explorer.
- Right-click an object, or a group of objects on a form or report, or right-click in an Editor, Library, or Script window, and choose Object Explorer.

To pin the Object Explorer to the Desktop

See also

To open the Object Explorer and keep it pinned to the Desktop whenever you open a Form Design, Library, or Script window,

1. Choose Edit|Developer Preferences, and click the Explorer tab.
2. Select Keep Pinned, and then click OK.

Pinning the Object Explorer keeps it open when you move between the Form Design window and Editor windows until you dismiss it. The Object Explorer automatically opens with the design window, and stays open until you leave the design window.

To pin the Object Explorer temporarily during work on a given form, choose View|Pin Explorer from the Object Explorer itself. When you leave the Form Design window and Editor windows associated with the form, the temporary preference is discarded.

About the object tree

See also

To display the object tree, click the Object Explorer button on the Toolbar, or choose Tools|Object Explorer. You can also open the Object Explorer by right-clicking an object, or a group of objects, and selecting Object Explorer.

Use the mouse or arrow keys to move around in the object tree. When you select a new object, the tabbed pane changes to display the methods, events, and properties for that object.

Forms

A form's object tree shows you the hierarchy of objects in your document (which objects are contained in other objects). The currently selected object appears at the far left, and the tree showing the container hierarchy extends to the right.

When you place an object in a form, Paradox gives it a default name that begins with a pound sign (#). The object tree shows objects you have placed and named, and objects you have placed but have not named.

Names of objects with ObjectPAL methods attached are underlined and marked with an asterisk. You can right-click an object and choose Object Explorer. Then click on the Methods tab on the right side of the Object Explorer window.

Reports

A report's object tree shows a diagram of the bands, fields, and design objects in your report and their relationships to one another.

To open the object tree

See also

1. Do one of the following:

- Click the Object Explorer button on the Toolbar.
- Choose Tools|Object Explorer.
- Right-click an object, or a group of objects, and select Object Explorer.

2. Choose View|Object Tree, or View|Both on the Object Explorer menu.

To view the document's structure

See also

To view the document's structure,

- Choose View|Object Tree on the Object Explorer menu.

To expand and collapse the tree,

- Click on a plus (+) icon to expand that node of the tree.
- Click on a minus (-) icon to collapse it.

When fully expanded, the object tree shows all objects you've placed in the current form.

To copy objects

See also

To copy an object to another object, do either of the following:

- Right-click the object in the object tree and choose Copy. Then select the container object in the object tree and choose Paste.
- Select the object in the object tree, and choose Edit|Copy from the Object Explorer menu. Then select the container object in the object tree, and choose Edit|Paste.

If the container cannot accept the object, because of containership rules, or because the object is too big, you'll hear a beep and the move will not be made.

To move objects

See also

To move an object into another object, do either of the following:

- Right-click the object in the object tree and choose Cut. Then select the container object in the object tree and choose Paste.
- Select the object in the object tree, and choose Edit|Cut from the Object Explorer menu. Then select the container object in the object tree, and choose Edit|Paste.

If the container cannot accept the object, because of containership rules, or because the object is too big, you'll hear a beep and the move will not be made.

To delete objects

See also

- Select the object and choose Edit|Delete from the Object Explorer menu, or press Delete.

▪

About the tabbed pane

See also

The tabbed pane contains separate pages that list the current object's built-in or custom methods, events, and properties, and it lets you define new custom methods and change an object's properties. The tabbed pane allows you to distinguish between methods, events, and properties of OLE controls and native Windows controls, and the form-level features.

Each item on the Methods and Events pages is edited in its own Editor window. Double-click an item on one of these pages, or Shift-click several items and press Enter to open several Editor windows at once. When you add code to an event or method, its name moves to the top of the list, and a little blue ball appears beside it to let you know you have attached custom code.

When you declare a variable, constant, or procedure in one of these windows, it is visible to all methods attached to that object.

F1 Help

Press F1 in the Object Explorer to get help on the selected item in the tabbed pane. If there is only one Help topic for the selected item, pressing F1 takes you directly to that topic. If Help contains multiple topics for the selected word, a Search dialog box appears, listing the topics available for that language element. Select a topic and choose Go To. You can also use the Alphabetical List of Methods.

■

About the Methods page

See also

The Methods page contains the following items:

Use	To declare
Uses	procedures used by the object's methods
Var	variables
Const	constants
Type	types
Proc	procedures
<New method>	Create a new method by double-clicking <New method>, entering its name in the New Method Name text box, and clicking OK. An Editor window opens for the new method.

Built-in event methods

Custom methods

Those objects that contain custom code are marked with a small blue ball.

Note: The Methods page might include prototypes for callable methods that cannot be overridden in an Editor.

Methods can be multiply selected to open multiple Editor windows. After selecting, press Enter.

▪

About the Events page

See also

The Events page lists the events associated with the selected object, including ObjectPAL built-in event methods.

In the Method Inspector in Paradox 5.0, everything (methods, events, var, const, and so on) appeared under "methods." Now, an Events category has been added to accommodate new ObjectPAL support for OCXs and native Windows controls.

OCXs have methods, events, and properties. Their control methods are called directly, just like functions; in addition, in the case of controls, events and methods are distinct and separate. So, an Events category is required. You'll find a control event on the Events page, while the methods associated with it appear on the Methods page.

Now, in ObjectPAL, many methods can be considered to be events, or event-like, even though they can be driven as if they were methods. Because the ObjectPAL built-in event methods are event-related (that is, they respond to events), they have been moved to the Events page and are now referred to in the documentation as built-in event methods.

When you add code to an event or method, its name moves to the top of the list, by default, and a little blue ball appears beside it to let you know you have attached custom code. You can change the way custom methods, OCXs, and built-in event methods sort in the Developer Preferences dialog box, Explorer page. (Choose Edit|Developer Preferences.)

▪

About methods and events for OLE controls

See also

OCXs are embedded in Paradox as form objects. Certain events, methods, and properties associated with OCXs are common to all visual objects on the form. To distinguish between OCX-specific methods and events and those methods and events associated with visual form objects, the Methods and Events pages use color to separate the OCX-specific entries form the Paradox ones. OCX-specific methods and events are in red. They're also marked with a white ball.

To view the tabbed pane

See also

1. Do one of the following:

- Click the Object Explorer button on the Toolbar.
- Choose Tools|Object Explorer.
- Right-click an object, or a group of objects, and select Object Explorer.

2. Choose View|Tabbed Pane, or View|Both on the Object Explorer menu.

To create a new method

See also

1. Right-click an object and choose Object Explorer from its menu.
2. Double-click <New method> on the Methods page.
3. Enter a name for the new method, and click OK.
   An Editor window opens for your new method.

To edit a method or event

See also

1. Select the method or event.
2. Do one of the following:

- Right-click and choose Edit Event.
- Press Enter.

An Editor window opens, where you can edit the method.

To copy a method or event name

See also

1. Select the method or event.
2. Right-click and choose Copy Event Name.
3. Place the insertion point where you want to copy the name.
4. Right-click and choose Paste.

To delete a method or event

See also

1. Select the method or event.
2. Do one of the following:

- Right-click and choose Delete Method or Delete Event.
- Press Delete.

This selection is dimmed if this task is unavailable for the selected event or method.

To attach methods to a form

See also

Attach frequently used custom methods to a form. It is more efficient than copying the method to each object that calls it.

1. Choose Tools|Object Explorer to display the object tree.
2. If the tabbed pane is hidden, choose View|Both from the Object Explorer.
3. Click on the form icon at the top of the object tree to select the form by itself.
4. On the Methods page, choose and edit a method or create a custom method as you would for any other object.

To change properties

See also

To change an object's properties in the Object Explorer,

1. Select the object in the object tree.
2. Click the Properties tab to display a list of the object's properties (appearing in ObjectPAL syntax).
3. Click once on a property to see a drop-down arrow or an ellipsis (...), or both. Either

- Click the arrow, or press Enter to see a drop-down list of choices.
- Click the ellipsis to get a dialog box where you can change several related properties at once (this is available, for example, with font-related properties.)

4. Select from the list, using either the mouse or the arrow keys, or make selections from the dialog box, and choose OK.

You can also right-click an object in the object tree to display a properties dialog box for that object. Use the dialog box to change properties.

Read-only properties

Not all properties can be edited. A small lock icon appears to the left of read-only properties.

▪

About the IDE

See also

When you work in the ObjectPAL integrated development environment (IDE), you are in either the Editor, the Debugger, the Object Explorer, or a design window.

The Editor is connected to the ObjectPAL compiler; the compiler translates the ObjectPAL code you write into machine code a computer can execute. When you use the Editor, the compiler can check your code and report any syntax errors so you can correct them before you try to run the application.

The Editor also works with the Debugger. These two, along with the design window and the Object Explorer provide you with an integrated development environment.

Using the Editor you can edit

- Methods (built-in or custom)
- Procedures
- Uses
- Types
- Constants
- Variables

With the Debugger you can debug methods or procedures.

The code you edit or debug can be attached to a Script, a Library, directly to a form, or to an object in a form.

To set developer preferences

See also

You can control many elements of the IDE by setting your preferences in the Developer Preferences dialog box.

1. Choose Edit|Developer Preferences.
2. Make your changes on any of the five pages, then click OK.

■

About the Editor

See also

The ObjectPAL Editor is a full featured Editor that includes color highlighting, incremental search, smart tab indent, multiple and group Undo, and many other features. It also supports BRIEF- and Epsilon-style editing. In an ObjectPAL Editor window you can write, edit, compile, and debug the ObjectPAL code attached to methods on a form, library, or script. The ObjectPAL Editor works the same whether you are working with an object, a form, a library, a script, or an SQL query.

In addition, the Editor is your gateway to the ObjectPAL language reference:

- You can choose View|ObjectPAL Quick Lookup to display the ObjectPAL language reference for each object type; the syntax for each method; properties and property values for each object; and constants for things like window attributes and error codes.
- Press F1 while the insertion point is in an ObjectPAL language element (in the ObjectPAL Quick Lookup, in an Editor window, or in the Object Explorer), to open a Help topic specific to that language element.

▪

About the Editor menus

See also

The Editor menus are described in Chapter 8 of the *Guide to ObjectPAL.* Or you can also press F1 when on a menu command to see what it does.

▪

Working in the Editor

See also

When an Editor window opens for the first time, some default text appears. For example, if you're editing the open method, the first line reads **`method open`** `(var eventInfo Event)`, the second line is blank, and the third line reads **`endMethod`**. If you accidently change the default text, you can edit it as your would any other text.

The insertion point is positioned on line 2 so you can start typing right away, but you don't have to start typing on that line. You can use the mouse or arrow keys to move the insertion point around, and you can insert blank lines by pressing Enter.

By default, keywords appear in bold, and comments in italics. Comments in code for the ObjectPAL built-in event methods are preceded by a double forward slash in addition to the semicolon to make them stand out. You can change colors and text attributes in the Developer Preferences dialog box on the Colors page.

The Editor does not automatically wrap lines of text. A line extends to the right as you type until you press Enter to begin a new line.

Note: Variables, constants, and procedures declared in a method's Editor window are visible only to that method. To make a variable, constant, or procedure visible to all of an object's methods, choose Uses, Type, Const , Var, and Proc (as many of these as you want) in the Object Explorer. Paradox opens a separate window for each item you choose. (This is called scoping.)

To start the Editor

See also

To start the Editor from a form or an object in a form,

1. Right-click an object in a form, and choose Object Explorer from its menu. (Or, select an object and choose Tools|Object Explorer.)

   The Object Explorer opens, listing the methods, events, and properties associated with the object. The Methods page also includes the items Uses, for declaring external routines; and Var, Type, Const, and Proc, for declaring variables, types, constants, and procedures.

   Note: A little blue ball appearing before an item indicates that it has custom code attached to it. A larger white ball indicates that the item is an OCX. A lock icon indicates that the item is read-only.

2. From the Methods or Events pages, select the item(s) you want to edit:

- To select several contiguous items, use Shift+click.
- To select several noncontiguous items, use Ctrl+click.

3. Press Enter. (If you're selecting only one item, you can double-click the item to open an Editor window.)

A separate Editor window opens for each item you selected. You can open as many windows as your system allows, in any order.

You must edit each method in its own window; however, you can edit more than one method at a time by opening multiple windows.

To start the Editor from a Library,

1. Choose File|Open|Library and select the library to open.
2. Right-click the Library window, and choose Object Explorer.
3. When the Object Explorer opens, choose the method you want to edit.

To start the Editor from a Script,

- Choose File|Open|Script.

When you open a Script, it is automatically in an Editor window.

The first line names the method and the last line ends it. The insertion point waits on the third line, where you can begin typing the method.

To move around the Editor with the keyboard

See also

Use the following keys to move around in the Editor:

Ctrl+left arrow	Moves the cursor one word to the left.
Ctrl+right arrow	Moves the cursor one word to the right.
Home	Moves the cursor to the beginning of a line.
End	Moves the cursor to the end of a line.
Ctrl+Home	Moves the cursor to the beginning of the text.
Ctrl+End	Moves the cursor to the end of the text.
Page up	Moves one screenful back.
Page down	Moves one screenful forward.
Backspace	Deletes the character to the left of the cursor.
Delete	Deletes the character to the right of the cursor.
Insert	Has no effect because the Editor is always in insert mode. As you type, characters are pushed to the right. You cannot overwrite characters.
Ctrl+C	Copies selected text to the clipboard.
Ctrl+X	Copies selected text to the clipboard and deletes it from the window.
Ctrl+V	Pastes text from the clipboard into your method.
Tab	Inserts a Tab character and pushes text to the right.

To select text

See also

You can select a block of text by dragging with the mouse, using the arrow keys with Shift held down, or clicking with Shift held down to extend the selection.

- To select a word, double-click it.
- To select an entire line, click to the left of the line and drag the insertion point. (The mouse is in position when the I-beam changes to an arrow.)

To select a block of text, either

- Click and drag the mouse
- Press Shift and use the arrow keys
- Click to indicate the starting position, then press Shift to extend the selection

The keymapping you choose in the Developer Preferences dialog box also affects selection. Press Shift+F1 while on a blank space in an Editor window to see the keystrokes for the keymap you chose.

When text is selected, what happens when you type a character (or paste from the Clipboard) depends on whether you checked Overwrite Blocks in the Developer Preferences dialog box.

Double-clicking to the left of a line toggles a breakpoint and selects the line. See Chapter 9 in the *Guide to ObjectPAL* for more information about breakpoints.

To search for text

See also

You can use Search|Incremental Search to find text in a Editor window, or you can use Search|Find or Search|Replace.

Using Incremental Search

Incremental Search will find text from the insertion point forward in either an Editor or a Debugger window.

1. Place the insertion point.
2. Choose Search|Incremental Search, and then type.

   The Editor highlights the first occurrence of the first character you type. Type another character, and the Editor highlights the first appearance of the pair of characters, and so on. The characters you type appear on the status bar.

   If, as you type, you create a string that has no match in the remainder of the current Editor or Debugger window, you'll hear a beep, and the last character you typed will not appear on the status bar.

   You can use the following keys with Incremental Search:

- Backspace removes the last letter of the search combination and moves back to a prior match.
- Ctrl+S searches for the next combination of the current search string.
- Esc stops incremental search and returns the Editor to its normal state.

Using Find, and Find and Replace

You can use these two commands (from the Search menu) to search for text from the insertion point forward (or backward if you check the Backwards option). The Find And Replace dialog box lets you replace the specified text with a specified value.

To leave the Editor

See also

How you exit the Editor depends on what you want to do next.

To continue designing your form or edit other methods,

- Double-click the Editor's Control menu (or choose Close from the Editor window's Control menu.
- Click the Save Source And Exit The Editor Toolbar button.

To run your code immediately and see your form in action, click the View Data button.

Note: Any Editor windows that are open when you run a form will open again upon returning to the design window.

Saving changes in the Editor

If the Prompt To Save option in the Developer Preferences dialog box is not checked, you are not prompted to save your changes when you close an Editor window or run a form with an Editor window open. All your changes are automatically saved. Choose Edit|Undo All Changes to discard changes before closing the Editor window. Changes you make in Editor windows are saved to disk when you save your form.

If the Prompt To Save option in the ObjectPAL Preferences dialog box (Display page) is checked, a confirmation dialog box lets you save or cancel your changes when you close an Editor window or run a form with an Editor window open.

■

About the ObjectPAL Quick Lookup

See also

The ObjectPAL Quick Lookup is a tabbed dialog box that displays

- Types and Methods
- Objects and Properties
- Constants

From the three pages of the dialog box, you can insert type and method names, object names and properties, and constants into your code at the insertion point (in the Editor). To do this, select the language element you want, then click the appropriate button in the dialog box.

All lists are in alphabetical order. When a panel has focus, you can type one letter to take you to the first item starting with that letter. If you check the checkbox Show All (at the bottom of each page), that page will show you the full spectrum of the ObjectPAL language, as appropriate. If you uncheck the box, you'll see a "beginner's" subset of the language. This check temporarily overrides the preference you set in the Developer Preferences dialog box (General page).

Types and Methods

The left side of this page displays a list of types. When you select a type, the right side lists the methods and procedures for that type. (Methods are marked by an "M"; procedures are marked by a "P.")

A panel at the bottom displays a prototype.

Objects and Properties

The left side of this page displays a list of objects. When you select an object, the right side displays the properties for that object.

A panel at the bottom displays possible values for a selected property. If a property has only one possible value, nothing is displayed.

Constants

The Constants page displays a list of constant types on the left. When you select a type, the right side displays a list of constants relevant to that category.

Constants let you specify things like colors, mouse shape, menu attributes, and window styles.

To use the ObjectPAL Quick Lookup

See also

To open the ObjectPAL Quick Lookup dialog box,

- Choose View|ObjectPAL Quick Lookup. You can click one of the following tabs:

Types and Methods

Objects and Properties

Constants

▪

About keywords

See also

Keywords are basic language elements. They are reserved words in Paradox, and they are selectable from the Program menu.

To use keywords

See also

1. Choose Program|Keywords to display the menu of keywords.
2. Choose an item from the Keywords menu to insert it into your code at the insertion point.

▪

About delivering applications

See also

When Paradox delivers a form, script, or library, it removes the ObjectPAL source code, but leaves the compiled code intact with the file. This lets others use the application but prevents them from seeing or changing your code.

The file-name extensions of delivered applications change as follows:

Application	Undelivered extension	Delivered extension
form	.FSL	.FDL
script	.SSL	.SDL
library	.LSL	.LDL

Note: A delivered form is also protected from design modifications. It cannot be opened in a design window. When you deliver a form, do not forget to deliver copies of all tables in its data model, along with any indexes and referential integrity files.

To deliver an application

See also

- With the form open, choose File|Deliver.

This removes the ObjectPAL source code, but leaves the compiled code intact with the file. Others can use the application, but delivering prevents them from seeing or changing your code.

▪

About Database Expert code examples

See also

The Paradox Database Expert produces small applications that were built using ObjectPAL. You can study the code for these applications to help you write your own code for the same or similar application tasks. To access the applications, run the Database Expert (Tools|Experts). The expert places the forms, tables, reports, and other files that comprise the applications into a directory you specify.

The forms in these applications are not "delivered," and the ObjectPAL code is available to you. You can cut and paste it into your own code, if you want.

Code in the applications built by the Database Expert can show you how to do these programming tasks:

- How to confirm the deletion of records
- How to place memo fields in Memo View
- How to open forms and reports
- How to base reports on queries
- How to use aliases
- How to allow for access to application help (both through the Toolbar and the keyboard)
- How to highlight records
- How to generate drop-down edit field values
- How to program an alphabet bar (Address Book and Contact Management applications)

Also see:

- Documentation on the common library used by all the Database Expert applications. The common library has examples for the following:
- Building a toolbar
- Placing a timestamp in a memo field
- Synchronizing records displayed in multiple forms
- Cascading deletes
- Coding Standards used for the applications.

▪

Coding standards

See also

Below are some general style guidelines used for the ObjectPAL code in the Database Expert applications.

Comments

All comments are formatted with a leading semicolon (;) and double slash (//). For example,

```
;//Comment here
```

Constants

All user-defined constants are formatted as initial lowercase indicating the data type followed in all uppercase. ObjectPAL constants are formatted with camel caps. Examples:

```
strDBNAME   ;//Database name as a string
DataBeginEdit     ;//An ObjectPAL constant
```

Variables

All variables are formatted as initial lowercase indicating the data type followed by camel caps. Examples:

```
strTableName      ;//TableName as a string
fCustomer   ;//Customer form
rInvoice    ;//Invoice report
libMain     ;//Main library
```

Common library

See also

All the applications built by the Database Expert share a common library. The library contains five custom methods. Described below are the custom methods and examples of how to call them.

LoadToolBar( )

Loads a common toolbar to all applications built by the Database Expert. The toolbar is a modification of the standard Toolbar and contains standard navigational buttons, plus insert record, delete record, undo, close form, and help buttons. **LoadToolBar** is called in the **setFocu**s method at the form level of all non-launcher forms. Example:

```
libMain.loadToolBar()
```

DateNotes(strFieldName String)

When called, this method enters the current date and time at the top of the indicated memo on the current form and positions the cursor on the line below it. Example:

```
libMain.DateNotes(fldNotes)
```

CascadeDeletes( )

When called, this method removes the current record and any child records that depend on it. This method does not attempt to remove records from tables that are marked as read-only in the data model. The method is especially useful when you have referential integrity, which forces you to delete detail records before you can delete master records. Example:

```
libMain.CascadeDeletes()
```

SetKeyValue(strFieldName String, anyValue AnyType)

When called, this method creates an entry in a DynArray to place the value of the current object. This must be called before calling **GetKeyValue**.

Generally, **SetKeyValue** is called in the **removeFocus** method of the form. Example:

```
libMain.SetKeyValue("Customer", Customer_Rec_ID.Value)
```

GetKeyValue(strFieldName String) AnyType

When called, this method returns the value that was set into a DynArray via **SetKeyValue**. This is used to help keep forms synchronized when moving between them.

Generally, **GetKeyValue** is called in the **setFocus** method of the form. Example:

```
Customer.locate("Customer", GetKeyValue("Customer Rec ID"))
```

SetKeyValue and GetKeyValue work together to synchonize forms. SetKeyValue places a key value in the library, which getKeyValue can then access in the future to locate the record referred to by getKeyValue.

▪

Use of aliases

See also

All forms create a common alias for the application in the **init** method that points to the directory where the form was loaded. Example:

```
var
   dynFile DynArray[] string
endVar

splitFullFileName(getFileName(), dynFile)
if (NOT addAlias(strDBNAME, "Standard", dynFile["DRIVE"] + dynFile["PATH"]))
then
  errorClear()
endIf
```

Whenever a form, report, library, table or help file is referred to, it uses this alias. This allows the application to work with any working directory. Example:

```
if (NOT libMain.open(strDBNAME + strLIBNAME)) then
    msgStop("Warning:", "Can't open " + strLIBNAME)
    close()
endIf
```

▪

Deleting records

See also

Whenever a user attempts to delete a record (either via Record | Delete, pressing Ctrl+Del, or using the Toolbar), a prompt is generated asking whether the user wants to delete the record or not. This is generally done on the table frame or multi-record object associated with the table. Example:

```
if  (eventInfo.id() = DataDeleteRecord) then
      if (msgQuestion("Warning", "Are you sure you want to delete the current
record?") = "No") then
           eventinfo.setErrorCode(peCannotDelete)
      endIf
endIf
```

▪

Accessing help

See also

All forms can access help via pressing F1, and all non-launcher forms can access help via the Toolbar as well.

F1 can be tested for in one of two methods.

action method

You can test for F1 in the **action** method on the form. Example:

```
if (eventinfo.id() = EditHelp then
   disableDefault
  if (NOT helpShowContext(strDBNAME + strHELP, 10)) then
      msgStop("Warning", "Cannot load help file")
  endIf
endIf
```

keyPhysical method

You can test for F1 in the **keyPhysical** method on the form. Example:

```
if (eventInfo.vChar() = "VK_F1")  then
  disableDefault
  if (NOT helpShowContext(strDBNAME + strHELP, 10)) then
     msgStop("Warning", "Cannot load help file")
  endIf
  disableDefault
endIf
```

To test for the help being selected from the Toolbar, code is placed in the **menuAction** method for the form. Example:

```
if (eventInfo.id() = MenuHelpContents) then
  disableDefault
  if (NOT helpShowContext(strDBNAME + strHELP, 10)) then
     msgStop("Warning", "Unable to load Help file")
  endIf
endIf
```

strDBNAME is an alias for the directory where the application, including the Help file, resides. strHELP is the name of the Help file. 10 is the context ID in the Help file.

▪

How forms are opened

See also

Most forms in the Database Expert applications are designed to be opened only once. If the form is not open, it is opened. If it is already opened, it is brought to the top. Example:

```
if (NOT fOrder.attach("Customer Order Form")) then
  if (NOT fOrder.open(strDBNAME + strORDERFORM)) then
    msgStop("Warning", "Cannot open " + strORDERFORM)
  endIf
else
  fOrder.bringToTop()
endIf
```

Sometimes the code allows a form to be opened more than once. Example:

```
if (NOT fOrder.open(strDBNAME + strORDERFORM)) then
  msgStop("Warning", "Cannot open " + strORDERFORM)
endIf
```

▪

How reports are opened

See also

Reports in the Database Expert applications are opened in a similar fashion to forms. Most are designed to be opened only once. What is different about them is that they open with fit width on (the equivalent of View|Zoom|Fit Width menu command). To accomplish this, reports are initially opened hidden, fit width is turned on, then the report is brought to top, which also displays it. Example:

```
if (NOT rCustomer.attach("Customer Report")) then
  if (rCustomer.open(strDBNAME + strCUSTRPT, WinstyleDefault +
WinStyleHidden)) then
     rCustomer.menuAction(MenuPropertiesZoomFitWidth)
     rCustomer.bringToTop()
  else
     msgStop("Warning", "Cannot open " + strCUSTRPT)
  endIf
else
  rCustomer.bringToTop()
endIf
```

Sometimes the code allows a report to be opened more than once. Example:

```
if (rCustomer.open(strDBNAME + strCUSTRPT, WinsStyleDefault +
WinStyleHidden)) then
  rCustomer.menuAction(MenuPropertiesZoomFitWidth)
  rCustomer.bringToTop()
else
  msgStop("Warning", "Cannot open " + strCUSTRPT)
endIf
```

▪

Reports based on queries

See also

Some reports when launched are based on queries. In order that as few fields as possible need to be protected by the Database Expert, a Check is placed under the table name. This causes all fields from the table to be included in the query.

The applications use two techniques to open reports based on queries. The first technique runs a query to a table and then opens the report based on the query. To use this technique, a report based on the query cannot be open already. So, if the report is already open, it is closed before the query is executed. Example:

```
if (rCustomer.attach("Customer Report")) then
  rCustomer.close()
endif
qCustomer =
  Query
     ~(strDBNAME)Customer | Customer Rec ID            |
     Check           | ~(Customer_Rec_ID.value) |
  endQuery
_recRepInfo.masterTable = ":priv:ANSWER"
_recRepInfo.name = strDBNAME + strCUSTRPT
if (rCustomer.open(_recRepInfo, WinStyleDefault + WinStyleHidden)) then
  rCustomer.menuAction(MenuPropertiesZoomFitWidth)
  rCustomer.bringToTop()
else
  msgStop("Warning", "Cannot open " + strCUSTRPT)
endIf
```

Second technique

A second technique is to open the report based on a query string instead. This technique has the advantage of not having to close an already open report because the using-a query-string guarantees a unique Answer table. Example:

```
strQuery =
  "Query
  " + strDBNAME + "Customer | Customer Rec ID        |
                                    Check                 | " +
string(Customer_Rec_ID.value) + " |
  endQuery"
_recRepInfo.queryString = strQuery
_recRepInfo.name = strDBNAME + strCUSTRPT
if (rCustomer.open(_recRepInfo, WinstyleDefault + WinStyleHidden)) then
  rCustomer.menuAction(MenuPropertiesZoomFitWidth)
  rCustomer.bringToTop()
else
  msgStop("Warning", "Cannot open " + strCUSTRPT)
endIf
```

Memo fields

See also

All memo fields are placed in Memo View when they are selected. This is done by putting code in the **arrive** method of the memo field. Example:

```
doDefault
self.action(EditEnterMemoView)
```

Record highlighting

See also

Numerous forms, such as the checkbook form, use record highlighting to help indicate the current record in a TableFrame.

To do record highlighting, code that sets the selected or unselected color must be placed in numerous methods. The following example shows how record highlighting is implemented in the Checkbook application. To indicate that the current record is selected, place the following in the **canDepart** method for the table frame, and in the **arrive** method of any standalone fields:

```
recCheckbk.color = Yellow
```

Similarly, in the **arrive** method of the record object, place:

```
self.color = Yellow
```

To indicate that the current record is not selected, place the following in the **canArrive** method of the table frame and in the **depart** method of any standalone fields:

```
recCheckbk color = Transparent
```

Similarly, in the **canDepart**, **depart** and **open** methods of the record object place:

```
self.color = Transparent.
```

Drop-down edit fields that are built on-the-fly

See also

Several forms, such as the Checkbook application form, contain drop-down edit fields that allow the user to select from values contained in a specific field of a table. To do this, a query is run in the **arrive** method of the list object of the field, and the drop-down edit field is populated from the query. Example:

```
var
  tcChecksQuery   tcursor
  qbeUniqueCategory     query
endVar

  qbeUniqueCategory =
    Query
      ~(strDBNAME + strCHECKBKTABLE) | Category      |
                                 | Check NOT Blank    |
    EndQuery

  ;// place the results of the query into a tcursor --> very fast!
  if (NOT qbeUniqueCategory.executeQBE(tcChecksQuery)) then
    msgInfo("Listing error" , "Failed to create Category List")
    return
  endIf
  ;// empty the list
  CategoryList.list.count = 0

  ;// set the list items pointer to the first one
  CategoryList.list.selection = 0

  ;// fill the list
  scan tcChecksQuery:
    CategoryList.list.selection = CategoryList.list.count + 1
    CategoryList.list.value = tcChecksQuery.(strCategory)
  endScan
```

Alphabet bar

See also

The Address Book application form and the Contact Management application form each have an alphabet bar that allows you to press a letter on the button bar and go to the first record that has a last name beginning with the letter pressed. If no such name is found, a prompt asks if you want to insert a new record.

The alphabet bar is a field with 26 buttons inside of it. The code to find last names is on the new value method of the field.

```
var
  dynAddressFilter      DynArray[] String
  lCaseFlag Logical
endVar

;// if the reason for the refresh is a value being edited on
;// the field (which in this case would be pressing on a button)
;//

    if (eventinfo.reason()= editValue) then
         doDefault
         lCaseFlag = isIgnoreCaseInLocate()
         ignoreCaseInLocate(Yes)
         if (NOT locatePattern("Last Name" , self.value + "..")) then
           if (msgQuestion("No last names starting with " + self.value, "Do
you want to add a new address?")) = "Yes" then
          mroAddressInfo.postaction(DataInsertRecord)
           endIf
       endIf
      ignoreCaseInLocate(lCaseFlag)
      ;// un-pop the button
      ;//
      self.value = ""
      First_Name.moveTo()
     endIf
```

▪

5.0 ObjectPAL properties

See also

AttachedHeader
AvgCharSize
BottomBorder (read-only)
Breakable
ByRows
CalculatedField
CheckedValue
ColumnPosition
ColumnWidth
CurrentColumn
CurrentRow
DeleteColumn
DeleteWhenEmpty
FirstRow
FrameObjects
GridLines.QueryLook
GridValue
GroupObjects
GroupRecords
Header
HeadingHeight
InsertColumn
InsertField
LeftBorder(read-only)
List.Value
Margins.Bottom
Margins.Left
Margins.Right
Margins.Top
NextTabStop
OtherBandName
PageSize
PageTiling
Picture
PositionalOrder
PrinterDocument
RefreshOption
RemoveGroupRepeats
RepeatHeader
RightBorder (read-only)
RowHeight
SeeMouseMove

Series.Marker.Size
Series.Marker.Style
SeriesName
ShowGrid
SnapToGrid
SpecialField
StandardToolbar
StartPageNumbers
SummaryModifier
TitleBoxName
TopBorder (read-only)
UncheckedValue
WideScrollBar
Width
XAxisName
XAxis.Ticks.TimeFormat
XAxis.Ticks.TimeStampFormat
Xseparation
YAxisName
YAxis.Graph_Title.Text
YAxis.Ticks.TimeFormat
YAxis.Ticks.TimeStampFormat
Yseparation
ZAxisName
ZAxis.Graph_Title.Text
ZAxis.Ticks.NumberFormat
ZAxis.Ticks.TimeFormat
ZAxis.Ticks.TimeStampFormat

▪

5.0 ObjectPAL constants

See also

ActionEditCommand
EditInsertObject
EditPasteLink
EditSaveCrosstab

AggModifier
CumulativeAgg
CumUniqueAgg
RegularAgg
UniqueAgg

DateRangeType
ByDay
ByMonth
ByQuarter
ByWeek
ByYear

FileBrowserFileType
fbDM
fbPrintStyle
fbScreenStyle
fbSQL

FrameStyle
Windows3dFrame
Windows3dGroup

MenuCommand
MenuChangedPriv
MenuChangedWork
MenuChangingPriv
MenuChangingWork
MenuFormViewData
MenuHelpToolbar
MenuHelpCoach
MenuOpenProjectView

PageTilingOption
StackPages
TileHorizontal
TileVertical

PrintColor
prnPrintColor
prnPrintMonochrome

PrintDuplex

prnHorizontal
prnSimplex
prnVertical

PrinterOrientation
prnLandscape
prnPortrait

PrinterSize
prn10x14
prn11x17
prnA3
prnA4
prnA4Small
prnA5
prnB4
prnB5
prnCSheet
prnDSheet
prnEnv9
prnEnv10
prnEnv11
prnEnv12
prnEnv14
prnEnvB4
prnEnvB5
prnEnvB6
prnEnvC3
prnEnvC4
prnEnvC5
prnEnvC6
prnEnvC65
prnEnvDL
prnEnvItaly
prnEnvMonarch
prnEnvPersonal
prnESheet
prnExecutive
prnFanfoldLegalGerman
prnFanfoldStandardGerman
prnFanfoldUS
prnFolio
prnLedger
prnLegal
prnLetter
prnLetterSmall
prnNote
prnQuarto
prnStatement
prnTabloid

PrintQuality
prnDraft
prnHigh
prnLow
prnMedium

PrintSource
prnauto
prnCassette
prnEnvelope
prnEnvManual
prnLargeCapacity
prnLargeFmt
prnLower
prnManual
prnMiddle
prnOnlyOne

prnSmallFmt
prnTractor
prnUpper

SpecialFieldType
DateField
NofFieldsField
NofPagesField
NofRecsField
PageNumField
RecordNoField
TableNameField
TimeField

UIObjectType
BandTool
PageBrkTool

▪

Version 7 ObjectPAL properties

See also

New properties in version 7

There are two new properties in version 7

Enabled

ProgID

▪

Version 7 ObjectPAL constants

See also

New constants in version 7

There are five new types of constants in version 7.

ToolbarBitmap Constants used with the Toolbar::addButton method

ToolbarButtonType Constants used with the Toolbar::addButton method

ToolbarClusterID Constants used with the Toolbar::addButton method

ToolbarState Constants used with the Toolbar::setState method

DesktopPreferenceTypes Constants

In addition, many new MenuCommand Constants were added.

▪

Syntax notation

See also

The following table shows the conventions for ObjectPAL syntax notation.

Convention	Sample	Meaning
Bold font	**beep( )**	Required element (method name or parentheses). Type exactly as shown. Parentheses are required, even if the method takes no arguments.
Bold italic font	***tableName***	Required element (argument). Replace with a variable, expression, or literal value.
[ ] (Square brackets)	[, ***fieldName***]	Informational element indicating an optional argument. You choose whether to include this argument.
* (Asterisk)	[, ***fieldName***] *	Informational element indicating a repeatable argument. You choose whether to repeat this argument.
{|} (Braces and bar)	{ Yes|No }	You *must* choose one of the values separated by the vertical bar.

▪

Required elements

See also

In ObjectPAL syntax, required elements are shown in **bold** or ***bold italic*** type. For example, in the following prototype the required elements are: the method name **load**, the parentheses, and the argument ***formName***. The rest of the prototype consists of informational elements.

```
load ( const formName String ) Logical
```

The following elements are required when shown as part of the prototype:

Required element	Description
Name	The name of the method or procedure.
Parentheses	Parentheses are required, even if the method or procedure takes no arguments.
Argument	If an argument is shown as part of the syntax, it must be included unless it is enclosed by square brackets (which make it optional). An argument can be a variable, an expression, or a literal (hard-coded) value. Lists of arguments are separated by commas.

▪

Informational elements

See also

Informational elements are not part of the ObjectPAL syntax you type for the method or procedure; they just tell you how the method or procedure works. The following table describes ObjectPAL informational elements.

Element	Description
Square brackets	Square brackets indicate an optional argument. If a prototype shows an argument enclosed in square brackets, you can include that argument or not, depending on what you want to do. For example, the square brackets in the following prototype indicate that *formTitle* is an optional argument. **attach (** [const ***formTitle*** String] **)** Logical There is one exception to this rule: when an argument is an array (or DynArray), the syntax for the argument shows square brackets following the Array (or DynArray) keyword. For instance, the following syntax shows that **enumPrinters** takes a resizeable array as an argument: **enumPrinters (** var ***printers*** Array[ ] String **)** Logical
Keywords	Keywords shown in normal type provide information about the arguments for a method or procedure. An argument preceded by the keyword var is passed by reference. An argument preceded by the keyword const is passed as a constant. An argument itself, without either keyword, is passed by value. The keyword that follows each argument specifies its data type (for example, String, Number, Table, or Logical). If a method or procedure returns a value, the keyword at the end of the prototype specifies its data type. Most, but not all ObjectPAL methods and procedures return values.
Asterisks	An asterisk (*) indicates that an argument can be repeated. For example, the following prototype shows that **message** takes one required argument, *reqTxt*, and one or more optional arguments, represented by *optTxt*. **message (** const ***reqTxt*** String [**,** const ***optTxt*** String] * **)**

▪

ObjectPAL prototypes

See also

Syntax statements (also called prototypes) are presented for each ObjectPAL method and procedure. An ObjectPAL prototype consists of required elements (shown in **bold** or ***bold italic*** type) and informational elements (shown in normal type). You must include required elements in the code you type, but don't include the informational elements.

For example, here is a prototype:

sample (var ***argOne*** Type [, const ***argTwo*** Type] **)** Type

The method name **sample**, the argument ***argOne***, and the parentheses are required. The argument ***argTwo*** is optional, and the rest of the prototype consists of informational elements. In ObjectPAL code, the following statements are valid:

```
; One argument, variable x stores the return value.
x = sample(custName)

; Two arguments, the return value is not used.
sample(custName, custAddress)
```

▪

Alternate syntax

See also

ObjectPAL supports an alternate syntax. The standard syntax uses dot notation to specify an object, a method name, and one or more arguments. For example,

object.methodName (***argument*** [**,** ***argument***] **)**

where **object** is an object name or UIObject variable, **methodName** represents the name of the method, and *argument* represents one or more arguments.

The alternate syntax does not use dot notation. Instead, it specifies the object as the first argument to the method. For example,

methodName (***object*** **,** ***argument*** [**,** ***argument***] **)**

For example, the following statement uses the standard ObjectPAL syntax to return a lowercase version of a string:

```
theString.lower()
```

The following statement uses the alternate syntax:

```
lower(theString)
```

For clarity and consistency, it's best to use standard syntax whenever possible. However, the alternate syntax can be convenient in some situations, for example, when defining the calculation for a calculated field.

▪

Using ObjectPAL in calculated fields

See also

A calculated field can use any of the following elements:

- Literal values.
- Variables, provided they are declared within the scope of the calculated field, and have been assigned a value.
- Object properties.
- Basic language elements.
- Custom methods attached to other objects (or to the field itself). You must first declare a UIObject variable within the scope of the calculated field and use an attach statement to associate the variable with a UIObject.
- Any method or procedure in the ObjectPAL run-time library (RTL) that returns a value (including a Logical value).
- Special functions (like Sum and Avg) provided specifically for use in calculated fields.

Note: ObjectPAL supports an alternate syntax that can be useful when defining a calculated field.

The following table describes these elements.

Element	Comments
5	Literal value.
"a"	Literal value.
x	Variable. Must be declared within the scope of the calculated field. Must be assigned a value.
x + 5	Simple expression. Rules for working with variables apply.
self.Name	Property. Displays the field's name as a String.
theBox.Color	Property. Displays an integer value representing the object's color.
iif(State.Value = "CA", 0.075, 0)	Basic language element **iif**. Value of calculated field depends on value of State field object.
uio.objCustomMethod()	Custom method attached to another object. The custom method must return a value.
tc.open("orders.db")	RTL method. Field displays True if the open succeeds; otherwise, it displays False. TCursor must be declared within the scope of the field.
Avg([DIVEITEM.Sale Price])	Special function. Operates on the Sale price field of the *Diveitem* table. The table must be in the form's data model. Quotes are not used, spaces are allowed.
tc.cAverage("Sale Price")	RTL method. The TCursor must be declared and opened previously. The table does not have to be in the data model. If a field name contains spaces, quotes are required.

▪

Derived methods

See also

Many object types include methods derived from similar methods defined for another type. For example, the Script type includes methods derived from the Form type. The diagram below shows that the Script type includes eleven methods: seven methods derived from the Form type, and four methods defined specifically for the Script type. The methods derived from the Form type are listed with the other Form methods, but the information applies equally to the Script type.

Some of the new methods for this version are derived from methods in other types. For example, the new **save** method for the Script type is derived from the Form type, but the new method operates on Script variables, the older one on Form variables. In cases like this, Help displays information about the original method, and does not duplicate it for the additional type. So, for example, when you're browsing the methods for the Script type and request help on the **save** method, Help displays information about the **save** method defined for the Form type. All the information that applies to forms also applies to scripts.

Methods for the Script Type

Form	⟵	**Script**
deliver		**attach**
enumSource		**create**
enumSourceToFile		**load**
methodDelete		**run**
methodGet		
methodSet		
save		

To print ObjectPAL Reference topics

You can print any screen in the ObjectPAL Reference or all the methods and examples in a type.

Because each topic is sent to the printer as a separate print job, you may want to set Form Feed and Banner off if your printer has these options.

To print a screen

1. Right-click the screen and choose Print Topic from the menu.

   The Print dialog box appears.

2. Change the Name and Properties of the printer, if necessary.
3. Choose OK.

   The Help system prints the selected topic.

To print all the methods and examples in a type

1. Click  to display the Alphabetical list of ObjectPAL types screen.
2. Click the name of the type that you want to print.
3. Click the print icon.
   The Print dialog box appears.
4. Change the Name and Properties of the printer, if necessary.
5. Choose OK.
   The Help system prints the selected topic.

■

Alphabetical list of ObjectPAL types

{button A,JI(`',`opaltype_a')} {button B,JI(`',`opaltype_b')} {button C,JI(`',`opaltype_c')} {button D,JI(`',`opaltype_d')} {button E,JI(`',`opaltype_e')} {button F,JI(`',`opaltype_f')} {button G,JI(`',`opaltype_g')} {button H,JI(`',`opaltype_k')} {button I,JI(`',`opaltype_k')} {button J,JI(`',`opaltype_k')} {button K,JI(`',`opaltype_k')} {button L,JI(`',`opaltype_l')} {button M,JI(`',`opaltype_m')} {button N,JI(`',`opaltype_n')} {button O,JI(`',`opaltype_o')} {button P,JI(`',`opaltype_p')} {button Q,JI(`',`opaltype_q')} {button R,JI(`',`opaltype_r')} {button S,JI(`',`opaltype_s')} {button T,JI(`',`opaltype_t')} {button U,JI(`',`opaltype_u')} {button V,JI(`',`opaltype_v')} {button W,JI(`',`opaltype_v')} {button X,JI(`',`opaltype_v')} {button Y,JI(`',`opaltype_v')} {button Z,JI(`',`opaltype_v')}

See also

Choose a type to see a list of the methods and procedures of that type. For each method and procedure you'll find syntax, a description, and sample code.

A

ActionEvent

AnyType

Application

Array

B

Binary

C

Currency

D

Database

DataTransfer

Date

DateTime

DDE

DynArray

E

ErrorEvent

Event

F

FileSystem

Form

G

Graphic

H-K

KeyEvent

L

Library

Logical

LongInt

ValueEvent

Object type categories

List of data model object types
List of data types
List of design object types
List of display managers
List of event types
List of system data objects

List of data model objects

Choose a type to see a list of the methods and procedures of that type. For each method and procedure you'll find syntax, a description, and sample code.

Database

Query

Table

TCursor

SQL

List of system data objects

Choose a type to see a list of the methods and procedures of that type. For each method and procedure you'll find syntax, a description, and sample code.

DDE

FileSystem

Library

Session

Script

System

TextStream

List of data types

Choose a type to see a list of the methods and procedures of that type. For each method and procedure you'll find syntax, a description, and sample code.

AnyType	DynArray	OLE
Array	Graphic	Point
Binary	Logical	Record
Currency	LongInt	SmallInt
Date	Memo	String
DateTime	Number	Time

List of design object types

Choose a type to see a list of the methods and procedures of that type. For each method and procedure you'll find syntax, a description, and sample code.

List of display managers

Choose a type to see a list of the methods and procedures of that type. For each method and procedure you'll find syntax, a description, and sample code.

Application

Form

Report

Script

TableView

List of event types

Choose a type to see a list of the methods and procedures of that type. For each method and procedure you'll find syntax, a description, and sample code.

ActionEvent	MouseEvent
ErrorEvent	MoveEvent
Event	StatusEvent
KeyEvent	TimerEvent
MenuEvent	ValueEvent

TimerEvent type

Methods in the TimerEvent type process information used by the timer method built into every design object. Use **setTimer**, defined for the UIObject type, to specify when to send timer events to an object, then modify the object's built-in **timer** method to control how the object responds when a timer goes off. Use **killTimer**, defined for the UIObject type, to turn off an object's timer. The following example shows how to use these methods with TimerEvents.

In this example, assume that a form contains a multi-record object bound to the *Customer* table. The record container in the multi-record object is named *custRecordMRO*.

Suppose you have a data-entry program, and you want to give the user 60 seconds to edit a record. After 60 seconds, you want to alert the user. To accomplish this, the built-in **action** method for *custRecordMRO* tests every action. If the action is DataArriveRecord, the method stops any old timers with **killTimer** and sets a new timer for the record object with **setTimer**. When the timer goes off, a message pops up alerting the user. To make it easy to change the time, a constant is defined in the Const window for *custRecordMRO*, as follows:

```
; custRecordMRO::Const
const
  alertTime = 60000     ; data-entry alert at 60 seconds
endConst
```

The following code is for the **action** method for *custRecordMRO*.

```
; custRecordMRO::action
method action(var eventInfo ActionEvent)
if eventInfo.id() = DataArriveRecord then ; when opening to a new record
  self.killTimer()         ; just in case it hasn't expired
                           ; yet, kill the old timer
  self.setTimer(alertTime) ; start timer for this record
endif
endMethod
```

This code is attached to the **timer** method for *custRecordMRO*.

```
; custRecordMRO::timer
method timer(var eventInfo TimerEvent)
self.killTimer()
beep()
msgInfo("Alert", "You have been processing this record for " +
                 "one minute now.")
endMethod
```

The Timer type consists only of derived methods from the Event type.

Methods for the TimerEvent Type

Event	←	**TimerEvent**
errorCode		(All TimerEvent methods are derived methods from the Event type.)
getTarget		
isFirstTime		
isPreFilter		
isTargetSelf		

reason
setErrorCode
setReason

Alphabetical list of 4.5 methods

See also

Database::beginTransaction

Database::commitTransaction

System::dlgExport
(moved to Data Transfer Type in version 7)

System::dlgImportASCIIFix
(moved to Data Transfer Type in version 7)

System::dlgImportASCIIVar
(moved to Data Transfer Type in version 7)

System::dlgImportSpreadsheet
(moved to Data Transfer Type in version 7)

Form::dmGetProperty

Form::dmLinkToFields

Form::dmLinkToIndex

Form::dmSetProperty

Form::dmUnlink

Session::enumAliasLoginInfo

System::errorHasErrorCode

System::errorHasNativeErrorCode

System::errorNativeCode

SQL::executeSQL

System::executeString

System::exportASCIIFix
(moved to Data Transfer Type in version 7)

System::exportASCIIVar
(moved to Data Transfer Type in version 7)

System::exportSpreadsheet
(moved to Data Transfer Type in version 7)

TCursor::forceRefresh

UIObject::forceRefresh

Session::getAliasProperty

System::importASCIIFix
(moved to Data Transfer Type in version 7)

System::importASCIIVar
(moved to Data Transfer Type in version 7)

System::importSpreadsheet
(moved to Data Transfer Type in version 7)

TCursor::isOnSQLServer

TCursor::isOpenOnUniqueIndex

Database::rollBackTransaction

System::sendKeys
System::sendKeysActionID
Session::setAliasPassword
Session::setAliasProperty
TCursor::setBatchOff
TCursor::setBatchOn

Database::transactionActive

TCursor::update

SQL::writeSQL

4.5 methods by type

See also

Database::beginTransaction
Database::commitTransaction
Database::rollBackTransaction
Database::transactionActive

Form::dmGetProperty
Form::dmLinkToFields
Form::dmLinkToIndex
Form::dmSetProperty
Form::dmUnlink

Session::enumAliasLoginInfo
Session::getAliasProperty
Session::setAliasPassword
Session::setAliasProperty

SQL::executeSQL
SQL::writeSQL

System::dlgExport
(moved to Data Transfer Type in version 7)
System::dlgImportASCIIFix
(moved to Data Transfer Type in version 7)
System::dlgImportASCIIVar
(moved to Data Transfer Type in version 7)
System::dlgImportSpreadsheet
(moved to Data Transfer Type in version 7)
System::errorHasErrorCode
System::errorHasNativeErrorCode
System::errorNativeCode
System::executeString
System::exportASCIIFix
(moved to Data Transfer Type in version 7)
System::exportASCIIVar
(moved to Data Transfer Type in version 7)
System::exportSpreadsheet
(moved to Data Transfer Type in version 7)
System::importASCIIFix
(moved to Data Transfer Type in version 7)
System::importASCIIVar
(moved to Data Transfer Type in version 7)
System::importSpreadsheet
(moved to Data Transfer Type in version 7)
System::sendKeysActionID

System::sendKeys

TCursor::forceRefresh
TCursor::isOnSQLServer
TCursor::isOpenOnUniqueIndex
TCursor::setBatchOff
TCursor::setBatchOn
TCursor::update

UIObject::forceRefresh

■

Alphabetical list of 5.0 methods

See also

The following table lists methods that were added or changed for version 5.0. Some of these are derived methods from other types.

New	Changed
Session::addProjectAlias	Session::addAlias
UIObject::bringToFront	TCursor::attach
OLE::canLinkFromClipboard	Database::beginTransaction
Binary::clipboardErase	Table::cCount
Binary::clipboardHasFormat	TCursor::cCount
System::compileInformation	Table::create
UIObject::copyToToolbar	Form::dmGet
Library::create	Form::dmGetProperty
Script::create (derived from Form type)	Form::dmHasTable
Table::createIndex	Form::dmLinkToFields
TCursor::createIndex	Form::dmLinkToIndex
Library::deliver (derived from Form type)	Form::dmPut
Report::deliver (derived from Form type)	Form::dmRemoveTable
Script::deliver (derived from Form type)	Form::dmSetProperty
System::desktopMenu	Form::dmUnlink
Form::disablePreviousError	Session::enumAliasNames
Report::dmAddTable (derived from Form type)	Session::enumDatabaseTables
Form::dmAttach	Session::enumDriverCapabilities
TCursor::dmAttach	Table::enumFieldStruct
Form::dmBuildQueryString	TCursor::enumFieldStruct
Report::dmBuildQueryString (derived from Form type)	Session::enumFolder
Form::dmEnumLinkFields	Table::enumIndexStruct
Report::dmEnumLinkFields (derived from Form type)	TCursor::enumIndexStruct
Report::dmGetProperty (derived from Form type)	Session:enumOpenDatabases
Report::dmHasTable (derived from Form type)	Table::enumRefIntStruct
Report::dmLinkToFields (derived from Form type)	TCursor::enumRefIntStruct

Report::dmLinkToIndex
(derived from Form type)

Report::dmRemoveTable
(derived from Form type)

Form::dmResync

Report::dmSetProperty
(derived from Form type)

Report::dmUnlink
(derived from Form type)

Table::dropGenFilter

TCursor::dropGenFilter

UIObject::dropGenFilter

System::enableExtendedCharacters

Binary::enumClipboardFormats

Form::enumDataModel

Report::enumDataModel
(derived from Form type)

Database::enumFamily

System::enumPrinters

System::enumRTLErrors

OLE::enumServerClassNames

Report::enumSource
(derived from Form type)

Script::enumSource
(derived from Form type)

Report::enumSourceToFile
(derived from Form type)

Script::enumSourceToFile
(derived from Form type)

Report::enumTableLinks
(derived from Form type)

System::exportParadoxDOS
(moved to Data Transfer Type in version 7)

System::formatGetSpec

System::formatStringToDate

System::formatStringToNumber

Script::formReturn
(derived from Form type)

AnyType::fromHex

System::getDefaultPrinterStyleSheet

System::getDefaultScreenStyleSheet

Form::enumTableLinks

Table::enumSecStruct

TCursor::enumSecStruct

UIObject::enumUIObjectProperties

Session::enumUsers

Table::fieldType

TCursor::fieldType

String::format

Query::getQueryRestartOptions
(previously in the Database type)

Query::isExecuteQBELocal
(previously in the Database type)

Form::load

Report::load

TCursor::nRecords

Report::open

Report::print

Query::readFromFile (replaces executeQBEFile)

SQL::readFromFile (replaces executeSQLFile)

Query::readFromString (replaces executeQBEString)

SQL::readFromString (replaces executeSQLString)

TCursor::recNo

TCursor::seqNo

Query::setQueryRestartOptions
(previously in the Database type)

System::sleep

Report::getFileName
(derived from Form type)

Table::getGenFilter

TCursor::getGenFilter

UIObject::getGenFilter

TCursor::getIndexName

System::getLanguageDriver

Form::getProtoProperty

Report::getProtoProperty
(derived from Form type)

SQL::getQueryRestartOptions

TCursor::getRange

Table::getRange

UIObject::getRange

Form::getSelectedObjects

Form::getStyleSheet

Report::getStyleSheet
(derived from Form type)

System::getUserLevel

Form::hideToolbar

OLE::insertObject

TCursor::instantiateView

SQL::isAssigned

Form::isDesign

Report::isDesign
(derived from Form type)

OLE::isLinked

Database::isSQLServer

Form::isToolbarShowing

TCursor::isView

OLE::linkFromClipboard

Library::load
(derived from Form type)

Report::load
(derived from Form type)

Script::load
(derived from Form type)

Session::loadProjectAliases

Report::menuAction
(derived from Form type)

Library::methodDelete
(derived from Form type)

Script::methodDelete

(derived from Form type)

Library::methodGet
(derived from Form type)

Script::methodGet
(derived from Form type)

Library::methodSet
(derived from Form type)

Script::methodSet
(derived from Form type)

Report::moveTo
(derived from Form type)

System::printerGetInfo

System::printerGetOptions

System::printerSetCurrent

System::printerSetOptions

System::projectViewerClose

System::projectViewerIsOpen

System::projectViewerOpen

Binary::readFromClipboard

Query::readFromFile (replaces Database::executeQBEFile)

Query::readFromString (replaces Database::executeQBEString)

Session::removeProjectAlias

Script::run
(derived from Form type)

Library::save
(derived from Form type)

Script::save
(derived from Form type)

Session::saveProjectAliases

Form::saveStyleSheet

Report::saveStyleSheet
(derived from Form type)

Form::selectCurrentTool

Report::selectCurrentTool (derived from Form type)

UIObject::sendToBack

System::setDefaultPrinterStyleSheet

System::setDefaultScreenStyleSheet

Table::setGenFilter

TCursor::setGenFilter)

UIObject::setGenFilter

Form::setMenu

Report::setMenu

FileSystem::setPrivDir

Form::setProtoProperty

Report::setProtoProperty
(derived from Form type)

SQL::setQueryRestartOptions

Table::setRange

TCursor::setRange

UIObject::setRange

Form::setSelectedObjects

Form::setStyleSheet

Report::setStyleSheet
(derived from Form type)

System::setUserLevel

FileSystem::setWorkingDir

Form::showToolbar

AnyType::toHex

OLE::updateLinkNow

Query::wantInMemoryTCursor

SQL::wantInMemoryTCursor

Report::windowClientHandle
(derived from Form type)

Binary::writeToClipboard

5.0 methods by type

See also

The following table lists methods that were added or changed for version 5.0. Some of these are derived methods from other types.

New	Changed
AnyType::fromHex	Database::beginTransaction
AnyType::toHex	Form::dmGet
Binary::clipboardErase	Form::dmGetProperty
Binary::clipboardHasFormat	Form::dmHasTable
Binary::enumClipboardFormats	Form::dmLinkToFields
Binary::readFromClipboard	Form::dmLinkToIndex
Binary::writeToClipboard	Form::dmPut
Database::enumFamily	Form::dmRemoveTable
Database::isSQLServer	Form::dmSetProperty
FileSystem::setPrivDir	Form::dmUnlink
FileSystem::setWorkingDir	Form::enumTableLinks
Form::dmAttach	Form::load
Form::dmBuildQueryString	Query::executeQBE
Form::dmEnumLinkFields	Query::getQueryRestartOptions (previously in the Database type)
Form::dmResync	Query::isExecuteQBELocal (previously in the Database type)
Form::enumDataModel	Query::readFromFile (replaces executeQBEFile)
Form::getProtoProperty	Query::readFromString (replaces executeQBEString)
Form::getSelectedObjects	Query::setQueryRestartOptions (previously in the Database type)
Form::getStyleSheet	Query::writeQBE
Form::hideToolbar	Report::load
Form::isDesign	Report::open
Form::isToolbarShowing	Report::print
Form::saveStyleSheet	Session::addAlias
Form::selectCurrentTool	Session::enumAliasNames
Form::setMenu	Session::enumDatabaseTables
Form::setProtoProperty	Session::enumDriverCapabilities
Form::setSelectedObjects	Session::enumFolder
Form::setStyleSheet	Session::enumUsers
Form::showToolbar	Session:enumOpenDatabases
Library::create	SQL::readFromFile (replaces executeSQLFile)

Library::deliver
(derived from Form type)

Library::load
(derived from Form type)

Library::methodDelete
(derived from Form type)

Library::methodGet
(derived from Form type)

Library::methodSet
(derived from Form type)

Library::save
(derived from Form type)

OLE::canLinkFromClipboard

OLE::enumServerClassNames

OLE::insertObject

OLE::isLinked

OLE::linkFromClipboard

OLE::updateLinkNow

Query::readFromFile (replaces Database::executeQBEFile)

Query::readFromString (replaces Database::executeQBEString)

Query::wantInMemoryTCursor

Report::deliver
(derived from Form type)

Report::dmAddTable
(derived from Form type)

Report::dmBuildQueryString
(derived from Form type)

Report::dmEnumLinkFields
(derived from Form type)

Report::dmGetProperty
(derived from Form type)

Report::dmHasTable
(derived from Form type)

Report::dmLinkToFields
(derived from Form type)

Report::dmLinkToIndex
(derived from Form type)

Report::dmRemoveTable
(derived from Form type)

Report::dmSetProperty
(derived from Form type)

Report::dmUnlink
(derived from Form type)

SQL::readFromString (replaces executeSQLString)

String::format

System::sleep

Table::cCount

Table::create

Table::enumFieldStruct

Table::enumIndexStruct

Table::enumRefIntStruct

Table::enumSecStruct

Table::fieldType

TCursor::attach

TCursor::cCount

TCursor::enumFieldStruct

TCursor::enumIndexStruct

TCursor::enumRefIntStruct

TCursor::enumSecStruct

TCursor::fieldType

TCursor::nRecords

TCursor::recNo

TCursor::seqNo

UIObject::enumUIObjectProperties

Report::enumDataModel
(derived from Form type)

Report::enumSource
(derived from Form type)

Report::enumSourceToFile
(derived from Form type)

Report::enumTableLinks
(derived from Form type)

Report::getFileName
(derived from Form type)

Report::getProtoProperty
(derived from Form type)

Report::getStyleSheet
(derived from Form type)

Report::isDesign
(derived from Form type)

Report::load
(derived from Form type)

Report::menuAction
(derived from Form type)

Report::moveTo
(derived from Form type)

Report::saveStyleSheet
(derived from Form type)

Report::selectCurrentTool (derived
from Form type)

Report::setMenu

Report::setProtoProperty
(derived from Form type)

Report::setStyleSheet
(derived from Form type)

Report::windowClientHandle
(derived from Form type)

Script::create
(derived from Form type)

Script::deliver
(derived from Form type)

Script::enumSource
(derived from Form type)

Script::enumSourceToFile
(derived from Form type)

Script::formReturn
(derived from Form type)

Script::load
(derived from Form type)

Script::methodDelete
(derived from Form type)

Script::methodGet
(derived from Form type)

Script::methodSet
(derived from Form type)

Script::run
(derived from Form type)

Script::save
(derived from Form type)

Session::addProjectAlias

Session::loadProjectAliases

Session::removeProjectAlias

Session::saveProjectAliases

SQL::isAssigned

SQL::getQueryRestartOptions

SQL::setQueryRestartOptions

SQL::wantInMemoryTCursor

System::compileInformation

System::desktopMenu

Form::disablePreviousError

System::enableExtendedCharacters

System::enumPrinters

System::enumRTLErrors

System::exportParadoxDOS
(moved to Data Transfer Type in version 7)

System::formatGetSpec

System::formatStringToDate

System::formatStringToNumber

System::getDefaultPrinterStyleSheet

System::getDefaultScreenStyleSheet

System::getLanguageDriver

System::getUserLevel

System::printerGetInfo

System::printerGetOptions

System::printerSetCurrent

System::printerSetOptions

System::projectViewerClose

System::projectViewerIsOpen

System::projectViewerOpen

System::setDefaultPrinterStyleSheet

System::setDefaultScreenStyleSheet
System::setUserLevel
Table::createIndex
Table::dropGenFilter
Table::getGenFilter
Table::getRange
Table::setGenFilter
Table::setRange
TCursor::createIndex
TCursor::dmAttach
TCursor::dropGenFilter
TCursor::getGenFilter
TCursor::getIndexName
TCursor::getRange
TCursor::instantiateView
TCursor::isView
TCursor::setGenFilter)
TCursor::setRange
UIObject::bringToFront
UIObject::copyToToolbar
UIObject::dropGenFilter
UIObject::getGenFilter
UIObject::getRange
UIObject::sendToBack
UIObject::setGenFilter
UIObject::setRange

▪

Alphabetical list of version 7 methods

See also

The following table lists methods that were added or changed for version 7. Some of these are derived methods from other types.

New	Changed
Mail::addAddress	UIObject::create
Mail::addAttachment	Date::date
Toolbar::addButton	System::dlgExport (moved to Data Transfer Type in version 7)
Mail::addressBook	System::dlgImportAsciiFix (moved to Data Transfer Type in version 7)
Mail::addressBookTo	System::dlgImportAsciiVar (moved to Data Transfer Type in version 7)
TCursor::aliasName	System::dlgImportSpreadSheet (moved to Data Transfer Type in version 7)
DataTransfer::appendASCIIFix	System::exportASCIIFix (moved to Data Transfer Type in version 7)
DataTransfer::appendASCIIVar	System::exportASCIIVar (moved to Data Transfer Type in version 7)
Query::appendRow	System::exportParadoxDOS (moved to Data Transfer Type in version 7)
Query::appendTable	System::exportSpreadsheet (moved to Data Transfer Type in version 7)
OleAuto::attach	System::importASCIIFix (moved to Data Transfer Type in version 7)
Toolbar::attach	System::importASCIIVar (moved to Data Transfer Type in version 7)
Query::checkField	System::importSpreadsheet (moved to Data Transfer Type in version 7)
Query::checkRow	Built-in::init
Query::clearCheck	System::sysInfo
OleAuto::close	
Toolbar::create	
Query::createAuxTables	
Query::createQBEString	

Toolbar::createTabbed

System::deleteRegistryKey

DataTransfer::dlgExport
(moved from System Type in version 7)

DataTransfer::dlgImport

DataTransfer::dlgImportAsciiFix
(moved from System Type in version 7)

DataTransfer::dlgImportAsciiVar
(moved from System Type in version 7)

DataTransfer::dlgImportSpreadSheet
(moved from System Type in version 7)

DataTransfer::dlgImportTable

DataTransfer::empty

Mail::empty

Toolbar::empty

Mail::emptyAddresses

Mail::emptyAttachments

OleAuto::enumAutomationServers

OleAuto::enumConstants

OleAuto::enumConstantValues

OleAuto::enumControls

System::enumDesktopWindowHandles

OleAuto::enumEvents

System::enumExperts

Query::enumFieldStruct

OleAuto::enumMethods

OleAuto::enumObjects

OleAuto::enumProperties

System::enumRegistryKeys

System::enumRegistryValueNames

OleAuto::enumServerInfo

DataTransfer::enumSourcePageList

DataTransfer::enumSourceRangeList

System::enumWindowHandles

DataTransfer::exportASCIIFix
(moved from System Type in version 7)

DataTransfer::exportASCIIVar
(moved from System Type in version

7)

DataTransfer::exportParadoxDOS (moved from System Type in version 7)

DataTransfer::exportSpreadsheet (moved from System Type in version 7)

OleAuto::first

System::formatStringToDateTime

System::formatStringToTime

Mail::getAddress

Mail::getAddressCount

Query::getAnswerFieldOrder

Query::getAnswerName

Query::getAnswerSortOrder

DataTransfer::getAppend

Mail::getAttachment

Mail::getAttachmentCount

Query::getCheck

Query::getCriteria

System::getDesktopPreference

DataTransfer::getDestCharSet

DataTransfer::getDestDelimitedFields

DataTransfer::getDestDelimiter

DataTransfer::getDestFieldNamesFromFirst

DataTransfer::getDestName

DataTransfer::getDestSeparator

DataTransfer::getDestType

DataTransfer::getKeyviol

Database::getMaxRows

Mail::getMessage

Mail::getMessageType

Toolbar::getPosition

DataTransfer::getProblems

System::getRegistryValue

Query::getRowID

Query::getrowNo

DataTransfer::getSourceCharSet

DataTransfer::getSourceDelimitedFields

DataTransfer::getSourceDelimiter

DataTransfer::getSourceFieldNamesFromFirst

DataTransfer::getSourceName

DataTransfer::getSourceRange

DataTransfer::getSourceSeparator

DataTransfer::getSourceType

Toolbar::getState

Mail::getSubject

Query::getTableID

Query::getTableNo

Query::hasCriteria

Toolbar::hide

DataTransfer::importASCIIFix (moved from System Type in version 7)

DataTransfer::importASCIIVar (moved from System Type in version 7)

DataTransfer::importSpreadsheet (moved from System Type in version 7)

Query::insertRow

Query::insertTable

OleAuto::invoke

Form::isCompileWithDebug

Query::isCreateAuxTables

Query::isEmpty

System::isMousePersistent

Query::isQueryValid

FileSystem::isValidFile

Toolbar::isVisible

DataTransfer::loadDestSpec

DataTransfer::loadSourceSpec

Mail::logoff

Mail::logoffDlg

Mail::logon

Mail::logonDlg

OleAuto::next

OleAuto::open

OleAuto::openObjectTypeInfo

OleAuto::openTypeInfo

Memo::readFromClipboard
String::readFromClipboard
OleAuto::registerControl
Toolbar::remove
Toolbar::removeButton
Query::removeCriteria
Query::removeRow
Query::removeTable
System::runExpert
System::searchRegistry
Mail::send
Mail::sendDlg
Query::setAnswerFieldOrder
Query::setAnswerName
Query::setAnswerSortOrder
DataTransfer::setAppend
Form::setCompileWithDebug
Query::setCriteria
System::setDesktopPreference
DataTransfer::setDest
DataTransfer::setDestCharSet
DataTransfer::setDestDelimitedFields
DataTransfer::setDestDelimiter
DataTransfer::setDestFieldNamesFromFirst
DataTransfer::setDestSeparator
Form::setIcon
DataTransfer::setKeyviol
Query::setLanguageDriver
Database::setMaxRows
Mail::setMessage
Mail::setMessageType
System::setMouseShapeFromFile
Toolbar::setPosition
DataTransfer::setProblems
System::setRegistryValue
Query::setRowOp
DataTransfer::setSource
DataTransfer::setSourceCharSet
DataTransfer::setSourceDelimitedFields

▪

Version 7 methods by type

See also

The following table lists methods that were added or changed for version 7. Some of these are derived methods from other types.

New	Changed
DataTransfer::appendASCIIFix	Built-in::init
Database::getMaxRows	Date::date
Database::setMaxRows	System::dlgExport (moved to Data Transfer Type in version 7)
DataTransfer::appendASCIIVar	System::dlgImportAsciiFix (moved to Data Transfer Type in version 7)
DataTransfer::dlgExport (moved from System Type in version 7)	System::dlgImportAsciiVar (moved to Data Transfer Type in version 7)
DataTransfer::dlgImport	System::dlgImportSpreadSheet (moved to Data Transfer Type in version 7)
DataTransfer::dlgImportAsciiFix (moved from System Type in version 7)	System::exportASCIIFix (moved to Data Transfer Type in version 7)
DataTransfer::dlgImportAsciiVar (moved from System Type in version 7)	System::exportASCIIVar (moved to Data Transfer Type in version 7)
DataTransfer::dlgImportSpreadSheet (moved from System Type in version 7)	System::exportParadoxDOS (moved to Data Transfer Type in version 7)
DataTransfer::dlgImportTable	System::exportSpreadsheet (moved to Data Transfer Type in version 7)
DataTransfer::empty	System::importASCIIFix (moved to Data Transfer Type in version 7)
DataTransfer::enumSourcePageList	System::importASCIIVar (moved to Data Transfer Type in version 7)
DataTransfer::enumSourceRangeList	System::importSpreadsheet (moved to Data Transfer Type in version 7)
DataTransfer::exportASCIIFix (moved from System Type in version 7)	System::sysInfo
DataTransfer::exportASCIIVar (moved from System Type in version 7)	UIObject::create
DataTransfer::exportParadoxDOS	

(moved from System Type in version 7)

DataTransfer::exportSpreadsheet (moved from System Type in version 7)

DataTransfer::getAppend

DataTransfer::getDestCharSet

DataTransfer::getDestDelimitedFields

DataTransfer::getDestDelimiter

DataTransfer::getDestFieldNamesFromFirst

DataTransfer::getDestName

DataTransfer::getDestSeparator

DataTransfer::getDestType

DataTransfer::getKeyviol

DataTransfer::getProblems

DataTransfer::getSourceCharSet

DataTransfer::getSourceDelimitedFields

DataTransfer::getSourceDelimiter

DataTransfer::getSourceFieldNamesFromFirst

DataTransfer::getSourceName

DataTransfer::getSourceRange

DataTransfer::getSourceSeparator

DataTransfer::getSourceType

DataTransfer::importASCIIFix (moved from System Type in version 7)

DataTransfer::importASCIIVar (moved from System Type in version 7)

DataTransfer::importSpreadsheet (moved from System Type in version 7)

DataTransfer::loadDestSpec

DataTransfer::loadSourceSpec

DataTransfer::setAppend

DataTransfer::setDest

DataTransfer::setDestCharSet

DataTransfer::setDestDelimitedFields

DataTransfer::setDestDelimiter

DataTransfer::setDestFieldNamesFro

mFirst

DataTransfer::setDestSeparator

DataTransfer::setKeyviol

DataTransfer::setProblems

DataTransfer::setSource

DataTransfer::setSourceCharSet

DataTransfer::setSourceDelimitedFields

DataTransfer::setSourceDelimiter

DataTransfer::setSourceFieldNamesFromFirst

DataTransfer::setSourceRange

DataTransfer::setSourceSeparator

DataTransfer::transferData

FileSystem::isValidFile

FileSystem::shortName

Form::isCompileWithDebug

Form::setCompileWithDebug

Form::setIcon

Mail::addAddress

Mail::addAttachment

Mail::addressBook

Mail::addressBookTo

Mail::emptyAddresses

Mail::emptyAttachments

Mail::empty

Mail::getAddressCount

Mail::getAddress

Mail::getAttachmentCount

Mail::getAttachment

Mail::getMessage

Mail::getMessageType

Mail::getSubject

Mail::logoffDlg

Mail::logoff

Mail::logonDlg

Mail::logon

Mail::sendDlg

Mail::send

Mail::setMessage

Mail::setMessageType

Mail::setSubject
Memo::readFromClipboard
Memo::writeToClipboard
OleAuto::attach
OleAuto::close
OleAuto::enumAutomationServers
OleAuto::enumConstants
OleAuto::enumConstantValues
OleAuto::enumControls
OleAuto::enumEvents
OleAuto::enumMethods
OleAuto::enumObjects
OleAuto::enumProperties
OleAuto::enumServerInfo
OleAuto::first
OleAuto::invoke
OleAuto::next
OleAuto::open
OleAuto::openObjectTypeInfo
OleAuto::openTypeInfo
OleAuto::registerControl
OleAuto::unregisterControl
OleAuto::version
Query::appendRow
Query::appendTable
Query::checkField
Query::checkRow
Query::clearCheck
Query::createAuxTables
Query::createQBEString
Query::enumFieldStruct
Query::getAnswerFieldOrder
Query::getAnswerName
Query::getAnswerSortOrder
Query::getCheck
Query::getCriteria
Query::getRowID
Query::getrowNo
Query::getTableID
Query::getTableNo

Query::hasCriteria
Query::insertRow
Query::insertTable
Query::isCreateAuxTables
Query::isEmpty
Query::isQueryValid
Query::removeCriteria
Query::removeRow
Query::removeTable
Query::setAnswerFieldOrder
Query::setAnswerName
Query::setAnswerSortOrder
Query::setCriteria
Query::setLanguageDriver
Query::setRowOp
String::readFromClipboard
String::writeToClipboard
System::deleteRegistryKey
System::enumDesktopWindowHandles
System::enumExperts
System::enumRegistryKeys
System::enumRegistryValueNames
System::enumWindowHandles
System::formatStringToDateTime
System::formatStringToTime
System::getDesktopPreference
System::getRegistryValue
System::isMousePersistent
System::runExpert
System::searchRegistry
System::setDesktopPreference
System::setMouseShapeFromFile
System::setRegistryValue
TCursor::aliasName
Toolbar::addButton
Toolbar::attach
Toolbar::create
Toolbar::createTabbed
Toolbar::empty

Toolbar::getPosition
Toolbar::getState
Toolbar::hide
Toolbar::isVisible
Toolbar::removeButton
Toolbar::remove
Toolbar::setPosition
Toolbar::setState
Toolbar::show
Toolbar::unAttach

Ordering the ObjectPAL Reference

To order a printed version of this ObjectPAL reference material,

- In the U.S., please fax your order to 510-657-0186. For faster service, call toll-free 1-800-621-3132.
- In Canada, please fax your order to 1-800-825-2225. For faster service, call toll-free 1-800-461-3327.
- Or, choose File|Print Topic to print this topic, complete the information, and mail it to:

U.S. Customers	Canadian Customers
Documentation Order Processing Borland International, Inc. 100 Borland Way P. O. Box 660005 Scotts Valley, CA 95067-0005	Borland Canada 200 Konrad Crescent Markham, Ontario Canada L3R8T9

(Please print)

Product Name____

Name____

Company Name____

Street Address____

City____ State_____________

Zip/Postal Code____________________ Country__________________________

Daytime Phone (________)____

(in case we have a question about your order)

The cost of the ObjectPAL Reference is U.S.$29.95 plus $5.00 shipping and handling and sales tax where applicable.*

ObjectPAL Reference	$ 29.95	*Residents in CA, CO, CT, DC, FL, GA, IL, MA, MD, MI,
State sales tax	$_______	MN, MO, MC, NJ, NY, OH, PA, TN, TX, UT, VA, WA:
Freight*	$ 5.00	Please add appropriate sales tax. CO, MI, NY, PA,
TOTAL	$_______	TX, WA residents: please add tax to freight charges

Please make your check payable to Borland International, Inc. and enclose it with this form. Or fill in your credit card name, number, expiration date, your printed name, and signature.

___ Check enclosed ___ MasterCard ___ Visa ___ American Express

Card number __ __ __ __ - __ __ __ __ - __ __ __ __ - __ __ __ __

Expiration date __ __ - __ __

Name as it appears on the card____

Signature____

Please allow two to three weeks for delivery. Pricing and offer good in the United States only.

For international orders: Please refer to the Borland Registration card for the telephone number of the Borland office nearest you. Prices and shipping may vary by country.

▪

Types of constants

Choose from the following types of constants for more information:

ActionClass Constants
ActionDataCommand Constants
ActionEditCommand Constants
ActionFieldCommand Constants
ActionMoveCommand Constants
ActionSelectCommand Constants
AggModifier Constants
ButtonStyle Constants
ButtonType Constants
Color Constants
CompleteDisplay Constants
DateRangeType Constants
DesktopPreferenceTypes Constants
ErrorReason Constants
EventErrorCode Constants
ExecuteOption Constants
FieldDisplayType Constants
FileBrowserFileType Constants
FontAttribute Constants
FrameStyle Constants
General Constants
GraphBindType Constants
GraphicMagnification Constants
GraphLabelFormat Constants
GraphLegendPosition Constants
GraphMarker Constants
GraphTypeOverRide Constants
GraphType Constants
IdRange Constants
Keyboard Constants
KeyBoardState Constants
LibraryScope Constants
LineEnd Constants
LineStyle Constants
LineThickness Constants
LineType Constants
MenuChoiceAttribute Constants
MenuCommand Constants
MenuReason Constants
MouseShape Constants
MoveReason Constants
PageTilingOption Constants

PatternStyle Constants
PrintColor Constants
PrintDuplex Constants
PrinterOrientation Constants
PrinterSize Constants
PrintQuality Constants
PrintSource Constants
QueryRestartOption Constants
RasterOperation Constants
ReportOrientation Constants
ReportPrintPanel Constants
ReportPrintRestart Constants
SpecialFieldType Constants
StatusReason Constants
TableFrameStyle Constants
TextAlignment Constants
TextDesignSizing Constants
TextSpacing Constants
ToolbarBitmap Constants
ToolbarButtonType Constants
ToolbarClusterID Constants
ToolbarState Constants
UIObjectType Constants
ValueReason Constants
WindowStyle Constants

ActionClass constants

Constant	Data type	Description
DataAction	SmallInt	Data actions are for navigating in a table and for such tasks as record locking and record posting.
EditAction	SmallInt	Most Edit actions alter data within a field.
FieldAction	SmallInt	Field actions are a special category of Move action that enable movement between field objects.
MoveAction	SmallInt	Move actions have to do with moving within a field object.
SelectAction	SmallInt	Select actions are equivalent to Move actions.

ActionDataCommand constants

Constant	Data type	Description
DataArriveRecord	SmallInt	Indicates a change to the current record, regardless of the reason. Some possible reasons: navigation, editing, network refresh, and scrolling.
DataBegin	SmallInt	Moves to first record in the table associated with the given UIObject. Will force recursive action (DataUnlockRecord) if current record has been modified. If Error encountered, will call **error** method. Invoked by "First Record" button on Toolbar, or Record|First.
DataBeginEdit	SmallInt	Used to enter Edit mode on the form. Normally always allowed. Invoked by (1st) F9, View|Edit Data or Edit icon on Toolbar.
DataBeginFirstField	SmallInt	Moves to the first field in the first record of the table associated with the given UIObject. Invoked by Ctrl+Home.
DataCancelRecord	SmallInt	Used to discard changes to record. Succeeds by default, but user could block it. Invoked by Edit|Undo, Alt+Backspace, or Record|Cancel Changes menu item. Also used internally when moving off a locked but unmodified record.
DataDeleteRecord	SmallInt	Deletes the current record. Errors encountered will call **error** method. This action is irreversible except for dBASE tables. Invoked by Record|Delete or Ctrl+Del.
DataDesign	SmallInt	Switches from running the form to the Form Design window. Invoked by F8.
DataDitto	SmallInt	Copies into the current field the value of the corresponding field in the prior record. Invoked by Ctrl+D.
DataEnd	SmallInt	Moves to final record in the table associated with the given UIObject. Will force recursive action (DataUnlockRecord) if current record has been modified. Error encountered will call **error** method. Invoked by the "Last Record" button on Toolbar.
DataEndEdit	SmallInt	Used to exit Edit mode on the form. Invoked by (2nd) F9, Edit Data button on Toolbar, or View|View Data.
DataEndLastField	SmallInt	Moves to the last field of the last record of the table associated with a UIObject. Invoked by Ctrl+Home.
DataFastBackward	SmallInt	Moves backward one set of records (where a set is defined as the number of rows in a table frame or MRO). Invoked by Record|Previous Set, Shift+F11 or Previous Record Set button on Toolbar.
DataFastForward	SmallInt	Moves forward one set of records (where a set is defined as the number of rows in a table frame or MRO). Invoked by Record|Next Set, Shift+F11 or Next Record Set button on Toolbar.
DataHideDeleted	SmallInt	Alters the mode of the form so that deleted records will be hidden (available only for dBASE tables). Invoked by Form|Hide Deleted.

DataInsertRecord	SmallInt	Will insert a new (blank) record before the current record. Record state will appear as "locked", and blank record will not exist in the underlying table until the record is eventually modified and unlocked. Invoked by Record|Insert, or Ins. Note that records created in this way can be discarded either via DataDeleteRecord or DataCancelRecord before they have been unlocked. Moving off such a record without making any changes will internally use DataCancelRecord to discard it. Invoked by Ins or Record|Insert.
DataLockRecord	SmallInt	Used to lock the current record. Errors encountered will call **error** method. Invoked by F5.
DataLookup	SmallInt	Used to invoke lookup table for current field, to accept user's choice of new value, and, if appropriate, to update all corresponding fields governed by lookup. Available only for fields that have been defined as lookup fields. Invoked by Ctrl+Spacebar.
DataLookupMove	SmallInt	A special form of lookup which allows the user to choose a new master record for this detail. Invoked by Record|Move Help or Ctrl+Shift+Spacebar.
DataNextRecord	SmallInt	Moves (if possible) to next sequential record in the table associated with the UIObject. Will force recursive action (DataUnlockRecord) if current record has been modified. Errors encountered will call **error** method. Invoked by Record|Next, "Next Record" Toolbar button, F12, and so forth.
DataNextSet	SmallInt	Moves forward one set of records (where a set is defined as the number of rows in a table frame or MRO. Invoked by PgDn.
DataPostRecord	SmallInt	Just like DataUnlockRecord, but the record lock will not be released. As a consequence, if changes to key fields mean the record will move to a new position in the table, the table's position "flies with" that record (meaning it will still be the current record). Invoked by Ctrl+F5 or Record|PostRecord.
DataPrint	SmallInt	Prints a Form or Table window. Invoked by File|Print or the Print button on Toolbar.
DataPriorRecord	SmallInt	Moves (if possible) to previous record in the table associated with the UIObject. Will force recursive action(DataUnlockRecord) if current record has been modified. Errors encountered will call **error** method. Invoked by Record|Previous, "Prior Record" Toolbar button, F11, and so forth.
DataPriorSet	SmallInt	Moves backward one set of records (where a set is defined as the number of rows in a table frame or MRO, or 1 in the case of a single-record form). Will force recursive action (DataUnlockRecord) if current record has been modified. Errors encountered will call error method. Invoked by PgUp.
DataRecalc	SmallInt	Forces an object and all objects it contains to refetch and recompute all their data. Invoked by Ctrl+F3.
DataRefresh	SmallInt	Notification of a refresh of a value in a record displayed on

		the screen.
DataRefreshOutside	SmallInt	Notification of a refresh of a value in a record not displayed on the screen.
DataSaveCrosstab	SmallInt	Writes given crosstab to CROSSTAB.DB. Different from EditSaveCrosstab, which brings up a dialog box asking user the name of the crosstab table to create.
DataSearch	SmallInt	Opens a dialog box to allow the user to search for a specific value within a specified field. Invoked by Record|Locate|Value, or Ctrl+Z.
DataSearchNext	SmallInt	Will search for the next record containing the value last specified in response to the last DataSearch action. Invoked by Record|Locate Next, or Ctrl+A.
DataSearchRecord	SmallInt	Opens a dialog box to allow the user to search for a record by specifying the record number. Invoked by Record|Locate|Record Number.
DataSearchReplace	SmallInt	Opens a dialog box to allow the user to search for a specific value within a specified field and to replace it with a different value. Invoked by Record|Locate and Replace, or Ctrl+Shift+Z.
DataShowDeleted	SmallInt	Alters the mode of the form so that deleted records will be shown (available only for dBASE tables). They will look no different from normal records, but status line will reflect their state. Invoked by Form|Show Deleted.
DataTableView	SmallInt	Used to open a Table View of the master table of a form. If this form was originally invoked as preferred form of existing Table View, this just returns focus to that Table View. Invoked by F7, Table View button on Toolbar or View|Table View.
DataToggleDeleted	SmallInt	Used to reverse the state of "show deleted records" for dBASE tables.
DataToggleDeleteRecord	SmallInt	Used to reverse the deleted state of records in dBASE tables.
DataToggleEdit	SmallInt	Used to reverse the Edit state of the form. Recursively calls action (DataBeginEdit) or action (DataEndEdit) as appropriate. Invoked by F9, or Edit Data button on Toolbar.
DataToggleLockRecord	SmallInt	Used to reverse the lock state of the current record. Actually just recursively uses action (DataLockRecord) or action (DataUnlockRecord) as appropriate. Errors encountered will call **error** method.
DataUnDeleteRecord	SmallInt	For dBASE tables, will match previously deleted record as "undeleted."
DataUnlockRecord	SmallInt	Used to commit the record modifications to the table and then (if successful) to unlock the record. Error encountered will call **error** method. Invoked by Record|Unlock or Shift+F5.

ActionEditCommand constants

Constant	Data type	Description
EditCommitField	SmallInt	Write current field's modifications to record buffer (without leaving field).
EditCopySelection	SmallInt	Copies selected area of text to Clipboard. Invoked by Edit|Copy or Ctrl+Ins.
EditCopyToFile	SmallInt	Invokes a dialog box to copy selection to a file. Invoked by Edit|Copy To.
EditCutSelection	SmallInt	Copies selected area of text to Clipboard and deletes it. Invoked by Edit|Cut or Ctrl+Del.
EditDeleteBeginLine	SmallInt	Deletes from current position to beginning of line.
EditDeleteEndLine	SmallInt	Deletes from current position to end of line.
EditDeleteLeft	SmallInt	Deletes one character position to the left. Invoked by Backspace in Field View.
EditDeleteLeftWord	SmallInt	Deletes up to and including beginning of word to the left of current character position.
EditDeleteLine	SmallInt	Deletes line on which current position is found.
EditDeleteRight	SmallInt	Deletes one character position to the right. Invoked by Del in Field View.
EditDeleteRightWord	SmallInt	Deletes up to and including end of word to the right of current character position.
EditDeleteSelection	SmallInt	Deletes currently selected area of text. Invoked by Edit|Delete.
EditDeleteWord	SmallInt	Deletes word around the current position. Invoked by Ctrl+Backspace.
EditDropDownList	SmallInt	Used by drop-down edit fields. Will drop down the associated pick list. Invoked by Alt+Down or click edit field's list icon.
EditEnterFieldView	SmallInt	Enters Field View for current field (allowing arrow keys to move around within the field). Begins by moving current position to end of field and unhighlighting it. Invoked by F2, View|Field View, or Field View button on Toolbar.
EditEnterMemoView	SmallInt	Enters Memo View on memos or OLE fields. Invoked by Shift+F2 or View|Memo View.
EditEnterPersistFieldView	SmallInt	Enters Persistent Field View, meaning arrow keys always move within character positions within a field, even when moving to new fields. Invoked by Ctrl+F2 or View|Persistent Field View.
EditExitFieldView	SmallInt	Exits Field View (meaning arrow keys will move between fields again) and highlights entire field. Invoked by F2, View|Field View, or Field View button on Toolbar.
EditExitMemoView	SmallInt	Exits Memo View on memos or OLE fields, meaning Enter and Tab will once again move between fields. Invoked by Shift+F2 or View|Memo View.
EditExitPersistField View	SmallInt	Exits Persistent Field View, meaning arrow keys move between fields. Invoked by Ctrl+F2 or View|Persistent Field View.

EditHelp	SmallInt	Invokes the Help subsystem. Invoked by F1.
EditInsertBlank	SmallInt	Inserts a blank character at current position.
EditInsertLine	SmallInt	Inserts a blank line at current position.
EditInsertObject (5.0)	SmallInt	Used only by OLE fields, inserts linked or embedded object into current field.
EditLaunchServer	SmallInt	Used only by OLE fields, will invoke the server application appropriate for current field.
EditPaste	SmallInt	Paste from the Clipboard to current position (replacing current selection if appropriate). Invoked by Shift+Ins or Edit|Paste.
EditPasteFromFile	SmallInt	Invokes a dialog box, allowing user to select file to insert at current position. Invoked by Edit|Paste From.
EditPasteLink (5.0)	SmallInt	Used only by OLE fields, pastes an object from the Clipboard and establishes a link to the underlying file. Invoked by Edit|Paste Link.
EditProperties	SmallInt	Invokes the property inspection menu for given object. Only unbound field objects, bound graphic fields, and bound formatted memo fields support this. Invoked by mouse right-click, Properties|Current Object, or F6.
EditReplace	SmallInt	Toggles overstrike mode in a field object.
EditSaveCrosstab (5.0)	SmallInt	Invokes a dialog box to allow user to save a crosstab. Invoked by Edit|Save Crosstab.
EditTextSearch	SmallInt	Invokes a dialog box to allow user to search and replace text within current field. Invoked by Edit|Search Text.
EditToggleFieldView	SmallInt	Reverses current state of Field View. Recursively calls action (EditEnterFieldView) or action (EditExitFieldView). Invoked by F2, Field View button on Toolbar, or Edit|Field View.
EditUndoField	SmallInt	Discards current fields modifications and reverts to value in current record buffer. Invoked by Esc.

ActionFieldCommand constants

Constant	Data type	Description
FieldBackward	SmallInt	Used to move one field backward in tab order. This will search for the prior UIObject marked as a "Tab Stop" in left-right/top-down order. Invoked by Shift+Tab.
FieldDown	SmallInt	Used to move to field below current field, whether in Field View or not. Invoked by Alt+↓.
FieldEnter	SmallInt	Used to commit modifications to a field (if any) and to move one field forward in tab order. Invoked by Enter.
FieldFirst	SmallInt	Used to move to first field within a record. Invoked by Alt+Home.
FieldForward	SmallInt	Used to move one field forward in tab order. This will search for the next UIObject marked as a "Tab Stop" in left-right/top-down order. Invoked by Tab.
FieldGroupBackward	SmallInt	Used to move one "super" tab group backward (for example, between different table frames on the same form). Invoked by F3.
FieldGroupForward	SmallInt	Used to move one "super" tab group forward (for example, between different table frames on the same form). Invoked by F4.
FieldLast	SmallInt	Used to move to last field within a record. Invoked by Alt+End or by End (when not in Field View).
FieldLeft	SmallInt	Used to move to field to left of current field.
FieldNextPage	SmallInt	Used to move to next sequential page in multi-page form. Invoked by Form|Page|Next or Shift+F4.
FieldPriorPage	SmallInt	Used to move to prior page in multi-page form. Invoked by Form|Page|Previous or Shift+F3.
FieldRight	SmallInt	Used to move to field to right of current field, whether in Field View or not. Invoked by Alt+→.
FieldRotate	SmallInt	Used to rotate columns within a table frame. Invoked by Ctrl+R.
FieldUp	SmallInt	Used to move to field above current field, whether in Field View or not. Invoked by Alt+↑.

ActionMoveCommand constants

Constant	Data type	Description
MoveBegin	SmallInt	In Memo View, moves to beginning of document. Otherwise, moves to first field in first record of table. Invoked by Ctrl+Home.
MoveBeginLine	SmallInt	In Memo View, moves to beginning of line; otherwise, moves to first field in record. Invoked by Home.
MoveBottom	SmallInt	In Memo View, moves to bottom line of the text region. Otherwise, moves to last record in table.
MoveBottomLeft	SmallInt	In Memo View, moves to beginning of last line on screen.
MoveBottomRight	SmallInt	In Memo View, moves to end of last line on screen. Invoked by Ctrl+PgDn.
MoveDown	SmallInt	Moves down as appropriate. In Memo View, moves down one line on multiline fields. Otherwise, moves to next Tab Stop object below current object. Table frame objects move to next record. Invoked by ↓.
MoveEnd	SmallInt	In Memo View, moves to end of document; otherwise, moves to last field in last record of table. Invoked by Ctrl+End.
MoveEndLine	SmallInt	In Memo View, moves to end of line; otherwise, moves to last field in record. Invoked by End.
MoveLeft	SmallInt	Moves left as appropriate. In Memo View, moves one character position left; otherwise, moves to next Tab Stop object to left of current object. Invoked by ←.
MoveLeftWord	SmallInt	In Memo View, moves insertion point to beginning of word to the left of current insertion point. Invoked by Ctrl+←.
MoveRight	SmallInt	Moves right as appropriate. In Memo View, moves one character position right; otherwise, moves to next Tab Stop object to right of current object. Invoked by →.
MoveRightWord	SmallInt	In Memo View, moves insertion point to beginning of word to the right of current insertion point. Invoked by Ctrl+→.
MoveScrollDown	SmallInt	Scrolls image down (effectively moving viewing area up) by appropriate amount. Active fields scroll by even lines of text. Tables move to new record. In Memo View, scroll toward the bottom of the text. The insertion point remains on the same line of the display region unless the last line of the text is visible, in which case the insertion point moves down one line until the last line is reached. Invoked by Ctrl+↓.
MoveScrollLeft	SmallInt	Scrolls image to left (effectively moving viewing area to the right) by appropriate amount. Active fields scroll roughly one character position. Tables move to new column.
MoveScrollPageDown	SmallInt	Scrolls image down (effectively moving viewing area up) by logical size of object (for example, complete page of document). Invoked by PgDn.
MoveScrollPageLeft	SmallInt	Scrolls image left (effectively moving viewing area right) by logical size of object (for example, complete page of document).
MoveScrollPageRight	SmallInt	Scrolls image right (effectively moving viewing area left) by logical size of object (for example, complete page of document).
MoveScrollPageUp	SmallInt	Scrolls image up (effectively moving viewing area down) by logical size of object (for example, complete page of document). Invoked by PgUp.

MoveScrollRight	SmallInt	Scrolls image to right (effectively moving viewing area to the left) by appropriate amount. Active fields scroll roughly one character position. Tables move to new column.
MoveScrollScreenDown	SmallInt	Scrolls image down (effectively moving viewing area up) by size of viewing area (for example, size of field). In Memo View, moves down in the document by the height of the display area.
MoveScrollScreenLeft	SmallInt	Scrolls image left (effectively moving viewing area right) by size of viewing area (for example, size of field).
MoveScrollScreenRight	SmallInt	Scrolls image right (effectively moving viewing area left) by size of viewing area (for example, size of field).
MoveScrollScreenUp	SmallInt	Scrolls image up (effectively moving viewing area down) by size of viewing area (for example, size of field). In Memo View, moves up in the document by the height of the display area.
MoveScrollUp	SmallInt	Scroll image up (effectively moving viewing area down) by appropriate amount. Active fields scroll by even lines of text. In Memo View, scroll toward the top of the document by one line of text. The insertion point stays at the same line position unless the top line of the document is visible, in which case the insertion point moves up one line if it can. Invoked by Ctrl+↑.
MoveTop	SmallInt	In Memo View, move the insertion point to the first line of text visible in the display region; otherwise, moves to first record in table.
MoveTopLeft	SmallInt	In Memo View, moves to the top left of the display region; otherwise, moves to top left field. Invoked by Ctrl+PgUp.
MoveTopRight	SmallInt	In Memo View, moves to the top right of the display region; otherwise, moves to top right field.
MoveUp	SmallInt	Moves up as appropriate. In Memo View, moves up one line on multiline fields; otherwise, it moves to next Tab Stop object above current object. Table frame objects move to prior record. Invoked by ↑.

ActionSelectCommand constants

Constant	Data type	Description
SelectBegin	SmallInt	In Memo View, selects from current position to beginning of document; otherwise, selects from current position to first field in first record of table. Invoked by Shift+Ctrl+Home.
SelectBeginLine	SmallInt	In Memo View, selects from current position to beginning of line; otherwise, selects from current position to first field in record. Invoked by Shift+Home.
SelectBottom	SmallInt	In Field View and Memo View, select from current position to bottom of the display region; otherwise, selects from current position to last record in table.
SelectBottomLeft	SmallInt	In Memo View, selects from current position to beginning of last line in display region. Invoked by Shift+Ctrl+PgUp.
SelectBottomRight	SmallInt	In Memo View, selects from current position to end of last line in display region. Invoked by Shift+Ctrl+PgDn.
SelectDown	SmallInt	Selects down as appropriate. In Field View or Memo View, selects down one line on multiline fields. Cannot extend selection across fields in forms. Table frame objects select to next record. Invoked by Shift+■.
SelectEnd	SmallInt	In Field View or Memo View, selects from current position to end of document; otherwise, selects from current position to last field in last record of table. Invoked by Shift+Ctrl+End.
SelectEndLine	SmallInt	In Field View or Memo View, selects from current position to end of line; otherwise, selects from current position to last field in record. Invoked by Shift+End.
SelectLeft	SmallInt	Selects left as appropriate. In Field View or Memo View, selects one character position left; otherwise, selects next Tab Stop object to left of current object. Invoked by Shift+■.
SelectLeftWord	SmallInt	In Field View or Memo View, if the insertion point is between words, selects word to the left of insertion point. If the insertion point is within a word, selects to the beginning of that word. Invoked by Shift+Ctrl+■.
SelectRight	SmallInt	Selects right as appropriate. In Field View or Memo View, selects one character position right. Invoked by Shift+■.
SelectRightWord	SmallInt	In Field View or Memo View, selects to the beginning of the next following word. If the insertion point precedes one or more spaces or tabs, selection only includes those spaces or tabs. Invoked by Shift+Ctrl+■.
SelectScrollDown	SmallInt	Selects image down (effectively moving viewing area up) by appropriate amount. Active fields select even lines of text. Tables select new record. Invoked by Shift+Crtl+■.
SelectScrollLeft	SmallInt	Selects image on left (effectively moving viewing area to the right) by appropriate amount. Active fields select roughly one character position. Tables select to new column.
SelectScrollPageDown	SmallInt	Selects image down (effectively moving viewing area up) by logical size of object (for example, complete page of document).
SelectScrollPageLeft	SmallInt	Selects image left (effectively moving viewing area right) by logical size of object (for example, complete page of document).
SelectScrollPageRight	SmallInt	Selects image right (effectively moving viewing area left) by logical size of object (for example, complete page of document).

SelectScrollPageUp	SmallInt	Selects image up (effectively moving viewing area down) by logical size of object (for example, complete page of document).
SelectScrollRight	SmallInt	Selects image on right (effectively moving viewing area to the left) by appropriate amount. Active fields select roughly one character position. Tables select new column.
SelectScrollScreenDown	SmallInt	Selects image down (effectively moving viewing area up) by size of viewing area (for example, size of field). Invoked by Shift+PgDn.
SelectScrollScreenLeft	SmallInt	Selects image left (effectively moving viewing area right) by size of viewing area (for example, size of field).
SelectScrollScreenRight	SmallInt	Selects image right (effectively moving viewing area left) by size of viewing area (for example, size of field).
SelectScrollScreenUp	SmallInt	Selects image up (effectively moving viewing area down) by size of viewing area (for example, size of field). Invoked by Shift+PgUp.
SelectScrollUp	SmallInt	Select image up (effectively moving viewing area down) by appropriate amount. Active fields select by even lines of text.
SelectSelectAll	SmallInt	Selects the entire field.
SelectTop	SmallInt	In Field View or Memo View, selects from current position to top of display region; otherwise, selects from current position to first record in table.
SelectTopLeft	SmallInt	In Field View or Memo View, selects from current position to beginning of screen; otherwise, selects from current position to top left field. Invoked by Shift+Crtl+PgUp.
SelectTopRight	SmallInt	In Field View or Memo View, selects from current position to end of top line of screen; otherwise, selects from current position to top right field. Invoked by Shift+Crtl+PgDn.
SelectUp	SmallInt	Selects up as appropriate. In Field View or Memo View, selects up one line on multiline fields; otherwise, it selects next Tab Stop object above current object. Table frame objects select to prior record. Invoked by Shift+▪.

AggModifier constants

Constant	Data type	Description
CumulativeAgg	SmallInt	A cumulative summary that keeps a running total that extends from the start of the report to the end of the current group.
CumUniqueAgg	SmallInt	A cumulative summary that counts only the unique non-null values from the start of the report to the end of the current group.
RegularAgg	SmallInt	A normal summary that considers all non-null values in the set, including duplicates.
UniqueAgg	SmallInt	A unique summary that counts only the unique non-null values in the set. Duplicates are ignored.

ButtonStyle constants

Constant	Data type	Description
BorlandButton	SmallInt	Gives a button the 3D look of buttons in Borland products.
Windows3dButton	SmallInt	Gives a button the 3D look of buttons in other Windows products.
WindowsButton	SmallInt	Gives a button the flat look of buttons in other Windows products.

ButtonType constants

Constant	Data type	Description
CheckboxType	SmallInt	Displays a button as a check box.
PushButtonType	SmallInt	Displays a button as a push button.
RadioButtonType	SmallInt	Displays a button as a radio button.

Color constants

Constant	Data type
Black	LongInt
Blue	LongInt
Brown	LongInt
DarkBlue	LongInt
DarkCyan	LongInt
DarkGray	LongInt
DarkGreen	LongInt
DarkMagenta	LongInt
DarkRed	LongInt
Gray	LongInt
Green	LongInt
LightBlue	LongInt
Magenta	LongInt
Red	LongInt
Translucent	LongInt
Transparent	LongInt
White	LongInt
Yellow	LongInt

CompleteDisplay constants

Constant	Data type	Description
DisplayAll	SmallInt	Specifies CompleteDisplay for all field objects in the form.
DisplayCurrent	SmallInt	Specifies CompleteDisplay for the current field object.

DateRangeType constants

Constant	Data type	Description
ByDay	SmallInt	Group report records by day.
ByMonth	SmallInt	Group report records by month.
ByQuarter	SmallInt	Group report records by quarter (3 months).
ByWeek	SmallInt	Group report records by week.
ByYear	SmallInt	Group report records by year.

DesktopPreferenceTypes constants

Constant	Data type	Description
Section = prefProjectSection		
prefStartUpExpert	SmallInt	Experts page: Run Startup Expert each time Paradox loads
prefTitleName	SmallInt	General page: Title
prefExpertDefault	SmallInt	Experts page: Run experts when creating objects on documents
prefBackgroundName	SmallInt	General page: Background bitmap
prefTileBitmap	SmallInt	General page: Tile bitmap
prefSaveOnExit	SmallInt	General page: Desktop state: Save on exit
prefRestoreDesktop	SmallInt	General page: Desktop state: Restore on startup
prefSystemFont	SmallInt	General page: Default system font
prefScreenPageDesk	SmallInt	Forms/Reports page: On-screen size: Size to desktop
prefScreenPageWidth	SmallInt	Forms/Reports page: On-screen size: Width
prefScreenPageHeight	SmallInt	Forms/Reports page: On-screen size: Height
prefFormOpen	SmallInt	Forms/Reports page: Open default: Open forms in design mode
prefReportOpen	SmallInt	Forms/Reports page: Open default: Open reports in design mode
prefWarnOnDirChange	SmallInt	Advanced page: Don't show warning prompts when changing directories
prefBitmapButtons	SmallInt	Change to Borland-style buttons
prefAltKeyPadChars	SmallInt	Advanced page: Always use Alt + numeric keypad for character entry
prefExpandBranchs	SmallInt	Advanced page: Indicate expandable directory branches
prefScrollBarsInForms	SmallInt	Advanced page: Use scroll bars in form windows by default
prefBlankAsZeroName	SmallInt	Database page: Treat blank fields as zeros
prefRefreshRate	SmallInt	Database page: Refresh rate (seconds):
prefExpertsOnCreate	SmallInt	Forms/Reports page: New forms/reports: Always use expert
prefUserLevel	SmallInt	Developer Preferences: General page: ObjectPAL level
prefDeveloperMenu	SmallInt	Developer Preferences: General page: Show developer menus
prefEnableControlBreak	SmallInt	Developer Preferences: General page: Debugger settings: Enable Ctrl+Break
Section = prefQbeSection		
prefAuxOpts	SmallInt	Generate auxiliary tables
prefSqlRunMode	SmallInt	Query: Queries against remote tables
prefDefCheck	SmallInt	Default QBE check type
prefSqlConstrained	SmallInt	SQL answer constraints

Section = prefProjViewSection

prefOpenOnStartup	SmallInt	General: Project Viewer settings: Open Project Viewer on startup

ErrorReason constants

Constant	Data type	Description
ErrorCritical	SmallInt	Displays a message in a modal dialog box.
ErrorWarning	SmallInt	Displays a message in the status area.

EventErrorCode constants

Constant	Data type	Description
Can_Arrive	SmallInt	Grants permission to arrive at an object.
Can_Depart	SmallInt	Grants permission to leave an object.
CanNotArrive	SmallInt	Refuses permission to arrive at an object (blocks the move).
CanNotDepart	SmallInt	Refuses permission to leave an object (blocks the move).

ExecuteOption constants

Constant	Data type	Description
ExeHidden	SmallInt	Hides the application window and passes activation to another window.
ExeMinimized	SmallInt	Minimizes the application window and activates the top-level window in the window-manager's list.
ExeShowMaximized	SmallInt	Activates the application window and displays it as a maximized window.
ExeShowMinimized	SmallInt	Activates the application window and displays it minimized (as an icon).
ExeShowMinimizedNoActivate	SmallInt	Displays the application as an icon. The window that is currently active remains active.
ExeShowNoActivate	SmallInt	Displays the application window at its most recent size and position. The currently active window remains active.
ExeShowNormal	SmallInt	Activates and displays a window.

FieldDisplayType constants

Constant	Data type	Description
BitmapField	SmallInt	Enables a field object to display a bitmap.
CheckboxField	SmallInt	Displays a field as a check box.
ComboField	SmallInt	Displays a field as a drop-down edit list (also called a combo box).
EditField	SmallInt	Displays an unlabeled field.
LabeledField	SmallInt	Displays a labeled field.
ListField	SmallInt	Displays a list box.
OleField	SmallInt	Enables a field to contain OLE data.
RadioButtonField	SmallInt	Displays a field as one or more radio buttons.

FileBrowserFileType constants

Constant	Data type	Description
fbAllTables	LongInt	All table types supported by Paradox (*.db, *.dbf, etc.).
fbASCII	LongInt	Text files (*.txt).
fbBitmap	LongInt	Bitmap graphics (*.bmp).
fbDBase	LongInt	dBASE tables (*.dbf).
fbDM (5.0)	LongInt	Data model files (*.dm).
fbExcel	LongInt	Excel worksheets (*.xls).
fbFiles	LongInt	All files (*.*).
fbForm	LongInt	Paradox forms (*.fsl, *.fdl).
fbGraphic	LongInt	Graphic files (*.bmp, *.eps, *.gif, *.pcx, *.tif).
fbIni	LongInt	Initialization files (*.ini).
fbLibrary	LongInt	ObjectPAL libraries (*.lsl, *.ldl).
fbLotus1	LongInt	Lotus 1-2-3 version 1 worksheets (*.wks).
fbLotus2	LongInt	Lotus 1-2-3 version 2 worksheets (*.wk1).
fbParadox	LongInt	Paradox tables (*.db).
fbQuattro	LongInt	Quattro worksheets (*.wkq).
fbQuattroPro	LongInt	Quattro Pro worksheets (*.wq1).
fbQuattroProWindows	LongInt	Quattro Pro for Windows notebooks (*.wb1).
fbQuery	LongInt	Query files (*.qbe).
fbReport	LongInt	Paradox reports (*.rsl, *.rdl).
fbScreenStyle (5.0)	LongInt	Form style sheets (*.ft).
fbScript	LongInt	ObjectPAL scripts (*.ssl, *.sdl).
fbSQL (5.0)	LongInt	SQL files (*.sql).
fbTable	LongInt	All table types supported by Paradox (*.db, *.dbf, etc.).
fbTableView	LongInt	Paradox table view files (*.tv).
fbText	LongInt	All text files (*.txt, *.pxt, *.rtf).

FontAttribute constants

Constant	Data type	Description
FontAttribBold	SmallInt	Example: **bold**
FontAttribItalic	SmallInt	Example: *italic*
FontAttribNormal	SmallInt	Example: normal
FontAttribStrikeOut	SmallInt	Example: ~~strike out~~
FontAttribUnderline	SmallInt	Example: underline

FrameStyle constants

Constant	Data type	Description
DashDotDotFrame	SmallInt	A repeating sequence of one dash followed by two dots.
DashDotFrame	SmallInt	A repeating sequence of one dash followed by one dot.
DashedFrame	SmallInt	A repeating sequence of dashes.
DottedFrame	SmallInt	A repeating sequence of dots.
DoubleFrame	SmallInt	Two concentric boxes.
Inside3DFrame	SmallInt	The frame appears pushed into the form.
NoFrame	SmallInt	No frame.
Outside3DFrame	SmallInt	The frame appears popped out of the form.
ShadowFrame	SmallInt	A drop shadow.
SolidFrame	SmallInt	A single solid box (no dashes or dots).
WideInsideDoubleFrame	SmallInt	Two concentric boxes; the inside box is wide.
WideOutsideDoubleFrame	SmallInt	Two concentric boxes; the outside box is wide.
Windows3dFrame (5.0)	SmallInt	Uses the default Windows 3D frame style.
Windows3dGroup (5.0)	SmallInt	Uses the default Windows 3D group border.

General constants

Constant	Data type	Description
No	Logical	False
Off	Logical	False
On	Logical	True
Pi	Number	3.14159265358979323846
Yes	Logical	True

GraphBindType constants

Constant	Data type	Description
Graph1DSummary	SmallInt	Specifies a one-dimensional summary chart. Enables summary operators.
Graph2DSummary	SmallInt	Specifies a two-dimensional summary chart. Enables summary operators and group-by specification.
GraphTabular	SmallInt	Specifies a tabular chart (default).

GraphicMagnification constants

Constant	Data type	Description
Magnify100	SmallInt	Displays the chart at its actual size.
Magnify200	SmallInt	Displays the chart at twice its actual size.
Magnify25	SmallInt	Displays the chart at a quarter of its actual size.
Magnify400	SmallInt	Displays the chart at four times its actual size.
Magnify50	SmallInt	Displays the chart at half its actual size.
MagnifyBestFit	SmallInt	Resizes the chart as necessary to fit the chart in the frame.

GraphLabelFormat constants

Constant	Data type	Description
GraphHideY	SmallInt	Hide Y-value (2-D and 3-D Pie and Column charts only).
GraphPercent	SmallInt	Display Y-value as a percent (2-D and 3-D Pie and Column charts only).
GraphShowY	SmallInt	Display Y-value in the units used in the table (2-D and 3-D Pie and Column charts only).

GraphLegendPosition constants

Constant	Data type	Description
LegendCenter	SmallInt	Display the legend centered below the chart.
LegendLeft	SmallInt	Display the legend to the left of the chart.

GraphMarkers

Constant	Data type	Description
MarkerBoxedCross	SmallInt	Marker is a box with a cross in it.
MarkerBoxed_Plus	SmallInt	Marker is a box with a plus sign in it.
MarkerCross	SmallInt	Marker is a cross.
MarkerFilledBox	SmallInt	Marker is a filled box.
MarkerFilledCircle	SmallInt	Marker is a filled circle.
MarkerFilledDownTriangle	SmallInt	Marker is a filled triangle pointing down.
MarkerFilledTriangle	SmallInt	Marker is a filled triangle pointing up.
MarkerFilledTriangles	SmallInt	Marker is two filled triangles pointing at each other.
MarkerHollowBox	SmallInt	Marker is a hollow (unfilled) box.
MarkerHollowCircle	SmallInt	Marker is a hollow circle.
MarkerHollowDownTriangle	SmallInt	Marker is a hollow triangle pointing down.
MarkerHollowTriangle	SmallInt	Marker is a hollow triangle pointing up.
MarkerHollowTriangles	SmallInt	Marker is two hollow triangles pointing at each other.
MarkerHorizontalLine	SmallInt	Marker is a horizontal line.
MarkerPlus	SmallInt	Marker is a plus sign.
MarkerVerticalLine	SmallInt	Marker is a vertical line.

GraphTypeOverRide

Constant	Data type	Description
GraphArea	SmallInt	Displays specified series as an area chart.
GraphBar	SmallInt	Displays specified series as a bar chart.
GraphDefault	SmallInt	Displays specified series in the default chart type.
GraphLine	SmallInt	Displays specified series as a line chart.

GraphTypes

Constant	Data type	Description
Graph2DArea	SmallInt	2-dimensional area chart.
Graph2DBar	SmallInt	2-dimensional bar chart.
Graph2DColumns	SmallInt	2-dimensional column chart.
Graph2DLine	SmallInt	2-dimensional line chart.
Graph2DPie	SmallInt	2-dimensional pie chart.
Graph2DRotatedBar	SmallInt	2-dimensional rotated bar chart.
Graph2DStackedBar	SmallInt	2-dimensional stacked bar chart.
Graph3DArea	SmallInt	3-dimensional area chart.
Graph3DBar	SmallInt	3-dimensional bar chart.
Graph3DColumns	SmallInt	3-dimensional column chart.
Graph3DPie	SmallInt	3-dimensional pie chart.
Graph3DRibbon	SmallInt	3-dimensional ribbon chart.
Graph3DRotatedBar	SmallInt	3-dimensional rotated bar chart.
Graph3DStackedBar	SmallInt	3-dimensional stacked bar chart.
Graph3DStep	SmallInt	3-dimensional step chart.
Graph3DSurface	SmallInt	3-dimensional surface chart.
GraphXY	SmallInt	XY chart.

IdRanges

Constant	Data type	Description
UserAction	SmallInt	The minimum value for a user-defined action constant.
UserActionMax	SmallInt	The maximum value for a user-defined action constant.
UserError	SmallInt	The minimum value for a user-defined error constant.
UserErrorMax	SmallInt	The maximum value for a user-defined error constant.
UserMenu	SmallInt	The minimum value for a user-defined menu ID constant.
UserMenuMax	SmallInt	The maximum value for a user-defined menu ID constant.

Keyboard constants

Constant	Data type	Description
VK_ADD	SmallInt	Add key
VK_APPS	SmallInt	Application property inspection key
VK_BACK	SmallInt	Backspace key
VK_CANCEL	SmallInt	Used for control-break processing
VK_CAPITAL	SmallInt	Capital key
VK_CLEAR	SmallInt	Clear key
VK_CONTROL	SmallInt	Ctrl key
VK_DECIMAL	SmallInt	Decimal key
VK_DELETE	SmallInt	Delete key
VK_DIVIDE	SmallInt	Divide key
VK_DOWN	SmallInt	▪ key
VK_END	SmallInt	End key
VK_ESCAPE	SmallInt	Escape key
VK_EXECUTE	SmallInt	Execute key
VK_F1	SmallInt	F1 key
VK_F10	SmallInt	F10 key
VK_F11	SmallInt	F11 key
VK_F12	SmallInt	F12 key
VK_F13	SmallInt	F13 key
VK_F14	SmallInt	F14 key
VK_F15	SmallInt	F15 key
VK_F16	SmallInt	F16 key
VK_F2	SmallInt	F2 key
VK_F3	SmallInt	F3 key
VK_F4	SmallInt	F4 key
VK_F5	SmallInt	F5 key
VK_F6	SmallInt	F6 key
VK_F7	SmallInt	F7 key
VK_F8	SmallInt	F8 key
VK_F9	SmallInt	F9 key
VK_HELP	SmallInt	Help key
VK_HOME	SmallInt	Home key
VK_INSERT	SmallInt	Insert key
VK_LBUTTON	SmallInt	Left mouse button
VK_LEFT	SmallInt	▪ key
VK_MBUTTON	SmallInt	Middle mouse button (3-button mouse)
VK_MENU	SmallInt	Menu key
VK_MULTIPLY	SmallInt	Multiply key
VK_NEXT	SmallInt	Page Down key
VK_NUMLOCK	SmallInt	Num Lock key

VK_NUMPAD0	SmallInt	Key pad 0 key
VK_NUMPAD1	SmallInt	Key pad 1 key
VK_NUMPAD2	SmallInt	Key pad 2 key
VK_NUMPAD3	SmallInt	Key pad 3 key
VK_NUMPAD4	SmallInt	Key pad 4 key
VK_NUMPAD5	SmallInt	Key pad 5 key
VK_NUMPAD6	SmallInt	Key pad 6 key
VK_NUMPAD7	SmallInt	Key pad 7 key
VK_NUMPAD8	SmallInt	Key pad 8 key
VK_NUMPAD9	SmallInt	Key pad 9 key
VK_PAUSE	SmallInt	Pause key
VK_PRINT	SmallInt	OEM specific
VK_PRIOR	SmallInt	Page Up key
VK_RBUTTON	SmallInt	Right mouse button
VK_RETURN	SmallInt	Return key
VK_RIGHT	SmallInt	▪ key
VK_SELECT	SmallInt	Select key
VK_SEPARATOR	SmallInt	Separator key
VK_SHIFT	SmallInt	Shift key
VK_SNAPSHOT	SmallInt	Printscreen key for Windows 3.0 and later
VK_SPACE	SmallInt	Space
VK_SUBTRACT	SmallInt	Subtract key
VK_TAB	SmallInt	Tab key
VK_UP	SmallInt	▪ key

KeyBoardState constants

Constant	Data type	Description
Alt	SmallInt	Alt is pressed.
Control	SmallInt	Ctrl is pressed.
LeftButton	SmallInt	The left mouse button is clicked.
RightButton	SmallInt	The right mouse button is clicked.
Shift	SmallInt	Shift is pressed.

LibraryScope constants

Constant	Data type	Description
GlobalToDesktop	SmallInt	Makes variables in an ObjectPAL library available to one or more forms.
PrivateToForm	SmallInt	Makes variables in an ObjectPAL library available to one form only.

LineEnd constants

Constant	Data type	Description
ArrowBothEnds	SmallInt	Adds arrows to both ends of a line (only if LineType = StraightLine)
ArrowOneEnd	SmallInt	Adds an arrow to the terminal end of a line (only if LineType = StraightLine)
NoArrowEnd	SmallInt	Displays a line without arrows at either end

LineStyle constants

Constant	Data type	Description
DashDotDotLine	SmallInt	A repeating sequence of one dash followed by two dots.
DashDotLine	SmallInt	A repeating sequence of one dash followed by one dot.
DashedLine	SmallInt	A repeating sequence of dashes.
DottedLine	SmallInt	A repeating sequence of dots.
NoLine	SmallInt	No line.
SolidLine	SmallInt	An unbroken line.

LineThickness constants

Constant	Data type	Description
LWidth10Points	SmallInt	Specifies a thickness of 10 printer's points.
LWidth1Point	SmallInt	Specifies a thickness of 1 printer's point.
LWidth2Points	SmallInt	Specifies a thickness of 2 printer's points.
LWidth3Points	SmallInt	Specifies a thickness of 3 printer's points.
LWidth6Points	SmallInt	Specifies a thickness of 6 printer's points.
LWidthHairline	SmallInt	Specifies a very thin line.
LWidthHalfFoint	SmallInt	Specifies a thickness of one half of a printer's point.

LineType constants

Constant	Data type	Description
CurvedLine	SmallInt	Specifies a curved (elliptical) line.
StraightLine	SmallInt	Specifies a straight line.

MenuChoiceAttribute constants

Constant	Data type	Description
MenuChecked	SmallInt	Insert a checkmark before the menu item.
MenuDisabled	SmallInt	Menu item cannot be selected. Menu stays open.
MenuEnabled	SmallInt	Menu item can be selected. Menu closes.
MenuGrayed	SmallInt	Menu item displayed in gray characters (dimmed).
MenuHilited	SmallInt	Menu item is highlighted.
MenuNotChecked	SmallInt	Display menu item without a checkmark.
MenuNotGrayed	SmallInt	Display the menu item normally (not dimmed).
MenuNotHilited	SmallInt	Display menu item without a highlight.

MenuCommand constants

Constant	Data type	Description
MenuAddPage	SmallInt	Form|Add Page
MenuAlignBottom	SmallInt	Design|Align Bottom
MenuAlignCenter	SmallInt	Design|Align Center
MenuAlignMiddle	SmallInt	Design|Align Middle
MenuAlignLeft	SmallInt	Design|Align Left
MenuAlignRight	SmallInt	Design|Align Right
MenuAlignTop	SmallInt	Design|Align Top
MenuBuild (4.5)	SmallInt	Reports when the Desktop is building a form's menu.
MenuCanClose	SmallInt	Asks for permission to continue after choosing Close from the Control menu.
MenuChangedPriv (5.0)	SmallInt	Reports when the private directory (:PRIV:) has been changed. Forms that remain open after the change can use this information to make adjustments, as needed.
MenuChangedWork (5.0)	SmallInt	Reports when the working directory (:WORK:) has been changed. Forms that remain open after the change can use this information to make adjustments, as needed.
MenuChangingPriv (5.0)	SmallInt	Reports when the private directory (:PRIV:) is about to change. Setting the error code to a nonzero value allows a form to stay open after the change; setting the error code to zero closes the form before changing the directory. Same as choosing File|Private Directory.
MenuChangingWork (5.0)	SmallInt	Reports when the working directory (:WORK:) is about to change. Setting the error code to a nonzero value allows a form to stay open after the change; setting the error code to zero closes the form before changing the directory. Same as choosing File|Working Directory.
MenuCompileWithDebug	SmallInt	Program|CompileWithDebug
MenuControlClose	SmallInt	Same as choosing Close from the Control menu.
MenuControlKeyMenu	SmallInt	Control menu was invoked by a keypress.
MenuControlMaximize	SmallInt	Same as choosing Maximize from the Control menu.
MenuControlMinimize	SmallInt	Same as choosing Minimize from the Control menu.
MenuControlMouseMenu	SmallInt	Control menu was invoked by a mouse click.
MenuControlMove	SmallInt	Same as choosing Move from the Control menu.
MenuControlNextWindow	SmallInt	Same as choosing Next Window from the

<table>
<tr><td></td><td></td><td>Control menu.</td></tr>
<tr><td>MenuControlPrevWindow</td><td>SmallInt</td><td>Same as choosing Prev Window from the Control menu.</td></tr>
<tr><td>MenuControlRestore</td><td>SmallInt</td><td>Same as choosing Restore from the Control menu.</td></tr>
<tr><td>MenuControlSize</td><td>SmallInt</td><td>Same as choosing Size from the Control menu.</td></tr>
<tr><td>MenuCopyToolbar</td><td>SmallInt</td><td>Design|Copy to Toolbar</td></tr>
<tr><td>MenuDataModel</td><td>SmallInt</td><td>Form|Data Model</td></tr>
<tr><td>MenuDataModelDesigner</td><td>SmallInt</td><td>Tools|Data Model Designer</td></tr>
<tr><td>MenuDeliver</td><td>SmallInt</td><td>File|Deliver</td></tr>
<tr><td>MenuDesignBringFront</td><td>SmallInt</td><td>Design|Bring to Front</td></tr>
<tr><td>MenuDesignDuplicate</td><td>SmallInt</td><td>Design|Duplicate</td></tr>
<tr><td>MenuDesignGroup</td><td>SmallInt</td><td>Design|Group</td></tr>
<tr><td>MenuDesignLayout</td><td>SmallInt</td><td>Design|Design Layout</td></tr>
<tr><td>MenuDesignSendBack</td><td>SmallInt</td><td>Design|Send to Back</td></tr>
<tr><td>MenuEditCopy</td><td>SmallInt</td><td>Edit|Copy</td></tr>
<tr><td>MenuEditCopyTo</td><td>SmallInt</td><td>Edit|Copy To</td></tr>
<tr><td>MenuEditCut</td><td>SmallInt</td><td>Edit|Cut</td></tr>
<tr><td>MenuEditDelete</td><td>SmallInt</td><td>Edit|Delete</td></tr>
<tr><td>MenuEditLinks</td><td>SmallInt</td><td>For OLE objects only</td></tr>
<tr><td>MenuEditPaste</td><td>SmallInt</td><td>Edit|Paste</td></tr>
<tr><td>MenuEditUndo</td><td>SmallInt</td><td>Edit|Undo</td></tr>
<tr><td>MenuFieldFilter</td><td>SmallInt</td><td>Right-click Filter on Field object</td></tr>
<tr><td>MenuFieldPicture</td><td>SmallInt</td><td>Right-click Picture... on unbound fields</td></tr>
<tr><td>MenuFileAliases</td><td>SmallInt</td><td>Tools|Alias Manager...</td></tr>
<tr><td>MenuFileAutoRefresh</td><td>SmallInt</td><td>Edit|Preferences:Database:Refresh Rate</td></tr>
<tr><td>MenuFileExit</td><td>SmallInt</td><td>File|Exit</td></tr>
<tr><td>MenuFileExport</td><td>SmallInt</td><td>File|Export</td></tr>
<tr><td>MenuFileImport</td><td>SmallInt</td><td>File|Import</td></tr>
<tr><td>MenuFileMultiBlankZero</td><td>SmallInt</td><td>Edit|Preferences|Treat blank fields as zeros</td></tr>
<tr><td>MenuFileMultiUserDrivers</td><td>SmallInt</td><td>Edit|Preferences:BDE:Database driver list</td></tr>
<tr><td>MenuFileMultiUserInfo</td><td>SmallInt</td><td>Edit|Preferences|Database</td></tr>
<tr><td>MenuFileMultiUserLock</td><td>SmallInt</td><td>Tools|Set Locks</td></tr>
<tr><td>MenuFileMultiUserLockInfo</td><td>SmallInt</td><td>Tools|Display Locks</td></tr>
<tr><td>MenuFileMultiUserRetry</td><td>SmallInt</td><td>Edit|Preferences|Database|Set Retry</td></tr>
<tr><td>MenuFileMultiUserUserName</td><td>SmallInt</td><td>Edit|Preferences|Database|User name</td></tr>
<tr><td>MenuFileMultiUserWho</td><td>SmallInt</td><td>Edit|Preferences|Database|Current user list</td></tr>
<tr><td>MenuFilePrint</td><td>SmallInt</td><td>File|Print</td></tr>
<tr><td>MenuFilePrinterSetup</td><td>SmallInt</td><td>File|Printer Setup</td></tr>
<tr><td>MenuFilePrivateDir</td><td>SmallInt</td><td>Edit|Preferences|Database|Private directory</td></tr>
</table>

		(:PRIV:)
MenuFileTableAdd	SmallInt	Tools|Utilities|Add
MenuFileTableCopy	SmallInt	Tools|Utilities|Copy
MenuFileTableDelete	SmallInt	Tools|Utilities|Delete
MenuFileTableEmpty	SmallInt	Tools|Utilities|Empty
MenuFileTableInfoStructure	SmallInt	Tools|Utilities|Info Structure
MenuFileTablePasswords	SmallInt	Tools|Passwords
MenuFileTableRename	SmallInt	Tools|Utilities|Rename
MenuFileTableRestructure	SmallInt	Tools|Utilities|Restructure
MenuFileTableSort	SmallInt	Tools|Utilities|Sort
MenuFileTableSubtract	SmallInt	Tools|Utilities|Subtract
MenuFileWorkingDir	SmallInt	File|Working Directory
MenuFolderOpen	SmallInt	Tools|Project Viewer
MenuFormDesign	SmallInt	Design
MenuFormEditData	SmallInt	Form|Edit Data
MenuFormFieldView	SmallInt	Form|Field View
MenuFormFilter	SmallInt	Form|Filter
MenuFormMemoView	SmallInt	View|Memo View
MenuFormNew	SmallInt	File|New|Form
MenuFormOpen	SmallInt	File|Open|Form
MenuFormOrderRange	SmallInt	Form|Filter
MenuFormPageFirst	SmallInt	Form|Page|First
MenuFormPageGoto	SmallInt	Form|Page|Go To
MenuFormPageLast	SmallInt	Form|Page|Last
MenuFormPageNext	SmallInt	Form|Page|Next
MenuFormPagePrevious	SmallInt	Form|Page|Previous
MenuFormPersistView	SmallInt	View|Persistent Field View
MenuFormShowDeleted	SmallInt	Form|Show Deleted
MenuFormTableView	SmallInt	Form|Table View
MenuFormView	SmallInt	Form|View Data
MenuFormViewData (5.0)	SmallInt	Form|View Data
MenuHelpAbout	SmallInt	Help|About
MenuHelpCoach (5.0)	SmallInt	Help|Coaches
MenuHelpContents	SmallInt	Help|Contents
MenuHelpKeyboard	SmallInt	Help|Keyboard
MenuHelpSearch	SmallInt	Help|ObjectPAL reference
MenuHelpSupport	SmallInt	Help|Support Info
MenuHelpToolbar (5.0)	SmallInt	Help|Toolbar
MenuHelpUsingHelp	SmallInt	Help|Using Help
MenuInit	SmallInt	Generated by clicking a menu item

MenuInsertObject	SmallInt	For OLE objects only
MenuLibraryNew	SmallInt	File|New|Library
MenuLibraryOpen	SmallInt	File|Open|Library
MenuNoteBookAddPage	SmallInt	Form|Add Page
MenuNoteBookFirstPage	SmallInt	Form|Page|First
MenuNoteBookLastPage	SmallInt	Form|Page|Last
MenuNoteBookNextPage	SmallInt	Form|Page|Next
MenuNoteBookPriorPage	SmallInt	Form|Page|Previous
MenuNoteBookRotate	SmallInt	Form|Rotate Pages
MenuOpenProjectView (5.0)	SmallInt	Tools|Project Viewer
MenuPageLayout	SmallInt	Form|Page Layout
MenuPasteFrom	SmallInt	Edit|Paste From
MenuPasteLink	SmallInt	Edit|Paste Link
MenuPropertiesBandLabels	SmallInt	View|Band Labels
MenuPropertiesCurrent	SmallInt	Design|Current Object
MenuPropertiesCurrentDialog	SmallInt	Design|Current object
MenuPropertiesDesigner	SmallInt	Form|Design
MenuPropertiesDesktop	SmallInt	Form|View Data
MenuPropertiesExpandedRuler	SmallInt	Edit|Preferences:Designer:Expanded Ruler
MenuPropertiesFormRestoreDefaults	SmallInt	Form|Restore Defaults
MenuPropertiesFormSaveDefaults	SmallInt	Form|Save Defaults
MenuPropertiesGridSettings	SmallInt	Report|Settings|Grid
MenuPropertiesGroupRepeats	SmallInt	Report|Properties|Remove group repeat
MenuPropertiesHorizontalRuler	SmallInt	Edit|Preferences:Designer:Horizontal Ruler
MenuPropertiesMethods	SmallInt	Object Explorer
MenuPropertiesShowGrid	SmallInt	View|Grid
MenuPropertiesSizeandPos	SmallInt	View|Size and Position
MenuPropertiesSizeToFit	SmallInt	Report|Properties
MenuPropertiesSnapToGrid	SmallInt	Design|Snap to Grid
MenuPropertiesStyleSheet	SmallInt	Report|Style Sheet
MenuPropertiesVerticalRuler	SmallInt	Edit|Preferences:Designer:Vertical Ruler
MenuPropertiesWindow	SmallInt	View|Window Style
MenuPropertiesZoom100	SmallInt	View|Zoom|100%
MenuPropertiesZoom200	SmallInt	View|Zoom|200%
MenuPropertiesZoom25	SmallInt	View|Zoom|25%
MenuPropertiesZoom400	SmallInt	View|Zoom|400%
MenuPropertiesZoom50	SmallInt	View|Zoom|50%
MenuPropertiesZoomBestFit	SmallInt	View|Zoom|Best Fit
MenuPropertiesZoomFitHeight	SmallInt	View|Zoom|Fit Height
MenuPropertiesZoomFitWidth	SmallInt	View|Zoom|Fit Width

MenuQueryNew	SmallInt	File|New|Query
MenuQueryOpen	SmallInt	File|Open|Query
MenuRecordCancel	SmallInt	Record|Cancel Changes
MenuRecordDelete	SmallInt	Record|Delete
MenuRecordFastBackward	SmallInt	Record|Previous Set
MenuRecordFastForward	SmallInt	Record|Next Set
MenuRecordFirst	SmallInt	Record|First
MenuRecordInsert	SmallInt	Record|Insert
MenuRecordLast	SmallInt	Record|Last
MenuRecordLocateNext	SmallInt	Record|Locate Next
MenuRecordLocateRecordNumber	SmallInt	Record|Locate|Record Number
MenuRecordLocateSearchAndReplace	SmallInt	Record|Locate|and Replace
MenuRecordLocateValue	SmallInt	Record|Locate|Value
MenuRecordLock	SmallInt	Record|Lock
MenuRecordLookup	SmallInt	Record|Lookup Help
MenuRecordMove	SmallInt	Record|Move Help
MenuRecordNext	SmallInt	Record|Next
MenuRecordPost	SmallInt	Record|Post/Keep Locked
MenuRecordPrevious	SmallInt	Record|Previous
MenuReportAddBand	SmallInt	Report|Add Group Band
MenuReportNew	SmallInt	File|New|Report
MenuReportOpen	SmallInt	File|Open|Report
MenuReportPageFirst	SmallInt	Report|Page|First
MenuReportPageGoto	SmallInt	Report|Page|Go To
MenuReportPageLast	SmallInt	Report|Page|Last
MenuReportPageNext	SmallInt	Report|Page|Next
MenuReportPagePrevious	SmallInt	Report|Page|Previous
MenuReportPrintDesign	SmallInt	File|Print|Design
MenuReportRestartOpts	SmallInt	Report|Restart Options
MenuRotatePage	SmallInt	Report|Rotate Pages and Form|Rotate Pages
MenuSave	SmallInt	File|Save
MenuSaveAs	SmallInt	File|Save As...
MenuSaveCrossTab	SmallInt	Edit|Save Crosstab, must have a defined crosstab on a runtime form.
MenuScriptNew	SmallInt	File|New|Script
MenuScriptOpen	SmallInt	File|Open|Script
MenuSearchText	SmallInt	Edit|Search Text
MenuSelectAll	SmallInt	Edit|Select All
MenuSizeMaxHeight	SmallInt	Design|Adjust Size|Maximum Height
MenuSizeMaxWidth	SmallInt	Design|Adjust Size|Maximum Width
MenuSizeMinHeight	SmallInt	Design|Adjust Size|Minimum Height

MenuSizeMinWidth	SmallInt	Design|Adjust Size|Minimum Width
MenuSpaceHorz	SmallInt	Design|Adjust Spacing|Horizontal
MenuSpaceVert	SmallInt	Design|Adjust Spacing|Vertical
MenuSQLFileNew	SmallInt	File|New|SQL File
MenuSQLFileOpen	SmallInt	File|Open|SQL File
MenuStackPages	SmallInt	Form|Tile Pages|Stack
MenuTableNew	SmallInt	File|New|Table
MenuTableOpen	SmallInt	File|Open|Table
MenuTileHorizontal	SmallInt	Form|Tile Pages|Side-by-Side
MenuTileVertical	SmallInt	Form|Tile Pages|Top and Bottom
MenuWindowArrangeIcons	SmallInt	Window|Arrange Icons
MenuWindowCascade	SmallInt	Window|Cascade
MenuWindowCloseAll	SmallInt	Window|Close All
MenuWindowTile	SmallInt	Window|Tile

MenuReason constants

Constant	Data type	Description
MenuControl	SmallInt	Triggered by choosing an item from the control menu.
MenuDesktop	SmallInt	Triggered by choosing an item from a built-in Paradox menu.
MenuNormal	SmallInt	Triggered by choosing an item from a custom ObjectPAL menu or by clicking a Toolbar button.

MouseShape constants

Constant	Data type	Description
MouseArrow	LongInt	Standard pointer arrow
MouseCross	LongInt	Pointer is a cross
MouseIBeam	LongInt	Pointer is an I-beam (text insertion cursor)
MouseSize	LongInt	Pointer is normal size
MouseSizeNWSE	LongInt	Pointer is two-headed arrow pointing Northwest-Southeast
MouseSizeNESW	LongInt	Pointer is two-headed arrow pointing Northeast-Southwest
MouseSizeWE	LongInt	Pointer is two-headed arrow pointing East-West
MouseSizeNS	LongInt	Pointer is two-headed arrow pointing North-South
MouseNo	LongInt	Pointer is the international symbol for NO
MouseHand	LongInt	Pointer is a hand
MouseHelp	LongInt	Pointer is the standard arrow and a question mark
MouseDrag	LongInt	Pointer is the standard document drag and drop
MouseUpArrow	LongInt	Pointer is an arrow pointing up
MouseWait	LongInt	Pointer is an hourglass

MoveReason constants

Constant	Data type	Description
PalMove	SmallInt	Caused by an ObjectPAL statement
RefreshMove	SmallInt	Caused when data is updated, for example, by scrolling through a table
ShutDownMove	SmallInt	Caused when the form closes
StartupMove	SmallInt	Caused when the form opens
UserMove	SmallInt	Caused by the user

PageTilingOption constants

Constant	Data type	Description
StackPages	SmallInt	Pages are "stacked" one on top of the other.
TileHorizontal	SmallInt	Pages are tiled horizontally.
TileVertical	SmallInt	Pages are tiled vertically.

PatternStyles

Constant	Data type
BricksPattern	SmallInt
CrosshatchPattern	SmallInt
DiagonalCrosshatchPattern	SmallInt
DottedLinePattern	SmallInt
EmptyPattern	SmallInt
FuzzyStripesDownPattern	SmallInt
HeavyDotPattern	SmallInt
HorizontalLinesPattern	SmallInt
LatticePattern	SmallInt
LeftDiagonalLinesPattern	SmallInt
LightDotPattern	SmallInt
MaximumDotPattern	SmallInt
MediumDotPattern	SmallInt
RightDiagonalLinesPattern	SmallInt
ScalesPattern	SmallInt
StaggeredDashesPattern	SmallInt
ThickHorizontalLinesPattern	SmallInt
ThickStripesDownPattern	SmallInt
ThickStripesUpPattern	SmallInt
ThickVerticalLinesPattern	SmallInt
VerticalLinesPattern	SmallInt
VeryHeavyDotPattern	SmallInt
WeavePattern	SmallInt
ZigZagPattern	SmallInt

PrintColor constants

Constant	Data type	Description
prnColor	LongInt	Print in color (color printers only).
prnMonochrome	LongInt	Print in monochrome.

PrintDuplex constants

Constant	Data type	Description
prnHorizontal	LongInt	Double-sided printing where the left and right edges of consecutive pages can be bound. Also called "bind on edge" printing.
prnSimplex	LongInt	Single-sided printing.
prnVertical	LongInt	Double-sided printing where the top and bottom edges of consecutive pages can be bound. Also called "bind on top" printing.

PrinterOrientation constants

Constant	Data type	Description
prnLandscape	LongInt	Landscape (long) orientation.
prnPortrait	LongInt	Portrait (tall) orientation.

PrinterSize constants

Constant	Data type	Description
prn10x14	LongInt	10 by 14 inches.
prn11x17	LongInt	11 by 17 inches.
prnA3	LongInt	A3 297 x 420 mm.
prnA4	LongInt	A4 210 x 297 mm.
prnA4Small	LongInt	A4 Small 210 x 297 mm.
prnA5	LongInt	A5 148 x 210 mm.
prnB4	LongInt	B4 250 x 354.
prnB5	LongInt	B5 182 x 257 mm.
prnCSheet	LongInt	C size sheet.
prnDSheet	LongInt	D size sheet.
prnEnv9	LongInt	Envelope #9 3 7/8 x 8 7/8 inches.
prnEnv10	LongInt	Envelope #10 4 1/8 x 9 1/2 inches.
prnEnv11	LongInt	Envelope #11 4 1/2 x 10 3/8 inches.
prnEnv12	LongInt	Envelope #12 4 3/4 x 11 inches.
prnEnv14	LongInt	Envelope #14 5 x 11 1/2 inches.
prnEnvB4	LongInt	Envelope B4 250 x 353 mm.
prnEnvB5	LongInt	Envelope B5 176 x 250 mm.
prnEnvB6	LongInt	Envelope B6 176 x 125 mm.
prnEnvC3	LongInt	Envelope C3 324 x 458 mm.
prnEnvC4	LongInt	Envelope C4 229 x 324 mm.
prnEnvC5	LongInt	Envelope C5 162 x 229 mm.
prnEnvC6	LongInt	Envelope C6 114 x 162 mm.
prnEnvC65	LongInt	Envelope C65 114 x 229 mm.
prnEnvDL	LongInt	Envelope DL 110 x 220mm.
prnEnvItaly	LongInt	Envelope 110 x 230 mm.
prnEnvMonarch	LongInt	Envelope Monarch 3.875 x 7.5 inches.
prnEnvPersonal	LongInt	6 3/4 Envelope 3 5/8 x 6 1/2 inches.
prnESheet	LongInt	E size sheet.
prnExecutive	LongInt	Executive 7 1/4 x 10 1/2 inches.
prnFanfoldLegalGerman	LongInt	German Legal Fanfold 8 1/2 x 13 inches.
prnFanfoldStandardGerman	LongInt	German Std Fanfold 8 1/2 x 12 inches.
prnFanfoldUS	LongInt	US Std Fanfold 14 7/8 x 11 inches.
prnFolio	LongInt	Folio 8 1/2 x 13 inches.
prnLedger	LongInt	Ledger 17 x 11 inches.
prnLegal	LongInt	Legal 8 1/2 x 14 inches.
prnLetter	LongInt	Letter 8 1/2 x 11 inches.
prnLetterSmall	LongInt	Letter Small 8 1/2 x 11 inches.
prnNote	LongInt	Note 8 1/2 x 11 inches.

prnQuarto	LongInt	Quarto 215 x 275 mm.
prnStatement	LongInt	Statement 5 1/2 x 8 1/2 inches.
prnTabloid	LongInt	Tabloid 11 x 17 inches.

PrintQuality constants

Constant	Data type	Description
prnDraft	LongInt	Draft quality (lowest quality, fastest print time).
prnHigh	LongInt	High quality (highest quality, slowest print time).
prnLow	LongInt	Low quality.
prnMedium	LongInt	Medium quality.

PrintSource constants

Constant	Data type	Description
prnAuto	LongInt	Paper source selected automatically.
prnCassette	LongInt	Cassette.
prnEnvelope	LongInt	Envelope, automatic feed.
prnEnvManual	LongInt	Envelope, manual feed.
prnLargeCapacity	LongInt	Large capacity paper source.
prnLargeFmt	LongInt	Large format paper source.
prnLower	LongInt	Lower paper tray.
prnManual	LongInt	Manual feed.
prnMiddle	LongInt	Middle paper tray.
prnOnlyOne	LongInt	Single paper tray.
prnSmallFmt	LongInt	Small format paper source.
prnTractor	LongInt	Tractor feed paper.
prnUpper	LongInt	Upper paper tray.

QueryRestartOption constants

Constant	Data type	Description
QueryDefault	SmallInt	Use the options specified interactively using the Query Restart Options dialog box.
QueryLock	SmallInt	Lock all other users out of the tables needed while the query is running. If Paradox cannot lock a table, it does not run the query. This is the least polite to other users. And you must wait until all the locks can be secured before the query will run.
QueryNoLock	SmallInt	Run the query even if someone changes the data while it's running.
QueryRestart	SmallInt	Start the query over. Specify QueryRestart when you want to make sure you get a snapshot of the data as it existed at some instant. Another user might change the data after the query is completed but before the Answer table is displayed, but at least you got a snapshot. This is just the nature of multi-user work.

RasterOperation constants

Constant	Data type	Description
MergePaint	LongInt	Inverts the source graphic and combines it with the destination using the Boolean OR operator.
NotSourceCopy	LongInt	Inverts the source graphic and copies it to the destination.
NotSourceErase	LongInt	Combines the source graphic and the destination and inverts the result using the Boolean OR operator.
SourceAnd	LongInt	Combines the source graphic and the destination using the Boolean AND operator.
SourceCopy	LongInt	Copies an unchanged source graphic to the destination.
SourceErase	LongInt	Inverts the destination and combines it with the source graphic using the Boolean AND operator.
SourceInvert	LongInt	Combines the source graphic and the destination using the Boolean XOR operator.
SourcePaint	LongInt	Combines the source graphic and the destination using the Boolean OR operator.

ReportOrientation constants

Constant	Data type	Description
PrintDefault	SmallInt	Use the current Windows default orientation.
PrintLandscape	SmallInt	Use landscape (long) orientation.
PrintPortrait	SmallInt	Use portrait (tall) orientation.

ReportPrintPanel constants

Constant	Data type	Description
PrintClipToWidth	SmallInt	Clips (trims) all data that does not fit across the page (within the margins).
PrintHorizontalPanel	SmallInt	Prints additional pages as needed to fit all the data. Each of these pages immediately follows the page it extends.
PrintOverflowPages	SmallInt	Same as PrintHorizontalPanel.
PrintVerticalPanel	SmallInt	Creates a secondary page for each page of the report, even if it doesn't overflow.

ReportPrintRestart constants

Constant	Data type	Description
PrintFromCopy	SmallInt	Prints the report from copies of the tables in the report's data model.
PrintLock	SmallInt	Locks tables in the report's data model before printing.
PrintNoLock	SmallInt	Prints without locking tables in the report's table model.
PrintRestart	SmallInt	Restarts print job when data changes in tables in the report's data model.
PrintReturn	SmallInt	Cancel the print job when data changes in tables in the report's data model.

SpecialFieldType constants

Constant	Data type	Description
DateField	SmallInt	Displays the current system date.
NofFieldsField	SmallInt	Displays the number of fields in the current table.
NofPagesField	SmallInt	Displays the number of pages in the current form or report.
NofRecsField	SmallInt	Displays the number of records in the current table.
PageNumField	SmallInt	Displays the current page number.
RecordNoField	SmallInt	Displays the current record number.
TableNameField	SmallInt	Displays the name of the current table.
TimeField	SmallInt	Displays the current system time.

StatusReason constants

Constant	Data type	Description
ModeWindow1	SmallInt	The status bar area second from the left.
ModeWindow2	SmallInt	The status bar area third from the left.
ModeWindow3	SmallInt	The rightmost status bar area.
StatusWindow	SmallInt	The leftmost (and largest) status bar area.

TableFrameStyle constants

Constant	Data type	Description
tf3D	SmallInt	Table frame has a 3D frame.
tfDoubleLine	SmallInt	Table frame has a double-box frame.
tfNoGrid	SmallInt	Table frame has no grid.
tfSingleLine	SmallInt	Table frame has a box frame.
tfTripleLine	SmallInt	Table frame has a triple-box frame.

TextAlignment constants

Constant	Data type	Description
TextAlignBottom	SmallInt	Bottom of text is aligned (table window only).
TextAlignCenter	SmallInt	Text is centered horizontally.
TextAlignJustify	SmallInt	Text is justified right and left (does not apply to table window).
TextAlignLeft	SmallInt	Text is left-justified.
TextAlignRight	SmallInt	Text is right-justified.
TextAlignTop	SmallInt	Top of text is aligned (table window only).
TextAlignVCenter	SmallInt	Text is centered vertically (table window only).

TextDesignSizing constants

Constant	Data type	Description
TextFixedSized	SmallInt	Text box does not change size.
TextGrowOnly	SmallInt	Text box grows to accommodate text.
TextSizeToFit	SmallInt	Text box grows or shrinks as necessary to accommodate text.

TextSpacing constants

Constant	Data type	Description
TextDoubleSpacing	SmallInt	2 lines.
TextDoubleSpacing2	SmallInt	2.5 lines.
TextSingleSpacing	SmallInt	1 line.
TextSingleSpacing2	SmallInt	1.5 lines.
TextTripleSpacing	SmallInt	3 lines.

ToolbarBitmap constants

Constant	Data type	Description
BitmapAddBand	SmallInt	System bitmap
BitmapAddTable	SmallInt	System bitmap
BitmapAddToCat	SmallInt	System bitmap
BitmapAlignBottom	SmallInt	System bitmap
BitmapAlignCenter	SmallInt	System bitmap
BitmapAlignLeft	SmallInt	System bitmap
BitmapAlignMiddle	SmallInt	System bitmap
BitmapAlignRight	SmallInt	System bitmap
BitmapAlignTop	SmallInt	System bitmap
BitmapBookTool	SmallInt	System bitmap
BitmapBoxTool	SmallInt	System bitmap
BitmapBringToFront	SmallInt	System bitmap
BitmapButtonTool	SmallInt	System bitmap
BitmapCancel	SmallInt	System bitmap
BitmapChartTool	SmallInt	System bitmap
BitmapChkSyntax	SmallInt	System bitmap
BitmapCoEdit	SmallInt	System bitmap
BitmapCompile	SmallInt	System bitmap
BitmapDataBegin	SmallInt	System bitmap
BitmapDataEnd	SmallInt	System bitmap
BitmapDataModel	SmallInt	System bitmap
BitmapDataNextRecord	SmallInt	System bitmap
BitmapDataNextSet	SmallInt	System bitmap
BitmapDataPriorRecord	SmallInt	System bitmap
BitmapDataPriorSet	SmallInt	System bitmap
BitmapDelTable	SmallInt	System bitmap
BitmapDesignMode	SmallInt	System bitmap
BitmapDoJoin	SmallInt	System bitmap
BitmapDuplicate	SmallInt	System bitmap
BitmapEditAnswer	SmallInt	System bitmap
BitmapEditCopy	SmallInt	System bitmap
BitmapEditCut	SmallInt	System bitmap
BitmapEditPaste	SmallInt	System bitmap
BitmapEllipseTool	SmallInt	System bitmap
BitmapFieldTool	SmallInt	System bitmap
BitmapFilter	SmallInt	System bitmap
BitmapFirstPage	SmallInt	System bitmap
BitmapFldView	SmallInt	System bitmap

BitmapFontAttribBold	SmallInt	System bitmap
BitmapFontAttribItalic	SmallInt	System bitmap
BitmapFontAttribStrikeout	SmallInt	System bitmap
BitmapFontAttribUnderline	SmallInt	System bitmap
BitmapGotoPage	SmallInt	System bitmap
BitmapGraphicTool	SmallInt	System bitmap
BitmapGroup	SmallInt	System bitmap
BitmapHelp	SmallInt	System bitmap
BitmapHSpacing	SmallInt	System bitmap
BitmapLastPage	SmallInt	System bitmap
BitmapLineSpace1	SmallInt	System bitmap
BitmapLineSpace15	SmallInt	System bitmap
BitmapLineSpace2	SmallInt	System bitmap
BitmapLineSpace25	SmallInt	System bitmap
BitmapLineSpace3	SmallInt	System bitmap
BitmapLineSpace35	SmallInt	System bitmap
BitmapLineTool	SmallInt	System bitmap
BitmapLinkDm	SmallInt	System bitmap
BitmapLoadDm	SmallInt	System bitmap
BitmapMaxHeight	SmallInt	System bitmap
BitmapMaxWidth	SmallInt	System bitmap
BitmapMinHeight	SmallInt	System bitmap
BitmapMinWidth	SmallInt	System bitmap
BitmapNextPage	SmallInt	System bitmap
BitmapNextWarn	SmallInt	System bitmap
BitmapObjectTree	SmallInt	System bitmap
BitmapOk	SmallInt	System bitmap
BitmapOleTool	SmallInt	System bitmap
BitmapOpenExpert	SmallInt	System bitmap
BitmapOpenForm	SmallInt	System bitmap
BitmapOpenLibrary	SmallInt	System bitmap
BitmapOpenProject	SmallInt	System bitmap
BitmapOpenQbe	SmallInt	System bitmap
BitmapOpenReport	SmallInt	System bitmap
BitmapOpenScript	SmallInt	System bitmap
BitmapOpenSql	SmallInt	System bitmap
BitmapOpenTable	SmallInt	System bitmap
BitmapOpenTutor	SmallInt	System bitmap
BitmapPageBreak	SmallInt	System bitmap
BitmapPickTool	SmallInt	System bitmap

BitmapPrevPage	SmallInt	System bitmap
BitmapPrint	SmallInt	System bitmap
BitmapQuickForm	SmallInt	System bitmap
BitmapQuickGraph	SmallInt	System bitmap
BitmapQuickReport	SmallInt	System bitmap
BitmapQuickXTab	SmallInt	System bitmap
BitmapRecordTool	SmallInt	System bitmap
BitmapRemoveFromCat	SmallInt	System bitmap
BitmapRestructure	SmallInt	System bitmap
BitmapRun	SmallInt	System bitmap
BitmapSave	SmallInt	System bitmap
BitmapSaveDm	SmallInt	System bitmap
BitmapSendToBack	SmallInt	System bitmap
BitmapSetBreak	SmallInt	System bitmap
BitmapSetOrgin	SmallInt	System bitmap
BitmapSetWatch	SmallInt	System bitmap
BitmapShowSQL	SmallInt	System bitmap
BitmapSortAnswer	SmallInt	System bitmap
BitmapSpeedExit	SmallInt	System bitmap
BitmapSrchNext	SmallInt	System bitmap
BitmapSrchValue	SmallInt	System bitmap
BitmapStepInto	SmallInt	System bitmap
BitmapStepOver	SmallInt	System bitmap
BitmapStop	SmallInt	System bitmap
BitmapTableFrameTool	SmallInt	System bitmap
BitmapTButton	SmallInt	System bitmap
BitmapTComboBox	SmallInt	System bitmap
BitmapTextCenter	SmallInt	System bitmap
BitmapTextJustify	SmallInt	System bitmap
BitmapTextLeft	SmallInt	System bitmap
BitmapTextRight	SmallInt	System bitmap
BitmapTextTool	SmallInt	System bitmap
BitmapTGuage	SmallInt	System bitmap
BitmapTHeader	SmallInt	System bitmap
BitmapTListBox	SmallInt	System bitmap
BitmapTSpinEdit	SmallInt	System bitmap
BitmapViewBreak	SmallInt	System bitmap
BitmapViewCallStack	SmallInt	System bitmap
BitmapViewDebugger	SmallInt	System bitmap
BitmapViewMethods	SmallInt	System bitmap

BitmapViewSource	SmallInt	System bitmap
BitmapViewTracer	SmallInt	System bitmap
BitmapViewTypes	SmallInt	System bitmap
BitmapViewWatch	SmallInt	System bitmap
BitmapVSpacing	SmallInt	System bitmap
BitmapXtabTool	SmallInt	System bitmap

ToolbarButtonType constants

Constant	Data type	Description
ToolbarButtonPush	SmallInt	Specifies a pushbutton type toolbar button.
ToolbarButtonRadio	SmallInt	Specifies a radiobutton type toolbar button.
ToolbarButtonRepeat	SmallInt	Specifies a repeating pushbutton type toolbar button.
ToolbarButtonToggle	SmallInt	Specifies a toggle-action toolbar button

ToolbarClusterID constants

A cluster is a logical aggregation of buttons. There are 13 clusters in the system. Each cluster is always at the same position. The position of the cluster is expressed in 'Button widths' on a horizontal Toolbar. For example, the Mode cluster starts at a distance of 4 button widths from the left.

Constant	Data type	Description
ToolbarFileCluster	SmallInt	Specifies Toolbar cluster 0 (position 0)
ToolbarEditCluster	SmallInt	Specifies Toolbar cluster 1 (position 1, 2, 3)
ToolbarModeCluster	SmallInt	Specifies Toolbar cluster 2 (position 4, 5)
ToolbarToolCluster	SmallInt	Specifies Toolbar cluster 3 (position 6, 7)
ToolbarVCRCluster	SmallInt	Specifies Toolbar cluster 4 (position 8, 9)
ToolbarInterCluster	SmallInt	Specifies Toolbar cluster 5 (position 10, 11, 12, 13)
ToolbarInter2Cluster	SmallInt	Specifies Toolbar cluster 6 (position 14)
ToolbarQuickCluster	SmallInt	Specifies Toolbar cluster 7 (position 15, 16, 17)
ToolbarMiscCluster	SmallInt	Specifies Toolbar cluster 8 (position 18, 19)
ToolbarMisc2Cluster	SmallInt	Specifies Toolbar cluster 9 (position 20, 21)
ToolbarObjectCluster	SmallInt	Specifies Toolbar cluster 10 (position 22)
ToolbarProjectCluster	SmallInt	Specifies Toolbar cluster 11 (position 23)
ToolbarExpertCluster	SmallInt	Specifies Toolbar cluster 12 (position 24)

ToolbarState constants

Constant	Data type	Description
ToolbarStateBottom	SmallInt	Specifies a Toolbar docked at screen bottom.
ToolbarStateFloatHorizontal	SmallInt	Specifies a floating horizontal Toolbar.
ToolbarStateFloatVertical	SmallInt	Specifies a floating vertical Toolbar.
ToolbarStateLeft	SmallInt	Specifies a Toolbar docked at screen left.
ToolbarStateRight	SmallInt	Specifies a Toolbar docked at screen right.
ToolbarStateTop	SmallInt	Specifies a Toolbar docked at screen top.

UIObjectType constants

Constant	Data type	Description
BandTool (5.0)	SmallInt	Creates a report band.
BoxTool	SmallInt	Creates a box.
ButtonTool	SmallInt	Creates a button.
ChartTool	SmallInt	Creates a chart.
EllipseTool	SmallInt	Creates an ellipse.
FieldTool	SmallInt	Creates a field.
GraphicTool	SmallInt	Creates a graphic object.
LineTool	SmallInt	Creates a line.
OleTool	SmallInt	Creates an OLE object.
NoteBookTool (7)	SmallInt	Creates a tabbed notebook object.
PageBrkTool (5.0)	SmallInt	Creates a page break in a report.
RecordTool	SmallInt	Creates a record.
TableFrameTool	SmallInt	Creates a table frame.
TextTool	SmallInt	Creates a text box.
XtabTool	SmallInt	Creates a crosstab object.

ValueReason constants

Constant	Data type	Description
EditValue	SmallInt	The built-in **newValue** method of a radio button field, list, or drop-down edit list has been triggered (for example, by poking a radio button or choosing a list item), but the field value has not been committed (for example, by moving off the field).
FieldValue	SmallInt	A field's built-in **newValue** method has been triggered, and the value has been committed.
StartupValue	SmallInt	A field's built-in **newValue** method has been triggered because the form has opened.

WindowStyle constants

Constant	Data type	Description
WinDefaultCoordinate	LongInt	Displays a window at its default size and position.
WinStyleBorder	LongInt	Specifies a sizing border.
WinStyleControlMenu	LongInt	Specifies a system-control menu.
WinStyleDefault	LongInt	Specifies default displays attributes.
WinStyleDialog	LongInt	Specifies dialog box attributes.
WinStyleDialogFrame	LongInt	Specifies a dialog box frame.
WinStyleHScroll	LongInt	Specifies a horizontal scroll bar.
WinStyleHidden	LongInt	Makes a window invisible.
WinStyleMaximize	LongInt	Displays a window at full size.
WinStyleMaximizeButton	LongInt	Specifies a maximize button.
WinStyleMinimize	LongInt	Displays a window as an icon (minimized).
WinStyleMinimizeButton	LongInt	Specifies a minimize button.
WinStyleModal	LongInt	Makes a window modal.
WinStyleThickFrame	LongInt	Specifies a thick frame.
WinStyleTitleBar	LongInt	Specifies a title bar.
WinStyleVScroll	LongInt	Specifies a vertical scroll bar.

Basic language elements

See also

Basic language elements are the fundamental structural elements of ObjectPAL. Most of these elements are not bound to specific object types; they work for all object types. You can use these elements to assign values, call functions from DLLs, and to build control structures like **if...then...else...endIf** loops, **while...endWhile** loops, and **switch...case...endSwitch** structures. You can also declare methods, procedures, constants, variables, and data types.

; (comments)	method
{} (comments)	passEvent
= (equals)	proc
= (assignment)	quitLoop
const	return
disableDefault	scan
doDefault	switch
enableDefault	try
for	type
forEach	uses
if	var
iif	while
loop	

▪

; (comments) keyword

See also Example Basic Language Elements

Designates the beginning of a comment, which is text that is ignored by the compiler. The comment extends from the comment operator (;) to the end of the current line.

Syntax

```
; Comments
```

Description

Comments are useful for documenting code, which is a good programming practice.

▪

; (comments) example

```
;The following demonstrates the comment operator (;):
var
   x AnyType; declares the variable x of AnyType
endvar
x = 25      ; x gets a value of 25
; Comments that begin with the comment operator (;) extend only to
; the end of the current line.
```

▪

{ } (comments) keyword

See also Example Basic Language Elements

Designates a comment, which is text that is ignored by the compiler. The comment extends from the open brace { to the closing brace }; it does not end at the end of the line

Syntax

```
{Comments ...
...More Comments}
```

Description

Comments are useful for documenting code, which is a good programming practice.

{ } (comments) example

```
;The following demonstrates the comment braces { } operator:
var
   x AnyType {declares the variable x of AnyType}
endvar
x = 25       {x gets a value of 25}
{Comments that begin with the comment braces operator extend from the opening
brace to the closing brace, regardless of the number of lines occupied.}
```

▪

= (Assignment operator & Comparison operator) keyword

See also Example Basic Language Elements

Syntax

itemSpec **=** *expression*

Description

In an expression, the = is a comparison operator that tests whether the two operands are equal.

Otherwise, the **=** operator assigns the value of *expression* to *itemSpec*. Any previous value stored in *itemSpec* is lost. When assigning a value to an object, information in *itemSpec* can include the containership path.

When you use = with UIObjects, you assign the value of one UIObject to another UIObject. For example, suppose a form contains two fields, *fieldOne* and *fieldTwo*. The following statement copies the value of *fieldTwo* into *fieldOne*.

```
fieldOne = fieldTwo ; fieldOne gets the value of fieldTwo
```

You can also use = with UIObject variables. ObjectPAL uses **attach** the way C and Pascal use pointers. For example,

```
var ui UIObject endVar
ui.attach(fieldOne) ; tells ui to "point to" fieldOne
ui.view() ; displays the value of ui (same as fieldOne) in a dialog box.
ui = fieldTwo ; ui gets the value of fieldTwo (fieldOne value changes, too)
ui.view() ; displays the value of ui (same as fieldTwo) in a dialog box
ui.color = Red ; sets the color of ui and therefore of fieldOne to red
```

The following statement assigns to *ui* everything ObjectPAL knows about *fieldOne*:

```
ui.attach(fieldOne)
```

In contrast, the following statement assigns to *ui* (and to *fieldOne*) only the value of *fieldTwo*:

```
ui = fieldTwo
```

▪

= example

```
var
   x AnyType
   ar Array[5] AnyType
   w Logical
   y, z SmallInt
   fred, sam UIObject
endVar
x = 5.14                ; x gets a value of 5.14 (the data type is Number)
ar[1] = "Hello"         ; element 1 of ar gets the value of "Hello" (String)
y = 5                   ; y gets the value of 5
z = 12                  ; z gets the value of 12
x ="foo"                ; x gets a new value: the String "foo"
myTable.myField = y + z ; the field myField gets the value of y + z
amountField = tempAmountField
bigBox.bigCircle.smallBox.smallCircle.color = Blue
; the color property of smallCircle gets the value of Blue
; the first = assigns a value, all others compare
w = (y = z) ; w gets a value of True if y = z,
            ; otherwise, w gets a value of False
fred.attach(fieldOne) ; makes fred a "pointer" to fieldOne
sam = fred ; assigns the value of fred to sam
```

▪

const keyword

See also Example Basic Language Elements

Declares constants.

Syntax

```
const
   constName = { dataType ( value )|value }
endConst
```

Description

const declares one or more constant values, where *dataType*, if included, specifies the data type of the constant. If *dataType* is omitted, the data type is inferred from *value* as either a LongInt, a Number, a SmallInt, or a String.

Note: You declare constants in a **const...endConst** block in ObjectPAL code, or in the Const window in The Object Explorer.

▪

const example

```
const
   a = -1000                         ; SmallInt, inferred
   x = 123.45                        ; Number, inferred
   newYear = Date ("01/01/99")       ; Date, assigned
   companyName = String ("Borland")  ; String, assigned
endconst
```

disableDefault keyword

See also Example Basic Language Elements

Disables the default code for a built-in event method.

Syntax

```
disableDefault
```

Description

disableDefault prevents an event's built-in code from executing for the current call to a built-in event method. Normally, the built-in code executes implicitly at the end of a method, just before the **endMethod** keyword. Using **disableDefault** in a method disables the implicit call to the built-in code.

▪

disableDefault example

The following example sets the value of a field to "hello" when the user types a character. The call to **disableDefault** prevents the built-in code from executing, so the character does not display in the field. The **message** statement displays the character in the status bar.

```
method keyChar(var eventInfo KeyEvent)
   self.value = "hello"      ; hello appears in the field
   disableDefault            ; disable the built-in code
   message(eventInfo.char()) ; displays the character in the status bar
endMethod
```

▪

doDefault keyword

See also Example1 Example2 Basic Language Elements

Executes the default code for a built-in event method.

Syntax

`doDefault`

Description

doDefault executes the built-in code for an event immediately, instead of at the end of the method. Using **doDefault** in a method disables the implicit call to the built-in code. If a method contains more than one **doDefault** statement, only the first one executes; others are ignored.

As a general rule, if you attach code to an object's built-in **open** method, you should call **doDefault** before calling any other method or procedure. The call to **doDefault** executes the built-in code so you can be sure the object is completely opened and initialized.

doDefault example 1

The following example demonstrates the effect of a call to **doDefault**. In the following method, the button pushes in, waits two seconds, then the system beeps and the button pops out. The built-in code is called implicitly, just before the **endMethod** statement:

```
method pushButton(var eventInfo Event)
   sleep(2000)
   beep()
endMethod
```

In the following method, the call to **doDefault** makes the button pop out before it sleeps and beeps, and it disables the implicit code at the end of the method:

```
method pushButton(var eventInfo Event)
   doDefault
   sleep(2000)
   beep()
endMethod
```

▪

doDefault example 2

The following example shows how to call **doDefault** when you attach code to an object's built-in **open** method. The following code is attached to the built-in **open** method of an unbound field object named *greetingFld*. The code calls **doDefault** to execute the built-in code, then sets the value of the field object.

```
greetingFld::open
method open(var eventInfo Event)
   doDefault
   self.Value = "Hello " + getNetUserName()
endMethod
```

▪

enableDefault keyword

See also Example Basic Language Elements

Enables the default code for a built-in event method.

Syntax

`enableDefault`

Description

enableDefault allows the built-in code to execute normally at the end of a method, just before the **endMethod** statement. Compare **enableDefault** to **doDefault**, which executes the built-in code immediately.

▪

enableDefault example

```
method menuAction(var eventInfo MenuEvent)

var theChoice String endVar
disableDefault
theChoice = eventInfo.menuChoice()
switch
   case theChoice = "Open" : doOpen()
   case theChoice = "Quit" : doQuit()
   otherwise               : enableDefault
endSwitch
endMethod
```

▪

for keyword

See also   Example   Basic Language Elements

Executes a sequence of statements a specified number of times.

Syntax

```
for counter [ from startVal  [ to endVal ] [ step stepVal ]
       Statements
endFor
```

Description

for executes a sequence of *Statements* as many times as is specified by a counter, which is stored in *counter* and controlled by the optional **from**, **to**, and **step** keywords. Any combination of these can be used to specify the number of times the statements in the loop are executed. You don't have to declare *counter* explicitly, but a **for** loop runs faster if you do.

The arguments *startVal*, *endVal*, and *stepVal* are values or expressions representing the beginning counter value, ending counter value, and the number by which to increment the counter each time through the loop. These values can be any data type represented by AnyType, except Point, Memo, Graphic, String, OLE, and Binary. Also, *counter* must be a literal value or a single-valued variable; it can't be an array element or record field value.

You can use **for** without the **from**, **to**, and **step** keywords:

- If *startVal* is omitted, the counter starts at the current value of counter.
- If *endVal* is omitted, the **for** loop executes indefinitely.
- If *stepVal* is omitted, the counter increments by 1 each time through the loop.
- *startVal*, *endVal*, and *stepVal* are stored in a temporary buffer; they are not evaluated each time through the loop.

If **quitLoop** is used within the body of statements in the **for** loop, the **for ...endFor** loop is exited. If **loop** is used within the body of statements, statements following **loop** are skipped, the counter is incremented, and iteration continues from the top of the **for** loop.

If **step** is positive and a **to** clause is present, iteration continues as long as the value of *counter* is less than or equal to the value of *endVal*. If **step** is negative, iteration will continue as long as the value of *counter* is greater than or equal to the value of *endVal*. In either case, once the value of *counter* reaches or exceeds the limit set by **step**, the **for** loop stops executing, but *counter* keeps its value, as shown in the example.

If *counter* has not previously been assigned a value, **from** creates the variable and assigns to it the value of *startVal*.

▪

for example

Following is a simple **for** loop. Notice the value of the counter variable *i* after the **for** loop is completed:

```
var i SmallInt endVar
for i from 1 to 3
   i.view("Inside for loop") ; i = 1, i = 2, i = 3
endFor
i.view("Outside for loop")   ; i = 4
```

▪

forEach keyword

See also Example Basic Language Elements

Repeats the specified statement sequence over elements within a DynArray.

Syntax

```
forEach VarName in DynArrayName
      Statements
endForEach
```

Description

forEach steps through the elements in a DynArray. The argument *VarName* is a String variable used as a placeholder for the DynArray indexes. The argument *DynArrayName* is a DynArray variable that identifies the DynArray to step through. If *DynArrayName* does not exist, the **forEach** statement causes an error when the method is compiled. The *Statements* clause represents one or more ObjectPAL statements to execute for each index in the DynArray.

In general, you cannot use the **for** statement to step through a DynArray because the indexes of a DynArray are not necessarily integers. Because DynArray indexes are not integers, DynArray elements are not ordered sequentially. The **forEach** statement operates on DynArray elements in an arbitrary order. You should not rely on a specific ordering of indexes.

If the **quitLoop** statement is used within the body of statements in the **forEach** loop, the **forEach...endForEach** loop is exited. If the loop statement is used within the body of *Statements*, the statements following **loop** are skipped and iteration continues from the top of the **forEach** loop.

Do not call **removeItem** or **empty** to modify a DynArray in a **forEach** loop.

▪

forEach example

The following example uses the **forEach** statement to display the elements in the DynArray created by the **sysInfo** statement:

```
var
   SystemArray DynArray[] AnyType
   Element AnyType
endVar
sysInfo(SystemArray)
forEach Element IN SystemArray
   message(Element, " : ", SystemArray[Element])
   sleep(1500)
endForEach
```

▪

if keyword

See also Example Basic Language Elements

Executes one of two sequences of statements, depending on the value of a logical condition.

Syntax

```
if Condition then
      Statements1
[else
      Statements2 ]
endIf
```

Description

When ObjectPAL comes to an **if** statement, it evaluates whether the *Condition* is true. If so, it executes the statements listed in *Statements1* in sequence. If not, it skips *Statements1* and, if the optional **else** keyword is present, executes the statements in *Statements2*. In either case, execution continues after the **endIf** keyword.

An **if** construction can span several lines, especially if there are many statements in *Statements1* or *Statements2*. It is good practice to indent the **then** and **else** clauses to show the flow of control:

```
if Condition then
   Statements1
else
   Statements2
endIf
The following is an example of an if statement:
if Stock < 100 then
   AddStock()        ; execute a custom method called AddStock()
   Stock = Stock + 10 ; then, add 10 to the value of Stock
endIf
```

if statements can be nested; that is, any of the statements in *Statements1* or *Statements2* can also be **if** statements. Nested **if** statements must be fully contained within the controlling **if** structure, in other words, each nested **if** statement must have an **endIf** within the nest. Each **if...endIf** set must enclose code or code and another complete **if...endIf set**:

```
if Condition then
   if Condition then
      Condition
   endIf
endIf
```

if example

The following example provides code for a nested **if** statement:

```
if skillLevel = "Beginner" then
   if skillBox.color = "Red" or skillBox.color = "Yellow" then
      skillBox.color = "Green"
   endIf
endIf
```

▪

iif keyword

See also Example Basic Language Elements

Returns one of two values depending on the value of a logical condition.

Syntax

iif (*Condition*, *ValueIfTrue*, *ValueIfFalse* **)**

Description

iif (immediate **if**) allows branching within a single statement. You can use **iif** anywhere you can use any other expression. **iif** is especially useful in calculated fields on forms or reports because **if...endIf** statements are illegal there.

-

iif example

```
a = iif(x > 1, b, c) ; if x > 1, a = b; else a = c
```

▪

loop keyword

See also Example Basic Language Elements

Passes control to the top of the nearest enclosing **for**, **forEach**, **scan**, or **while** loop.

Syntax

```
loop
```

Description

When executed within a for, **forEach**, **scan**, or **while** structure, **loop** skips the statements between it and the **endFor**, **endForEach**, **endScan**, or **endWhile** and returns to the beginning of the structure. Otherwise, **loop** causes an error.

loop example

```
var x SmallInt endVar

for x from 1
   if x <> 5 then
      loop ; go back to for statement, get next value of x
      message("This never appears") ; this statement never executes
   else
      quitLoop                       ; break out of the loop
   endIf
endFor
message(x) ; displays 5
```

▪

method keyword

See also Example Basic Language Elements

Defines an ObjectPAL method.

Syntax

```
method Name ( parameterDesc [ , parameterDesc ] * ] ) [ returnType ]
[ const section ]
[ type section ]
[ var section ]
Statements
endMethod
```

Description

method marks the beginning of a method. At a minimum, you must provide the following:

- The method name, in *Name*
- Parentheses, even if the method has no arguments
- The *Statements* that comprise the method

The definition ends with the mandatory **endMethod** keyword.

Additionally, you can declare constants, data types, variables and procedures before the **method** keyword, and you can declare variables and constants after **method**.

Also optional are one or more parameter descriptions (up to a maximum of 29), represented in the prototype by *parameterDesc*, where each description takes the form

[**var|const**] ***parameter type***

The optional *returnType* declares the data type of the value returned by the method. *returnType* is optional because a method may or may not return a value. However, if the method returns a value, you must specify the data type of the value.

Methods and procedures are similar. The key differences are:

- Methods are visible and exportable to other objects, while procedures are private within a containership hierarchy.
- A method can contain a procedure definition, but a procedure can't contain a method definition.

Note: The scope of a method depends on where it is declared.

▪

method example

```
method pushButton (var eventInfo Event)
   var
      txt String
      myNum Number
   endVar
   myNum = 123.321
   txt = String(myNum)
   msgInfo("myNum = ", txt)
endMethod
```

▪

passEvent keyword

See also Example Basic Language Elements

Passes the event to the object's container.

Syntax

```
passEvent
```

Description

passEvent passes the event packet to the object's container. Using **passEvent** in a method does not affect the implicit call to the built-in code.

passEvent example

The code in the following example is attached to a field object. It executes when the pointer is in the field object. If Shift is held down when the mouse is pressed, the code calls **disableDefault** to prevent the built-in code from executing and calls **passEvent** to send the event to the field object's container. This technique is useful when you want several objects to respond the same way to a given event.

```
method mouseDown(var eventInfo MouseEvent)
   if eventInfo.isShiftKeyDown() then
      disableDefault
      passEvent ; let container handle it
   endIf
endMethod
```

▪

proc keyword

See also Example Basic Language Elements

Defines an ObjectPAL procedure.

Syntax

```
proc ProcName ( [ parameterDesc [ , parameterDesc ] * ] ) [ returnType ]
[ const section ]
[ type section ]
[ var section ]
Statements
endProc
```

Description

proc begins the definition of a procedure. You provide the following:

- The procedure name, in *ProcName*
- Parentheses, even if the procedure has no arguments
- Zero or more parameter descriptions (up to a maximum of 29), represented in the prototype by *parameterDesc*, where each description takes the form

  [**var**|**const**] ***parameter type***

- Use *returnType* to declare the data type of the value returned by the procedure (if it returns a value)
- Sections to declare variables, constants, and types
- The *Statements* that comprise the procedure

The definition ends with the mandatory **endProc** keyword.

You can use return in the body of a procedure to return a value to the calling method or procedure.

A procedure used in an expression must return a value, such as

```
x = NumValidRecs("Orders")    ; NumValidRecs is a procedure
```

Note: You declare procedures in a **proc...endProc** block in ObjectPAL code, or in the Proc window in The Object Explorer.

Procedures and methods are similar. The key differences are:

- Methods are visible and exportable to other objects, while procedures are private within a containership hierarchy.
- A method can contain a procedure definition, but a procedure can't contain a method definition.

Note: The scope of a procedure depends on where it is declared.

■

proc example

```
proc inc (x SmallInt) SmallInt
   return x + 1
endProc
method pushButton(var eventInfo Event)
   var x SmallInt endVar
   x = 5
   x = inc(x) ; calls the procedure
   message(x) ; displays 6
endMethod
```

▪

quitLoop keyword

See also Example Basic Language Elements

Terminates the **for**, **forEach**, **scan**, or **while** loop in which it appears.

Syntax

```
quitLoop
```

Description

quitLoop exits immediately from the closest enclosing **for**, **forEach**, **scan**, or **while** loop. The method continues with the statement following the closest **endFor**, **endForEach**, **endScan**, or **endWhile**.

quitLoop causes an error if executed outside of any **for**, **scan**, or **while** structure.

▪

quitLoop example

In the following example, **quitLoop** is used in a **for** loop that determines whether an array has any unassigned elements:

```
var
   myArray Array[12]
   notAssigned Logical
endVar
notAssigned = False
for i from 1 to myArray.length()
   if not isAssigned(myArray[i]) then
      notAssigned = True
      quitLoop
   endIf
endFor
```

▪

return keyword

See also Example Basic Language Elements

Returns control from a method or procedure, optionally passing back a value.

Syntax

```
return [ Expression ]
```

Description

return is used to return control from the current procedure or method to the procedure or method that called it, whether or not the method or procedure is declared to return a value. The following apply to **return**:

- If **return** is executed within the body of a procedure, the procedure is exited.
- If **return** is executed within a method (but outside the body of a procedure), the method is exited.

You can optionally return the value of *Expression* when returning from either a procedure or a method. If a procedure is called in an expression, then the procedure must return a value, which becomes the value of the procedure call.

```
y = myProc(x) + 3       ; myProc is a procedure
```

If a procedure is called in a standalone context, then any returned value is ignored. For example:

```
myProc(x)
```

If no *Expression* is supplied, **return** must not be followed by anything else on the line other than a comment.

The following data types *cannot* be returned: DDE, Database, Query, Session, Table, or TCursor.

It is not necessary to use **return** to pass control back to a higher-level method or procedure, since this happens automatically when a lower-level method or procedure finishes. However, if the method or procedure is declared to return a value, you must use **return** to return the value; the value won't be returned automatically.

▪

return example

This example adds 1 to the value of a variable and returns the new value to the calling method:

```
proc addOne (x SmallInt) SmallInt
   return x + 1
endProc
```

In a built-in event method, a **return** statement executes the built-in code unless you explicitly disable it. For example, the following code calls **return** when the user types a "?" into a field object. The call to **disableDefault** prevents the built-in code from displaying the "?" in the field object.

```
method keyChar(var eventInfo KeyEvent)
   if eventInfo.char() = "?" then
      disableDefault
      return
   endIf
endMethod
```

▪

scan keyword

[See also](#) [Example](#) [Basic Language Elements](#)

Scans the TCursor and executes ObjectPAL instructions.

Syntax

```
scan tcVar [ for booleanExpression ] :
      Statements
endScan
```

The colon is required, even if you omit the **for** keyword.

Description

scan scans *tcVar* (TCursor) and executes *Statements* (ObjectPAL instructions) for each record. **scan** always begins at the first record of the table, and steps through each record in sequence. When statements in the **scan** loop change an indexed field, that record moves to its sorted position in the table, so it's possible to encounter the same record more than once in the same loop.

If you supply the **for** clause, *Statements* execute only for those records that satisfy the condition; all others are skipped. If the table is empty or if no records meet the condition, the **scan** has no effect.

Note: The colon is required, even if you omit the **for** keyword.

scan is extremely powerful in that you can first prototype a statement sequence for a single record of a table, then place that sequence inside a **scan** loop to make it work on an entire table.

You can use **loop**, **return**, and **quitLoop** in the body of the **scan**. **loop** skips the remaining statements between it and **endScan**, moves to the next record, and returns to the top of the **scan** loop. **quitLoop** terminates the **scan** altogether, leaving the record being scanned as the current record.

Since **scan** repeats an entire statement sequence for each record, don't include actions that only need to be performed once for the table. Put those statements outside the **scan** loop. **scan** automatically moves from record to record through the table, so there's no need to call **nextRecord**.

■

scan example

The following example uses a **scan** loop to update the *Employee* table. It scans the Dept field of each record, and if the value is "Personnel", changes it to "Human Resources".

```
var
    empTC TCursor
endVar

empTC.open("employee.db") ; These statements need only be executed once,
empTC.edit()              ; so they're placed outside the loop.

scan empTC for empTC.Dept = "Personnel":  ; the colon is required
   empTC.Dept = "Human Resources"
endScan

empTC.endEdit()
empTC.close()
```

▪

switch keyword

See also Example Basic Language Elements

Executes one of a set of alternative statement sequences, depending on which of several conditions is met.

Syntax

```
switch
   CaseList
   [ otherWise: Statements ]
endSwitch
```

CaseList is any number of statements in the following form:

```
case Condition : Statements
```

Description

switch uses the values of the *Condition* statements in *CaseList* to determine which sequence of *Statements* should be executed, if any. **switch** works like multiple **if** statements, and each *CaseList* works like a single if statement.

The case *Conditions* are evaluated in the order in which they appear:

- If one has a value of True, the corresponding *Statements* sequence is executed and the rest are skipped.
- If none has the value True, and the optional **otherWise** clause is present, the *Statements* in **otherWise** are executed.
- If none has the value True and no **otherWise** clause is present, switch has no effect.

Thus, one set of *Statements* is executed at most. The method resumes with the next statement after **endSwitch**.

▪

switch example

The following example creates an array of 100 random numbers, then uses the bubble sort algorithm to sort them in numerical order:

```
method pushButton(var eventInfo Event)
var
   sz, i , itmp, j,k SmallInt
   a Array[100] SmallInt
   tmp Number
endVar

   sz = 100
   a.fill(0)

for i from 1 to sz step 1
   tmp = Rand()
   switch
         case tmp < .1 : a[i] = 1
         case tmp < .2 : a[i] = 2
         case tmp < .3 : a[i] = 3
         case tmp < .4 : a[i] = 4
         case tmp < .5 : a[i] = 5
         case tmp < .6 : a[i] = 6
         case tmp < .7 : a[i] = 7
         case tmp < .8 : a[i] = 8
         case tmp < .9 : a[i] = 9
         otherwise:      a[i] = 10
   endSwitch
endFor

for i from 1 to sz-1 step 1
   for j from 1 to sz-i step 1
      if a[j] <> a[j+1] then
         a.exchange(j, j+1)
      endIf
   endFor
endFor

endMethod
```

try keyword

See also   Example   Basic Language Elements

Marks a block of statements to try, and specifies a response should an error occur.

Syntax

```
try
   [ Statements ] ; the transaction block
onFail
   [ Statements ] ; the recovery block
   [ reTry ]        ; optional
EndTry
```

Description

The mechanism for building error recovery into an application is the **try...onFail** block.

The transaction block is a set of *Statements*, all of which you want to succeed. If the transaction succeeds, the program skips to **endTry**. If the transaction fails, the recovery block executes. You can call **reTry** to execute the transaction block again.

A trial is caused to fail by the program calling the System procedure **fail** at some point within the transaction block, or within procedures called by the transaction block. This stops system functions from returning status errors or null values to their callers.

A **fail** call can be nested in several procedure calls deep from where the block began. Their local variables are removed from the stack, and any special objects (such as large Text blocks) are deallocated. If reference objects (such as tables) are in use, they are closed, and any pending updates are canceled. It's as if the transaction had never started. What remains are changes to variables outside the block, or data added successfully to tables and committed before the failure occurred.

If during a recovery block you decide that the error code is not one you expected, or is more serious than can be handled at this level, call **fail** again to pass that error code. If no higher-level **try...onFail** block exists, the whole application fails, cancels existing actions, closes resources, and exits.

By default, a **try...onFail** block traps critical errors only. Use **errorTrapOnWarnings** if you want a **try...onFail** block to trap warnings, too. For more information about critical errors and warnings, refer to the *Guide to ObjectPAL*.

try example

The following example tries to set the Color property of some design objects, and uses a **try...onFail** block to handle the situation if the property cannot be set.

```
method pushButton(var eventInfo Event)
var s String endVar
box1.box2.color = Blue                        ; this works
s = "box5"                                    ; box5 doesn't exist

try
   box1.(s).color = Red                       ; try to set color of box5
onFail                                        ; handle the error
   msgStop("Error", "Couldn't find " + s)
   s = "box2"                                 ; box2 exists
   reTry                                      ; try again
endTry

s = "box6"                                    ; box6 doesn't exist
try
   box1.(s).color = Green
onFail
   fail(peObjectNotFound, "The object " + s + "does not exist.")
endTry
endMethod
```

▪

type keyword

See also Example Basic Language Elements

Declares data types.

Syntax

```
type
   [ newTypeName = existingType ] *
endType
```

Description

Using **type**, you can define new data types (based on existing ObjectPAL types). Once defined, you can use these types to declare variables in methods.

Note: You declare data types in a **type...endType** block in ObjectPAL code, or in the Type window in Methods page of The Object Explorer.

For example, an application to track the number of parts in a warehouse might declare a **type** *partQuantity*, then declare a variable to be of **type** *partQuantity*, like this:

```
type
   partQuantity = SmallInt ; declare a new type
endType

var                   ; use the new type to declare a variable
   pQty partQuantity ; pQty is a SmallInt
endVar                ; because partQuantity is a SmallInt
```

Later, if the number of parts approaches 32,767 (the maximum value of a SmallInt), you need only change the **type** definition, for example,

```
type
   partQuantity = LongInt ; change the declaration
endType

var                   ; use the new type to declare a variable
   pQty partQuantity ; pQty is now a LongInt
endVar                ; because partQuantity is a LongInt
```

▪

type example

A useful **type** is the **record**. Records defined in an object's type window have no connection to tables. Instead, they are similar to records in Pascal and STRUCTs in C, because they allow you to join several related elements of data together under one name. For example, the following code declares a **record** *Employee* that you can use to declare variables in methods and procedures:

```
type
   Employee = record
              LastName   String
              FirstName  String
              Title      String
              Salary     Currency
              DateHired  Date
              endRecord
endType
```

▪

uses keyword

See also Basic Language Elements Changes

Declares external ObjectPAL methods, types, and constants or Dynamic Link Library (DLL) routines to use in a method or procedure.

Syntax

Syntax for declaring external ObjectPAL methods, types and constants:

```
uses ObjectPAL
   [ "fileName"]*
endUses
```

Syntax for declaring DLL routines:

```
uses LibraryName
   [ routineName ( [parameterList] ) [returnType] [[callingConvention
["linkName"]]] ]*
enduses
```

Note: While the syntax shown above is different from the **uses** block syntax in version 5.0, any existing **uses** blocks will continue to work as they always have.

Description

The **uses** block, declared in an object's Uses window, makes methods, constants, and type definitions stored in external forms or libraries available to the object's methods and procedures. An ObjectPAL uses block is different from a DLL uses block, so they are discussed separately. A Uses window may contain multiple ObjectPAL or DLL **uses** blocks.

Changes to uses keyword

Changes for version 7

The uses keyword can now be used to specify types, methods and constants from an ObjectPAL form or library. In version 7 you can use all the the types, methods and constants in a specific library by specifying the filename of the form or library. You don't have to separately name each of the types, constants, and methods you want to use.

The syntax for specifying a DLL in a uses block now includes an optional calling convention that lets you control the type of call made to the DLL.

Note that any DLL compiled for 16-bit use (such as with Windows 3.1) will not work with Paradox 7. Paradox now requires 32-bit DLL's.

▪

ObjectPAL uses block

See also   Example1   Example2   Example3   Example4   Basic Language Elements

To use methods, constants or type definitions stored in an ObjectPAL library or attached to a form, write a **uses** block in an object's Uses window.

Syntax for specifying an ObjectPAL form or library

```
uses ObjectPAL
   ["fileName"]*
endUses
```

The keyword **ObjectPAL** is required to indicate that you are referencing ObjectPAL forms or libraries, rather than a Dynamic Link Library (DLL).

Specify the filename of each form or library name to reference. You may use an alias or path in each filename specified. Each filename must be surrounded by quotation marks and must include the file extension (.FSL or .LSL).   Each form or library that you reference must be in .FSL or .LSL form when the **uses** block is compiled.

You must open a form or library before calling a method from it; however, you may use constants and type definitions without opening the form or library.

Every form or library that you want to reference must be explicitly named in the **uses** block. You cannot, for example, have a form FORM1.FSL, with a **uses** block that references LIBRARY1.LSL, that in turn has a **uses** block that references LIBRARY2.LSL, and then use the constants, types, or method declarations defined in LIBRARY2.LSL in the code in FORM1.FSL. (In this case, you would add the **uses** block for FORM1.FSL shown below to use the constants, types, and methods from both LIBRARY1.LSL and LIBRARY2.LSL).

```
Uses ObjectPAL
   "LIBRARY1.LSL" "LIBRARY2.LSL"
endUses
```

Constants and type definitions defined in the **const** and **type** sections of a library are available for other forms, libraries, or scripts to access through a **uses** statement. All methods defined in a library are available after a library variable has been attached to the library containing the methods.

Constants and type definitions defined in the **const** and **type** sections at the *form level only* are available for other forms, libraries, or scripts to access through a **uses** statement. All methods defined on all objects of a form are available to be called, after a form variable has been attached to the form containing the methods.

Procedures and variables in external forms or libraries are not available.   If you need to access variables in libraries, use methods in the library to get and set the values of library variables, and then call those methods from your forms, libraries, or scripts to share global values.

When your code is compiled or saved, it reads the constants, types, or method declarations from the .FSL or .LSL files named in **uses** blocks. Delivered forms or libraries (.FDL and .LDL files) do not have the information required for this step, so you must have the .FSL or .LSL files available when you make changes to your code.

After you deliver your code, it will run without the .FSL or .LSL files it references. After it's saved, it will run without the .FSL or .LSL files, as long as you don't make changes to your code.

When you change constant or type information in a form or library that other forms, libraries, or scripts reference, all the forms, libraries, or scripts need to be recompiled to use the changed values.   To recompile your code, make sure you have Show Developer Menus enabled in Developer Preferences, then for each form, library, or script, open the file in Design Mode, select Program | Compile, then File | Save.

▪

ObjectPAL uses block example 1

The following example references an ObjectPAL library to calculate interest rates. The library, named MATHLIB.LSL, contains the method **calcInterest**, which takes two arguments: *intRate* and *nPeriods*. It returns the interest calculated.

The following code, attached to a button's Uses window, reads the declaration for the **calcInterest** method from MATHLIB.LSL so the button can use it.

```
uses ObjectPAL
   "mathlib.lsl"
endUses
```

The following code, attached to a button's built-in **pushButton** method, opens the library, reads the values of two fields on a tableframe, calls **calcInterest**, and then displays the results.

```
method pushButton(var eventInfo Event)
   var
      mathLib   Library
      iRate     Number
      nPeriods  SmallInt
      interest  Number
   endVar
   if mathLib.open("mathlib.lsl") then
      iRate = mortgage.intRate.value
      nPeriods = mortgage.nYears.value * 12
      interest = mathLib.calcInterest(iRate, nPeriods)
      interest.view("Interest")
   endIf
endMethod
```

In this example, dot notation specifies where to find the **calcInterest** method. The following statement says to look in the library represented by the Library variable *mathLib*.

```
interest = mathLib.calcInterest(iRate, nPeriods)
```

The concept for calling a method attached to another form is the same: use dot notation to specify the form to search for the method. The following assumes that the Form variable *codeForm* has been previously declared, the form has been opened, and the form was referenced in a **uses** block.

```
returnValue = codeForm.getObjHelp(self.name)
```

Note: With previous versions of Paradox, the uses block was used to declare external methods to call. The declarations are now read directly from the form or library that you are calling. You no longer have to maintain multiple copies of method declarations as they change, and Paradox will report parameter mismatches when you compile your code, rather than later as your code is run.

ObjectPAL uses block example 2

The following example references an ObjectPAL library named PARTS.LSL. The example shows how **uses** allows you to share constants and type declarations as well as method declarations from forms and libraries.

The library PARTS.LSL contains a **const** block, a **type** block, and a method using the constants and type definitions.

```
const
   DefaultPartName = "N/A"
   DefaultPartNumber = "000-00"
   DefaultPricePerUnit = 1.00
endConst

type
   PartRecordType = Record
      PartName      String
      PartNumber    String
      QtyOnHand     LongInt
      QtyOnOrder    LongInt
      PricePerUnit  Currency
   endRecord
endType

method NewPart(var newPartRecord PartRecordType)
   newPartRecord.PartName = DefaultPartName
   newPartRecord.PartNumber = DefaultPartNumber
   newPartRecord.QtyOnHand = 0
   newPartRecord.QtyOnOrder = 0
   newPartRecord.PricePerUnit = DefaultPricePerUnit
endMethod
```

The following code, attached to a button's Uses window, declares *DefaultPartName*, *DefaultPartNumber*, *DefaultPricePerUnit* and *PartRecordType* from the library and declares *NewPart* so the button can use them.

```
Uses ObjectPAL
   "parts.lsl"
endUses
```

The following code, attached to a button's built-in **pushButton** method, opens the library and calls the method with a *PartRecordType* variable. Note that *PartRecordType* is a type defined in the library, and declared automatically by the **uses** block.

```
method pushButton(var eventInfo Event)
   var
      partsLib    Library
      partRecord  PartRecordType
   endVar

   if partsLib.open("parts") then
      partsLib.newPart(partRecord)
   endIf
endMethod
```

▪

ObjectPAL uses block example 3

The following example references an ObjectPAL library named WINAPI.LSL. The example shows how to create a *Reference Library,* that is, a library that only gets accessed at compile-time for constant, type and method declarations. An ObjectPAL Reference Library contains no ObjectPAL code, only definitions.

Certain data structures, constants and method declarations that you develop in your Paradox applications can apply to several projects. The **uses** block allows applications to access centralized libraries that have been created solely for the purpose of defining the types, constants and method declarations used. Changes to types and constants automatically propagate to all projects referencing the information (after the projects are recompiled to include the change). An ObjectPAL Reference Library is in some ways analogous to a “header” (.H) file in the C and C++ programming languages.

The following code is attached to the Uses window in WINAPI.LSL. It declares calls made to the Windows API. These calls should not change, so it is helpful to have them defined in a single file that also does not change, where they can be referenced whenever needed.

```
Uses User32
   GetWindowText(hwin CLONG, title CPTR, nMaxLength CLONG) CLONG  [STDCALL
"GetWindowTextA"]
   GetActiveWindow() CHANDLE  [STDCALL "GetActiveWindow"]
   MessageBox(hwin CLONG, text CPTR, title CPTR, flags CLONG) CLONG  [STDCALL
"MessageBoxA"]
EndUses
```

The following code is attached to the Const window in WINAPI.LSL. It assigns a constant used in the *MessageBox* call to the Windows API.

```
Const
   MB_OK = 0
EndConst
```

The following code, attached to a **pushButton** method, calls the functions from the Windows API defined in WINAPI.LSL.

```
uses ObjectPAL
   "winapi.lsl"
enduses

method pushButton(var eventInfo Event)
var
   windowHandle   LongInt
   windowTitle   String
endvar

   windowTitle = fill(" ", 80)   ; reserve 80 characters for title
   windowHandle = GetActiveWindow()
   if GetWindowText(windowHandle, windowTitle, 80) > 0 then
      MessageBox(0, windowTitle, "Title of Active Window", MB_OK)
   endif

endmethod
```

Other objects (forms, libraries, or scripts) can also access WINAPI.LSL with a **uses** block, declaring the Windows functions in the system DLL, USER32.DLL. It is not necessary to have WINAPI.LSL around at run-time in either source (.LSL) or delivered (.LDL) form.

▪

ObjectPAL uses block example 4

The following example references an ObjectPAL library named PARTSHDR.LSL. The example shows how **uses** enables you to share constants and type declarations as well as method declarations from forms and libraries, how to use a Reference Library, and that you may need to use multiple **uses** blocks to declare all the information you need.

The library PARTSHDR.LSL contains a **const** block, and a **type** block. It defines some global constants and types that are to be used by several other forms and libraries. Paradox only needs to reference the information in PARTSHDR.LSL at compile-time, so it is a Reference Library.

```
const
   DefaultPartName = "N/A"
   DefaultPartNumber = "000-00"
   DefaultPricePerUnit = 1.00
endConst

type
   PartRecordType = Record
      PartName       String
      PartNumber       String
      QtyOnHand      LongInt
      QtyOnOrder       LongInt
      PricePerUnit   Currency
   endRecord
endType
```

The library PARTS.LSL declares the **NewPart** method. It declares constants and type declarations through a **uses** block that references PARTSHDR.LSL.

```
uses ObjectPAL
   "partshdr.lsl"
endUses

method NewPart(var newPartRecord PartRecordType)
   newPartRecord.PartName = DefaultPartName
   newPartRecord.PartNumber = DefaultPartNumber
   newPartRecord.QtyOnHand = 0
   newPartRecord.QtyOnOrder = 0
   newPartRecord.PricePerUnit = DefaultPricePerUnit
endMethod
```

The following code, attached to a button's Uses window, declares *DefaultPartName*, *DefaultPartNumber*, *DefaultPricePerUnit* and *PartRecordType* from PARTSHDR.LSL and *NewPart* from PARTS.LSL so the button can use them.

```
Uses ObjectPAL
   "partshdr.lsl" "parts.lsl"
endUses
```

Even though PARTS.LSL has a **uses** block that references PARTSHDR.LSL, the **uses** block for this button must explicitly include the reference to PARTSHDR.LSL. An indirect reference is not sufficient. Every object that needs to declare constants, type definitions, or methods from external forms or libraries must declare the forms or libraries directly in its own **uses** block, or have such a definition included in the **uses** block of one of its containers.

The following code, attached to a button's built-in **pushButton** method, opens the library and calls the method with a *PartRecordType* variable.

```
method pushButton(var eventInfo Event)
   var
      partsLib   Library
      partRecord   PartRecordType
   endVar

   if partsLib.open("parts") then
      partsLib.newPart(partRecord)
   endIf

   partRecord.view() ; display the record to show the changed values

endMethod
```

▪

DLL uses block

See also Example1 Example2 Basic Language Elements

To use routines stored in a Dynamic Link Library (DLL), write a DLL **uses** block in one of the following places:

- A design object's Uses window
- A window for a built-in method
- A window for a custom method
- A window for a custom procedure

Where you write the block depends on the desired scope (availability) of the routine.

No matter where you write it, the basic structure (shown in the following prototype) is the same:

Syntax for specifying a DLL

```
uses libraryName
[ routineName ( [parameterList] ) [returnType] [[callingConvention
["linkName"]]] ]*
endUses
```

The required elements in a DLL uses block are *libraryName* and an optional list of routines. Each routine must be specified with a *routineName* and the left and right parentheses. All other arguments are optional.

The argument *libraryName* specifies the DLL filename. Paradox assumes a file extension of DLL or EXE.

Each routine that you declare must include a *routineName*, the name you use in your ObjectPAL code to call the external routine.

The optional *parameterList* specifies zero or more argument names and data types.

If the routine returns a value, *returnType* specifies the return value's data type.

The *callingConvention* for a DLL call can be PASCAL, STDCALL, or CDECL.

The *linkName* argument is the name of the routine as it is defined in the DLL. It is dependent on the calling convention and is case-sensitive in Windows95 and Windows NT.

Windows searches for the DLL *libraryName* in this order:

1. The current directory.

2. The Windows directory. You can use the FileSystem procedure **windowsDir** to get the path to this directory (typically, it's C:\WINDOWS).

3. The Windows system directory. You can use the FileSystem procedure **windowsSystemDir** to get the path to this directory (typically, it's C:\WINDOWS\SYSTEM).

4. The directories listed in the PATH environment variable. Refer to your DOS documentation for more information.

5. The list of directories mapped in a network.

Advanced Windows programmers: If you're calling a routine from a previously-loaded DLL (for example, a DLL loaded automatically by Windows), you can use *libraryName* to specify the DLL's module name instead of the filename. Consult your programming language's documentation for more information about DLL module names.

A DLL **uses** block can contain one or more *routineNames*, and each *routineName* can have its own *parameterList*. A *parameterList* specifies zero or more argument names and data types. If the routine returns a value, the *returnType* specifies the return value's data type. ObjectPAL checks your specifications for these arguments for *exact* matches with those declared in the routine; that's all the checking it does.

The routines must fit one of the following descriptions:

- Routines written in assembly language, C, C++, or Pascal and stored in or a Windows Dynamic-link library (DLL). A DLL is a library of executable code or data that you can link to your application at runtime. Using DLLs you can add features and functions without modifying your compiled ObjectPAL application.
- Routines from the Windows API (Application Programming Interface). The Windows system is itself made up of several DLL's. You can use Paradox to access routines within the DLL's that comprise the Windows system.

Declare a **uses** block in an object's Uses window, and within that window, declare one **uses** block for each DLL you want to use. You don't have to declare every routine the DLL contains, just the ones you want to use. Once declared, routines are available to all methods attached to that object, to all objects that object contains, and to forms or libraries that reference the declarations through an ObjectPAL **uses** block.

In a **uses** block, declare the data types of parameters and return types using the following keywords:

Data type	Uses keyword	ObjectPAL type	C/C++ type	Pascal type
16-bit integer	CWORD	SmallInt	short (short int)	Integer
32-bit integer	CLONG	LongInt	long (long int)	Longint
Natural integer (*)	CLONG (*)	LongInt (*)	int	Integer
64-bit floating-point number	CDOUBLE	Number	double	Double
80-bit floating-point number	CLONGDOUBLE	Number	long double	Extended
string pointer	CPTR	String	char *	Pchar
binary or graphic data	CHANDLE	Binary, Graphic	HANDLE (Windows)	Thandle

(*) The size of a natural integer is dependent upon the compiler you use to create your DLL. With Windows 95 and Windows NT, natural integers in C and Pascal are 32-bit integers, and map into a CLONG. If your compiler uses 16-bit integers, then the arguments map into CWORD, and you must declare the arguments as CWORD.

The ObjectPAL keywords CWORD, CLONG, CDOUBLE, CLONGDOUBLE, CPTR, and CHANDLE are valid only within a DLL **uses** block. Don't use them anywhere else. They are used by Paradox to convert between the more complex (and powerful) ObjectPAL data types and the corresponding data types in C or Pascal.

Note: Do not modify any passed CPTR. If you change the contents of a string passed as a CPTR, the string must not grow beyond the size it had when it was passed to your routine.

▪

Calling External routines

See also Example uses

Previous versions of Windows (3.1 and earlier) and Paradox used the Pascal calling convention (PASCAL). Windows 95 and Windows NT use a different calling convention. Paradox supports this calling convention, STDCALL, as well as PASCAL, and the C calling convention, CDECL. Paradox defaults to STDCALL.

Convention	Push order	Restore stack	Link name	Used by
PASCAL	Left first	Callee	Uppercase	Pascal
CDECL	Right first	Caller	'_' prepended	C/C++
STDCALL	Right first	Callee	No change	Windows 95, Windows NT

When you declare routines to be called from a DLL, you must match the calling convention that the routines were declared with. All calls to functions in the Windows 95 API or the Windows NT API are case-sensitive and require the use of the STDCALL calling convention.

Calls to functions written in Pascal should be declared with the PASCAL calling convention and calls to C functions should be declared CDECL, unless the routines were explicitly declared to use a different convention when the DLL was compiled. For example, a C routine might be declared __stdcall, in which case you declare it STDCALL in the **uses** block.

If you do not include a link name in the declaration, the routine name will be used in the call, with any changes listed in the Link Name column in the table above.

When passing a value to a C procedure, the ObjectPAL variable must be declared and typed explicitly. However, AnyType is not allowed.

All C and C++ functions that you want ObjectPAL to call must be exported in the .DEF file, or tagged with _export in the function declaration.

Using C++

Calling DLL modules written in C++ requires either the use of a C linkage specification or the "mangled" name in the **uses** block.

To specify a C++ function with C linkage, the modules must be in one of the following forms:

```
extern "C" declaration
extern "C" { declarations }
```

For example, if a C module contains these functions:

```
char *SCopy(char*, char*);
void ClearScreen(void);
```

they must be declared in a C++ module in one of the following ways to have C linkage.

```
extern "C" char *SCopy(char*, char*);
extern "C" void ClearScreen(void);
```

or

```
extern "C" {
char *SCopy(char*, char*);
void ClearScreen(void);
}
```

Otherwise, you can specify the mangled name of the routine to call. The mangled name can be found by using a dumping file on the .OBJ file produced by your compiler.

For example, if a Borland C++ module (named MyLib) contains the function

```
int __cdecl MyFunction(int arg)
```

then you can use this uses block to declare the DLL routine.

```
uses MyLib
   MyFunction(CLONG arg) CLONG  [CDECL "@MyFunction$qi"]
enduses
```

All C or C++ functions that you want to call from ObjectPAL must be exported, either through use of a .DEF file, or through the use of the _export modifier. See your C or C++ compiler documentation for more information on exporting functions when creating DLLs.

▪

Passing by value

See also uses

The following table presents the syntaxes for passing various data types by value to a C procedure. ObjectPAL passes and returns floating-point values by value, as required by the Borland C++ compiler. Other C compilers may have different requirements. To ensure compatibility with any C compiler, pass values by pointer.

It is assumed that these ObjectPAL variables have been declared: si SmallInt, li LongInt, nu Number, st String, gr Graphic, and bi Binary

C data type	C syntax	In USES block	ObjectPAL call
long double	void __stdcall cproc(long double value)	cproc(numvar CLONGDOUBLE)	cproc(si) cproc(li) cproc(nu)
double	void __stdcall cproc(double value)	cproc(numvar CDOUBLE)	cproc(si) cproc(li) cproc(nu)
long int	void __stdcall cproc(long int value)	cproc(longvar CLONG)	cproc(si) cproc(li)
short int	void __stdcall cproc(short int value)	cproc(shortvar CWORD)	cproc(si)
int	void __stdcall cproc(int value)	cproc(longvar CWORD)	cproc(si)
(String)	void __stdcall cproc(char * value)	cproc(stringvar CPTR)	cproc(st)
(Graphic)	void __stdcall cproc(HANDLE value)	cproc(bitmapvar CHANDLE)	cproc(gr)
(Binary)	void __stdcall cproc(HANDLE value)	cproc(binaryvar CHANDLE)	cproc(bi)

■

Passing by pointer

See also uses

When ObjectPal passes information to a C procedure that takes pointers to information, the pointer points directly to the corresponding value in the ObjectPAL object (variables in ObjectPAL are treated as objects internally). For example, if you want an int * and you pass a LongInt, you will get a pointer that points directly to the int value inside the LongInt object. You can then modify the value of the LongInt through the pointer in your DLL. This could be an extremely dangerous operation, since you can corrupt ObjectPAL by overwriting memory (writing past the bounds of the memory pointer). Use caution when using pointers.

Use pointers to

- Change the information (this should be done by function return values if possible).
- Pass floating point values to C procedures that were not compiled using the Borland C compiler.

Different C compilers use different conventions for passing and returning floating point values (double and long double). The only way to pass compiler-independent information is by pointer.

The following table presents the syntaxes for passing various data types by pointer to a C procedure, with the assumption that these ObjectPAL variables have been declared: si SmallInt, li LongInt, nu Number, st String, gr Graphic, and bi Binary

C data type	C syntax	In USES block	ObjectPAL call
long double *	void __stdcall cproc(long double * value)	cproc(numvar CPTR)	cproc(nu)
long int *	void __stdcall cproc(long int * value)	cproc(longvar CPTR)	cproc(li)
int *	void __stdcall cproc(int * value)	cproc(longvar CPTR)	cproc(li)
short int *	void __stdcall cproc(short int * value)	cproc(shortvar CPTR)	cproc(si)
char *	void __stdcall cproc(char * value)	cproc(strvar CPTR)	cproc(st)

▪

Returning values

See also uses

The following table presents the syntaxes for returning values of various data types from a C procedure, with the assumption that these ObjectPAL variables have been declared: si SmallInt, li LongInt, nu Number, and st String

C data type	C syntax	In USES block	ObjectPAL call
long double	long double __stdcall cproc(void)	cproc() CLONGDOUBLE	nu = cproc()
double	double __stdcall cproc(void)	cproc() CDOUBLE	nu = cproc()
long int	long int __stdcall cproc(void)	cproc() CLONG	li = cproc()
short int	short int __stdcall cproc(void)	cproc() CWORD	si = cproc()
char *	char * __stdcall cproc(void)	cproc() CPTR	st = cproc()

▪

Notes on Graphic and Binary data (CHANDLE)

See also uses

Graphic and Binary data are passed via CHANDLE. In C terms this is a HANDLE typedef. A CHANDLE is a handle to Windows memory. To use such a handle, wrap it inside code like this:

```
void __stdcall cproc(HANDLE value)
{
   // declare ptr to point to Global Memory Block
   huge *ptr = (huge *) GlobalLock(value);

   // ... make use of ptr here
   // ... DO NOT use 'GlobalFree(value);'

   GlobalUnlock(value);
}
```

For a Binary variable, HANDLE is a handle to memory that directly contains the information contained in the binary BLOB. There is no "header" information. As with any strings you pass, you can read or modify the data, but you cannot change its size.

For a Graphic variable, HANDLE is a Windows Bitmap handle. Use this as you would any other bitmap HANDLE.

▪

DLL uses block example 1

This example references a DLL named MYSTUFF.DLL. To use a DLL routine in a method, declare variables to use as arguments, then call the routine. For example,

```
; this goes in an object's Uses window
uses myStuff ; reads routines from MYSTUFF.DLL
   doSomething(thisNum CLONG, thatNum CLONG) CDOUBLE ; declare a routine
endUses

; this modifies an object's mouseUp method
method mouseUp(var eventInfo MouseEvent)
var
   thisNum, thatNum LongInt ; declare variables to pass to the routine
   myResult Number
endVar

thisNum = 3,155,111
thatNum = 5,535,345
myResult = doSomething(thisNum, thatNum) ; call the routine, return a result
endMethod
```

In this example, notice how the variables in the method are declared as LongInt and Number, and so the arguments in the **uses** block are correspondingly declared as CLONG and CDOUBLE.

■

DLL uses block example 2

The following example uses routines from MINMAX.DLL, written using a Borland 32-bit Pascal compiler. The code for the DLL is as follows:

```
library MinMax;

function Min(x, y: integer): integer; stdcall; export;
begin
   if x < y then
      result := x
   else
      result := y;
end;

function Max(x, y: integer): integer; stdcall; export;
begin
   if x > y then
      result := x
   else
      result := y;
end;

exports
   Min, Max;

begin
end.
```

The following ObjectPAL code uses the routines in the DLL. The code for the Uses window appears first, followed by the code that modifies a button's **pushButton** method:

```
; the following goes in a button's Uses window
uses
   MinMax ; load routines from MINMAX.DLL
   Min (x CLONG, y CLONG) CLONG  [STDCALL]
   Max (x CLONG, y CLONG) CLONG  [STDCALL]
endUses
```

The following code modifies a button's built-in **pushButton** method:

```
method pushButton(var eventInfo Event)
var
   x, y, z  LongInt
endVar
   x = 2
   y = 6
   z = Min(x, y)          ; call Min from the DLL
   msgInfo("Min", z)
   z = Max(x, y)          ; call Max from the DLL
   msgInfo("Max", z)
endMethod
```

■

DLL uses block example 3

The following example shows how to use ObjectPAL to call a function from the Windows API. It calls the Windows API function MessageBox to display a dialog box.

The following code is attached to a button's Uses window.

```
Uses USER32     ; The MessageBox function is in
                ; the Windows system DLL USER32.DLL
                ; usually found in C:\WINDOWS\SYSTEM
   MessageBox(hWnd CHANDLE, lpText CPTR, lpCaption CPTR, wType CLONG) CLONG
endUses
```

The following code is attached to a button's built-in **pushButton** method. It calls MessageBox, passing it zero for the window handle (so that it's not connected to any particular window), text for the message and the caption, and another zero to signify an OK style message box.   The return value is ignored.

```
method pushButton(var eventInfo event)
   MessageBox(0,
              "Your message here",
              "Your caption here",
              0)
endMethod
```

See the Windows API reference for further information on the parameters for this and other Windows API function calls.

▪

var keyword

See also Example Basic Language Elements

Declares variables.

Syntax

```
var
   [ varName [ , varName ] * varType ]*
endVar
```

Description

The **var...endVar** block declares variables by associating a variable name *varName* with a data type *varType*. When you declare more than one variable of the same type on the same line, use commas to separate the names.

A variable's scope depends on the block in which it is declared.

Note: You declare variables in a **var...endVar** block in ObjectPAL code, or in the Var window in the Methods page of The Object Explorer.

▪

var example

```
var
   myChars, xx String
   myNum Number
   orders, sales, parts TCursor
   proteus AnyType
   myBox UIObject
   a, b Array[5] SmallInt
   myOtherNum Number
endVar
```

while keyword

See also Example Basic Language Elements

Repeats a sequence of statements as long as a specified condition is True.

Syntax

```
while Condition
   [ Statements ]
endWhile
```

Description

while starts by evaluating the logical expression *Condition*. If *Condition* is False, the *Statements* are not executed. If the value is True, the *Statements* between the *Condition* and **endWhile** are executed in sequence. Control then returns to the top of the loop, and the *Condition* is evaluated again. The steps are repeated until the *Condition* evaluates to False, at which point the loop is exited and control advances to the next statement after **endWhile**.

You can use **loop** within the body of the **while** to force control back to the top of the **loop**, skipping the statements between **loop** and **endWhile**. You can also use **quitLoop** to jump out of the loop altogether. You can also nest **while** statements within each other to any level.

while and **for** are similar but are generally used for different reasons. Use **for** to execute a sequence of statements a known number of times. Use **while** to execute a sequence of statements an arbitrary number of times.

▪

while example

```
; this example creates an array of last names
var
   myNames TCursor
   namesArray Array[] String
   n SmallInt
endVar

myNames.open("names.db")
namesArray.grow(1)
namesArray[1] = myNames."Last name"
n=1

while myNames.nextRec()
   n = n + 1
   namesArray.grow(1)
   namesArray[n] = myNames."Last name"
endWhile
```

Keywords

See also

The keywords in this list cannot be used to name objects, variables, arrays, methods, or procedures. The case of the words is irrelevant; they cannot be used in any combination of uppercase or lowercase.

Also, as a general rule, you should not use object type names, names of basic language elements, names of methods and procedures in the run-time library, or names of built-in event methods.

Keywords		
active	endTry	proc
and	endType	query
array	endUses	quitLoop
as	endVar	record
case	endWhile	refIntStruct
caseInsensitive	for	retry
const	forEach	return
container	from	scan
create	if	secStruct
Database	iif	self
descending	in	sort
disableDefault	index	step
doDefault	indexStruct	struct
DynArray	is	subject
else	key	switch
enableDefault	lastMouseClicked	switchMenu
endConst	lastMouseRightClicked	tag
endCreate	like	then
endFor	loop	to
endForEach	maintained	try
endIf	method	type
endIndex	not	unique
endMethod	ObjectPAL	uses
endProc	of	var
endQuery	on	where
endRecord	onFail	while
endScan	or	with
endSort	otherwise	without
endSwitch	passEvent	
endSwitchMenu	primary	

▪

Built-in object variables

See also

ObjectPAL provides built-in object variables. You can use them to refer to UIObjects. They are particularly useful for creating generalized code. For example, when the following statement executes, it sets the color of the active object (the object that has focus)▪you don't have to specify the object by name.

```
active.Color = Blue
```

The built-in object variables are

active

container

lastMouseClicked

lastMouseRightClicked

self

subject

▪

Properties and property values

See also

Choose from the following properties for more information:

Alignment
Arrived
AttachedHeader
AutoAppend
AvgCharSize
BlankRecord
Border
BottomBorder (read-only)
Breakable
ButtonType
ByRows
CalculatedField
Caption
CenterLabel
CheckedValue
Class
Color
Columnar
ColumnPosition
ColumnWidth
CompleteDisplay
ContainerName
ControlMenu
CurrentColumn
CurrentRecordMarker.Color
CurrentRecordMarker.LineStyle
CurrentRecordMarker.Show
CurrentRow
CursorColumn
CursorLine
CursorPos
DataSource
Default
DefineGroup
DeleteColumn
Deleted
DeleteWhenEmpty
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable
KeyField
LabelText
LeftBorder (read-only)
Line.Color
Line.LineStyle
Line.Thickness
LineEnds
LineSpacing
LineStyle
LineType
List.Count
List.Selection
List.Value
Locked
LookupTable
LookupType
Magnification
Manager
MarkerPos
Margins.Bottom
Margins.Left
Margins.Right
Margins.Top
MaximizeButton
Maximum
MemoView
MinimizeButton
Minimum
Modal
MouseActivate
Name
NCols
Next
NextTabStop
NoEcho
NRecords
NRows
OtherBandName
OverStrike
Owner
PageSize

Design.SizeToFit
DesignModified
DesignSizing
DesktopForm
DialogForm
DialogFrame
DisplayType
Editing
Enabled
End
FieldName
FieldNo
FieldRights
FieldSize
FieldType
FieldUnits2
FieldValid
FieldView
First
FirstRow
FitHeight
FitWidth
FlyAway
Focus
Font.Color
Font.Size
Font.Style
Font.Typeface
Format.DateFormat
Format.LogicalFormat
Format.NumberFormat
Format.TimeFormat
Format.TimeStampFormat
Frame.Color
Frame.Style
Frame.Thickness
FrameObjects
FullName
FullSize
Grid.Color
Grid.GridStyle
Grid.RecordDivider
GridLines.Color
GridLines.ColumnLines
GridLines.HeadingLines
PageTiling
Pattern.Color
Pattern.Style
PersistView
Picture (enhanced for 5.0)
Position
PositionalOrder
PrecedePageHeader
Prev
PrinterDocument
PrintOn1stPage
ProgID
Range
RasterOperation
Readonly
RecNo
Refresh
RefreshOption
RemoveGroupRepeats
RepeatHeader
Required
RightBorder (read-only)
RowHeight
RowNo
Scroll
SeeMouseMove
Select
SelectedText
SeqNo
ShowGrid
Shrinkable
Size
SizeToFit
SnapToGrid
SortOrder
SpecialField
StandardMenu
StandardToolbar
Start
StartPageNumbers
Style
SummaryModifier
TabStop
TableName
Text

GridLines.LineStyle
GridLines.QueryLook
GridLines.RowLines
GridLines.Spacing
GridValue
GroupObjects
GroupRecords
Header
HeadingHeight
Headings
HorizontalScrollBar
IndexField
InsertColumn
InsertField
Inserting
Invisible
Justification

ThickFrame
Thickness
Title
TopBorder (read-only)
TopLine
Touched
Translucent
UncheckedValue
Value
VerticalScrollBar
Visible
WideScrollBar
Width
WordWrap
Xseparation
Yseparation

Alignment property

Data type
SmallInt

Description
Specifies the position of text relative to a field object or a text object.

Values
TextAlignCenter, TextAlignJustify, TextAlignLeft, TextAlignRight

Arrived property

Data type
Logical

Description
Specifies whether the focus has arrived at the object.

Values
True, False

AttachedHeader property

Data type

Logical

Description

Determines whether a table frame's header is attached to the table frame. True = attached; False = not attached, and selection handles appear around the header object.

Values

True, False

AutoAppend property

Data type

Logical

Description

Setting an object's AutoAppend property has the same effect as right-clicking a table in the Data Model dialog box and choosing Auto-Append from its menu. This property is valid only for an object bound to a table in the data model.

By default, when you're editing data in a form and you move the cursor past the end of a table, Paradox automatically inserts a blank record (in other words, AutoAppend = True). You can prevent this by setting the object's AutoAppend property to False. When AutoAppend is set to False, you can insert a record by pressing Ins, by choosing Record|Insert, or by executing an ObjectPAL statement such as active.action(DataInsertRecord).

Values

True, False

AvgCharSize property

Data type

Point

Description

Read-only property that gives the average width and height of a character in the current font.

Values

>0

BlankRecord property

Data type

Logical

Description

Reports whether a record is blank.

Values

True, False

Border property

Data type
Logical

Description
Reports whether a form's window has a border.

Values
True, False

BottomBorder property

Data type
LongInt

Description
Returns the size of an object's bottom border (in twips). An object's border represents the area in which you cannot embed another object.

Values
N/A

Breakable property

Data type
Logical

Description
Specifies whether an object can be split across page breaks in a report. Read-only for elliptical lines.

Values
True, False

ButtonType property

Data type
SmallInt

Description
Specifies the display type of a button.

Values
CheckBoxType, PushButtonType, RadioButtonType

ByRows property

Data type
Logical

Description
Determines the record layout of a multi-record object in a form or report. If ByRows = True, record layout is top-down, then left-right. If ByRows = False, record layout is left-right, then top-down.

Values
True, False

CalculatedField property

Data type
Logical

Description
Specifies whether a field is a calculated field.

Values
True, False

Caption property

Data type
Logical

Description
Reports whether a form has a caption (title) bar.

Values
True, False

CenterLabel property

Data type
Logical

Description
Specifies whether a label is centered within a button.

Values
True, False

CheckedValue property

Data type

String

Description

Specifies the string that the checkbox or radio button will write to its parent field object when a checkbox or radio button is chosen.

Values

N/A

Class property

Data type
String

Description
Returns the class of a UIObject.

Values
Band, Bitmap, Box, Button, Cell, Chart, Crosstab, EditRegion, Ellipse, Field, Form, Group, Header, Line, List, Multirecord, OLE, Page, Record, TableFrame, Text

Color property

Data type
LongInt

Description
Specifies the display color of an object.

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Columnar property

Data type
Logical

Description
The Columnar property is valid for multi-record objects in Report Design windows. It is not valid for objects in forms. When Columnar is True, each individual record expands or contracts individually when you print or preview the report. This means that the multi-record object does not display the records in a fixed-size grid. When Columnar is False, the multi-record object displays the records in a fixed grid. By setting Columnar to True, you can usually fit more records on a single page than you can with Columnar set to False.

Values
True, False

ColumnPosition property

Data type
SmallInt

Description
Specifies the position (starting with 1) to which you want to move the current column of a table frame.

Values
>0

ColumnWidth property

Data type
LongInt

Description
Specifies the width in twips of the current column of a table frame.

Values
>0

CompleteDisplay property

Data type
Logical

Description
Specifies whether to display the complete contents of a field.

Values
True, False

ContainerName property

Data type
String

Description
Reports the name of an object's container.

Values
N/A

ControlMenu property

Data type
Logical

Description
Read/write. Specifies whether a form has a Control menu. If ControlMenu is True (the default value), the form has one; otherwise, it does not.

Values
True, False

CurrentColumn property

Data type
SmallInt

Description
At run time, this property applies to crosstabs, multi-record objects, and table frames. It determines the active column. It is especially useful for working with crosstabs: all cells in a crosstab have the same name, so it is almost impossible to address the one you want. Using CurrentColumn and CurrentRow, you can move to the cell you want and then have your code address the active object, its container, etc.

At design time, it also applies to table frames (the table frame need not be selected). Setting this property in the designer does not necessarily cause any change in visual appearance. In design time, it is useful for controlling column rotation and resizing.

Values
>0

CurrentRecordMarker.Color property

Data type
LongInt

Description
Specifies the display color of the current record of a table view.

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

CurrentRecordMarker.LineStyle property

Data type
SmallInt

Description
Specifies the style of the line that marks the current record in a table view.

Values
DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine

CurrentRecordMarker.Show property

Data type
Logical

Description
Specifies whether to highlight the current record in a table view.

Values
True, False

CurrentRow property

Data type
SmallInt

Description
At run time, this property applies to crosstabs, multirecord objects, and table frames. It determines the active row. It is especially useful for working with crosstabs: all cells in a crosstab have the same name, so it is almost impossible to address the one you want. Using CurrentColumn and CurrentRow, you can move to the cell you want and then have your code address the active object, its container, etc.

At design time, it also applies to table frames (the table frame need not be selected). Setting this property in the designer does not necessarily cause any change in visual appearance. In design time, it is useful for controlling column rotation and resizing.

Values
>0

CursorColumn property

Data type

LongInt

Description

The horizontal position of the insertion point in a field object, where position 0 is to the left of the first character.

Values

N/A

CursorLine property

Data type

LongInt

Description

The vertical position of the insertion point in a field object, where the first line is line 1.

Values

N/A

CursorPos property

Data type
LongInt

Description
The position of the insertion point in a field object, relative to the first character in the field. Counting begins with 0, the position to the left of the first character.

Values
N/A

DataSource property

Data type

String

Description

The name of the table that provides items in a list. It fills a list with values from a specified field (column) of a table.

Values

N/A

Default property

Data type
String

Description
The default value of a field.

Values
N/A

DefineGroup property

Data type

Logical

Description

Specifies whether a report band defines a group.

Values

True, False

DeleteColumn property

Data type
SmallInt

Description
Write-only. Specifies the number of the column to delete from a table frame.

Values
>0

Deleted property

Data type
Logical

Description
Reports whether a record in a dBASE table has been flagged as deleted.

Values
True, False

DeleteWhenEmpty property

Data type
Logical

Description
Causes the field to be deleted from the report if it has no data (including any labels, buttons, and so on).

Values
True, False

Design.ContainObjects property

Data type
Logical

Description
Specifies whether an object can contain other objects.

Values
True, False

Design.PinHorizontal property

Data type
Logical

Description
Specifies whether to prevent an object from moving horizontally.

Values
True, False

Design.PinVertical property

Data type

Logical

Description

Specifies whether to prevent an object from moving vertically.

Values

True, False

Design.Selectable property

Data type

Logical

Description

The Design.Selectable property is valid for UIObjects in forms and reports in design windows and at runtime. It specifies whether an object can be selected. When Design.Selectable is True, you can select the object, and selection handles appear around the object. When Design.Selectable is False, you cannot select the object, so there are no handles; however, you can still right-click the object to view its menu.

Values

True, False

Design.SizeToFit property

Data type
Logical

Description
Specifies whether the object will change size to accommodate its contents.

Values
True, False

DesignModified property

Data type
Logical

Description
Paradox sets a form's (or report's) DesignModified property to True when its design is changed, whether by an interactive user or ObjectPAL statements, either in a design window or at run time. When you close a form (or report) and DesignModified is True, Paradox displays a dialog box asking if you want to save it. When DesignModified is False, Paradox closes the it without displaying the dialog box or saving it, and changes are lost.

Values
True, False

DesignSizing property

Data type
SmallInt

Description
Specifies design time sizing for a text box.

Values
TextFixedSize, TextGrowOnly, TextSizeToFit

DesktopForm property

Data type

Logical

Description

Specifies whether a form's menus are used by other forms on the Desktop.

Values

True, False

DialogForm property

Data type
Logical

Description
Specifies whether a form will open as a dialog box.

Values
True, False

DialogFrame property

Data type
Logical

Description
When True and DialogForm and Border are also True, form has a conventional dialog box frame.

Values
True, False

DisplayType property

Data type
SmallInt

Description
Returns the display type of a field object.

Values
CheckBoxField, ComboField, EditField, LabeledField, ListField, RadioButtonField

Editing property

Data type

Logical

Description

Valid for objects (including forms) bound to tables. Editing returns True if the table associated with an object is in Edit mode. For field objects, Editing indicates whether that field is active and using a temporary "edit" object (such as being in field view).

When you put a form into Edit mode (for example, by pressing F9 or clicking the Edit Data button on the Toolbar), you automatically put all associated tables into Edit mode.

Values

True, False

Enabled property

Data type

Logical

Description

Specifies whether a UI object is enabled (true) or disabled (false). When a UI object is disabled the text in the object becomes grayed. The object no longer responds to mouse clicks and it can't be moved to with TAB key (as if the TABBABLE property were off). All objects contained by a disabled UI object are also disabled (set to FALSE).

Values

True, False

End property

Data type
Point

Description
Specifies the coordinates of the end of a line. See also: Start.

Values
N/A

FieldName property

Data type
String

Description
Specifies the name of the field to which a field object or list is bound.

This is not a new property; it was altered slightly in version 5.0 so that, in addition to setting a value like "Quant" or "Bookord->Quant" or "[Bookord.Quant]", you can also specify "sum(Bookord.Quant)", thus allowing aggregated values to be placed. This property's return value is no longer simply the field name (like "Quant"), but a string just as you would see if you clicked the Copy Field button in the Define Field dialog.

To use these new features to greatest advantage, do either (or both) of the following:

- Change code that uses expressions like `active.TableName + "." + active.FieldName` to just `active.FieldName`.
- Use calculations like these:

```
actField = active.FieldName
x = actField.SubStr( actField.search( "." ) + 1, actField.Size() )
```

Values
N/A

FieldNo property

Data type
SmallInt

Description
Reports the position of a field in a table, where the first field is field 1.

Values
N/A

FieldRights property

Data type
String

Description
Reports the user's field rights.

Values
ReadOnly, ReadWrite, All

FieldSize property

Data type
SmallInt

Description
The field's size (for alphanumeric and dBASE number fields).

Values
N/A

FieldType property

Data type
String

Description
The field's data type.

Values
N/A

FieldUnits2 property

Data type
SmallInt

Description
Indicates the number of decimal places in a dBASE number field. For a Paradox table (or any other driver or field type that does not require field units to be specified), this method returns 0.

Values
N/A

FieldValid property

Data type
Logical

Description
Reports whether a field passes its own value checks.

Values
True, False

FieldView property

Data type
Logical

Description
Reports whether a field is in Field View.

Values
True, False

First property

Data type
String

Description
Returns the name of the first child object in a container.

Values
N/A

FirstRow property

Data type

String

Description

Read-only. FirstRow returns the name of the first record object within a table frame or multi-record object.

Values

N/A

FitHeight property

Data type
Logical

Description
Specifies whether an edit region expands vertically to accommodate text.

Values
True, False

FitWidth property

Data type
Logical

Description
Specifies whether an edit region or crosstab cell expands horizontally to accommodate text.

Values
True, False

FlyAway property

Data type
Logical

Description
Reports whether a record has moved to its sorted position in a table.

Values
True, False

Focus property

Data type
Logical

Description
Reports whether an object's built-in **setFocus** method has been called. Set to False when the object's built-in **removeFocus** method is called.

Values
True, False

Font.Color property

Data type
LongInt

Description
Specifies the color of characters displayed in a field object or a text object.

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Font.Size property

Data type
SmallInt

Description
Specifies (in printer's points) the size of characters displayed in a field object or a text object.

Values
>0

Font.Style property

Data type

SmallInt

Description

Specifies the style of characters displayed in a field object or a text object.

Values

FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

Font.Typeface property

Data type
String

Description
Specifies the typeface of characters displayed in a field object or a text object.

Values
Depends on system.

Format.DateFormat property

Data type
String

Description
Specifies the format for date values.

Values
Format specification

Format.LogicalFormat property

Data type
String

Description
Specifies the format for logical values.

Values
Format specification

Format.NumberFormat property

Data type
String

Description
Specifies the format for number values.

Values
Format specification

Format.TimeFormat property

Data type
N/A

Description
Specifies the format for time values.

Values
Format specification

Format.TimeStampFormat property

Data type
String

Description
Specifies the format for time stamps.

Values
Format specification

Frame.Color property

Data type
LongInt

Description
Specifies the color of an object's frame.

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Frame.Style property

Data type
SmallInt

Description
Specifies the style of an object's frame.

Values
3DWindows, 3DWindowsGroup, DashDotDotFrame, DashDotFrame, DashedFrame, DottedFrame, DoubleFrame, Inside3DFrame, NoFrame, Outside3DFrame, ShadowFrame, SolidFrame, WideInsideDoubleFrame, WideOutsideDoubleFrame

Frame.Thickness property

Data type
SmallInt

Description
Specifies the thickness of an object's frame in pixels.

Values
N/A

FrameObjects property

Data type

Logical

Description

Specifies whether the dotted frame shows around objects in the designers. If FrameObjects is true, creates a 1 pixel region in which embedding cannot occur.

Values

True, False

FullName property

Data type
String

Description
Returns the full name (including containership path) of an object in a form.

Values
N/A

FullSize property

Data type
Point

Description
In scrolling object, returns actual size if you could see the whole thing.

Values
N/A

Grid.Color property

Data type
LongInt

Description
Specifies the color of the grid in a table frame.

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Grid.GridStyle property

Data type
SmallInt

Description
Specifies the style of grid lines in a table frame.

Values
tf3D, tfDoubleLine, tfNoGrid, tfSingleLine, tfTripleLine

Grid.RecordDivider property

Data type
Logical

Description
Specifies whether dividing lines are displayed between records in a table frame.

Values
True, False

GridLines.Color property

Data type
LongInt

Description
Specifies the color of grid lines in a Table window.

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

GridLines.ColumnLines property

Data type
Logical

Description
Specifies whether to display column lines in a Table window.

Values
True, False

GridLines.HeadingLines property

Data type
Logical

Description
Specifies whether to display heading lines in a Table window.

Values
True, False

GridLines.LineStyle property

Data type
SmallInt

Description
Specifies the style of grid lines in a Table window.

Values
DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine

GridLines.QueryLook property

Data type
Logical

Description
Specifies whether a Table window displays grid lines in the same style as a Query Editor window.

Values
True, False

GridLines.RowLines property

Data type

Logical

Description

Specifies whether to display grid lines in a Table window.

Values

True, False

GridLines.Spacing property

Data type
SmallInt

Description
Specifies the spacing of grid lines in a Table window.

Values
TextSingleSpacing, TextDoubleSpacing, TextTripleSpacing

GridValue property

Data type

Point

Description

GridValue determines the minimum grid interval in twips. Note there is no control (except through the UI) of the number of these intervals shown as major or minor intervals when ShowGrid is on. This does, however, give more control of the granularity of the grid than does the dialog in the UI.

Values

>0

GroupObjects property

Data type
Logical

Description
Write-only. Specifies whether to group selected objects in forms and reports. If true, GroupObjects is identical to choosing the "Group" menu item. If false, GroupObjects is identical to choosing the "Ungroup" menu item.

Values
True, False

GroupRecords property

Data type
SmallInt

Description
GroupRecords determines the number of records in a group in a report.

Values
>0

Header property

Data type
String

Description
Read-only. Header returns the name of a table frame's header object (if it has one).

Values
N/A

HeadingHeight property

Data type
LongInt

Description
Determines (in twips) the height of the heading in a Table window.

Values
>0

Headings property

Data type
String

Description
Specifies which report headings to print.

Values
"GroupOnly", "PageAndGroup"

HorizontalScrollBar property

Data type

Logical

Description

Specifies whether a table frame has a horizontal scroll bar.

Values

True, False

IndexField property

Data type

Logical

Description

Reports whether a field object is bound to an indexed field in a table.

Values

True, False

InsertColumn property

Data type

SmallInt

Description

Write-only. Specifies where (which column) to insert a column in a table frame.

Values

>0

InsertField property

Data type

Point

Description

Write-only property. Inserts a field object into a text box. Specify the field size using a Point value. The upper left corner of the field is at (0, 0) relative to the text box.

Values

N/A

Inserting property

Data type
Logical

Description
Returns True when a record is being inserted anywhere in a form.

Values
True, False

Invisible property

Data type

Logical

Description

Invisible applies only to boxes and lines in reports. Like the Visible property, which applies to objects in forms, Invisible determines whether an object is visible at run time. However, unlike Visible, when you use Invisible to hide an object, contained objects are not hidden. Set Invisible to True to hide an object, set Invisible to False to show an object.

Values

True, False

Justification property

Data type
SmallInt

Description
Specifies the justification of data in a Table window.

Values
TextAlignTop, TextAlignBottom, TextAlignVCenter, TextAlignLeft, TextAlignRight, TextAlignCenter

KeyField property

Data type
Logical

Description
Reports whether a field object is bound to a key field in a table.

Values
True, False

LabelText property

Data type
String

Description
Specifies the text displayed in a button's label.

Values
N/A

LeftBorder property

Data type

LongInt

Description

Returns the size of an object's left border (in twips). An object's border represents the area in which you cannot embed another object.

Values

N/A

Line.Color property

Data type
LongInt

Description
Specifies the color of a line for an ellipse.

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Line.LineStyle property

Data type
SmallInt

Description
Specifies the style of a line for an ellipse.

Values
DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine

Line.Thickness property

Data type
SmallInt

Description
Specifies the thickness of a line for an ellipse.

Values
N/A

LineEnds property

Data type
SmallInt

Description
Specifies whether (or where) to place arrows at the end of a line.

Values
ArrowBothEnds, ArrowOneEnd, NoArrowEnd

LineSpacing property

Data type
SmallInt

Description
Specifies the number of blank lines to print between each line of text in a field object or a text object.

Values
TextDoubleSpacing, TextDoubleSpacing2, TextSingleSpacing, TextSingleSpacing2, TextTripleSpacing

LineStyle property

Data type
SmallInt

Description
Specifies the style of a line.

Values
DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine

LineType property

Data type
SmallInt

Description
Specifies the type of a line.

Values
CurvedLine, StraightLine

List.Count property

Data type
SmallInt

Description
Specifies the number of items in a list.

Values
N/A

List.Selection property

Data type
SmallInt

Description
Specifies the item selected from a list.

Values
N/A

List.Value property

Data type
AnyType

Description
Determines the value of an item in a list.

Values
N/A

Locked property

Data type
Logical

Description
Reports whether the table bound to a design object is locked.

Values
True, False

LookupTable property

Data type
String

Description
The name of the lookup table for a field object.

Values
N/A

LookupType property

Data type
String

Description
Specifies the type of table lookup.

Values
JustCurrentField, AllCorresponding

Magnification property

Data type
SmallInt

Description
Specifies the display magnification of a bitmap object. You can also enter a literal value.

Values
Magnify25, Magnify50, Magnify100, Magnify200, Magnify400, MagnifyBestFit

Manager property

Data type

String

Description

Returns UIObject name of a form.

Values

N/A

MarkerPos property

Data type
LongInt

Description
The "other end" of a selection. See also CursorPos.

Values
N/A

Margins.Bottom property

Data type
LongInt

Description
Determines the height of a report's bottom margin in twips.

Values
>0

Margins.Left property

Data type
LongInt

Description
Determines the width of a report's left margin in twips.

Values
>0

Margins.Right property

Data type
LongInt

Description
Determines the width of a report's right margin in twips.

Values
>0

Margins.Top property

Data type
LongInt

Description
Determines the height of a report's top margin in twips.

Values
>0

MaximizeButton property

Data type
Logical

Description
Determines whether a form's window has a maximize box.

Values
True, False

Maximum property

Data type
String

Description
Specifies the maximum value allowed in a field.

Values
N/A

MemoView property

Data type
Logical

Description
Specifies whether a field object is in Memo View mode.

Values
True, False

MinimizeButton property

Data type
Logical

Description
Reports or specifies whether a form's window has a minimize box.

Values
True, False

Minimum property

Data type
String

Description
Specifies the maximum value allowed in a field.

Values
N/A

Modal property

Data type

Logical

Description

Specifies whether a dialog form is modal. A modal dialog box retains focus until you close it. Further, you cannot resize nor move a modal dialog box.

Values

True, False

MouseActivate property

Data type
Logical

Description
Specifies whether a dialog form gets focus as the result of a MouseEvent.

Values
True, False

NCols property

Data type
SmallInt

Description
Returns the number of columns in a table frame or multi-record object.

Values
N/A

NRecords property

Data type

LongInt

Description

Reports the number of records in the table bound to a design object. This property returns the number of records in the underlying table, not the number of records displayed in the object.

Note: When you read NRecords after setting a filter, the returned value does not represent the number of records in the filtered set. To get that information, attach a TCursor to the UIObject and call **cCount**. When you read NRecords after setting a range, the returned value represents the number of records in the set defined by the range.

Values

N/A

NRows property

Data type

SmallInt

Description

Returns the number of rows in a table frame or multi-record object.

Values

N/A

Name property

Data type
String

Description
Specifies the name of a design object.

Values
N/A

Next property

Data type
String

Description
Returns the name of the next object in the same container.

Values
N/A

NextTabStop property

Data type
String

Description
Determines the object name of the next tab stop on a form or report. When you read NextTabStop, it returns a UIObject; when you write it, you must assign a String.

Values
N/A

NoEcho property

Data type
Logical

Description
Specifies whether characters typed into a field object are displayed.

Values
True, False

OtherBandName property

Data type

String

Description

Returns the name of a report band's counterpart (the "other" band in a pair of bands). Given a header band, it returns the name of the corresponding footer band. Given a footer band, it returns the name of the corresponding header band. Given a record band, it returns the name of that record band.

Values

N/A

OverStrike property

Data type

Logical

Description

Specifies whether a field or text object is in overstrike (as opposed to insert) mode.

Values

True, False

Owner property

Data type
String

Description
Name of the logical container of an object, irrespective of intermediate cosmetic objects.

Values
N/A

PageSize property

Data type
Point

Description
PageSize determines the size of a page in a report in a design window. This property was added because there is no page object in a banded report designer.

Values
>0

PageTiling property

Data type
SmallInt

Description
PageTiling determines how to arrange pages in a form. It uses PageTiling constants.

Values
StackPages, TileHorizontal, TileVertical

Pattern.Color property

Data type
LongInt

Description
Specifies the color of a pattern.

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Pattern.Style property

Data type

SmallInt

Description

Specifies the style of a pattern.

Values

BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

PersistView property

Data type

Logical

Description

Specifies whether to remain in Field View or Memo View.

Values

True, False

Picture property

Data type

String

Description

A template that formats the value of a field. You can use picture string characters to define a picture for a bound or unbound field object in a form (this feature was added in version 5.0). The maximum length of an ObjectPAL picture string is 255 characters; the maximum size of a picture defined interactively is 175 characters. If a bound field object has a picture defined in the data model, the property does not override it: the table picture takes precedence.

Values

N/A

Position property

Data type
Point

Description
Specifies the coordinates of the upper left corner of a design object, relative to its container.

Values
N/A

PositionalOrder property

Data type

SmallInt

Description

For Page objects, specifies the page number. For Band objects, specifies the position the band occupies, counting from the top. The Report header always has PositionalOrder of 1, the Page header always has PositionalOrder of 2, and so on. Only the PositionalOrder of group bands can be set. Can be used to rearrange groups or to swap header and footer.

Values

>0

PrecedePageHeader property

Data type
Logical

Description
Specifies whether a report band should appear before the page header.

Values
True, False

Prev property

Data type
String

Description
Returns the name of the previous object in the same container.

Values
N/A

PrinterDocument property

Data type

Logical

Description

Specifies whether the document is designed for printer or screen. The setting determines which fonts are used, and which frame widths are available for the various frame styles.

Values

True, False

PrintOn1stPage property

Data type
Logical

Description
Specifies whether to print a report band on the first page of the report.

Values
True, False

ProgID property

Data type
String

Description
Specifies the internal identifier of a system object or OLE Automation object. When the object is a Paradox Native Windows Control (NWC), such as the TrackBar or ProgressBar, the 'ProgID' will return that string (literally, "TrackBar", etc.) When the object is an OCX, ProgID returns the OLE 2.0 ProgID.

Values
N/A

Range property

Data type
SmallInt

Description
Range is used on report bands to convert a group on a field to a group on a range of that field (or to change the range on a range group). Semantics are the same as the range value in the define group dialog (dependent on field type). The DateRangeType constants are provided for use with date and timestamp fields: ByDay, ByWeek, ByMonth, ByQuarter, and ByYear.

Values
Depends on field type.

RasterOperation property

Data type
LongInt

Description
Specifies how to blend the colors in two overlapping design objects.

Values
MergePaint, NotSourceCopy, NotSourceErase, SourceAnd, SourceCopy, SourceErase, SourceInvert, SourcePaint

Readonly property

Data type

Logical

Description

Specifies whether a field object is read-only.

Values

True, False

RecNo property

Data type
LongInt

Description
Reports the position of a record. (This can be a time-consuming operation for dBASE tables.)

Values
N/A

Refresh property

Data type
Logical

Description
Reports when data displayed onscreen is being changed, either across a network, by an ObjectPAL statement, or user action.

Values
True, False

RefreshOption property

Data type
SmallInt

Description
Determines what to do when data changes while printing a report. It uses ReportPrintRestart constants.

Values
PrintFromCopy, PrintLock, PrintNoLock, PrintRestart, PrintReturn

RemoveGroupRepeats property

Data type
Logical

Description
Use RemoveGroupRepeats to retain or suppress repeated group values within a record band. When RemoveGroupRepeats is False, Paradox displays the value of the grouped field for each record, including duplicates, in the record band. When it is True, Paradox prints the value for the first record of the group only.

Values
True, False

RepeatHeader property

Data type
Logical

Description
Determines whether the header is repeated on each page of a report. If True, the header is repeated; otherwise, it is not repeated.

Values
True, False

Required property

Data type
Logical

Description
Reports whether a field object must be assigned a value for the record to be valid.

Values
True, False

RightBorder property

Data type

LongInt

Description

Returns the size of an object's right border (in twips). An object's border represents the area in which you cannot embed another object.

Values

N/A

RowHeight property

Data type
LongInt

Description
Determines the height (in twips) of a row in a Table window.

Values
>0

RowNo property

Data type
SmallInt

Description
Reports the row number of a record displayed in a table frame, multi-record object, or table view, starting with 1.

Values
N/A

Scroll property

Data type
Point

Description
How far you've scrolled.

Values
N/A

SeeMouseMove property

Data type

Logical

Description

When SeeMouseMove is True, the form responds to mouse movements (mouseEnter, mouseMove, and mouseExit) even when the form does not have focus. This can be useful in forms used as custom Toolbars. When SeeMouseMove is False (the default), the form does not respond to mouse movements unless it has focus. This property is saved with the form.

Values

True, False

Select property

Data type
Logical

Description
The Select property is valid for UIObjects in a Form Design window or a Report Design window. It specifies whether an object is selected. When Select is True, selection handles appear around the frame of the selected object, just as if you had selected it interactively. When Select is False, there are no handles. This property is not valid while the form or report is running.

Values
True, False

SelectedText property

Data type
String

Description
Returns the selected text in a field object.

Values
N/A

SeqNo property

Data type

LongInt

Description

The actual sequence number of a record as displayed, taking filters and indexes into account. Returns <N/A> when table is from a dynaset or has a filter.

Values

N/A

ShowGrid property

Data type
Logical

Description
Specifies whether the form or report grid is visible.

Values
True, False

Shrinkable property

Data type
Logical

Description
Specifies whether a report band can be shrunk.

Values
True, False

Size property

Data type
Point

Description
Specifies the coordinates of the lower right corner of a design object, relative to its upper left corner.

Values
N/A

SizeToFit property

Data type

Logical

Description

When True, the form opens at the size of the underlying page. When False, the form opens at a preset default size.

Values

True, False

SnapToGrid property

Data type

Logical

Description

When SnapToGrid is True, design objects jump to the closest minor division of the grid when moved or resized. Internally generated resizes (such as when you add text to a text object or define a field object) do not snap to the grid.

When SnapToGrid is False, object size and position are not affected by the grid.

Values

True, False

SortOrder property

Data type
Logical

Description
Specifies the sort order of a report. True = Descending order, False = Ascending order.

Values
Ascending, Descending

SpecialField property

Data type
SmallInt

Description
SpecialField determines the type of a special field. It uses SpecialFieldTypes constants. On the types that require a table, the current table for the field is used. If there is no current table, the master table of the form or report is assumed.

Values
DateField, NofFieldsField, NofPagesField, NofRecsField, PageNumField, RecordNoField, TableNameField, TimeField

StandardMenu property

Data type
Logical

Description
Specifies whether a form or report uses the standard Paradox menus. Set it to True to use Paradox menus; otherwise, set it to False.

Values
True, False

StandardToolbar property

Data type
Logical

Description
Specifies whether a form or report uses the standard Toolbar. Set it to True to use the standard Toolbar; otherwise, set it to False.

Values
True, False

Start property

Data type
Point

Description
Specifies the coordinates of the start of a line. See also: End.

Values
N/A

StartPageNumbers property

Data type
SmallInt

Description
Determines the starting value for page numbers in a report.

Values
>0

Style property

Data type
SmallInt

Description
Reports or specifies the display style of a button.

Values
BorlandButton, Windows3dButton, WindowsButton

SummaryModifier property

Data type

SmallInt

Description

SummaryModifier determines how to modify aggregator fields in reports. It uses AggModifiers constants.

Values

CumulativeAgg, CumUniqueAgg, RegularAgg, UniqueAgg

TabStop property

Data type
Logical

Description
Specifies whether a field object is a tab stop.

Values
True, False

TableName property

Data type
String

Description
Specifies the name of a table to which a design object is bound.

Values
N/A

Text property

Data type
String

Description
Specifies the characters displayed in a text object.

Values
N/A

ThickFrame property

Data type
Logical

Description
When True, and DialogForm and Border also True, specifies a thick window frame instead of the usual pixel-wide frame.

Values
True, False

Thickness property

Data type
SmallInt

Description
Specifies the thickness of a line.

Values
LWidth10Points, LWidth1Point, LWidth2Points, LWidth3Points, LWidth6Points, LWidthHairline, LWidthHalfPoint

Title property

Data type
String

Description
Specifies the text of a form's caption.

Values
N/A

TopBorder property

Data type

LongInt

Description

Returns the size of an object's top border (in twips). An object's border represents the area in which you cannot embed another object.

Values

N/A

TopLine property

Data type
LongInt

Description
The number of the line currently displayed at the top of a text object.

Values
N/A

Touched property

Data type

Logical

Description

True when user has made changes not yet committed.

Values

True, False

Translucent property

Data type
Logical

Description
Specifies whether the color of an object is translucent.

Values
True, False

UncheckedValue property

Data type

String

Description

Specifies the value that a button will write to its parent field object when the button is unchecked. This property is supported only by checkboxes.

Values

N/A

Value property

Data type
String

Description
Specifies the value of a design object.

Values
N/A

VerticalScrollBar property

Data type

Logical

Description

Specifies whether an object has a vertical scroll bar. Not valid for all UIObjects. Refer to chart.

Values

True, False

Visible property

Data type
Logical

Description
Specifies whether an object is displayed in a form at run time. Setting Visible to True makes the object (and the objects it contains) visible; setting Visible to False hides the object and the objects it contains.

Values
True, False

WideScrollBar property

Data type

Logical

Description

Determines whether a scroll bar is wide or narrow (the default). If True, the scroll bar is wide; otherwise, it is narrow.

Values

True, False

Width property

Data type
LongInt

Description
Determines the width (in twips) of a column in a Table window.

Values
>0

WordWrap property

Data type
Logical

Description
Specifies whether to wrap lines that exceed the width of a field object.

Values
True, False

Xseparation property

Data type
LongInt

Description
Specifies the distance (in twips) between records in the indicated direction.

Values
>0

Yseparation property

Data type
LongInt

Description
Specifies the distance (in twips) between records in the indicated direction.

Values
>0

▪

Properties unique to chart objects

See also

Choose from the following chart object properties for more information.

BackWall.Color
BackWall.Pattern.Color
BackWall.Pattern.Style
Background.Color
Background.Pattern.Color
Background.Pattern.Style
BaseFloor.Color
BaseFloor.Pattern.Color
BaseFloor.Pattern.Style
BindType
CurrentSeries
CurrentSlice
GraphType
Label.Font.Color
Label.Font.Size
Label.Font.Style
Label.Font.Typeface
Label.LabelFormat
Label.LabelLocation
Label.NumberFormat
LeftWall.Color
LeftWall.Pattern.Color
LeftWall.Pattern.Style
LegendBox.Color
LegendBox.Font.Color
LegendBox.Font.Size
LegendBox.Font.Style
LegendBox.Font.Typeface
LegendBox.LegendPos
LegendBox.Pattern.Color
LegendBox.Pattern.Style
MaxGroups
MaxXValues
MinXValues
Options.Elevation
Options.Rotation
Options.ShowAxes
Options.ShowGrid
Options.ShowLabels
Options.ShowLegend
Options.ShowTitle

Series.Color
Series.Graph_Title.Font.Color
Series.Graph_Title.Font.Size
Series.Graph_Title.Font.Style
Series.Graph_Title.Font.Typeface
Series.Graph_Title.Text
Series.Graph_Title.UseDefault
Series.Line.Color
Series.Line.LineStyle
Series.Line.Thickness
Series.Marker.Size
Series.Marker.Style
Series.Pattern.Color
Series.Pattern.Style
Series.TypeOverride
SeriesName
Slice.Color
Slice.Explode
Slice.Pattern.Color
Slice.Pattern.Style
TitleBox.Color
TitleBox.Graph_Title.Font.Color
TitleBox.Graph_Title.Font.Size
TitleBox.Graph_Title.Font.Style
TitleBox.Graph_Title.Font.Typeface
TitleBox.Graph_Title.Text
TitleBox.Graph_Title.UseDefault
TitleBox.Pattern.Color
TitleBox.Pattern.Style
TitleBox.Subtitle.Font.Color
TitleBox.Subtitle.Font.Size
TitleBox.Subtitle.Font.Style
TitleBox.Subtitle.Font.Typeface
TitleBox.Subtitle.Text
TitleBox.Subtitle.UseDefault
TitleBoxName
XAxisName
XAxis.Graph_Title.Font.Color
XAxis.Graph_Title.Font.Size
XAxis.Graph_Title.Font.Style
XAxis.Graph_Title.Font.Typeface
XAxis.Graph_Title.Text
XAxis.Graph_Title.UseDefault
XAxis.Scale.AutoScale
XAxis.Scale.HighValue

XAxis.Scale.Increment
XAxis.Scale.Logarithmic
XAxis.Scale.LowValue
XAxis.Ticks.Alternate
XAxis.Ticks.DateFormat
XAxis.Ticks.Font.Color
XAxis.Ticks.Font.Size
XAxis.Ticks.Font.Style
XAxis.Ticks.Font.Typeface
XAxis.Ticks.NumberFormat
XAxis.Ticks.TimeFormat
XAxis.Ticks.TimeStampFormat
YAxisName
YAxis.Graph_Title.Font.Color
YAxis.Graph_Title.Font.Size
YAxis.Graph_Title.Font.Style
YAxis.Graph_Title.Font.Typeface
YAxis.Graph_Title.Text
YAxis.Graph_Title.UseDefault
YAxis.Scale.AutoScale
YAxis.Scale.HighValue
YAxis.Scale.Increment
YAxis.Scale.Logarithmic
YAxis.Scale.LowValue
YAxis.Ticks.Alternate
YAxis.Ticks.DateFormat
YAxis.Ticks.Font.Color
YAxis.Ticks.Font.Size
YAxis.Ticks.Font.Style
YAxis.Ticks.Font.Typeface
YAxis.Ticks.NumberFormat
YAxis.Ticks.TimeFormat
YAxis.Ticks.TimeStampFormat
ZAxisName
ZAxis.Graph_Title.Font.Color
ZAxis.Graph_Title.Font.Size
ZAxis.Graph_Title.Font.Style
ZAxis.Graph_Title.Font.Typeface
ZAxis.Graph_Title.Text
ZAxis.Graph_Title.UseDefault
ZAxis.Scale.AutoScale
ZAxis.Scale.HighValue
ZAxis.Scale.Increment
ZAxis.Scale.Logarithmic
ZAxis.Scale.LowValue

ZAxis.Ticks.Alternate
ZAxis.Ticks.DateFormat
ZAxis.Ticks.Font.Color
ZAxis.Ticks.Font.Size
ZAxis.Ticks.Font.Style
ZAxis.Ticks.Font.Typeface
ZAxis.Ticks.NumberFormat
ZAxis.Ticks.TimeFormat
ZAxis.Ticks.TimeStampFormat

BackWall.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

BackWall.Pattern.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

BackWall.Pattern.Style

This property applies only to chart objects.

Data type
SmallInt

Values
BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

Background.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Background.Pattern.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Background.Pattern.Style

This property applies only to chart objects.

Data type
SmallInt

Values
BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

BaseFloor.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

BaseFloor.Pattern.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

BaseFloor.Pattern.Style

This property applies only to chart objects.

Data type
SmallInt

Values
BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

BindType

This property applies only to chart objects.

Data type
SmallInt

Values
Graph1DSummary, Graph2DSummary, GraphTabular

CurrentSeries

This property applies only to chart objects.

Data type
SmallInt

Values
N/A

CurrentSlice

This property applies only to chart objects.

Data type
SmallInt

Values
N/A

GraphType

This property applies only to chart objects.

Data type
SmallInt

Values
Graph2DArea, Graph2DBar, Graph2DColumns, Graph2DLine, Graph2DPie, Graph2DRotatedBar, Graph2DStackedBar, Graph3DArea, Graph3DBar, Graph3DColumns, Graph3DPie, Graph3DRibbon, Graph3DRotatedBar, Graph3DStackedBar, Graph3DStep, Graph3DSurface, GraphXY

Label.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Label.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

Label.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

Label.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

Label.LabelFormat

This property applies only to chart objects.

Data type
SmallInt

Values
GraphHideY, GraphPercent, GraphShowY

Label.LabelLocation

This property applies only to chart objects.

Data type
SmallInt

Values
Above, Below, Bottom, Center, Left, Middle, Right, Top

Label.NumberFormat

This property applies only to chart objects.

Data type
N/A

Values
Format specification

LeftWall.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

LeftWall.Pattern.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

LeftWall.Pattern.Style

This property applies only to chart objects.

Data type
SmallInt

Values
BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

LegendBox.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

LegendBox.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

LegendBox.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

LegendBox.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

LegendBox.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

LegendBox.LegendPos

This property applies only to chart objects.

Data type
SmallInt

Values
Bottom, Right

LegendBox.Pattern.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

LegendBox.Pattern.Style

This property applies only to chart objects.

Data type
SmallInt

Values
BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

MaxGroups

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on chart

MaxXValues

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on chart

MinXValues

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on chart

Options.Elevation

This property applies only to chart objects.

Data type
SmallInt

Values
0 to 90 degrees

Options.Rotation

This property applies only to chart objects.

Data type
SmallInt

Values
0 to 90 degrees

Options.ShowAxes

This property applies only to chart objects.

Data type
Logical

Values
True, False

Options.ShowGrid

This property applies only to chart objects.

Data type
Logical

Values
True, False

Options.ShowLabels

This property applies only to chart objects.

Data type
Logical

Values
True, False

Options.ShowLegend

This property applies only to chart objects.

Data type
Logical

Values
True, False

Options.ShowTitle

This property applies only to chart objects.

Data type
Logical

Values
True, False

Series.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Series.Graph_Title.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Series.Graph_Title.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

Series.Graph_Title.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

Series.Graph_Title.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

Series.Graph_Title.Text

This property applies only to chart objects.

Data type
String

Values
N/A

Series.Graph_Title.UseDefault

This property applies only to chart objects.

Data type
Logical

Values
True, False

Series.Line.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Series.Line.LineStyle

This property applies only to chart objects.

Data type
SmallInt

Values
DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine

Series.Line.Thickness

This property applies only to chart objects.

Data type
SmallInt

Values
LWidth10Points, LWidth1Point, LWidth2Points, LWidth3Points, LWidth6Points, LWidthHairline, LWidthHalfPoint

Series.Marker.Size

This property applies only to chart objects.

Data type
SmallInt

Values
MarkerSize0, MarkerSize2, MarkerSize4, MarkerSize8, MarkerSize12, MarkerSize18, MarkerSize24, MarkerSize36, MarkerSize54, MarkerSize72

Series.Marker.Style

This property applies only to chart objects.

Data type
SmallInt

Values
MarkerBoxedCross, MarkerBoxed_Plus, MarkerCross, MarkerFilledBox, MarkerFilledCircle, MarkerFilledDownTriangle, MarkerFilledTriangle, MarkerFilledTriangles, MarkerHollowBox, MarkerHollowCircle, MarkerHollowDownTriangle, MarkerHollowTriangle, MarkerHollowTriangles, MarkerHorizontalLine, MarkerPlus, MarkerVerticalLine

Series.Pattern.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Series.Pattern.Style

This property applies only to chart objects.

Data type
SmallInt

Values
BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

Series.TypeOverride

This property applies only to chart objects.

Data type
SmallInt

Values
Graph2DArea, Graph2DBar, Graph2DLine, None

SeriesName

This property applies only to chart objects.

Data type
String

Values
N/A

Slice.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Slice.Explode

This property applies only to chart objects.

Data type
Logical

Values
True, False

Slice.Pattern.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Slice.Pattern.Style

This property applies only to chart objects.

Data type
SmallInt

Values
BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

TitleBox.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

TitleBox.Graph_Title.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

TitleBox.Graph_Title.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

TitleBox.Graph_Title.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

TitleBox.Graph_Title.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

TitleBox.Graph_Title.Text

This property applies only to chart objects.

Data type
String

Values
N/A

TitleBox.Graph_Title.UseDefault

This property applies only to chart objects.

Data type
Logical

Values
True, False

TitleBox.Pattern.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

TitleBox.Pattern.Style

This property applies only to chart objects.

Data type
SmallInt

Values
BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern

TitleBox.Subtitle.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

TitleBox.Subtitle.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

TitleBox.Subtitle.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

TitleBox.Subtitle.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

TitleBox.Subtitle.Text

This property applies only to chart objects.

Data type
String

Values
N/A

TitleBox.Subtitle.UseDefault

This property applies only to chart objects.

Data type
Logical

Values
True, False

TitleBoxName

This property applies only to chart objects.

Data type
String

Values
N/A

XAxisName

This property applies only to chart objects.

Data type
String

Values
N/A

XAxis.Graph_Title.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

XAxis.Graph_Title.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

XAxis.Graph_Title.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

XAxis.Graph_Title.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

XAxis.Graph_Title.Text

This property applies only to chart objects.

Data type
String

Values
N/A

XAxis.Graph_Title.UseDefault

This property applies only to chart objects.

Data type
Logical

Values
True, False

XAxis.Scale.AutoScale

This property applies only to chart objects.

Data type
Logical

Values
True, False

XAxis.Scale.HighValue

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

XAxis.Scale.Increment

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

XAxis.Scale.Logarithmic

This property applies only to chart objects.

Data type
Logical

Values
True, False

XAxis.Scale.LowValue

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

XAxis.Ticks.Alternate

This property applies only to chart objects.

Data type
Logical

Values
True, False

XAxis.Ticks.DateFormat

This property applies only to chart objects.

Data type
N/A

Values
Format specification

XAxis.Ticks.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

XAxis.Ticks.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

XAxis.Ticks.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

XAxis.Ticks.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

XAxis.Ticks.NumberFormat

This property applies only to chart objects.

Data type
N/A

Values
Format specification

XAxis.Ticks.TimeFormat

This property applies only to chart objects.

Data type
String

Values
N/A

XAxis.Ticks.TimeStampFormat

This property applies only to chart objects.

Data type
String

Values
N/A

YAxisName

This property applies only to chart objects.

Data type
String

Values
N/A

YAxis.Graph_Title.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

YAxis.Graph_Title.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

YAxis.Graph_Title.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

YAxis.Graph_Title.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

YAxis.Graph_Title.Text

This property applies only to chart objects.

Data type
String

Values
N/A

YAxis.Graph_Title.UseDefault

This property applies only to chart objects.

Data type
Logical

Values
True, False

YAxis.Scale.AutoScale

This property applies only to chart objects.

Data type
Logical

Values
True, False

YAxis.Scale.HighValue

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

YAxis.Scale.Increment

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

YAxis.Scale.Logarithmic

This property applies only to chart objects.

Data type
Logical

Values
True, False

YAxis.Scale.LowValue

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

YAxis.Ticks.Alternate

This property applies only to chart objects.

Data type
Logical

Values
True, False

YAxis.Ticks.DateFormat

This property applies only to chart objects.

Data type
N/A

Values
Format specification

YAxis.Ticks.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

YAxis.Ticks.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

YAxis.Ticks.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

YAxis.Ticks.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

YAxis.Ticks.NumberFormat

This property applies only to chart objects.

Data type
N/A

Values
Format specification

YAxis.Ticks.TimeFormat

This property applies only to chart objects.

Data type
String

Values
N/A

YAxis.Ticks.TimeStampFormat

This property applies only to chart objects.

Data type

String

Values

N/A

ZAxisName

This property applies only to chart objects.

Data type
String

Values
N/A

ZAxis.Graph_Title.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

ZAxis.Graph_Title.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

ZAxis.Graph_Title.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

ZAxis.Graph_Title.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

ZAxis.Graph_Title.Text

This property applies only to chart objects.

Data type
String

Values
N/A

ZAxis.Graph_Title.UseDefault

This property applies only to chart objects.

Data type
Logical

Values
True, False

ZAxis.Scale.AutoScale

This property applies only to chart objects.

Data type
Logical

Values
True, False

ZAxis.Scale.HighValue

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

ZAxis.Scale.Increment

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

ZAxis.Scale.Logarithmic

This property applies only to chart objects.

Data type
Logical

Values
True, False

ZAxis.Scale.LowValue

This property applies only to chart objects.

Data type
Number

Values
Depends on chart

ZAxis.Ticks.Alternate

This property applies only to chart objects.

Data type
Logical

Values
True, False

ZAxis.Ticks.DateFormat

This property applies only to chart objects.

Data type
String

Values
N/A

ZAxis.Ticks.Font.Color

This property applies only to chart objects.

Data type
LongInt

Values
Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

ZAxis.Ticks.Font.Size

This property applies only to chart objects.

Data type
SmallInt

Values
Depends on system

ZAxis.Ticks.Font.Style

This property applies only to chart objects.

Data type
SmallInt

Values
FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline

ZAxis.Ticks.Font.Typeface

This property applies only to chart objects.

Data type
String

Values
Depends on system

ZAxis.Ticks.NumberFormat

This property applies only to chart objects.

Data type
String

Values
N/A

ZAxis.Ticks.TimeFormat

This property applies only to chart objects.

Data type
String

Values
N/A

ZAxis.Ticks.TimeStampFormat

This property applies only to chart objects.

Data type
String

Values
N/A

▪

UIObject properties

See also

Choose from the following UIObjects for a list of applicable properties.

Band

Bitmap

Box

Button

Cell

Crosstab

EditRegion

Ellipse

Field

Form

Graph

Group

Header

Line

List

Multirecord

OLE

Page

Record

TableFrame

TableView

Text

TVData

TVHeading

▪

Band properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Breakable
Class (read-only)
ContainerName (read-only)
DefineGroup
Enabled (7)
FieldName
First (read-only)
Focus (read-only)
FullName (read-only)
FullSize (read-only)
GroupRecords (5.0)
Headings
LeftBorder (read-only) (5.0)
Manager (read-only)
Name
Next (read-only)
OtherBandName (read-only) (5.0)
Owner (read-only)
Position
PositionalOrder (5.0)
PrecedePageHeader
Prev (read-only)
PrintOn1stPage
Range (5.0)
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Shrinkable
Size
SortOrder
StartPageNumbers (read-only) (5.0)
TopBorder (read-only) (5.0)

▪

Bitmap properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
Enabled (7)
First (read-only)
Focus (read-only)
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
HorizontalScrollBar
LeftBorder (read-only) (5.0)
Magnification
Manager (read-only)
Name
Next (read-only)
NextTabStop (5.0)
Owner (read-only)
Position
Prev (read-only)
RasterOperation
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Size
TopBorder (read-only) (5.0)
Value
VerticalScrollBar
Visible

-

Box properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Breakable
Class (read-only)
Color
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Enabled (7)
First (read-only)
Focus (read-only)
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
Invisible (5.0)
LeftBorder (read-only) (5.0)
Manager (read-only)
Name
Next (read-only)
Owner (read-only)
Pattern.Color
Pattern.Style
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Select (4.5)
Shrinkable
Size
TopBorder (read-only) (5.0)
Translucent
Visible

-

Button properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
ButtonType
CenterLabel
CheckedValue (5.0)
Class (read-only)
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Enabled (7)
First (read-only)
FitHeight
FitWidth
Focus (read-only)
FullName (read-only)
FullSize (read-only)
LabelText
LeftBorder (read-only) (5.0)
Manager (read-only)
Name
Next (read-only)
NextTabStop (5.0)
Owner (read-only)
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Select (4.5)
Size
Style
TabStop
TopBorder (read-only) (5.0)
UncheckedValue (5.0)
Value
Visible

-

Cell properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
Color
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
Enabled (7)
First (read-only)
FitWidth
Focus (read-only)
FullName (read-only)
FullSize (read-only)
HorizontalScrollBar
LeftBorder (read-only) (5.0)
Manager (read-only)
Name
Next (read-only)
NextTabStop (5.0)
Owner (read-only)
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Size
TopBorder (read-only) (5.0)
VerticalScrollBar

▪

Crosstab properties

Arrived (read-only)
Breakable
Class (read-only)
Color
ContainerName (read-only)
CurrentColumn (5.0)
CurrentRow (5.0)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
Enabled (7)
First (read-only)
FitHeight
FitWidth
Focus (read-only)
FullName (read-only)
FullSize (read-only)
Grid.Color
Grid.GridStyle
HorizontalScrollBar
Manager (read-only)
Name
Next (read-only)
NextTabStop (5.0)
Owner (read-only)
Position
Prev (read-only)
Scroll
Select (4.5)
Size
TableName
Touched (read-only)
Translucent
VerticalScrollBar
Visible

-

EditRegion properties

Alignment
Arrived (read-only)
AvgCharSize (read-only) (5.0)
BottomBorder (read-only) (5.0)
Breakable
Class (read-only)
Color
CompleteDisplay
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
DisplayType (read-only)
Enabled (7)
First (read-only)
FitHeight
FitWidth
Focus (read-only)
Font.Color
Font.Size
Font.Style
Font.Typeface
Format.DateFormat
Format.LogicalFormat
Format.NumberFormat
Format.TimeFormat
Format.TimeStampFormat
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
LeftBorder (read-only) (5.0)
Magnification
Manager (read-only)
Name
Next (read-only)
NextTabStop (5.0)
NoEcho
Owner (read-only)
Position

Prev (read-only)
RasterOperation
Readonly
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Size
TabStop
Text
TopBorder (read-only) (5.0)
Translucent
Value
Visible
WordWrap

-

Ellipse properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
Color
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Enabled (7)
First (read-only)
Focus (read-only)
FullName (read-only)
FullSize (read-only)
LeftBorder (read-only) (5.0)
Line.Color
Line.LineStyle
Line.Thickness
Manager (read-only)
Name
Next (read-only)
NextTabStop (5.0)
Owner (read-only)
Pattern.Color
Pattern.Style
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Select (4.5)
Shrinkable
Size
TopBorder (read-only) (5.0)
Translucent
Visible

▪

Field properties

Alignment
Arrived (read-only)
AutoAppend (4.5)
AvgCharSize (read-only) (5.0)
BlankRecord (read-only)
BottomBorder (read-only) (5.0)
Breakable
CalculatedField (5.0)
Class (read-only)
Color
CompleteDisplay
ContainerName (read-only)
CursorColumn
CursorLine
CursorPos
Default (read-only)
Deleted (read-only)
DeleteWhenEmpty (5.0)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
DisplayType
Editing (read-only) (4.5)
Enabled (7)
FieldName
FieldNo (read-only)
FieldRights (read-only)
FieldSize (read-only)
FieldType (read-only)
FieldUnits2 (read-only)
FieldValid (read-only)
First (read-only)
FitHeight
FitWidth
FlyAway (read-only)
Focus (read-only)
Font.Color
Font.Size
Font.Style
Font.Typeface
Format.DateFormat

Format.LogicalFormat
Format.NumberFormat
Format.TimeFormat
Format.TimeStampFormat
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
HorizontalScrollBar
IndexField (read-only)
Inserting (read-only)
KeyField (read-only)
LabelText
LeftBorder (read-only) (5.0)
Locked (read-only)
LookupTable (read-only)
LookupType (read-only)
Magnification
Manager (read-only)
MarkerPos
Maximum (read-only)
Minimum (read-only)
Name
Next (read-only)
NextTabStop (5.0)
NoEcho
NRecords (read-only)
OverStrike
Owner (read-only)
Pattern.Color
Pattern.Style
Picture (enhanced for 5.0)
Position
Prev (read-only)
RasterOperation
Readonly
RecNo (read-only)
Refresh (read-only)
Required (read-only)
RightBorder (read-only) (5.0)
RowNo (read-only)
Scroll
Select (4.5)
SelectedText

SeqNo (read-only)
Size
SpecialField (5.0)
SummaryModifier (5.0)
TableName
TabStop
Text
TopBorder (read-only) (5.0)
TopLine
Touched (read-only)
Translucent
Value
VerticalScrollBar
Visible
WideScrollBar (5.0)
WordWrap

-

Form properties

Arrived (read-only)
AutoAppend (4.5)
BlankRecord (read-only)
Border
Caption
Class (read-only)
ContainerName (read-only)
ControlMenu
Deleted (read-only)
DesignModified (4.5)
DesktopForm
DialogForm
DialogFrame
Editing (read-only) (4.5)
Enabled (7)
FieldView (read-only)
First (read-only)
FlyAway (read-only)
Focus (read-only)
FrameObjects (5.0)
FullName (read-only)
FullSize (read-only)
GridValue (5.0)
GroupObjects (5.0)
HorizontalScrollBar
Inserting (read-only)
Locked (read-only)
Manager (read-only)
MaximizeButton
MemoView (read-only)
MinimizeButton
Modal
MouseActivate
Name
Next (read-only)
NRecords (read-only)
PageTiling (5.0)
PersistView (read-only)
Position
Prev (read-only)
PrinterDocument (5.0)
RecNo (read-only)
Refresh (read-only)

Scroll
SeeMouseMove (5.0)
SeqNo (read-only)
ShowGrid (5.0)
Size (read-only)
SizeToFit
SnapToGrid (5.0)
StandardMenu
StandardToolbar (5.0)
TableName (read-only)
ThickFrame
Title
Touched (read-only)
VerticalScrollBar

▪

Chart properties

Arrived (read-only)
Background.Color
Background.Pattern.Color
Background.Pattern.Style
BackWall.Color
BackWall.Pattern.Color
BackWall.Pattern.Style
BaseFloor.Color
BaseFloor.Pattern.Color
BaseFloor.Pattern.Style
BindType
BottomBorder (read-only) (5.0)
Class (read-only)
Color
ContainerName (read-only)
CurrentSeries
CurrentSlice
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Enabled (7)
First (read-only)
Focus (read-only)
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
GraphType
Label.Font.Color
Label.Font.Size
Label.Font.Style
Label.Font.Typeface
Label.LabelFormat
Label.LabelLocation
Label.NumberFormat
LeftBorder (read-only) (5.0)
LeftWall.Color
LeftWall.Pattern.Color
LeftWall.Pattern.Style
LegendBox.Color
LegendBox.Font.Color

LegendBox.Font.Size
LegendBox.Font.Style
LegendBox.Font.Typeface
LegendBox.LegendPos
LegendBox.Pattern.Color
LegendBox.Pattern.Style
Manager (read-only)
MaxGroups
MaxXValues
MinXValues
Name
Next (read-only)
NextTabStop (5.0)
Options.Elevation
Options.Rotation
Options.ShowAxes
Options.ShowGrid
Options.ShowLabels
Options.ShowLegend
Options.ShowTitle
Owner (read-only)
Pattern.Color
Pattern.Style
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Series.Color
Series.Graph_Title.Font.Color
Series.Graph_Title.Font.Size
Series.Graph_Title.Font.Style
Series.Graph_Title.Font.Typeface
Series.Graph_Title.Text
Series.Graph_Title.UseDefault
Series.Line.Color
Series.Line.LineStyle
Series.Line.Thickness
Series.Marker.Size (5.0)
Series.Marker.Style (5.0)
Series.Pattern.Color
Series.Pattern.Style
Series.TypeOverride
SeriesName (5.0)
Size

Slice.Color
Slice.Explode
Slice.Pattern.Color
Slice.Pattern.Style
TableName
TabStop
TitleBox.Color
TitleBox.Graph_Title.Font.Color
TitleBox.Graph_Title.Font.Size
TitleBox.Graph_Title.Font.Style
TitleBox.Graph_Title.Font.Typeface
TitleBox.Graph_Title.Text
TitleBox.Graph_Title.UseDefault
TitleBox.Pattern.Color
TitleBox.Pattern.Style
TitleBox.Subtitle.Font.Color
TitleBox.Subtitle.Font.Size
TitleBox.Subtitle.Font.Style
TitleBox.Subtitle.Font.Typeface
TitleBox.Subtitle.Text
TitleBox.Subtitle.UseDefault
TitleBoxName (5.0)
TopBorder (read-only) (5.0)
Touched (read-only)
Translucent
Visible
XAxisName (5.0)
XAxis.Graph_Title.Font.Color
XAxis.Graph_Title.Font.Size
XAxis.Graph_Title.Font.Style
XAxis.Graph_Title.Font.Typeface
XAxis.Graph_Title.Text
XAxis.Graph_Title.UseDefault
XAxis.Scale.AutoScale
XAxis.Scale.HighValue
XAxis.Scale.Increment
XAxis.Scale.Logarithmic
XAxis.Scale.LowValue
XAxis.Ticks.Alternate
XAxis.Ticks.DateFormat
XAxis.Ticks.Font.Color
XAxis.Ticks.Font.Size
XAxis.Ticks.Font.Style
XAxis.Ticks.Font.Typeface
XAxis.Ticks.NumberFormat

XAxis.Ticks.TimeFormat (5.0)
XAxis.Ticks.TimeStampFormat (5.0)
YAxisName (5.0)
YAxis.Graph_Title.Font.Color
YAxis.Graph_Title.Font.Size
YAxis.Graph_Title.Font.Style
YAxis.Graph_Title.Font.Typeface
YAxis.Graph_Title.Text (5.0)
YAxis.Graph_Title.UseDefault
YAxis.Scale.AutoScale
YAxis.Scale.HighValue
YAxis.Scale.Increment
YAxis.Scale.Logarithmic
YAxis.Scale.LowValue
YAxis.Ticks.Alternate
YAxis.Ticks.DateFormat
YAxis.Ticks.Font.Color
YAxis.Ticks.Font.Size
YAxis.Ticks.Font.Style
YAxis.Ticks.Font.Typeface
YAxis.Ticks.NumberFormat
YAxis.Ticks.TimeFormat (5.0)
YAxis.Ticks.TimeStampFormat (5.0)
ZAxisName (5.0)
ZAxis.Graph_Title.Font.Color
ZAxis.Graph_Title.Font.Size
ZAxis.Graph_Title.Font.Style
ZAxis.Graph_Title.Font.Typeface
ZAxis.Graph_Title.Text (5.0)
ZAxis.Graph_Title.UseDefault
ZAxis.Scale.AutoScale
ZAxis.Scale.HighValue
ZAxis.Scale.Increment
ZAxis.Scale.Logarithmic
ZAxis.Scale.LowValue
ZAxis.Ticks.Alternate
ZAxis.Ticks.DateFormat
ZAxis.Ticks.Font.Color
ZAxis.Ticks.Font.Size
ZAxis.Ticks.Font.Style
ZAxis.Ticks.Font.Typeface
ZAxis.Ticks.NumberFormat (5.0)
ZAxis.Ticks.TimeFormat (5.0)
ZAxis.Ticks.TimeStampFormat (5.0)

-

Group properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Breakable
Class (read-only)
ContainerName (read-only)
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Enabled (7)
First (read-only)
Focus (read-only)
FullName (read-only)
FullSize (read-only)
LeftBorder (read-only) (5.0)
Manager (read-only)
Name
Next (read-only)
Owner (read-only)
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Select (4.5)
Size
TopBorder (read-only) (5.0)
Visible

▪

Header properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
Color
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Enabled (7)
First (read-only)
Focus (read-only)
FullName (read-only)
FullSize (read-only)
Invisible
LeftBorder (read-only) (5.0)
Manager (read-only)
Name
Next (read-only)
NextTabStop (5.0)
Owner (read-only)
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Size
TopBorder (read-only) (5.0)
Translucent
Visible

-

Line properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Breakable
Class (read-only)
Color
ContainerName (read-only)
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Enabled (7)
End
First (read-only)
Focus (read-only)
FullName (read-only)
FullSize (read-only)
Invisible
LeftBorder (read-only) (5.0)
LineEnds
LineStyle
LineType
Manager (read-only)
Name
Next (read-only)
Owner (read-only)
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Select (4.5)
Size
Start
Thickness
TopBorder (read-only) (5.0)
Visible

▪

List properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
ContainerName (read-only)
DataSource
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Enabled (7)
First (read-only)
FitHeight
FitWidth
Focus (read-only)
FullName (read-only)
FullSize (read-only)
LeftBorder (read-only) (5.0)
List.Count
List.Selection
List.Value
Manager (read-only)
Name
Next (read-only)
Owner (read-only)
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Size
TopBorder (read-only) (5.0)
Visible
WideScrollBar (5.0)

▪

Multi-record properties

Arrived (read-only)
AutoAppend (4.5)
BlankRecord (read-only)
BottomBorder (read-only) (5.0)
Breakable
ByRows (5.0)
Class (read-only)
Color
Columnar (4.5)
ContainerName (read-only)
CurrentColumn (5.0)
CurrentRow (5.0)
Deleted (read-only)
Editing (4.5)
Enabled (7)
First (read-only)
FirstRow (read-only) (5.0)
FitHeight
FitWidth
FlyAway (read-only)
Focus (read-only)
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
Inserting (read-only)
LeftBorder (read-only) (5.0)
Locked (read-only)
Manager (read-only)
Name
NCols
Next (read-only)
NRecords (read-only)
NRows
Owner (read-only)
Pattern.Color
Pattern.Style
Position
Prev (read-only)
Readonly (read-only)
RecNo (read-only)
Refresh (read-only)

RightBorder (read-only) (5.0)
RowNo (read-only)
Scroll
Select (4.5)
SeqNo (read-only)
Size
TableName
TopBorder (read-only) (5.0)
Touched
Translucent
VerticalScrollBar
Visible
WideScrollBar (5.0)
Xseparation (5.0)
Yseparation (5.0)

▪

OLE properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
Enabled (7)
First (read-only)
Focus (read-only)
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
HorizontalScrollBar
LeftBorder (read-only) (5.0)
Magnification
Manager (read-only)
Name
Next (read-only)
NextTabStop (5.0)
Owner (read-only)
Position
Prev (read-only)
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Size
TopBorder (read-only) (5.0)
Value
VerticalScrollBar
Visible
WideScrollBar (5.0)

-

Page properties

Arrived (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
Color
ContainerName (read-only)
Enabled (7)
First (read-only)
Focus (read-only)
FullName (read-only)
FullSize (read-only)
LeftBorder (read-only) (5.0)
Manager (read-only)
Name
Next (read-only)
Owner (read-only)
Pattern.Color
Pattern.Style
Position (read-only)
PositionalOrder (5.0)
Prev (read-only)
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Size
TopBorder (read-only) (5.0)
Translucent
Visible

▪

Record properties

Arrived (read-only)
AutoAppend (4.5)
BlankRecord (read-only)
BottomBorder (read-only) (5.0)
Breakable
Class (read-only)
Color
ContainerName (read-only)
Deleted (read-only)
DeleteWhenEmpty (5.0)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
Editing (read-only) (4.5)
Enabled (7)
First (read-only)
FitHeight
FitWidth
FlyAway (read-only)
Focus (read-only)
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
Inserting (read-only)
LeftBorder (read-only) (5.0)
Locked (read-only)
Manager (read-only)
Name
Next (read-only)
NRecords (read-only)
Owner (read-only)
Pattern.Color
Pattern.Style
Position
Prev (read-only)
Readonly
RecNo (read-only)
Refresh (read-only)
RightBorder (read-only) (5.0)

RowNo (read-only)
Scroll
Select (4.5)
SeqNo (read-only)
Shrinkable
Size
TableName (read-only)
TopBorder (read-only) (5.0)
Touched
Translucent
Visible

■

TableFrame properties

Arrived (read-only)
AttachedHeader (read-only) (5.0)
AutoAppend (4.5)
BlankRecord (read-only)
BottomBorder (read-only) (5.0)
Breakable
Class (read-only)
Color
ColumnPosition (5.0)
ColumnWidth (5.0)
ContainerName (read-only)
CurrentColumn (5.0)
CurrentRow (5.0)
DeleteColumn (5.0)
Deleted (read-only)
DeleteWhenEmpty (5.0)
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
Editing (read-only) (4.5)
Enabled (7)
First (read-only)
FirstRow (read-only) (5.0)
FitHeight
FitWidth
FlyAway (read-only)
Focus (read-only)
FullName (read-only)
FullSize (read-only)
Grid.Color
Grid.GridStyle
Grid.RecordDivider
Header (read-only) (5.0)
HorizontalScrollBar
InsertColumn (5.0)
Inserting (read-only)
LeftBorder (read-only) (5.0)
Locked (read-only)
Manager (read-only)
Name
NCols
Next (read-only)

NextTabStop (5.0)
NRecords (read-only)
NRows
Owner (read-only)
Pattern.Color
Pattern.Style
PersistView (read-only)
Position
Prev (read-only)
Readonly (read-only)
RecNo (read-only)
Refresh (read-only)
RepeatHeader (5.0)
RightBorder (read-only) (5.0)
RowNo (read-only)
Scroll
Select (4.5)
SeqNo (read-only)
Size
TableName
TopBorder (read-only) (5.0)
Touched
Translucent
VerticalScrollBar
Visible
WideScrollBar (5.0)

▪

TableView properties

Arrived (read-only)
BlankRecord (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
Color
ContainerName (read-only)
CurrentRecordMarker.Color
CurrentRecordMarker.LineStyle
CurrentRecordMarker.Show
Deleted (read-only)
Editing (read-only) (4.5)
Enabled (7)
FieldName (read-only)
FieldNo (read-only)
FieldView (read-only)
Focus (read-only)
FullName (read-only)
FullSize (read-only)
GridLines.Color
GridLines.ColumnLines
GridLines.HeadingLines
GridLines.LineStyle
GridLines.QueryLook (5.0)
GridLines.RowLines
GridLines.Spacing
HeadingHeight (5.0)
Inserting (read-only)
LeftBorder (read-only) (5.0)
Locked (read-only)
Manager (read-only)
MemoView (read-only)
Name (read-only)
NCols (read-only)
NRecords (read-only)
NRows (read-only)
Owner (read-only)
PersistView (read-only)
Position
Readonly (read-only)
RecNo (read-only)
Refresh (read-only)
RightBorder (read-only) (5.0)
RowHeight (5.0)

RowNo
Scroll
Select (4.5)
SeqNo (read-only)
Size
TableName (read-only)
TopBorder (read-only) (5.0)
Touched (read-only)

▪

Text properties

Alignment
Arrived (read-only)
AvgCharSize (read-only) (5.0)
BottomBorder (read-only) (5.0)
Breakable
Class (read-only)
Color
ContainerName (read-only)
CursorColumn
CursorLine
CursorPos
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
DesignSizing
Enabled (7)
First (read-only)
FitHeight
FitWidth
Focus (read-only)
Font.Color
Font.Size
Font.Style
Font.Typeface
Frame.Color
Frame.Style
Frame.Thickness
FullName (read-only)
FullSize (read-only)
HorizontalScrollBar
InsertField (5.0)
LeftBorder (read-only) (5.0)
LineSpacing
Manager (read-only)
MarkerPos
Name
Next (read-only)
OverStrike
Owner (read-only)
Pattern.Color
Pattern.Style

Position
Prev (read-only)
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
SelectedText
Size
Text
TopBorder (read-only) (5.0)
TopLine
Translucent
Value
VerticalScrollBar
Visible
WideScrollBar (5.0)
WordWrap

▪

TVData properties

Alignment
Arrived (read-only)
BlankRecord (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
Color
CompleteDisplay
ContainerName (read-only)
Default (read-only)
Deleted (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
Editing (read-only) (4.5)
Enabled (7)
FieldName (read-only)
FieldNo (read-only)
FieldRights (read-only)
FieldSize (read-only)
FieldType (read-only)
FieldUnits2 (read-only)
First (read-only)
Focus (read-only)
Font.Color
Font.Size
Font.Style
Font.Typeface
Format.DateFormat
Format.LogicalFormat
Format.NumberFormat
Format.TimeFormat
Format.TimeStampFormat
FullName (read-only)
FullSize (read-only)
IndexField (read-only)
Inserting
KeyField (read-only)
LeftBorder (read-only) (5.0)
Locked (read-only)
LookupTable (read-only)
LookupType (read-only)

Magnification
Manager (read-only)
Maximum (read-only)
Minimum (read-only)
Name
NextTabStop (5.0)
NRecords (read-only)
Owner (read-only)
Picture (read-only)
Position
RecNo (read-only)
Refresh (read-only)
Required (read-only)
RightBorder (read-only) (5.0)
RowNo (read-only)
Scroll
Select (4.5)
SeqNo (read-only)
Size
TableName (read-only)
TopBorder (read-only) (5.0)
Touched (read-only)
Value
Width (5.0)

▪

TVHeading properties

Alignment
Arrived (read-only)
BottomBorder (read-only) (5.0)
Class (read-only)
Color
ContainerName (read-only)
Design.ContainObjects
Design.PinHorizontal
Design.PinVertical
Design.Selectable (4.5)
Design.SizeToFit
Enabled (7)
Focus (read-only)
Font.Color
Font.Size
Font.Style
Font.Typeface
FullName (read-only)
FullSize (read-only)
LeftBorder (read-only) (5.0)
Manager (read-only)
Name
Owner (read-only)
Position
RightBorder (read-only) (5.0)
Scroll
Select (4.5)
Size
TopBorder (read-only) (5.0)

■

List of built-in event methods

Changes See also ■

The following table lists built-in event methods for internal and external events, and the special built-in event methods. Choose one of the following methods for more information:

Internal	External	Special
arrive	action	pushButton
canArrive	error	changeValue
canDepart	init	newValue
close	keyChar	
depart	keyPhysical	
mouseEnter	menuAction	
mouseExit	mouseClick	
open	mouseDouble	
removeFocus	mouseDown	
setFocus	mouseMove	
timer	mouseRightDouble	
	mouseRightDown	
	mouseRightUp	
	mouseUp	
	status	

Changes to Built-in event methods
The init method is new in version 7.

■

About built-in event methods

See also Built-in event methods

Every object in a form (and the form itself) has built-in event methods for handling events. Built-in event methods have the same names as the events that trigger them. For example, changing a value trigger's an object's built-in **changeValue** method, pressing the mouse button triggers the built-in **mouseDown** method, and releasing the mouse button triggers the built-in **mouseUp** method. The behavior of an object is simply the combined effects of its built-in event methods.

There are three kinds of built-in event methods in ObjectPAL:

- Built-in event methods for internal events
- Built-in event methods for external events
- Special built-in event methods

Built-in event methods for internal events

Internal events are generated internally by ObjectPAL. Like all events, internal events go to the form first, which dispatches them to the target object. Internal events do not bubble up through the containership hierarchy.

Built-in event methods for external events

External events are typically generated by user actions, although they can also be generated by ObjectPAL statements. Processing for all external events begins with the form, which acts as a dispatcher. Any external event that can not be handled by an object bubbles up the containership hierarchy.

Note: Unless otherwise noted, the default behavior for most objects and built-in event methods is to pass the event up the containership hierarchy.

Special built-in event methods

Special built-in event methods are additional methods built into a few specific objects.

Attaching code to built-in event methods

See also Built-in event methods

You can attach code to any built-in event method by opening an ObjectPAL Editor window and typing some code. For example, every design object has a built-in **mouseClick** method that performs some default behavior when you click that object. To change that behavior, right-click the object, choose Object Explorer from the menu, then choose the Events tabbed page, choose **mouseClick** from the list of event methods to open the **mouseClick** ObjectPAL Editor window. Type your code and save it. Now your code executes whenever this object's **mouseClick** method is called.

The built-in code executes too, *after* your code (just before the **endMethod** keyword).

The built-in code is implicit and executes automatically. But, if you want to change the default behavior for example, call the built-in code *before* your code, or block it from executing

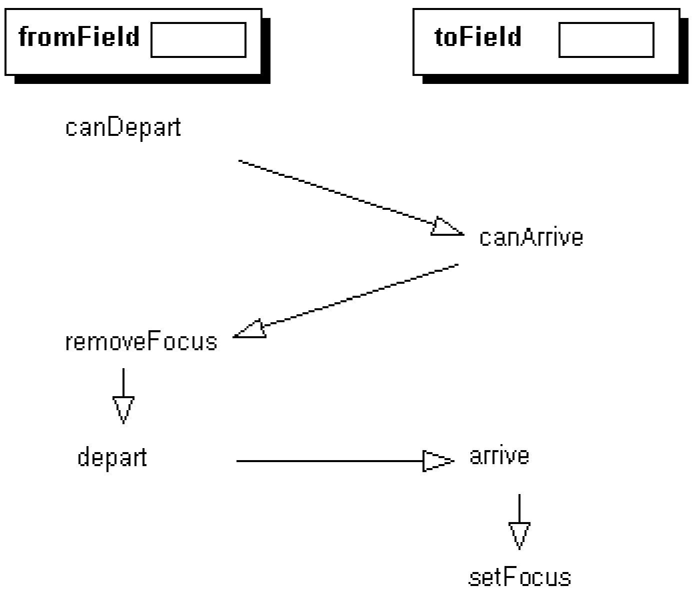
you can. First, though, you should understand the default behavior for each built-in event method.

As a general rule, if you attach code to an object's built-in **open** method, you should call **doDefault** before calling any other method or procedure. The call to **doDefault** executes the built-in code so you can be sure the object is completely opened and initialized.

Sequence of execution

See also Built-in event methods

The following figure shows the sequence in which built-in event methods execute when you move from one field object to another (for example, by pressing Tab). In this example, the field object you're moving from is named *fromField*, the one you're moving to is named *toField*.



▪

init method

Built-in event methods

The **init** built-in event method in Paradox 7 allows the ObjectPAL user to initialize a form before the open method is executed. You don't have to change existing code. All existing forms work as they did before.

Previous versions of Paradox required that initialization code be put in the **open** method. In Paradox 7, the **open** is initiated by code in the doDefault of **init**. The **init** method is called once when a form goes into run mode, that is, when the form is toggled from design to run or when the form is loaded directly into run. The call of the **init** method occurs at the same point that the call to **open** occurred previously. There is no change in the timing of the **open**.

The **init** method is supported only by the form object. No other UIObjects within the form have the **init** method. Unlike all other form-level methods, **init** does not have a "PreFilter" clause.

When you bring up a form in the method editor, you will see:

```
method init (var eventInfo Event)

endMethod
```

rather than the normal "`if eventInfo.isPreFilter()`", etc.

Example

To setup tables before the **open** method opens them, write the following code :

```
method init(var eventInfo Event)
createMyTables()
endMethod
```

To let the system open the tables before processing them, write this code :

```
method init(var eventInfo Event)
doDefault
tc.attach(tableFrame)     ; etc. etc.
endMethod
```

The **open** method is used for this sort of pre-processing. However, the PreFilter clause allows the **open** method to be called once for every object on the form, spending excessive time opening all the objects on the form. Also, naive users found themselves executing things like queries hundreds of times because they chose the wrong clause of the PreFilter.

The **init** method can stop the form from executing (as the **open** method can) by doing:

```
eventInfo.setErrorCode(1)       ; any nonzero error code will work
```

The **init** method calls the **open** method, so any error code returned from the **open** method propagates to **init**. Any forms that return error codes in their **open** continue to stop the form from executing.

▪

open

Built-in event methods

open is called once for every object on the form, starting from the form and working down each container in turn. For every object, the default code for the **open** method calls the **open** method for each of its child objects (that is, the objects one level below it in the containership hierarchy). In other words, by default, the form's **open** calls the **open** for each page in the form, and each page's **open** calls **open** for each object on the page, and so on.

Note: The form's **open** method opens all tables in the form's data model *before* any other objects are opened. If aliases are required to open any of the tables, specify them before executing the default code for **open**.

An error from any object prevents a form from opening. Also, explicitly setting an error code in an **open** method's event packet prevents a form from opening.

As a general rule, if you attach code to an object's built-in **open** method, you should call **doDefault** before calling any other method or procedure. The call to **doDefault** executes the built-in code so you can be sure the object is completely opened and initialized.

Note: Paradox compiles out of date forms (forms created in an earlier version of Paradox) before opening them. In this case, tables in the form's data model are opened before any ObjectPAL code executes.

-

close

Built-in event methods

close is called once for every object on a form being closed. By default, a form's **close** method closes all tables attached to the form.

▪

canArrive

Built-in event methods

canArrive is called when moving to an object. It asks whether the object (usually a field object) can be made active. Paradox calls this method by working through the object's containers, triggering **canArrive** for each object until it reaches the object itself. At any level of the containership, an error denies permission, blocking the move.

By default, Paradox blocks **canArrive** for objects that are not tab stops, and for a crosstab object if the target is not a cell.

▪

arrive

Built-in event methods

arrive is called only after the target and its containers have allowed a **canArrive**. As with **canArrive**, Paradox calls **arrive** on the target object's containers, finishing at the target. Pages, table frames, and multi-record objects all move to the first tab stop object they contain if they are the final destination of the move.

A successful **arrive** on a record or a field object makes it current (and opens an editing window for the field object, if appropriate).

arrive can have further effects on a field object, depending on its display type. If it's a drop-down edit list, focus moves to the list. If it's a radio button, focus moves to the first button.

▪

setFocus

Built-in event methods

setFocus is called after a successful **arrive** or when focus is returned to the form after going away to another window. This method is called for each of the active object's containers, starting with the outermost container, before it is called for the active object itself.

On an edit field, the default code for **setFocus** highlights the current selection and starts blinking the insertion point. At this time, also, the object's Focus property is set to True, and the form displays a status message reporting the number of the current record and the total number of records.

When a button gets focus, a rectangle displays around the label.

▪

canDepart

Built-in event methods

canDepart is called when trying to move off any object. This is the place to put code to block a departure: any error blocks the move. Field objects try to post their contents (triggering **changeValue**), and record objects try to commit the current record if changes have been made. If the record is locked, the form calls **action(DataUnlockRecord)** or **action(DataPostRecord)**.

Switching to another window does not move off an object, it only changes focus. Use **setFocus** and **removeFocus** to respond to focus events.

▪

removeFocus

Built-in event methods

removeFocus removes the flashing insertion point and highlight (if appropriate) from a field object, and removes the rectangle from a button. The object's Focus property is set to False.

This method is called for the active object and all its containers, starting with the active object, when the user activates some other window or moves to some other object.

depart

Built-in event methods

depart is called after all containers of the current object have granted permission to leave the field via **canDepart** and **removeFocus.** Use **canDepart** to block a move; **depart** is reserved for closing edit regions, repainting, and performing general cleanup.

▪

mouseEnter

Built-in event methods

mouseEnter is called whenever the pointer crosses into an object. It is called only on the transition into the object, not on every move across it.

By default, field objects set the pointer to the I-beam, and form, page, and button objects set it to an arrow.

If a button was the last object to receive a click and the mouse button is still down, the button's value toggles between True and False.

▪

mouseExit

Built-in event methods

mouseExit is called when the pointer leaves an object. An object that sets the shape of the pointer on **mouseEnter** sets it to an arrow in **mouseExit**.

If a button was the last object to receive a click and the mouse button is still down, its value toggles between True and False.

▪

timer

Built-in event methods

timer is called each time a timer interval elapses. Use the UIObject method **setTimer** to set timer intervals.

▪

mouseDown

Built-in event methods

mouseDown is called when the logical left mouse button is pressed. The event packet for this method contains the mouse coordinates in twips, relative to the target object.

An active field object enters Field View, positions the insertion point and begins a drag-selection.

When the form handles a **mouseDown**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseDown** to the object the mouse was pointing at.

This method toggles a button's value between True and False.

▪

mouseUp

Built-in event methods

mouseUp is called when the left mouse button is released. It's called for the last object to receive a **mouseDown**, even if the button is released outside the object, so the object always sees the **mouseDown**/**mouseUp** pair.

An active field object ends the selection; a field object that is not active performs a **self.moveTo()**.

When the form handles a **mouseUp**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseUp** to the object that received the last click.

This method toggles a button's value between True and False. If **mouseUp** is called and the pointer is inside a button, it triggers the button's **pushButton** method. For any other type of object, it triggers the object's built-in **mouseClick** method.

▪

mouseDouble

Built-in event methods

mouseDouble is called when the left mouse button is double-clicked. In Windows convention, a **mouseDown** and **mouseUp** are delivered first.

Field objects enter field view on a **mouseDouble**.

When the form handles a **mouseUp**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseDouble** to the object that received the last click.

▪

mouseRightDown

Built-in event methods

mouseRightDown is called when the logical left mouse button is pressed. The event packet for this method contains the mouse coordinates in twips, relative to the target object.

An active field object enters Field View, positions the insertion point and begins a drag-selection.

When the form handles a **mouseRightDown**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseRightDown** to the object the mouse was pointing at.

This method toggles a button's value between True and False.

▪

mouseRightUp

Built-in event methods

mouseRightUp is called when the left mouse button is released. It's called for the last object to receive a **mouseRightDown**, even if the button is released outside the object, so the object always sees the **mouseRightDown**/**mouseRightUp** pair.

When the form handles a **mouseRightUp**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseRightUp** to the object that received the last click.

This method toggles a button's value between True and False. If **mouseRightUp** is called and the pointer is inside a button, it triggers the button's **pushButton** method.

In addition, the following field objects display a pop-up menu when they get a **mouseRightUp**: formatted memo, graphic, OLE, and unbound (undefined).

▪

mouseRightDouble

Built-in event methods

mouseRightDouble is called when the left mouse button is double-clicked. In Windows convention, a **mouseRightDown** and **mouseRightUp** are delivered first.

When the form handles a **mouseRightUp**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseRightDouble** to the object that received the last click.

▪

mouseClick

Built-in event methods

mouseClick is called when the logical left mouse button is pressed and released when the pointer is inside the boundaries of an object. **mouseClick** is not called if the user moves the mouse outside the object before releasing the mouse button.

mouseClick is actually generated from within the **mouseUp** method.

The mapping from **mouseUp** to **mouseClick** happens at the first container object which does something with **mouseUp**. In other words, **mouseUp** in a box bubbles to its container, and so forth. Only the field object, the button object, the list object, and the form intercept **mouseUp**, so those are the spots where the translation occurs. If you click on an object inside a button, that object's **mouseClick** will be called. If that object allows the default (bubbling), then the button will ultimately receive that **mouseClick**, triggering its own **pushButton** method. In this way, you can have code execute on objects you click inside the button, but still trigger the button's **pushButton** method. Setting the error code in **mouseUp** will inhibit the **mouseClick**, and setting the error code in **mouseClick** will inhibit a **pushButton**.

▪

mouseMove

Built-in event methods

mouseMove is called whenever the mouse moves within an object. The event packet for this method contains the coordinates of the pointer (in twips).

An active edit field checks the state of *Shift*. If *Shift* is down (physically or logically), the selection is extended. An active graphic field scrolls the graphic, if necessary.

When you press and hold the mouse button inside an object, **mouseMove** is called until you release it, even when the pointer moves outside the object.

When the form handles a **mouseMove**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse.

▪

keyPhysical

Built-in event methods

keyPhysical is called when a key is pressed and each time a key autorepeats. It goes to the form first, and the form dispatches it to the active object. Then, the active object's built-in code sorts out whether a keystroke represents an action or a character to display in a field, and calls the appropriate **action** or **keyChar** method.

For example, suppose a field object within a table object is active, and the user presses Enter. The keystroke triggers **keyPhysical**, which interprets it as a request for an action and maps it to **action(FieldEnter)**, which in turn triggers the built-in **action** method. In contrast, when the user presses K, the keystroke triggers **keyPhysical**, which interprets it as a character and triggers the **keyChar** method.

Technically, the event packet for **keyPhysical** contains information from the Windows WM_KEYDOWN message and an optional WM_CHAR. Therefore, it provides both the virtual key code as well as the ANSI character. Although you can attach code to this method, it's best if you don't, except for special character handling. For example, if you want to intercept the F9 key explicitly (rather than handle the eventual **action(DataToggleEdit)**) this is the method to use.

▪

keyChar

Built-in event methods

keyChar is called when a **keyPhysical** is not interpreted by Paradox. That is, **keyChar** gets called for every **keyPhysical** that does not map to an action (see the **action** built-in event method). It goes to the form first, and the form dispatches it to the active object.

When editing a field, the system locks the record before inserting the first character.

If a button receives a **keyChar** equal to pressing Spacebar (for example, **keyChar(VK_SPACE)**), it calls the button's **pushButton** method.

▪

menuAction

Built-in event methods

menuAction is called whenever the user chooses an item from a menu (or clicks a Toolbar button that executes a menu action). It goes to the form first, and the form dispatches it to the active object.

▪

error

Built-in event methods

error is called when an error occurs. By default, objects (except the form) pass errors to their containers. You can attach code to the default method to make an object handle an error, or pass it, or both. For more information about errors and error handling, refer to the *Guide to ObjectPAL.*

▪

status

Built-in event methods

status is called before a message is displayed in one of the areas in the status bar. Among other things, you can attach code to the built-in **status** method to redirect messages to other areas, or to change the text of the message.

action

Built-in event methods

action is called when **keyPhysical** maps some keystroke to an action, when **menuAction** maps a menu choice to an action, or when other methods want some action performed. It goes to the form first, and the form dispatches it to the target object. For example, by default, pressing F2 in a field triggers **action(EditToggleFieldView)** after its **keyPhysical** method executes, and clicking the Forward navigation button triggers **action(DataNextRecord)** after its **menuAction** method executes.

action is very important method. It is discussed in detail in the *Guide to ObjectPAL*.

-

pushButton

Built-in event methods

Defined only for button objects and fields displayed as list boxes, **pushButton** is called when the user releases the mouse on a button. This method is actually not called directly by the form, but by the default **mouseUp** method for buttons. You can also call this method directly to accomplish the normal action associated with pressing a button object.

By default, buttons change their appearance when clicked. For example, a push button pushes in and pops out, check boxes check or uncheck, and radio buttons push in or pop out. Focus moves to a button when you click it (unless its Tab Stop property is set to False).

When a button's Tab Stop property is set to True and the button is the active object, there are two ways to trigger its **pushButton** method using the keyboard:

- Press Spacebar. The button keeps focus.
- Press Enter. Focus moves to the next object in the tab order.

▪

changeValue

Built-in event methods

Defined only for field objects, **changeValue** asks for permission to change the value of a field. It is called before the value is stored, so you can check the value and do something with it (such as performing additional validity checks). It is not called when someone changes a value across a network or through a lookup with fill-all-corresponding.

The following statement triggers Quant's **changeValue** method, even if Quant is already 10, and it triggers it immediately, without waiting for the method to finish executing.

```
Quant = 10
```

Using changeValue with field objects

The built-in code for **changeValue** commits the changes to the value; until the built-in code executes, Paradox uses the old, unchanged value. For example, suppose a field object has a value of 10, and you move to it and enter a value of 23. Then, when you move off that field object, you trigger its **changeValue** method, to which the following code has been attached:

```
method changeValue(var eventInfo ValueEvent)
   msgInfo("before the change", self.value) ; displays 10
   doDefault
   msgInfo("after the change", self.value) ; displays 23
endMethod
```

When this method executes, the first dialog box displays the old value, 10, because the built-in code has not yet executed. Then, the call to **doDefault** executes the built-in code, which commits the changed value, and the second dialog box displays the changed value.

Within an object's **changeValue** method, you can use the ValueEvent::**newValue** method (not the same as the built-in **newValue** event method) to get the incoming value before the built-in code executes. For example, suppose as before that a field object has a value of 10, and you move to it, enter a value of 23, and trigger its **changeValue** method, to which the following code has been attached:

```
method changeValue(var eventInfo ValueEvent)
   msgInfo("before the change", self.value) ; Displays 10
   msgInfo("the new (incoming) value",
           eventInfo.newValue())             ; Displays 23
   doDefault
   msgInfo("after the change", self.value)  ; Displays 23
endMethod
```

The first dialog box displays the old, unchanged value. The second dialog box calls **eventInfo.newValue** to display the new, incoming value, but that value has not yet been committed. The call to **doDefault** executes the built-in code, which commits the change, and the third dialog box displays the changed value.

You can block an attempted change to a value by calling **eventInfo.setErrorCode** and specifying a non-zero value. You can also test (or alter) the incoming value (for example, to round up to the nearest dollar amount), using **eventInfo.setNewValue**. For example, the following code is attached to a field object's built-in **changeValue** method. It executes when the user changes the field's value and attempts to post (commit) the change. The code calls **eventInfo.newValue** to get the user's value before it is posted. If it is greater than 50, the call to **eventInfo.setErrorCode** prevents the user from posting the value or leaving the field.

```
method changeValue(var eventInfo ValueEvent)
   var
      atNewVal   AnyType
   endVar

   atNewVal = eventInfo.newValue()
   if atNewVal > 50 then
      eventInfo.setErrorCode(CanNotDepart)
      message("Enter a value less than 50.")
   endIf
endMethod
```

▪

newValue

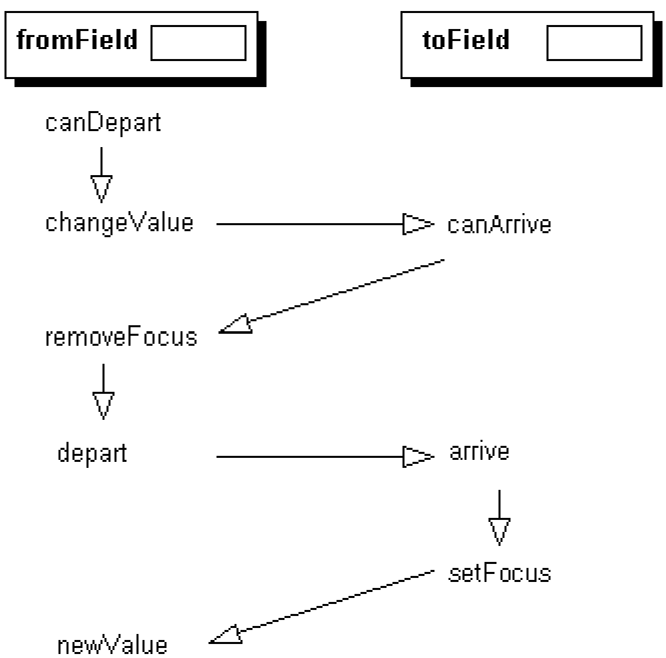
Built-in event methods

Defined only for field objects, **newValue** is called to report that a field object has a new value. For example, when scrolling through a table, moving to the next record triggers **newValue**. When a field is displayed as radio buttons, **newValue** is called when you click a button. Note that simply typing into a field object does not trigger **newValue** (but does set the Touched property to True). In any case, **changeValue** is not called until you try to move off the field object or otherwise try to commit changes. Also, a form's **open** method triggers **newValue** for each field object in the form.

Labeled and unlabeled field objects

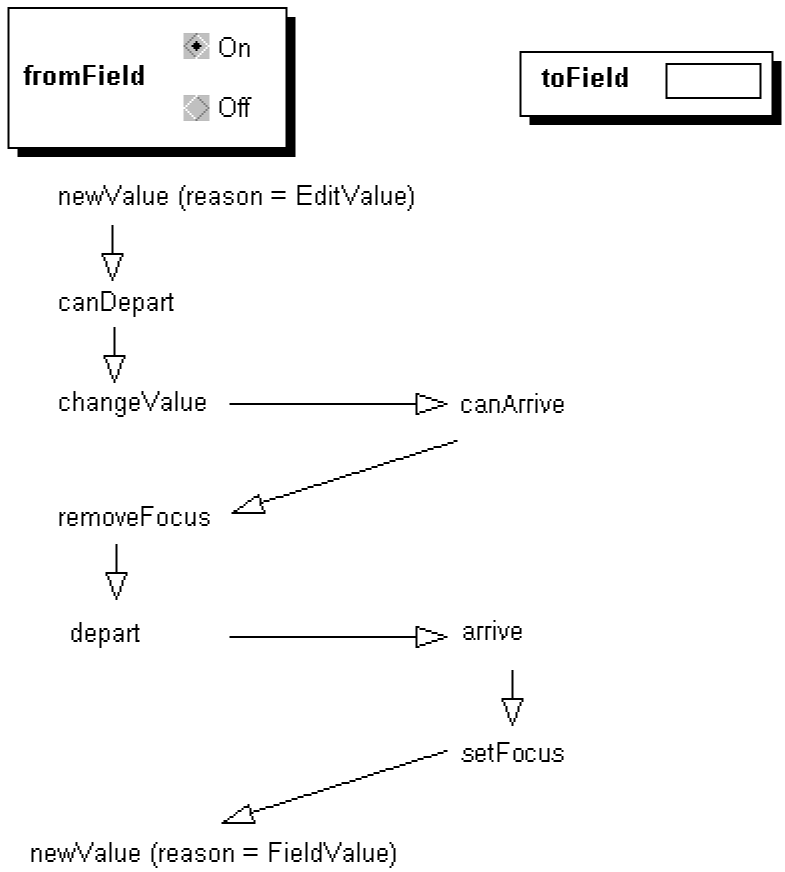
The following figure shows the sequence in which built-in event methods execute when you move from one field object (labeled or unlabeled) to another after editing a value. The field object you're moving from is named fromField; the one you're moving to is named toField.

Note: **newValue** is called when Paradox needs to refresh the value of the field object (in this case, to update the display). Calls to **newValue** are not part of the **canDepart** sequence. However, **newValue** is not necessarily the last method to execute.



Radio buttons and lists

The following figure shows the sequence in which built-in event methods execute when you move from one field object (radio buttons, list, or drop-down edit list) to another after editing a value. The sequence is the same as for regular fields, except for an additional **newValue** call when you choose a radio button or a list item. The table also gives the ValueReason constant for each **newValue**. The field object you're moving from is named fromField; the one you're moving to is named toField.


fromField
On
Off
toField
newValue (reason = EditValue)
canDepart
changeValue
canArrive
removeFocus
depart
arrive
setFocus
newValue (reason = FieldValue)

ActionEvent type

See also

ActionEvents are generated primarily by editing and navigating in a table. The ActionEvent type includes several methods defined for the Event type.

The only built-in event method that is triggered by an ActionEvent is **action**. Typically, when you work with ActionEvents, you'll also work with ObjectPAL action constants. For example, to prevent users from editing a table, you could do something like this:

```
; thisTableFrame::action
method action(var eventInfo ActionEvent)
; If the user tries to switch to Edit mode, display a dialog box
if eventInfo.id() = DataBeginEdit then  ; DataBeginEdit is a constant.
  msgStop("Stop", "You can't edit this table.")
  eventInfo.setErrorCode(UserError) ; UserError is a constant.
endif
endMethod
```

The action constants are grouped as follows:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use user-defined action constants.

For more information and examples, refer to the *Guide to ObjectPAL.*

The ActionEvent type includes several derived methods from the Event type.

Methods for the ActionEvent type

Event	**ActionEvent**
errorCode	**actionClass**
getTarget	**id**
isFirstTime	**setId**
isPreFilter	
isTargetSelf	
reason	
setErrorCode	
setReason	

User-defined action constants

ActionEvent Type

You can define your own action constants, but you must keep them within a specific range. Because this range is subject to change in future versions of Paradox, ObjectPAL provides the IdRanges constants UserAction and UserActionMax to represent the minimum and maximum values allowed.

For example, suppose that you want to define two action constants, ThisAction and ThatAction. In a Const window, define values for your custom constants as follows:

```
Const
   ThisAction = 1
   ThatAction = 2
EndConst
```

Then, to use one of these constants, add it to UserAction. For example,

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = UserAction + ThisAction then
      doSomething()
   endIf
endMethod
```

By adding UserAction to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, use the value of UserActionMax. One way to check the value is with a **message** statement:

```
message(UserActionMax)
```

In this version of Paradox, the difference between UserAction and UserActionMax is 2047. That means the largest value you can use for an action constant is UserAction + 2047.

▪

actionClass method

See also Example ActionEvent Type

Returns the class number of an ActionEvent.

Syntax

actionClass ( ) SmallInt

Description

actionClass returns an integer value representing an ActionEvent class. Use ActionClasses constants to find out which class the integer value represents.

actionClass example

The following example uses **actionClass** to prevent the user from making any changes to a field object. This code is attached to a field's built-in **action** method. See **id** for an example that traps for the user entering Edit mode.

```
; Site_Notes::action
method action(var eventInfo ActionEvent)
; check for any attempt to edit, and block it
if eventInfo.actionClass() = EditAction then
  ; allow user to start and end field view
  if NOT (eventInfo.id() = EditEnterFieldView) AND
     NOT (eventInfo.id() = EditToggleFieldView) AND
     NOT (eventInfo.id() = EditExitFieldView) then
     eventInfo.setErrorCode(UserError)
     beep()
     message("Sorry. Can't make changes to this field.")
  endif
endif
endMethod
```

id method

See also Example ActionEvent Type
Beginner

Returns the ID number of an ActionEvent.

Syntax

```
id ( ) SmallInt
```

Description

id returns the ID number of an ActionEvent. ObjectPAL defines constants for these ID numbers (for example, DataBeginEdit), so you don't have to remember numeric values.

The action constants are grouped as follows:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use user-defined action constants.

id example

The following example uses **id** to prevent the user from entering Edit mode on a form. This code is attached to a form's built-in **action** method:

```
; thisForm::action
method action(var eventInfo ActionEvent)
if eventInfo.isPreFilter() then
    ; code here executes for each object in form
else
    ; code here executes just for form itself
    if eventInfo.id() = DataBeginEdit then
      eventInfo.setErrorCode(UserError)     ; don't start Edit mode
      msgStop("Sorry", "View only - can't edit this form")
    endif
endif
endMethod
```

▪

setId method

See also Example ActionEvent Type

Specifies an ActionEvent.

Syntax

`setId ( const actionId SmallInt )`

Description

setId specifies the ActionEvent represented by the constant *actionId*. ObjectPAL provides constants (for example, DataNextRecord) for ActionEvents so you don't have to remember numeric values.

The action constants grouped as follows:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use user-defined action constants.

▪

setId example

In the following example, the Toolbar record-movement buttons are remapped to move within a memo field. Assume that a form contains a multi-record object, *SITES*, bound to the *Sites* table. The following code is attached to the **action** method for the *Site_Notes* field object:

```
; Site_Notes::action
method action(var eventInfo ActionEvent)
var
  actID  SmallInt
endVar
; if Site Notes is in Field View, remap record-movement
; actions to move within the memo field
if self.Editing then
   actID = eventInfo.id()
   switch
      case actID = DataPriorRecord  : eventInfo.setId(MoveBeginLine)
      case actID = DataNextRecord   : eventInfo.setID(MoveEndLine)
      case actID = DataFastBackward : eventInfo.setID(MoveBegin)
      case actID = DataFastForward  : eventInfo.setID(MoveEnd)
      case actID = DataBegin        : eventInfo.setID(FieldBackward)
      case actID = DataEnd          : eventInfo.setID(FieldForward)
   endswitch
endif
endMethod
```

AnyType type

Changes

An AnyType variable can store any one of the data types listed in the following table.

Type	Description
AnyType	A catch-all for basic data types
Binary	Machine-readable data
Currency	Used to manipulate currency values
Date	Calendar data
DateTime	Calendar and clock data combined
Graphic	A bitmap image
Logical	True or False
LongInt	Used to represent relatively large integer values
Memo	Holds lots of text
Number	Floating-point values
OLE	A link to another application
Point	Information about a location on the screen
SmallInt	Used to represent relatively small integer values
String	Letters
Time	Clock data

An AnyType can never be a complex type such as TCursor or TextStream. An AnyType variable inherits characteristics from the value assigned to it. That is, it behaves like a String when assigned a String value, like a Number when assigned a Number value, and so forth.

AnyType data objects are included in ObjectPAL so you can use variables for basic data types without declaring them first. (Remember that it's better to declare variables whenever possible.)

Methods for the AnyType type

AnyType
blank
dataType
fromHex
isAssigned
isBlank
isFixedType
toHex
unAssign
view

Changes to AnyType type methods

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
fromHex	(None)
toHex	

▪

blank method/procedure

See also Example AnyType Type
Beginner

Returns a blank value.

Syntax

```
1. ( Method ) blank ( )
2. ( Procedure ) blank ( ) AnyType
```

Description

blank generates a blank value to assign to a variable or field. A blank value is not the same as a numeric value of zero, but you can use Session type method **blankAsZero** to treat blank values as zeros in certain calculations. You can use the Session type method **isBlankZero** to find out whether Blank=Zero is on or off.

▪

blank example

The following example assumes that a form has a table frame bound to the *Lineitem* table, and a button named *thisButton*. When a user presses *thisButton*, the code scans the Qty field in *Lineitem*, and replaces non-blank values with blank values. This code is attached to the built-in **pushButton** method for *thisButton*:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

if tc.attach(LINEITEM) then                  ; attach tc to table frame
  tc.edit()                                  ; edit the table frame
  scan tc for tc.Qty.isBlank() = False :     ; look for non-blank Qty fields
    tc.Qty.blank()                           ; put a blank value in Qty
  endScan
  tc.endEdit()                               ; end edit mode
endif

endMethod
```

dataType method

See also   Example   AnyType Type

Returns a string representing the data type of a variable.

Syntax

```
dataType ( ) String
```

Description

dataType returns a string representing the data type of a variable or expression: Binary, Currency, Date, DateTime, Graphic, Logical, LongInt, Memo, Number, OLE, Point, SmallInt, String, or Time. In comparison statements, you need to use one of the string values shown here. For example, the following is coded incorrectly because it compares "String" with "string".

```
var s AnyType endVar
s = "This is a String data type."
msgInfo("Test", s.dataType() = "string")  ; displays False - should use
"String"
```

Note: This method works for all ObjectPAL types, not just AnyType.

■

dataType example

The following example assumes a form has a button and a graphic field named *bmpField*. The following code loads a DynArray with several different types of data, then uses **dataType** to display the data type of each value in the DynArray. This code is attached to the button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  mixedTypes DynArray[] AnyType
endVar

mixedTypes["Make"]  = "Ford"          ; String
mixedTypes["Model"] = "Cobra"         ; String
mixedTypes["Year"]  = 1969            ; SmallInt (not Date)
mixedTypes["Color"] = Black           ; LongInt - used here as a constant
mixedTypes["Photo"] = bmpField.value  ; Graphic

forEach element in mixedTypes    ; display a message for each element

  msgInfo("dataType(" + element + ")", dataType(mixedTypes[element]))

endForEach

endMethod
```

■

fromHex procedure

See also Example AnyType Type

Converts a hexadecimal number to a decimal number.

Syntax

```
fromHex ( const value String ) LongInt
```

Description

fromHex converts a hexadecimal number *value* to a decimal number. The value must range from 0x00000000 to 0xFFFFFFFF.

fromHex example

In the following example, the **pushButton** method for a button named *convertHex* converts a hexadecimal string variable to a decimal number.

```
; convertHex::pushButton
method pushButton(var eventInfo Event)
  var
     s  String
     li LongInt
  endVar

  ;Hexadecimal value to convert.
  s = "0x0756B5B3"
  s.view("Hex value to convert")
  li = fromHex(s)
  li.view("0x0756B5B3") ; Displays 123123123.
endMethod
```

■

isAssigned method

See also Example AnyType Type

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

isAssigned returns True if the variable has been assigned a value; otherwise, it returns False.

Note: This method works for all ObjectPAL types, not just AnyType.

isAssigned example

The following example uses **isAssigned** to test the value of *i* before assigning a value to it. If *i* has been assigned, this code increments *i* by one. The following code goes in a button's Var window:

```
; thisButton::var
var
  i SmallInt
endVar
```

This code is attached to the button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

if i.isAssigned() then ; if i has a value
  i = i + 1            ; increment i
else
  i = 1                ; otherwise, initialize i to 1
endif
                       ; now show the value of i
message("The value of i is : " + String(i))

endMethod
```

▪

isBlank method

See also　　Example　　AnyType Type
Beginner

Reports whether an expression has a blank value.

Syntax

`isBlank ( ) Logical`

Description

isBlank returns True if the expression has a blank value; otherwise, it returns False. Blank string values are denoted by "". Other blank values can be generated using **blank**. Note that blank values are not the same as 0, spaces ("   "), or unassigned values.

isBlank example

The following code (attached to a button's **pushButton** method) uses **isBlank** to test various values, and displays the results in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

msgInfo("Is the empty string blank?", isBlank(""))       ; True
msgInfo("Is a string of spaces blank?", isBlank("   ")) ; False
msgInfo("Is 5 a blank?", isBlank(5))                     ; False
msgInfo("Is blank blank?", isBlank(blank()))             ; True

endMethod
```

▪

isFixedType method

See also Example AnyType Type

Reports whether a variable's data type has been explicitly declared.

Syntax

```
isFixedType ( ) Logical
```

Description

isFixedType returns True if the variable has been declared using a **var...Endvar** block; otherwise, it returns False.

isFixedType example

The following code demonstrates when **isFixedType** returns True. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  x SmallInt                   ; declare x
endVar

message(x.isFixedType())       ; displays True
sleep(2000)

testMe = 4                     ; testMe was not declared
message(testMe.isFixedType()) ; displays False

endMethod
```

▪

toHex procedure

See also Example AnyType Type

Converts a decimal number to a hexadecimal number.

Syntax

```
toHex ( const value LongInt ) String
```

Description

toHex converts a decimal number *value* to a hexadecimal number.

toHex example

In the following example, the **pushButton** method for a button named *convertDecimal* converts a long integer value to a hexadecimal string.

```
; convertDecimal::pushButton
method pushButton(var eventInfo Event)
  var
    s  String
    li LongInt
  endVar

  li = 123123123
  li.view("Value to convert")
  s = toHex(li)
  s.view("123123123") ; Displays 0x0756B5B3.
endMethod
```

▪

unAssign method

See also Example AnyType Type

Sets a variable's state to unAssigned.

Syntax

```
unAssign ( )
```

Description

unAssign sets a variable's state to unAssigned. The unAssigned state is not the same as a value of 0, nor is it the same as Blank.

unAssign example

The following example demonstrates **unAssign**. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  x AnyType
endVar

msgInfo("Is x assigned?", x.isAssigned()) ; displays False
x = 5
msgInfo("Is x assigned?", x.isAssigned()) ; displays True
x.unAssign()
msgInfo("Is x assigned?", x.isAssigned()) ; displays False

endMethod
```

view method

See also Example1 Example2 AnyType Type
Beginner

Displays in a dialog box the value of a variable.

Syntax

```
view ( [ const title String ] )
```

Description

view displays, in a modal dialog box, the value of a variable. ObjectPAL execution suspends until the user closes this dialog box. You have the option to specify, in *title*, a title for the dialog box. If you don't specify a title, the variable's data type appears.

The user can change the value displayed in a **view** dialog box, as long as the data type is not an Array, DynArray, or Record. **view** cannot display Binary, Graphic, Memo, or OLE AnyTypes. The following table summarizes AnyTypes that can be displayed, and those which the user can modify.

Type	Can be viewed	Can be modified
Binary	no	no
Currency	yes	yes
Date	yes	yes
DateTime	yes	yes
Graphic	no	no
Logical	yes	yes
LongInt	yes	yes
Memo	no	no
Number	yes	yes
OLE	no	no
Point	yes	yes
SmallInt	yes	yes
String	yes	yes
Time	yes	yes

▪

view example 1

The following example shows how to tell whether the user clicked the OK button or the Cancel button (or closed the **view** dialog box another way, for example, by pressing Esc). When the user clicks OK, the value displayed in the dialog box is assigned to the variable. If the user closes the dialog box any other way, the value is not assigned.

The following code assigns an initial value to the variable, then tests to see if the user has changed it. If the user has entered a valid value, the value is entered into the *ShipVia* field object in a form bound to the *Orders* table (assuming the form is in edit mode).

This code is attached to a field object's built-in **arrive** method:

```
; shipViaFld::pushButton
method arrive(var eventInfo MoveEvent)
   var
      stShipVia,
      stPrompt  String
   endVar

   stPrompt  = "Enter Express or Regular."
   stShipVia = stPrompt

   stShipVia.view("Ship via:")

   if stShipVia = stPrompt then
         ; User closed the dialog box without changing the value.
         return
   else
         ; User entered a value and clicked OK.
         if stShipVia = "Express" or
               stShipVia = "Regular" then
               orders.shipVia.Value = stShipVia ; Or self.Value = stShipVia
         else
               msgStop("Stop", stPrompt)
         endIf
   endIf
endMethod
```

▪

view example 2

The following example uses a **view** dialog box to prompt the user for a date. If the user enters a valid date, the code displays the day of the week for that date; otherwise, an error message is displayed.

```
; showDOW::pushButton
method pushButton(var eventInfo Event)
var
  theDate AnyType
  fullDays Array[7] String
endvar

fullDays[1] = "Sunday"
fullDays[2] = "Monday"
fullDays[3] = "Tuesday"
fullDays[4] = "Wednesday"
fullDays[5] = "Thursday"
fullDays[6] = "Friday"
fullDays[7] = "Saturday"

 ; initialize theDay variable
theDate = today()
 ; now show today's date in a dialog and prompt the user to enter a new date
theDate.view("Enter a Date")

 ; it's possible the user could enter an invalid date (like "Saturday")
 ; so this try..fail block attempts to convert theDate to a Date with
 ; dateVal() and if successful, displays the day of the week that
 ; theDate falls on
try
  msgInfo("Day of the week", String(theDate) + " falls on a\n" +
              fullDays[dowOrd(dateVal(theDate))])
onfail
  msgStop("Error!", theDate + " is not a valid date.")
endtry

endMethod
```

Application type

Changes

An Application variable provides a handle for working with the Desktop window of the current Paradox application. You can use an Application variable in your code to control the size, position, and appearance of the Desktop. Methods added in version 5.0 let you change the working directory and the private directory at run time.

Although you can have more than one application running at the same time, Application objects can't communicate or operate on each other. An Application variable refers to the current Paradox Desktop only; you can, however, use Session variables to open multiple channels to the database engine (see the Session type).

Since there can be only one current application, to get an application handle, you merely declare a variable of type Application. While an Application variable is in scope, it serves as a handle: you use that variable to access the methods in the Application type. For instance, in the following example, an Application variable called *thisApp* is declared, then used in the method's code.

```
; downSize::pushButton
method pushButton(var eventInfo Event)
var
  thisApp     Application
endVar
thisApp.maximize() ; Maximize the Desktop.
endMethod
```

The Application type consists of derived methods from the Form type.

Methods for the Application type

Form	Application
bringToTop	The Application type consists of derived methods from the Form type.
getPosition	
getTitle	
hide	
isMaximized	
isMinimized	
isVisible	
maximize	
minimize	
setIcon	
setPosition	
setTitle	
show	
windowClientHandle	
windowHandle	

Changes to Application type methods

Changes for version 7

The Form::setIcon method is new for version 7 and is a derived method of the Application type.

New
setIcon

Array type

An Array holds values (called *items* or *elements*) in *cells* similar to the way mail slots hold mail. An ObjectPAL array is one-dimensional, like a single row of slots, where each slot holds one item.

To use arrays in methods, you must declare them by specifying a name, size (number of items), and a data type for the items.

Note: In ObjectPAL, array items are counted beginning with 1, not with 0, as in some other languages.

Note: ObjectPAL also supports dynamic arrays. See the method and procedures for DynArray for more information.

The Array type includes several derived methods from the AnyType type.

Methods for the Array type

AnyType	**Array**
blank	**addLast**
dataType	**append**
isAssigned	**contains**
isBlank	**countOf**
isFixedType	**empty**
	exchange
	fill
	grow
	indexOf
	insert
	insertAfter
	insertBefore
	insertFirst
	isResizeable
	remove
	removeAllItems
	removeItem
	replaceItem
	setSize
	size
	view

▪

addLast method

See also Example Array Type

Inserts an item at the end of a resizeable array.

Syntax

addLast (const *value* AnyType)

Description

addLast inserts *value* after the last item in a resizeable array. The array grows, if necessary, to make room for the new item. If you need to add more than one element to an array, it is usually preferable to use **grow** or **setSize** to allocate more space in the array rather than several **addLast** statements. For example, the following code uses **addLast** in a **for** loop to add 10 new elements to the *ar* array. Note that this use of **addLast** forces ObjectPAL to re-allocate space in the array 10 times; once each cycle through the loop.

```
for i from 11 to 20
  ar.addLast(i * 10)
endfor
```

The following code accomplishes the same as the previous code, but executes faster because ObjectPAL allocates space only once:

```
ar.grow(10)    ; increase array size by 10 elements
for i from 11 to 20
  ar[i] = (i * 10)
endfor
```

▪

addLast example

The following example adds an element to a resizeable array each time *thisButton* is pressed. The **pushButton** method for *thisButton* increments the value of the newest element by 10 and displays the contents of the array in a **view** dialog box. The code immediately following goes in the Var window for *thisButton*:

```
; thisButton::Var
var
  ar Array[] SmallInt  ; declare ar as a resizeable array
  i  SmallInt          ; incrementing variable
endVar
```

The following code is attached to the built-in **pushButton** method for *thisButton*:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

                       ; initialize or increment i
i = iif(isAssigned(i), i + 10, 0)

if ar.size() = 0 then  ; true if this is the first time the button was
pressed
  ar.setSize(0)        ; initialize size
endif

ar.addLast(i)          ; add another element to ar, and assign
                       ; the new element with the value of i

  ; display size of array in the title, and the value of
  ; each element in a view dialog box
ar.view("Size of ar array is " + strVal(ar.size()))

endMethod
```

▪

append method

See also Example Array Type

Appends the contents of one array to another.

Syntax

append (**`const` ***newArray* `Array[ ] String` **)**

Description

append appends the items of *newArray* to a resizeable array. The array grows, if necessary, to make room for the added items.

▪

append example

The following code creates two resizeable arrays, *addMe* and *baseArray*, and loads them with numeric values. The following example demonstrates **append** by appending the *addMe* array to *baseArray*, then displays the results in a **view** dialog box. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   baseArray, addMe Array[] SmallInt
   i SmallInt
endVar

baseArray.setSize(3)
addMe.setSize(3)        ; now both arrays can store 3 values
for i from 1 to 3
   baseArray[i] = i     ; baseArray[1] = 1, [2] = 2, [3] = 3
   addMe[i] = (i + 3)   ;     addMe[1] = 4, [2] = 5, [3] = 6
endFor

baseArray.append(addMe) ; add the addMe array to baseArray
                        ; this grows baseArray to 6 elements

  ; now display the size of baseArray in the title of a view dialog
  ; and show baseArray elements within the dialog
baseArray.view("baseArray size: " + strVal(baseArray.size()))
endMethod
```

▪

contains method

[See also] [Example] [Array Type]

Searches the items of an array for a pattern of characters.

Syntax

contains **(** const ***value*** AnyType **)** Logical

Description

contains returns True if any item of an array exactly matches *value*; otherwise, it returns False.

▪

contains example

The following example defines and loads a resizeable array named *dogs* when a form opens. Once the form's **open** method loads the array with dog names, the code displays the contents of the array in a dialog box. A button on the form contains code that uses the **contains** method to search the array for a particular name. If **contains** doesn't find the name, the built-in **pushButton** method attached to the button uses **insertFirst** to add the name to the top of the array.

The following code is attached to the form's Var window:

```
; thisForm::Var
var
  dogs Array[] String   ; resizeable array
endVar
```

The following code is attached to the form's built-in **open** method:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

    dogs.setSize(4)      ; now dogs can store 4 values
    dogs[1] = "Bruno"    ; add some dog names
    dogs[2] = "Frodo"
    dogs[3] = "Yipper"
    dogs[4] = "Juneau"

      ; show the contents of the dogs array in a view dialog box
    dogs.view("dogs is initialized with these values")
endif
endMethod
```

This code is attached to the button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

if dogs.contains("Bandit") = False then
  dogs.insertFirst("Bandit")  ; add new name to the top of the list
                              ; display contents of the array in a dialog box
  dogs.view("dogs size: " + strVal(dogs.size()))
else                          ; "Bandit" must already exist
  msgInfo("Once is enough", "The dogs array already contains Bandit.")
endif

endMethod
```

▪

countOf method

See also Example Array Type

Counts the occurrences of a value in an array.

Syntax

countOf (** const ***value* AnyType **)** LongInt

Description

countOf compares *value* to each item in an array and returns the number of exact matches, or 0 if no match is found.

▪

countOf example

This code (attached to a button's **pushButton** method) creates and loads a fixed-size array, then uses **countOf** to display the number of like values in the array:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  zoo Array[4] String
  i   SmallInt
endVar
for i from 1 to 3
  zoo[i] = "cat"        ; add three "cat" values
endFor
zoo[4] = "dog"          ; add one "dog" value

msgInfo("How many cats?", zoo.countOf("cat"))   ; displays 3
msgInfo("How many dogs?", zoo.countOf("dog"))   ; displays 1
msgInfo("How many apes?", zoo.countOf("ape"))   ; displays 0

endMethod
```

▪

empty method

See also Example Array Type

Removes all items from an array.

Syntax

`empty ( )`

Description

empty removes all items from an array. A fixed-size array stays the same size, and all items become unassigned. A resizeable array is reset to a size of 0.

▪

empty example

The following example shows how **empty** functions for a fixed-size array. The code immediately following declares a fixed-size array in a form's Var window. This array is global to all objects on the form.

```
; thisForm::Var
Var
  ar Array[5] AnyType  ; declare a fixed-size array
endVar
```

The following code is attached to a button's **pushButton** method. When this button (*fillButton*) is pressed, the code assigns numeric values to each element in the *ar* array:

```
; fillButton::pushButton
method pushButton(var eventInfo Event)
ar[1] = 234    ; load the array with numbers
ar[2] = 356
ar[3] = 98
ar[4] = 989
ar[5] = 2341
               ; view the contents of the array
ar.view("Contents of the ar array")
endMethod
```

The following code is attached to a button's **pushButton** method. When this button (*emptyButton*) is pressed, the code empties the *ar* array and displays the contents of the array. Since *ar* is a fixed-size array, the number of elements does not change; there are still five elements, but each value becomes unassigned.

```
; emptyButton::pushButton
method pushButton(var eventInfo Event)
ar.empty()       ; empty the ar array
                 ; view the contents of the array
ar.view("Contents of the ar array")
endMethod
```

▪

exchange method

See also Example Array Type

Swaps the contents of two cells in an array.

Syntax

`exchange (` `const` ***`index1`*** `LongInt, const` ***`index2`*** `LongInt` **`)`**

Description

exchange swaps the contents of the cells at *index1* and *index2* in an array.

▪

exchange example

See the example for **indexOf**.

■

fill method

See also Example Array Type

Fills an array with a value.

Syntax

`fill ( ``const `***`value`***` AnyType `**`)`**

Description

fill assigns *value* to every item of an array.

-

fill example

This code creates a fixed-size array and fills the array with string values. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myArray Array[4] String
endVar

myArray.fill("Hello")  ; fill myArray with Hello
myArray.view()         ; display four Hello's in a dialog

endMethod
```

▪

grow method

See also Example Array Type

Increases the size of a resizeable array.

Syntax

```
grow ( const increment LongInt )
```

Description

grow appends *increment* cells to a resizeable array, or removes cells if the value of *increment* is negative. If you try to remove more cells than the array has, an error occurs.

grow example

The following example uses **grow** to increase and shrink the size of a resizeable array. This code is attached to a button's **pushButton** method.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ar Array[] SmallInt
endVar

ar.setSize(2)
ar[1] = 6
ar[2] = 123
message(ar.size()) ; displays 2
sleep(1000)
ar.grow(3)
message(ar.size()) ; displays 5
sleep(1000)
ar.grow(-3)
message(ar.size()) ; displays 2
sleep(1000)

endMethod
```

indexOf method

See also Example Array Type

Returns the position of an item in an array.

Syntax

indexOf (**const ***value* AnyType **)** LongInt

Description

indexOf returns the index of the first occurrence of *value* in an array, or 0 if an exact match is not found.

indexOf example

The following example assumes a form has an undefined field object named *thisField*. When a user right-clicks on the field, a pop-up menu appears, offering a list of payment types. The item selected is inserted into the field. When the user next right-clicks on the field, the last menu item selected is the first in the list of menu choices. The following code goes in the Var window for *thisField*:

```
; thisField::Var
Var
  payArray  Array[5] String
  payMenu   PopUpMenu
endVar
```

The following code is attached to the **open** method for *thisField*. When the field first opens, this code assigns values to the array that is used for the pop-up menu:

```
; thisField::open
method open(var eventInfo Event)
payArray[1] = "Check"      ; initialize array elements
payArray[2] = "Cash"
payArray[3] = "Visa"
payArray[4] = "MasterCard"
payArray[5] = "AmEx"
endMethod
```

The following code is attached to the **mouseRightUp** method for *thisField*. This code displays the pop-up menu and inserts the selection into *thisField*. The **indexOf** method is used here to get the ordinal value of the selected menu item; the selection is then moved, with the **exchange** method, to the beginning of the array.

```
; thisField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  choiceIndex SmallInt
  choice      String
endVar

disableDefault                ; don't display the normal menu
payMenu.addArray(payArray)    ; add the array to the pop-up menu
choice = payMenu.show()       ; show the menu - assign selection to choice
self.value = choice           ; enter menu selection into field

  ; now prepare the pop-up menu for the next right click
payMenu.empty()                         ; empty the menu
choiceIndex = payArray.indexOf(choice)  ; get the array index of the selection
payArray.exchange(choiceIndex, 1)       ; move the selection to the top
endMethod
```

▪

insert method

[See also](#) [Example](#) [Array Type](#)

Inserts one or more empty cells into an array.

Syntax

insert (const ***index*** LongInt [**,** const ***numberOfItems*** LongInt] **)**

Description

insert inserts *numberOfItems* empty cells into a resizeable array. if *numberOfItems* is not specified, one cell is inserted. Indexes of subsequent items are increased by the number of inserted cells.

▪

insert example

The following example inserts empty elements at two locations in a resizeable array and displays the results. This code is attached to a button's **pushbutton** method:

```
; thisbutton::pushbutton
method pushbutton(var eventinfo event)
var
  myArray Array[] SmallInt
endVar
myArray.setSize(20)   ; allocates space for 20 items
myArray.fill(1)       ; fills the array with 1's
myArray.insert(5)     ; inserts an empty cell at position 5
myArray.insert(12, 4) ; inserts 4 empty cells at position 12
myArray.view()
endMethod
```

▪

insertAfter method

See also Example Array Type

Inserts an item into an array after a specified item.

Syntax

insertAfter (const ***keyItem*** AnyType**,** const ***insertedItem*** AnyType **)**

Description

insertAfter inserts *insertedItem* into a resizeable array at a position one greater than the first occurrence of *keyItem*. If *keyItem* is not found, *insertedItem* is not inserted, and indexes do not change. If *insertedItem* is inserted, indexes of subsequent items increase by 1.

insertAfter example

The following example loads a resizeable array, then uses **insertAfter** to insert a new element after an existing array element. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  zoo Array[] String
endVar
zoo.setSize(0)
zoo.addLast("ape")      ; [1] = "ape"
zoo.addLast("cow")      ; [2] = "cow"
zoo.addLast("dog")      ; [3] = "dog"

zoo.insertAfter("ape", "bear")
  ; displays size: 4 in the title; zoo[ape, bear, cow, dog]
zoo.view("zoo size: " + strVal(zoo.size()))

endMethod
```

▪

insertBefore method

See also   Example   Array Type

Inserts an item into an array before a specified item.

Syntax

`insertBefore ( ``const `***`keyItem`***` AnyType, const `***`insertedItem`***` AnyType `**`)`**

Description

insertBefore searches a resizeable array for *keyItem*, and inserts *insertedItem* at *keyItem*'s position. Indexes of *keyItem* (and subsequent items) are increased by 1. If *keyItem* is not found, *insertedItem* is not inserted, and indexes do not change.

insertBefore example

The following example adds an element to a resizeable array with **insertBefore**. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  foodChain Array[] String
endVar

foodChain.grow(3)         ; start array out with 3 elements
foodChain[1] = "Hawk"
foodChain[2] = "Snake"
foodChain[3] = "Fly"

  ; insert an element - this increases the array to 4 elements
foodChain.insertBefore("Fly", "Frog")
  ; displays size: 4 in title; [Hawk, Snake, Frog, Fly]
foodChain.view("foodChain size: " + strVal(foodChain.size()))

endMethod
```

▪

insertFirst method

See also Example Array Type

Inserts an item at the beginning of an array.

Syntax

`insertFirst ( const value AnyType )`

Description

insertFirst inserts value at the beginning of a resizeable array. Indexes of subsequent items are increased by 1.

▪

insertFirst example

The following example creates a resizeable array, then adds a new element to the beginning of the array. This code is attached to a button's built-in **pushButton** method:

```
method pushButton(var eventInfo Event)
var
  myZoo Array[] String
endVar
myZoo.setSize(2)    ; start the array with two elements
myZoo[1] = "lion"
myZoo[2] = "tiger"

                    ; insert an element at beginning of array -
                    ; this increases the array to three elements
myZoo.insertFirst("bear")
                    ; displays size: 3 in title; [bear, lion, tiger]
myZoo.view("myZoo size: " + strVal(myZoo.size()))

endMethod
```

▪

isResizeable method

See also Example Array Type

Reports whether an array can be resized.

Syntax

`isResizeable ( ) Logical`

Description

isResizeable returns True if an array can be resized; otherwise, it returns False.

isResizeable example

This code checks to see if a particular array can be resized before attempting to increase its size. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myArray Array[] String
endVar
if myArray.isResizeable() = True then  ; if array can be resized
  myArray.grow(5)                       ; add 5 cells to it
else
  msgStop("Problem", "Array cannot be resized.")
endif
endMethod
```

remove method

[See also](#) [Example1](#) [Example2](#) [Array Type](#)

Removes one or more items from an array.

Syntax

remove (const ***index*** SmallInt [const ***numberOfItems*** SmallInt] **)**

Description

remove deletes *numberOfItems* items (or 1 item, if *numberOfItems* is not specified) at index in an array. Indexes of subsequent items are decreased by *numberOfItems* (or 1, if *numberOfItems* is not specified).

■

remove example 1

The following example removes a single item from a resizeable array. Note that it is common to use the **indexOf** method to determine which element you want to remove. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myZoo Array[] String
endVar

myZoo.setSize(3)          ; start myZoo out with three elements
myZoo[1] = "lion"
myZoo[2] = "tiger"
myZoo[3] = "bear"

myZoo.remove(myZoo.indexOf("tiger"))  ; same as myZoo.remove(2)

                          ; title displays size: 2
                          ; dialog displays myZoo[lion, bear]
myZoo.view("myZoo size: " + strVal(myZoo.size()))

endMethod
```

▪

remove example 2

The following example shows how to use **remove** to eliminate more than one element from a resizeable array. This code is attached to a button's **pushButton** method:

```
; thatButton::pushButton
method pushButton(var eventInfo Event)
var
  myNums Array[] SmallInt
  i      SmallInt
endVar

myNums.grow(9)      ; start myNums with nine elements
for i from 1 to 9   ; assign nine elements
   myNums[i] = i
endFor

                    ; displays myNums[1, 2, 3, 4, 5, 6, 7, 8, 9]
myNums.view("Before removing elements")
                    ; remove four items, starting with third element
myNums.remove(3, 4) ; myNums = [1, 2, 7, 8, 9]
                    ; displays myNums[1, 2, 7, 8, 9]
myNums.view("After removing elements")
endMethod
```

▪

removeAllItems method

See also Example Array Type

Removes all occurrences of an array item.

Syntax

removeAllItems (const ***value*** AnyType **)**

Description

removeAllItems deletes all occurrences of *value* from an array. Indexes of subsequent items are decreased by 1.

removeAllItems example

The following example shows how **removeAllItems** works with a resizeable array. The following code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   myZoo Array[] String
endVar
myZoo.setSize(5)
myZoo[1] = "ape"
myZoo[2] = "cow"
myZoo[3] = "pig"
myZoo[4] = "cow"
myZoo[5] = "lion"

  ; display current contents of array in a dialog
myZoo.view("Before removing elements")

  ; removes all occurrences of cow
myZoo.removeAllItems("cow")

  ; now,
  ; myZoo[1] = "ape"
  ; myZoo[2] = "pig"
  ; myZoo[3] = "lion"

  ; display new contents of array in a dialog
myZoo.view("After removing elements")

endMethod
```

▪

removeItem method

See also   Example   Array Type

Deletes a specified item from an array.

Syntax

removeItem (**const ***value* AnyType **)**

Description

removeItem deletes the first occurrence of *value* from an array. Indexes of subsequent items are decreased by 1.

removeItem example

The following example uses **removeItem** to eliminate an item from a resizeable array. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myZoo Array[] String
endVar

myZoo.setSize(4)
myZoo[1] = "ape"
myZoo[2] = "lion"
myZoo[3] = "tiger"
myZoo[4] = "lion"

  ; this displays [ape, lion, tiger, lion]
myZoo.view("Before removing a lion")

  ; remove first occurrence of "lion"
myZoo.removeItem("lion")

  ; this displays [ape, tiger, lion] in a dialog
myZoo.view("After removing a lion")

endMethod
```

▪

replaceItem method

See also Example Array Type

Overwrites an item in an array with another item.

Syntax

replaceItem (const ***keyItem*** AnyType, const ***newItem*** AnyType **)**

Description

replaceItem searches an array for *keyItem*, and replaces the first occurrence of *keyItem* with *newItem*.

replaceItem example

The following example replaces an item in a resizeable array, and displays the initial value and the results in a dialog box. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  foodChain Array[] String
endVar

foodChain.setSize(3)
foodChain[1] = "Shark"
foodChain[2] = "Elephant"
foodChain[3] = "Minnow"

  ; display contents of array in a dialog box
foodChain.view("Before replaceItem...")

foodChain.replaceItem("Elephant", "Tuna")
  ; display contents of array in a dialog box ([Shark, Tuna, Minnow])
foodChain.view("After replaceItem...")

endMethod
```

▪

setSize method

See also   Example   Array Type

Specifies the size of an array.

Syntax

```
setSize ( const size LongInt )
```

Description

setSize saves space for *size* items in a resizeable array. If **setSize** makes the array smaller, the array is truncated.

setSize example

The following example declares a resizeable array in the variable declaration section, then uses **setSize** to initialize the size of the array to three elements. The code fills each element of the array, then issues **setSize** again, this time to resize the array to two elements. The result of making the array smaller (shown in a dialog box) is the elimination of the third (and last) element. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myArray Array[] SmallInt
endVar

myArray.setSize(3)          ; size is 3

myArray[1] = 123
myArray[2] = 2353
myArray[3] = 18

  ; display size: 3 in title; [123, 2353, 18] in a dialog box
myArray.view("myArray size: " + strVal(myArray.size()))

myArray.setSize(2)          ; size is 2- myArray[3] truncated

  ; display size: 2 in title; [123, 2353] in a dialog box
myArray.view("Now myArray size: " + strVal(myArray.size()))

endMethod
```

▪

size method

See also   Example   Array Type

Returns the number of items in an array.

Syntax

`size ( ) LongInt`

Description

size returns the total number of items in an array, even if one or more elements are blank.

▪

size example

See the example for **setSize**.

view method

See also Example Array Type

Displays in a dialog box the contents of an array.

Syntax

```
view ( [ const title String ] )
```

Description

view displays in a modal dialog box the contents of an array. ObjectPAL execution suspends until the user closes this dialog box. You have the option to specify, in *title*, a title for the dialog box. If you omit title, the title is "Array."

Unlike many other data types, Array values displayed in a view dialog box can not be changed interactively. See AnyType for more information regarding other data types and the **view** method.

■

view example

The following example displays the contents of an array in a dialog box without a custom title, then with a custom title. Note that *title* can be any expression that evaluates to a string. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ar Array[]  SmallInt
  i           SmallInt
endVar

ar.setSize(10)
for i from 1 to 10
  ar[i] = i * 10
endfor

ar.view()         ; displays 10, 20, 30, etc (no title)
                  ; this displays "ar size: 10" in the title
ar.view("ar size: " + strVal(ar.size()))

endMethod
```

Binary type

Changes

A binary object (sometimes called a binary large object, or BLOB) contains data that only a computer can read and interpret. An example of a binary object is a sound file: a human can't read or interpret the file in its raw form, but a computer can.

When you declare a Binary variable, you create a handle to a binary object, a variable you can refer to in your code to move binary data back and forth between a disk file and a binary field in a table, or from a disk file or a table to a method or procedure.

The Binary type includes several derived methods from the AnyType type.

Methods for the Binary type

AnyType	**Binary**
blank	**clipboardErase**
dataType	**clipboardHasFormat**
isAssigned	**enumClipboardFormats**
isBlank	**readFromClipboard**
isFixedType	**readFromFile**
	size
	writeToClipboard
	writeToFile

Changes to Binary type methods

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
clipboardErase	(None)
clipboardHasFormat	
enumClipboardFormats	
readFromClipboard	
writeToClipboard	

clipboardErase method

See also Example Binary Type

Clears the Windows Clipboard.

Syntax

`clipboardErase ( )`

Description

clipboardErase clears the Windows Clipboard on the user's system.

▪

clipboardErase example

See the example for **clipboardHasFormat**.

▪

clipboardHasFormat procedure

See also Example Binary Type

Reports whether a format name is on the Windows Clipboard.

Syntax

`clipboardHasFormat ( const formatName String ) Logical`

Description

clipboardHasFormat returns True if the format name *formatName* is in the Windows Clipboard on a user's system; otherwise, it returns False.

clipboardHasFormat example

In the following example, the **pushButton** method for a button named *clearClipboard* checks the Windows Clipboard for a Borland Form Object and if it is there, clears the Clipboard.

```
;btnClearClipboard::pushButton
method pushButton(var eventInfo Event)
   var
      b   Binary
   endVar

   if clipboardHasFormat("Borland Form Object") then
      b.clipBoardErase()
      message("Clipboard cleared")
   else
      message("Borland form object not on Clipboard")
   endIf
endMethod
```

▪

enumClipboardFormats method

See also Example Binary Type

Creates an array listing the formats currently on the Windows Clipboard.

Syntax

`enumClipboardFormats ( var formatNames Array[ ] String ) SmallInt`

Description

enumClipboardFormats creates an array *formatNames* that lists the formats currently on the Windows Clipboard on the user's system. You must declare the array before you call this method.

▪

enumClipboardFormats example

The following code writes the Clipboard format names to an array named *ar*, then displays *ar* in a **view** dialog box.

```
;btnShowClipboard :: pushButton
method pushButton(var eventInfo Event)
   var
      b    Binary
      ar   Array[] String
   endVar

   b.enumClipboardFormats( ar )
   ar.view("Formats in Windows Clipboard")
endmethod
```

▪

readFromClipboard method

See also Example Binary Type

Reads a binary object from the Clipboard.

Syntax

`readFromClipboard ( const clipboardFormat String ) Logical`

Description

readFromClipboard reads a binary object *clipboardFormat* from the Clipboard to a variable of type Binary. If the Clipboard contains a Binary object that can be copied to the Binary variable, **readFromClipboard** returns True. If the Clipboard is empty or does not contain a valid Binary, **readFromClipboard** returns False.

▪

readFromClipboard example

See the example for **writeToClipboard**.

▪

readFromFile method

See also Example Binary Type

Reads data from a file and stores it in a Binary variable.

Syntax

readFromFile (const *fileName* String) Logical

Description

readFromFile reads binary data from the disk file named in *fileName*. This method returns True if successful; otherwise, it returns False.

▪

readFromFile example

The following statements declare a Binary variable *theSound*, read binary data from a file into *theSound*, then assign the value of the variable to a Binary field in a table. (Assume SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, B.)

```
; getFile::pushButton
method pushButton(var eventInfo Event)
var
  soundsTC TCursor
  theSound Binary
endVar
if theSound.readFromFile("noise.bin") then ; True if readFromFile succeeds
  if soundsTC.open("sounds.db") then
     soundsTC.edit()
     soundsTC.insertRecord()
     soundsTC.SoundName = "Noise"
     soundsTC.SoundData = theSound  ; put file contents in a binary field
     soundsTC.endEdit()
     soundsTC.close()
  endIf
endIf

endMethod
```

▪

size method

See also Example Binary Type

Returns the number of bytes in a Binary variable.

Syntax

```
size ( ) LongInt
```

Description

size returns a value representing the number of bytes stored in a Binary variable.

▪

size example

The following example steps through the records in a table that contain Binary fields. The example tests the **size** of each Binary field. If there's enough free disk space, the code writes the data to a disk file. (Assume SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, B.) This code is attached to a custom method named *writeBinFiles*:

```
method writeBinFiles()
var
  binVar    Binary
  fs        FileSystem
  soundsTC  TCursor
  freeSpace LongInt
endVar

if soundsTC.open("Sounds.db") then
  scan soundsTC for not isBlank(soundsTC.SoundData) :
    binVar = soundsTC.SoundData     ; binVar = SoundData field value
    freeSpace = fs.freeDiskSpace("B")
    if freeSpace > binVar.size() then     ; if there's room on B:
binVar.writeToFile(soundsTC.SoundName)     ; write binVar to file
    else          ; else the file won't fit on B:
      msgStop("Stop", "The disk in drive B: is full.")
      return
    endIf
  endScan
endif

endMethod
```

▪

writeToClipboard method

See also Example Binary Type

Writes a binary object to the Clipboard.

Syntax

`writeToClipboard ( const clipboardFormat String ) Logical`

Description

writeToClipboard writes (copies) a binary object to the Windows Clipboard. Specify the Clipboard format to use with the parameter *clipboardFormat*. **writeToClipboard** returns True if successful; otherwise, it returns False.

writeToClipboard example

In the following example, a form contains two buttons. The button named *btnStoreClip* stores part of the Windows Clipboard, the Native portion, to a file called *NATIVE.CLP*. The second button named *btnRetrieveClip* retrieves the clip from the file and writes it to the clipboard.

The following code is attached to *btnStoreClip*.

```
;btnStoreClip :: pushButton
method pushButton(var eventInfo Event)
   var
      b   Binary
   endVar

   if not b.readFromClipboard("Native") then
      msgInfo("Instructions", "First copy something to Clipboard.")
   endIf

   b.writeToFile("Native.clp")
endmethod
```

The following code is attached to *btnRetrieveClip*.

```
;btnRetrieveClip :: pushButton
method pushButton(var eventInfo Event)
   var
      b   Binary
   endVar

   if not b.readFromFile("Native.clp") then
      beep()
      message("File does not exist")
   endIf

   b.writeToClipBoard("Native")
endMethod
```

writeToFile method

See also Example Binary Type

Writes the data stored in a Binary variable to a disk file.

Syntax

writeToFile (const ***fileName*** String **)** Logical

Description

writeToFile writes the data stored in a Binary variable to the disk file specified in *fileName*. This method returns True if successful; otherwise, it returns False.

▪

writeToFile example

The following example steps through the records in a table that contains Binary fields. It tests the size of each Binary field. If there's enough free disk space, the code writes the data to a disk file. (Assume SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, B.) This code is attached to a custom method named *writeBinFiles*:

```
method writeBinFiles()
var
  binVar    Binary
  fs        FileSystem
  soundsTC  TCursor
  freeSpace LongInt
endVar

if soundsTC.open("Sounds.db") then
  scan soundsTC for not isBlank(soundsTC.SoundData) :
    binVar = soundsTC.SoundData   ; binVar = SoundData field value
    freeSpace = fs.freeDiskSpace("B")
    if freeSpace > binVar.size() then         ; if there's room on B:
      binVar.writeToFile(soundsTC.SoundName) ; write binVar to file
    else                                     ; else the file won't fit
                                             ; on B:
      msgStop("Stop", "The disk in drive B: is full.")
      return
    endIf
  endScan
endif

endMethod
```

Currency type

Currency values can range from ± 3.4E-4930 to ± 1.1E4930 (precise to eighteen decimal places). The number of decimal places displayed depends on the user's Control Panel settings. However, the value stored in a table does not a table stores the full eighteen decimal places.

The Currency type includes several derived methods from the Number and AnyType types.

Methods for the Currency type

AnyType	Number	**Currency**
blank	abs	**currency**
dataType	acos	
isAssigned	asin	
isBlank	atan	
isFixedType	atan2	
view	ceil	
	cos	
	cosh	
	exp	
	floor	
	fraction	
	fv	
	ln	
	log	
	max	
	min	
	mod	
	number	
	numVal	
	pmt	
	pow	
	pow10	
	pv	
	rand	
	round	
	sin	
	sinh	
	sqrt	
	tan	
	tanh	
	truncate	

▪

currency procedure

See also Example1 Example2 Currency Type

Casts a value as Currency.

Syntax

currency (const ***value*** AnyType **)** Currency

Description

currency casts (converts) the data type of *value* to Currency.

▪

currency example 1

In the following example, a number is stored to a String variable, then cast to a Currency type for use in a calculation. The **pushButton** method for *showDouble* displays the type of the variable, then calculates and displays the result of the string cast as Currency and multiplied by 2:

```
; showDouble::pushButton
method pushButton(var eventInfo Event)

var
  numstr   String
endVar

numStr = "12.34"
msgInfo("The data type of numStr is:", dataType(numStr))
; before multiplying numStr by two, it must be cast
; to a numeric type
msgInfo("Double " + numStr, currency(numStr) * 2)
endMethod
```

▪

currency example 2

In the following example, the **pushButton** method for the *watchPrecision* button calculates a number using variables of type Number, then performs the same calculation with the values cast to Currency. The result of the two calculations varies slightly.

```
; watchPrecision::pushButton
method pushButton(var eventInfo Event)

var
  x, y, z Number
endVar

x = 1.2 / 3.323        ; stores greatest precision
y = 4.9 / 7.3
z = 2.0 * x * y        ; calculates on full values
msgInfo("Result of Number calculation",
         format("W14.6", z))     ; displays .484790
x = Currency(1.2 / 3.323)        ; stores precision to 6th decimal place
y = Currency(4.9 / 7.3)
z = 2.0 * x * y                  ; calculates on 6 decimal precision values
msgInfo("Result of Currency calculation",
         format("W14.6", z))     ; displays .484791

endMethod
```

▪

Database type

Changes ▪

A Database variable provides a handle to a database (a directory). When you start a Paradox application, Paradox opens the *default database* (the working directory). The default database stores the path to the current working directory. If all you want to do is work with those tables, you don't have to open any other database. To work with tables stored elsewhere, declare a Database variable and use an **open** statement to create a handle to another database. (You could specify the full path to each table each time you wanted to use it, but code that uses Database variables is easier to maintain.)

Using **open** and an alias, you can specify which database to open, as shown in the following example:

```
var
   custInfo Database
endVar
; addAlias is defined for the Session type
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")
custInfo.open("CustomerInfo") ; opens the CustomerInfo database
                              ; CustomerInfo must be a valid alias
```

Paradox now knows about two databases: the default database and CustomerInfo. The variable *custInfo* is a *handle* to the CustomerInfo database▪that is, you can use *custInfo* in statements to refer to the CustomerInfo database. For example, suppose you have two files named ORDERS.DB (one in your working directory, and one in CustomerInfo), and you want to find out if these files are tables. The following example tests ORDERS.DB in the working directory first, then uses *custInfo* as a handle for the CustomerInfo database and tests ORDER.DB there:

```
var
   custInfo Database
endVar
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")
custInfo.open("CustomerInfo")

if isTable("orders.db") then         ; test ORDERS.DB in the default database
   msgInfo("Working directory", "ORDERS.DB is a table.")
endIf

if custInfo.isTable("orders.db") then    ; use custInfo as a handle for
                                         ; the CustomerInfo database
   msgInfo("CustomerInfo", "ORDERS.DB is a table.")
endIf
```

If you use **open** but don't specify a database, Paradox assumes you want a handle for the default database. For example, this syntax gives you a handle for the default database, which you could pass to a custom method that requires a database handle.

```
var defaultDb Database endVar
defaultDb.open() ; opens the default database
```

Using a handle to the default database can also make code more readable, especially when you're working with several databases at once.

Methods for the Database type

DataBase
beginTransaction
close
commitTransaction
delete

enumFamily
getMaxRows
isAssigned
isSQLServer
isTable
open
setMaxRows
rollBackTransaction
transactionActive

Changes to Database type methods

The following table lists the new methods for version 7.

New
getMaxRows
setMaxRows

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
enumFamily	beginTransaction
isSQLServer	Database::**executeQBE** was removed. Use Query::executeQBE instead.
	Database::**executeQBEFile** was replaced by Query::readFromFile
	Database::**executeQBEString** was replaced by Query::readFromString
	getQueryRestartOptions was moved to the Query type.
	Database::**isExecuteQBEFileLocal** was removed. Use Query::isExecuteQBELocal instead.
	Database::**isExecuteQBEStringLocal** was removed. Use Query::isExecuteQBELocal instead.
	setQueryRestartOptions was moved to the Query type.
	isExecuteQBELocal was moved to the Query type.
	Database::**writeQBE** was removed. Use Query::writeQBE instead.

beginTransaction method

See also Example Database Type

Starts a transaction.

Syntax

```
beginTransaction ( [ const isoLevel String ] ) Logical
```

Description

beginTransaction starts a transaction on a database that supports transactions. Standard databases (for example, Paradox and dBASE) do not support transactions. Most SQL databases do support transactions.

The optional argument *isoLevel* (added in version 5.0) specifies an isolation level to use when transactions are supported. If you do not specify an isolation level in *isoLevel,* the highest (most isolated) isolation level supported by the server is used. The following table lists valid values for *isoLevel* from lowest isolation level to highest.

***isoLevel* value**	**Description**
DirtyRead	The transaction can read uncommitted changes made by other transactions.
ReadCommitted	Changes made by other transactions affect data read by this transaction.
RepeatableRead	Data previously read in this transaction is not affected by changes made by other transactions.

Only one transaction is allowed for each database. This method returns True if successful; otherwise, it returns False. While the transaction is active, statements (except passthrough SQL statements) that operate on any table associated with the specified database are included as part of the transaction.

▪

beginTransaction example

The following example processes a withdrawal of cash from an automatic teller machine. The call to **beginTransaction** starts a transaction consisting of three operations: debiting the customer's account, debiting the cash on hand, and dispensing cash to the customer. The result of each operation is stored in a DynArray. When all the operations have been completed, this code checks each item in the DynArray and either calls **commitTransaction** (if all items are True) or **rollbackTransaction** (if any item is False).

This example uses **beginTransaction**, **commitTransaction**, **rollbackTransaction**, **transactionActive**, **enumAliasNames**, and **getAliasProperty**.

```
method pushButton(var eventInfo Event)
   var
      db                   Database
      opResult             DynArray[] Logical
      Element              AnyType
      All_OK               Logical
      serverType,
      myAlias,
      custID               String
      aliasNamTC           TCursor
      xAmount              Currency
      xDate                Date
      xTime                Time
   endVar

   ; initialize variables
   myAlias = "ITCHY"
   custID = "RHALL001"
   xAmount = Currency(120.00)
   xDate = today() ; returns current date
   xTime = time()  ; returns current time

   ; use alias to get database handle to server
   if not db.open(myAlias) then
         errorShow("Could not open the database.")
      return ; exit the method
   endIf

   if db.transactionActive() then
      db.commitTransaction()       ; commit any previous transaction
   endIf

   db.beginTransaction()           ; begin a transaction

   ; execute the operations for this transaction
   ; debitAccount, debitCashOnHand, and dispenseCash
   ; are custom procs assumed to be defined elsewhere
   ; after calling debitAccount and debitCashOnHand, the code
   ; calls transactionActive to check the transaction status
   ; before calling dispenseCash

   opResult["Debit customer account"] =
         debitAccount(custID, xAmount)
   opResult["Debit cash on hand"] =
         debitCashOnHand(xAmount, xDate, xTime)

   ; the following if...then...else block is not required
   ; it's included to show one way to use transactionActive

   if db.transactionActive() then  ; make sure everything is OK
      msgInfo("Transaction Status", "In a Transaction")
   else
      errorShow("NOT in a Transaction")
      return
   endIf

   opResult["Dispense cash"] = dispenseCash(xAmount)
```

```
   All_OK = True                  ; initialize to True

   forEach element in opResult   ; Check operation results
      if opResult[element] = False then
         All_OK = False
         quitLoop
      endIf
   endForEach

   ; inform user of transaction status
   if All_OK then
      if db.commitTransaction() then
         msgInfo("Transaction Status","Transaction committed.")
      else
         errorShow("Transaction NOT committed")
      endIf
   else
      if msgQuestion("Transaction failed",
                     "View results?") = "Yes" then
         opResult.view("Operation results")
      endIf

      if db.rollbackTransaction() then
         msgInfo("Transaction Status",
                 "Transaction rolled back.")
      else
         errorShow("Transaction NOT rolled back.")
      endIf
   endIf

endMethod
```

close method

See also Example Database Type

Closes a database.

Syntax

`close ( ) Logical`

Description

close ends the association between a Database variable and a database, making the variable unassigned. **close** returns True if it succeeds; otherwise, it returns False.

close example

The following code opens the database with the alias *someTables*. If the *Orders* table doesn't exist in *someTables*, this code closes *someTables* and opens another database with the alias *moreTables*. This code assumes that both aliases have been defined elsewhere and are valid.

```
; sumButton::pushButton
method pushButton(var eventInfo Event)
var
  db Database
  tc TCursor
endVar
db.open("someTables")               ; open the database alias someTables
if db.isTable("Orders.db") then     ; if Orders.db is in the database,
  tc.open("Orders.db", db)          ; open a TCursor for it
                                    ; calculate the total balance due
  msgInfo("Balance Due", tc.cSum("Balance Due"))
else
  db.close()                        ; close someTables database
  db.open("moreTables")             ; and open another one
  if db.isTable("Orders.db") then
    tc.open("Orders.db", db)
    msgInfo("Balance Due", tc.cSum("Balance Due"))
  endIf
endIf
endMethod
```

▪

commitTransaction method

See also Example Database Type

Commits all changes within a transaction.

Syntax

```
commitTransaction ( ) Logical
```

Description

commitTransaction commits all changes made within a transaction on a database that supports transactions. Standard databases (for example, Paradox and dBASE) do not support transactions. Most SQL databases do support transactions.

commitTransaction returns True if successful; otherwise, returns False. This method does not check the results of the operations in the transaction; it's up to you to evaluate the results of the operations and decide whether to commit the transaction or roll it back.

commitTransaction example

See the example for **beginTransaction**.

▪

delete method/procedure

See also   Example   Database Type

Deletes a table from a database.

Syntax

```
1. delete ( const tableName String [ , const tableType String ] ) Logical
2. delete ( const tableVar Table ) Logical
```

Description

delete removes a table and any associated index files or table view files from the database without asking for confirmation. If you use syntax 1, and if the file extension is not standard or not supplied, you can use the optional argument *tableType* to specify the type of the table to delete ("Paradox" or "dBASE"). If *tableType* is not specified or not standard, "Paradox" is assumed. If you use syntax 2, you can use the argument *tableVar* to specify a Table variable. However, this method uses only the name and type of the table described by the Table variable, not its database association.

The operation cannot be undone. This method returns True if the table is successfully deleted; otherwise, it returns False. If the table is open, **delete** fails.

delete example

In the following example, the **pushButton** method for *delTable* deletes a table from the database with the alias *megaData*.

```
; delTable::pushButton
method pushButton(var eventInfo Event)
var
  myDb Database
  tableName String
endVar
tableName = "OldTable.dbf"
myDb.open("megadata")
if isTable(tableName) then
  myDb.delete(tableName, "dBASE") ; removes OldTable.dbf from megadata
endif
endMethod
```

▪

enumFamily method/procedure

See also Example Database Type

Lists the files in a table family.

Type

Database

Syntax

`enumFamily ( var members DynArray[ ] String, const tableName String ) Logical`

Description

enumFamily lists the files in the table family of the table *tableName*. It assigns values to the dynamic array *members* that you pass as an argument. The value of *tableName* must include a file extension if the actual table name includes one. In other words, if you specify "ORDERS", this method *will not* list family for ORDERS.DB; instead, it will look for a table named ORDERS.

The indexes of the resulting DynArray are the full file names (for example, "C:\PDOXWIN\TABLES\ORDERS.DB") of the family members, and the corresponding value is one of the following strings:

Blobfile

Form

Index

Report

SecondaryIndex

SecondaryIndex2

Table

Unknown

ValCheck

▪

enumFamily example

The following code writes the family information from the *Orders* table to a dynamic array *dyn* then a **forEach** loop displays each element in a message information dialog box.

```
;btnFamilyInfo :: pushButton
method pushButton(var eventInfo Event)
   var
      dyn        DynArray[] String
      sElement   String
   endVar

   enumFamily(dyn, "ORDERS.DB")

   forEach sElement in dyn
      msgInfo(sElement, dyn[sElement])
   endForEach
   ; You could also do dyn.view().
endmethod
```

▪

getMaxRows method

See also Example Database Type

Retrieves the setting of **setMaxRows**.

Syntax

```
getMaxRows ( const maxRows LongInt ) Logical
```

Description

getMaxRows retrieves the setting on the maximum number of rows that will be returned from an SQL server in response to any query. The maximum is set by the **setMaxRows** method.

▪

getMaxRows example

This example puts a 1000 record limit on the query if the maximum is set to less than 1000.

```
var myQBE Query
endvar
   if getMaxRows() < 1000 then
      setMaxRows(1000)
   endif
   myQBE = Query
         Customer    |Customer No |Name   |
                     | Check      |A..    |
   endQuery
```

▪

isAssigned method

See also Example Database Type

Reports whether a Database variable has been assigned a value.

Syntax

`isAssigned ( )` `Logical`

Description

isAssigned returns True if the Database variable has been assigned a value; otherwise, it returns False.

▪

isAssigned example

For the following example, a form has an unassigned field named coRating and a button named *showRating*. Code attached to *showRating*'s **pushButton** method uses **isAssigned** to determine whether the Database variable *db* is assigned. If it's not, a database alias is established and assigned to the Database variable. Once the variable is defined, the code opens a TCursor for the *NewCust* table contained in the database. The TCursor locates a value in the Company field, then displays that company's credit rating in the *coRating* field on the form. Following is the code attached to the **pushButton** method for *showRating*:

```
; showRating::pushButton
method pushButton(var eventInfo Event)
var
  db Database
  tc TCursor
endVar

if not isAssigned(db) then
  addAlias("myTables", "Standard", "c:\\pdoxwin\\myTables")
  db.open("myTables")
endif

tc.open("NewCust.dbf", db)
if tc.locatePattern("Company", "Thompson's..") then
  coRating.value = tc.Rating
else
  message("Error", "Thompson's.. not found.")
endif

endMethod
```

▪

isSQLServer method

See also Example Database Type

Reports whether a Database is opened on a SQL server.

Syntax

`isSQLServer ( ) Logical`

Description

isSQLServer returns True if the Database variable opened on a SQL server; otherwise, it returns False.

isSQLServer example

In the following example, a database variable is opened on an alias. The code then checks to see if the database variable points to an SQL server and displays the results.

```
; showRating::pushButton
method pushButton(var eventInfo Event)
   var
      db Database
   endVar

   db.open(":fred:")

   if db.isSQLServer() then
      msgInfo(":FRED:", "Is on a SQL server.")
   else
      msgInfo(":FRED:", "Is not on a SQL server.")
   endIf

endMethod
```

▪

isTable method/procedure

See also   Example   Database Type

Reports whether a table exists in a database.

Syntax

```
1. isTable ( const tableName String [ , const tableType String ] ) Logical
2. isTable ( const tableVar Table ) Logical
```

Description

isTable returns True if a specified table is found in the database; otherwise, it returns False.

If you use syntax 1, you can specify a table name and a table type in arguments *tableName* and *tableType*. If you use syntax 2, you can specify a Table variable in *tableVar.* However, this method uses only the name and type of the table described by the Table variable, not the database association.

■

isTable example

The following code uses **isTable** to determine whether the *Orders* table exists in a given database. This code is attached to the built-in **pushButton** method for *thisButton*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  db        Database
  testMe    String
  testMeToo Table
  myTable   TableView
endVar

db.open()                        ; opens the default database
testMe = "Orders.db"
if db.isTable(testMe) then
  myTable.open(testMe)
else
  message(testMe, " is not a table!")
endIf

testMeToo.attach("sales.db")
if testMeToo.isTable() then
  tot = testMeToo.cSum("Total sales")
  msgInfo("total sales:", tot)
endMethod
```

▪

open method/procedure

See also Example Database Type

Opens a database.

Syntax

```
1. open ( ) Logical
2. open ( const aliasName String ) Logical
3. open ( const ses Session ) Logical
4. open ( const aliasName String, const ses Session ) Logical
5. open ( [ const aliasName String, ] [ const ses Session, ]
            [ const parms DynArray ] ) Logical
```

Description

open opens a database. In syntax 1, where no arguments are given, **open** opens the default database in the current session. In syntax 2, you specify in *aliasName* a database to open in the current session. Syntax 3 lets you open the default database in the session specified in *ses*. Use syntax 4 to open a specified database in a specified session. In syntax 5, the *parms* argument represents a list of parameters and values to use when opening a database on a SQL server. The items in the parameter list correspond to the fields in the Alias Manager dialog box for a given alias. The items will vary depending on the type of server you're connecting to; refer to your server documentation for more information.

If you use syntax 2, 4 or 5, *aliasName* must be a valid alias in the current session or the *ses* session. The colons around the alias name are optional.

Syntaxes 3, 4 and 5 require that a valid session variable has been opened; the current session is assumed in syntax 1 and 2.

When you use syntax 5, the settings in the *parms* DynArray override values set previously, both in code and interactively. For example, if the OPEN MODE parameter was previously set to READ/WRITE, the following statement would set it to READ ONLY when you open the database.

```
dbParmsDA["OPEN MODE"] = "READ ONLY"
```

When you use *parms* to specify parameters, the Alias Manager dialog box does not open, even if a password is required.

open returns True if it is able to open the specified database; otherwise, it returns False.

open example

For the following example, the **pushButton** method for *thisButton* opens four databases in the current session.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dDb, myDb, pDb, rDb  Database
  dbParmsDA            DynArray[] AnyType
  currSes              Session
endVar

currSes.open()          ; get a handle to the current session

dDb.open()              ; associate dDb with the default database
myDb.open("custInfo")   ; associate myDb with the Custinfo database
                        ; (custInfo is an alias defined elsewhere)
pDb.open("PRIV")        ; associate pDb with the Private directory

; specify parameters for SQL database
dbParmsDA["OPEN MODE"] = "READ/WRITE"
dbParmsDA["Password"]  = "tycobb"

rDb.open("remote", currSes, dbParmsDA) ; (remote is an alias defined
elsewhere)
endMethod
```

rollBackTransaction method

See also   Example   Database Type

Rolls back (undoes) all changes within a transaction on a server when transactions are supported.

Syntax

```
rollbackTransaction ( ) Logical
```

Description

rollbackTransaction rolls back (undoes) the effects of all operations within a transaction. Returns True if successful; otherwise, returns False.

▪

rollBackTransaction example

See the example for **beginTransaction**.

▪

setMaxRows method

See also Example Database Type

Sets the maximum number of rows that can be retrieved by one query.

Syntax

setMaxRows (**const **maxRows LongInt **)** Logical

Description

setMaxRows sets the maximum number of rows that will be returned from an SQL server in response to any query. The argument *maxRows* is a long integer that specifies the maximum number of rows returned. **setMaxRows** returns True if the maximum number of rows specified by *maxRows* is successfully set.

setMaxRows is similar to the BDE configuration option MAX ROWS. MAX ROWS is set in the BDE Configuration file's DB OPEN section and sets the maximum number of rows that the SQL driver will attempt to fetch for any single SQL statement sent to the server. If a request is made that exceeds the maximum specified by MAX ROWS, the error generated is DBIERR_ROWFETCHLIMIT.

The maximum specified with the **setMaxRows** method can exceed that specified by the MAX ROWS BDE configuration option.

If no **setMaxRows** method is issued or if the *maxRows* argument is set to -1, no limit on rows is imposed by Paradox and only the BDE MAX ROWS limit (if present) is imposed.

■

setMaxRows example

This example puts a 1000 record limit on the query if the maximum is set to less than 1000.

```
var myQBE Query
endvar
   if getMaxRows() < 1000 then
      setMaxRows(1000)
   endif
   myQBE = Query
         Customer       |Customer No |Name   |
                        | Check      |A..    |
   endQuery
```

▪

transactionActive method

See also Example Database Type

Reports whether a transaction is currently active in a specified database.

Syntax

```
transactionActive ( ) Logical
```

Description

transactionActive reports whether a transaction is currently active in a specified database. Paradox allows only one active transaction for each database, so it's a good idea to call **transactionActive** before beginning a transaction.

▪

transactionActive example

See the example for **beginTransaction**.

Date type

Changes

In ObjectPAL, date values can be represented in either month/day/year, day-Month-year, or day.month.year format. Dates must be cast (explicitly declared). For example,

```
var
  d Date
endVar
d = date("12/21/1997")
```

assigns to *d* the date December 21, 1997. Don't omit the quotes around the date value▪if you do, ObjectPAL performs division on the values.

The Date type includes methods defined for the AnyType type and the DateTime type. Refer to AnyType and DateTime for more information.

Date values are formatted as specified by the **formatSetDateDefault** method (System type), or by ObjectPAL formatting statements.

Dates in the 20th century can be specified using two digits for the year, as in

```
myDay = date("11/09/59")        ; November 9, 1959
```

Dates in the 2nd through the 10th centuries must include three digits of the year (as in 12/17/243); dates in the 11th through 19th centuries must have four digits (12/17/1043). The year cannot be omitted completely.

Note: Paradox treats all dates in the B.C. era as leap years. Support for B.C. era dates was added in version 5.0.

The Date type includes several derived methods from the DateTime and AnyType types.

The Date type includes several methods defined for the DateTime type. Refer to DateTime for more information.

Methods for the Date type

AnyType	DateTime	**Date**
blank	day	**date**
dataType	daysInMonth	**dateVal**
isAssigned	dow	**today**
isBlank	dowOrd	
isFixedType	doy	
view	isLeapYear	
	month	
	moy	
	year	

Changes to Date type methods

The following table lists new methods and methods that were changed for version 7.

New	Changed
(None)	date

▪

date method

See also Example Date Type
Beginner

Returns a Date value.

Syntax

```
1. date ( const value AnyType ) Date
2. date ( ) Date
3. date ( month SmallInt, day SmallInt, year SmallInt ) Date
```

Description

date casts (converts) **value** as a date. If the date supplied in **value** is invalid, the method fails. If you do not supply **value**, **date** returns the current system date as a Date value.

If you use syntax 3, the month can range from 1 to 12. The day range depends on the month and can range from 1 to 28 or 31. The year can range from -9999 to 9999. The year must be entered without abbreviations. That is, all four digits must be used (1995 rather than 95). An error is returned if a specific value is not within the required range. For example, entering 40 for the day generates a 'Bad Day Specification'.

▪

date example

The following example casts a string value as a date, uses the date value in a calculation, then displays the result in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  s String
  d Date
endVar

s = "11/11/99"    ; s is a String value
d = date(s) + 7   ; convert String type to a Date type
                  ; and add 7 days

d.view()          ; show value of d in a dialog box (11/18/99)
                  ; dialog box title displays "Date"
endMethod
```

▪

dateVal procedure

See also Example Date Type

Returns a value as a date.

Syntax

dateVal (const ***value*** AnyType **)** Date

Description

dateVal returns a value as a date.

■

dateVal example

In the following example, the **pushButton** method for a button uses **dateVal** to get the date equivalent of a String value and displays the value in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  s String
  d Date
endVar

s = "11/11/99"   ; s is a String value
d = dateVal(s)   ; d holds the date equivalent of s

d.view()         ; show value of d in a dialog box (11/11/99)
                 ; dialog box title displays "Date"
endMethod
```

today procedure

See also   Example   Date Type

Returns the current date.

Syntax

```
today ( ) Date
```

Description

today returns the current date, according to the system clock/calendar.

▪

today example

The following example displays the current date in a dialog box:

```
; CurrentDate::pushButton
method pushButton(var eventInfo Event)
msgInfo("Today's Date", today())       ; displays the current date
endMethod
```

DateTime type

A DateTime variable stores data in the form hour-minute-second-millisecond year-month-day. DateTime values are used only in ObjectPAL calculations; you cannot store a DateTime value in a Paradox table. DateTime values must be cast (explicitly declared). For example, the following statements assign to the DateTime variable *dt* a time of 10 minutes and 40 seconds past eleven o'clock and a date of December 21, 1997.

```
var dt DateTime endVar
dt = DateTime("11:10:40 am 12/21/97")
```

The quotes around the value are required.

You can use the following characters as separators: blank, tab, space, comma (,), hyphen (-), slash (/), period (.), colon (:), and semicolon (;). DateTime values are formatted as specified by the **formatSetDateTimeDefault** procedure (System type), or by ObjectPAL formatting statements.

You must specify a DateTime value completely; you can't omit any of the fields, but you can specify a value of zero for any field.

See also methods and procedures defined for the Date type and the Time type. Also, both the Date and Time types include methods derived from the DateTime type.

The DateTime type includes several derived methods from the AnyType type.

Methods for the DateTime type

AnyType	**DateTime**
blank	**dateTime**
dataType	**day**
isAssigned	**daysInMonth**
isBlank	**dow**
isFixedType	**dowOrd**
view	**doy**
	hour
	isLeapYear
	milliSec
	minute
	month
	moy
	second
	year

▪

dateTime method

See also Example DateTime Type
Beginner

Returns a DateTime value.

Syntax

```
1. dateTime ( const value AnyType ) DateTime
2. dateTime ( ) DateTime
```

Description

dateTime casts (converts) *value* as a DateTime data type. If *value* is not supplied, **dateTime** returns the current system date and time as a DateTime value.

▪

dateTime example

The following statements assign to the DateTime variable *dt* a time of 10 minutes and 40 seconds past eleven o'clock and a date of December 21, 1997. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
var dt DateTime endVar
dt = dateTime("11:10:40 am 12/21/97")
```

The quotes around the value are required.

You can use the following characters as separators: blank, tab, space, comma (,), hyphen (-), slash (/), period (.), colon (:), and semicolon (;). DateTime values are formatted as specified by **formatSetDateTimeDefault** (System type), or by ObjectPAL formatting statements.

You must specify a DateTime value completely; you cannot omit any of the fields, but you can specify a value of zero for any field.

▪

day method

See also Example DateTime Type
Beginner

Extracts the day of the month from a date.

Syntax

```
day ( ) SmallInt
```

Description

day extracts the day of the month from a DateTime value and returns a value between 1 and 31. If the DateTime is invalid, the method fails.

day example

In the following example, a button's **pushButton** method displays the current day of the month in a dialog box. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  theDay DateTime
endVar
theDay = DateTime("12:00:00 am 12/22/92")

  ; displays 22 in a dialog box
msgInfo("Day of the month", theDay.day())

endMethod
```

daysInMonth method

See also Example DateTime Type
Beginner

Returns the number of days in a month.

Syntax

```
daysInMonth ( ) SmallInt
```

Description

Given a valid DateTime value, **daysInMonth** returns the number of days in that month. If the DateTime is not valid, the method fails.

daysInMonth example

In the following example, the **pushButton** method for the *FebDays* button displays the number of days in February 1992. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; FebDays::pushButton
method pushButton(var eventInfo Event)
var
  daysInFeb SmallInt
endVar
daysInFeb = daysInMonth(DateTime("5:15:35 AM 2/1/92"))
msgInfo("Number of days", "There are " + String(daysInFeb) +
            " days in February 1992")

  ; displays "There are 29 days in February 1992" in a dialog box
  ; (1992 is a leap year)
endMethod
```

■

dow method

See also Example DateTime Type
Beginner

Returns the day of the week.

Syntax
`dow ( )` `String`

Description
Given a valid DateTime value, **dow** returns the first three letters of the day of the week of that DateTime. If the DateTime is not valid, the method fails.

dow example

The following example displays, in a dialog box, the day of week for a given DateTime. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; showDay::pushButton
method pushButton(var eventInfo Event)
var
  theDate DateTime
endVar

theDate = DateTime("11:20:15 pm 3/9/93")

  ; displays "Tue" in a dialog box
msgInfo("Day of Week", strVal(theDate) + " falls on a " + dow(theDate))

endMethod
```

dowOrd method

See also Example DateTime Type
Beginner

Returns the number of a day of the week.

Syntax
dowOrd ( ) SmallInt

Description
Given a valid DateTime value, **dowOrd** returns an integer from 1 to 7 representing that day's position in the week. Sunday is day 1, Monday is day 2, and so on. If the DateTime is not valid, the method fails.

dowOrd example

The following example displays the day of the week as an entire word (such as "Monday") rather than an abbreviation or a number. This code uses **dowOrd** to retrieve the appropriate subscript of a fixed array, then displays the value of the array element in a dialog box. This code is attached to the **pushButton** method for the *fullDay* button. This example assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; fullDay::pushButton
method pushButton(var eventInfo Event)
var
  fullDays Array[7] String
  givenDate         DateTime
endVar

fullDays[1] = "Sunday"
fullDays[2] = "Monday"
fullDays[3] = "Tuesday"
fullDays[4] = "Wednesday"
fullDays[5] = "Thursday"
fullDays[6] = "Friday"
fullDays[7] = "Saturday"

givenDate = DateTime("5:35:20 AM 12/25/93")
  ; this displays "Saturday" in a dialog box
msgInfo("Day of the week", fullDays[dowOrd(givenDate)])

endMethod
```

▪

doy method

See also Example DateTime Type
Beginner

Returns the number of a day of the year.

Syntax
`doy ( ) SmallInt`

Description
Given a valid DateTime, **doy** returns an integer from 1 to 366 representing that day's position in the year. January 1 is day 1, February 1 is day 32, and so on. If the DateTime is not valid, the method fails.

doy example

The following example displays a day's position in a specified year. This code is attached to a button's **pushButton** method. This example assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  theDate DateTime
endVar

theDate = DateTime("5:35:20 AM 6/1/92")

  ; this displays "5:35:20, 6/1/92 is
  ; 153 days past the first of the year"
msgInfo("Date", String(theDate) + " is " + String(theDate.doy()) +
                " days past the first of the year.")

endMethod
```

hour method

See also Example DateTime Type
Beginner

Extracts as a number the hour from a DateTime.

Syntax

`hour ( ) SmallInt`

Description

Given a valid DateTime, **hour** returns an integer representing the hour of the day in 24-hour format. This method fails if the DateTime is not valid.

hour example

The following code extracts the hour from a given DateTime and displays it in a dialog box. Note that even though the DateTime given is in 12-hour format, **hour** returns the 24-hour equivalent.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dt DateTime
endVar

dt = DateTime("8:15:18 pm 12/29/92")
msgInfo("Hour", dt.hour())     ; displays 20 in a dialog

endMethod
```

isLeapYear method

See also Example DateTime Type
Beginner

Reports whether a year has 366 days.

Syntax

```
isLeapYear ( ) Logical
```

Description

Given a valid DateTime, **isLeapYear** returns True if the year within DateTime has 366 days; otherwise, it returns False. This method fails if the DateTime is not valid.

Note: Paradox treats all dates in the B.C. era as leap years, so **isLeapYear** always returns True for B.C. era dates. Support for B.C. era dates was added in version 5.0.

▪

isLeapYear example

For the following example, the **pushButton** method for the *testLeapYr* button displays a True if the given DateTime is a leap year; otherwise the method displays False. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  bDay      DateTime
  leapYear Logical
endVar

bDay = DateTime("5:35:20 AM 6/1/92")

leapYear = bDay.isLeapYear()
leapYear.view("bDay")          ; displays True

endMethod
```

milliSec method

See also Example DateTime Type
Beginner

Extracts as a number the milliseconds from a DateTime.

Syntax

`milliSec ( ) SmallInt`

Description

Given a valid DateTime, **milliSec** returns an integer representing the milliseconds. This method fails if the DateTime is not valid.

milliSec example

The following example constructs a DateTime value from integer calculations, then displays the milliseconds portion of the DateTime in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dt DateTime
  oneSecond, oneMinute, oneHour LongInt
endVar
oneSecond = 1000                ; milliseconds
oneMinute = oneSecond * 60
oneHour   = oneMinute * 60

  ; the following statement assigns dt a DateTime value
  ; of "1:20:30.4 pm 00/00/00" (the statement does not
  ; assign a date, so DateTime sets date portion to 0)
dt = DateTime(13 * oneHour   +
              20 * oneMinute +  ; specifies 1:20 pm
              30 * oneSecond +  ; + 30 seconds
              400)              ; + 400 milliseconds

msgInfo("Milliseconds", dt.milliSec())   ; displays 400

endMethod
```

minute method

See also   Example   DateTime Type
Beginner

Extracts as a number the minutes from a DateTime.

Syntax

minute ( ) SmallInt

Description

Given a valid DateTime, **minute** returns an integer representing the minutes. This method fails if the DateTime is not valid.

minute example

For the following example, the **pushButton** method for *thisButton* displays the minutes portion of a given DateTime. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dt DateTime
endVar

dt = DateTime("9:20:15 am 8/2/93")

msgInfo("Minutes", dt.minute())    ; displays 20

endMethod
```

month method

See also Example DateTime Type
Beginner

Extracts as a number the month from a DateTime.

Syntax
`month ( ) SmallInt`

Description
Given a valid DateTime, **month** returns an integer representing the position in the year of that date's month. January is month 1, February is month 2, and so on. This method fails if the DateTime is not valid.

■

month example

The following example displays the month of the year as an entire word (such as "August") rather than an abbreviation or a number. This code uses **month** to retrieve the appropriate subscript of a fixed array, then displays the value of the array element in a dialog box. This code is attached to the **pushButton** method for the *fullMonth* button. This example assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; fullMonth::pushButton
method pushButton(var eventInfo Event)
var
  fullMonth Array[12] String
  orderDate DateTime
endVar

fullMonth[1] =  "January"
fullMonth[2] =  "February"
fullMonth[3] =  "March"
fullMonth[4] =  "April"
fullMonth[5] =  "May"
fullMonth[6] =  "June"
fullMonth[7] =  "July"
fullMonth[8] =  "August"
fullMonth[9] =  "September"
fullMonth[10] = "October"
fullMonth[11] = "November"
fullMonth[12] = "December"

orderDate = DateTime("5:35:20 AM 9/18/93")

  ; this displays "September" in a dialog box
msgInfo("Order Month", fullMonth[month(orderDate)])

endMethod
```

▪

moy method

See also Example DateTime Type
Beginner

Extracts as a string the month from a DateTime.

Syntax

`moy ( ) String`

Description

Given a valid DateTime, **moy** returns the first three letters of the name of that date's month. This method fails if the DateTime is not valid.

moy example

For the following example, the **pushButton** method for *thisButton* displays the abbreviated month name of a specified DateTime. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  orderDate DateTime
endVar

orderDate = DateTime("2:09:00 AM 3/3/97")
msgInfo("Order date", orderDate.moy())    ; displays Mar

endMethod
```

▪

second method

See also Example DateTime Type
Beginner

Extracts as a number the seconds from a DateTime.

Syntax

`second ( ) SmallInt`

Description

Given a valid DateTime, **second** returns an integer representing the seconds. This method fails if the DateTime is not valid.

▪

second example

The following example constructs a DateTime value from integer calculations, then displays the seconds portion of the DateTime in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dt DateTime
  oneSecond, oneMinute, oneHour LongInt
endVar
oneSecond = 1000                    ; milliseconds
oneMinute = oneSecond * 60
oneHour   = oneMinute * 60

  ; the following statement assigns dt a DateTime value
  ; of "1:20:30.4 pm 00/00/00" (the statement does not
  ; assign a date, so DateTime sets date portion to 0)
dt = DateTime(13 * oneHour   +
              20 * oneMinute +   ; specifies 1:20 pm
              30 * oneSecond +   ; + 30 seconds
              400)               ; + 400 milliseconds

msgInfo("Seconds", dt.second())  ; displays 30

endMethod
```

▪

year method

See also Example DateTime Type
Beginner

Extracts as a number the year from a DateTime.

Syntax
`year ( ) SmallInt`

Description
Given a valid DateTime, **year** returns an integer representing the year within the DateTime. If the DateTime is invalid, this method fails.

year example

For the following example, the **pushButton** method for the *yearButton* button displays the four-digit year for a specified DateTime. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; yearButton::pushButton
method pushButton(var eventInfo Event)
var
  orderDate DateTime
endVar

orderDate = DateTime("2:15:24 pm 3/3/97")
msgInfo("Order date", orderDate.year())   ; displays 1997

endMethod
```

DDE type

Dynamic data exchange (DDE) is a Windows protocol that lets Paradox share data with other applications that behave according to the DDE protocol. Using DDE methods, you have access to data created and stored in another application. You can also use DDE methods to send commands and data to other applications.

Note: When you use DDE to access Paradox from another application, the application name for Paradox is PDXWIN32.

Note: Paradox and ObjectPAL also support OLE, another protocol for sharing data between applications. Refer to the OLE type and to the *Guide to ObjectPAL*, and to About OLE in the User's Guide help for more information.

Methods for the DDE type

DDE

close

execute

open

setItem

▪

close method

See also Example DDE Type

Closes a DDE link.

Syntax

```
close ( )
```

Description

close ends a DDE conversation by closing the link between Paradox and the other application. It does not affect the other application.

▪

close example

This example closes the DDE link between Paradox and the other application, but the other application stays open. To close the other application, you can use execute with the application-specific command (if available) before you use close. The following example gets data from a Quattro Pro for Windows worksheet, then uses the Quattro Pro for Windows macro command {FileExit} to close Quattro Pro for Windows before the close method is called.

```
var
   ddeVar  DDE
   Winery  AnyType
endVar

ddeVar.open("QPW", "C:\\QPW\\SAMPLES\\WINES.WB2", "$A:$C$2")

Winery = ddeVar
msginfo("First Winery", Winery)

ddeVar.execute("{FileExit}")

ddeVar.close()
```

■

execute method

See also   Example   DDE Type

Sends a command via a DDE link.

Syntax

`execute` `( const` ***`command`*** `String )`

Description

execute sends the string *command* to an application via a DDE link. The nature of *command* will vary from one application to another. For example, a string that makes perfect sense to a word processing program may not be understood by a spreadsheet, and spreadsheets from different manufacturers may use different commands to perform similar activities.

▪

execute example

The following example uses the Quattro Pro for Windows macro command INDICATE to set the status line indicator, and the macro command LET to set cells of the Wines worksheet representing Year, Quarter, and Winery.

```
var
   ddeVar   DDE
endVar

ddeVar.open("QPW", "C:\\QPW\\SAMPLES\\WINES.WB2")

ddeVar.execute("{INDICATE \"NewInfo\"}")
ddeVar.execute("{LET A146,1993}")
ddeVar.execute("{LET B146,Q1:string}")
ddeVar.execute("{LET C146,Duckhorn:string}")

ddeVar.close()
```

open method

See also Example DDE Type

Opens a DDE link to another application.

Syntax

1. **open (** const ***server*** String **)** Logical
2. **open (** const ***server*** String, const ***topic*** String **)** Logical
3. **open (** const ***server*** String, const ***topic*** String, const ***item*** String **)** Logical

Description

open creates a DDE link to the application *server*, and tells *server* to open the document *topic* (optional) at a location specified in *item* (optional).

This method returns True if application *server* is successfully opened; otherwise it returns False. If the server application cannot open *topic*, or if *item* cannot be found in *topic*, this method fails.

The nature of *item* varies from one application to another. For example, a string that makes perfect sense to a word processing program may not be understood by a spreadsheet, and spreadsheets from different manufacturers may use different commands to perform similar activities.

open example

Opens two DDE links and stores the values in ObjectPAL variables.

```
var
   d1, d2                DDE
   Appellation, Region   AnyType
endVar

d1.open("QPW", "C:\\QPW\\SAMPLES\\WINES.WB2", "$A:$D$2")
d2.open("QPW", "C:\\QPW\\SAMPLES\\WINES.WB2", "$A:$E$2")

Appellation = d1
Region = d2

msgInfo("Wines Information",
        "Appellation is: " + String(Appellation) +
        ", Region is " + String(Region))

d1.close()
d2.close()
```

setItem method

See also Example DDE Type

Specifies an item in a DDE conversation.

Syntax

`setItem ( ``const `***`item`***` String `**`)`**

Description

setItem is used in a DDE link with application and topic established. The argument *item* specifies a new item. The nature of *item* varies from application to application. For example, a string that makes perfect sense to a word processing program may not be understood by a spreadsheet, and spreadsheets from different manufacturers may use different commands to do the same thing.

▪

setItem example

The following example uses **setItem** twice to get the values of two cells in a QPW worksheet.

```
var
   winesLink            DDE
   Appellation, Region  AnyType
endVar

; link to the QPW worksheet
winesLink.open("QPW", "C:\\QPW\\SAMPLES\\WINES.WB2")

winesLink.setItem("$A:$D$2")     ;// item is cell A:D2
Appellation = winesLink          ;// sets Appellation = cell D2

winesLink.setItem("$A:$E$2")     ;// item is cell A:E2
Region = winesLink               ;// sets Region = cell E2

msgInfo("Wines Information",
        "Appellation is: " + String(Appellation)+
        ", Region is " + String(Region))

winesLink.close()
```

DynArray type

A DynArray is a flexibly structured dynamic array. A dynamic array is a compact storage structure for any combination of data types. Using a DynArray, you can look up values quickly, even when the dynamic array contains a large number of items.

These arrays are dynamic because you do not specify their size; the dimensions of a DynArray automatically change as items are added to it or removed from it. A DynArray's size is limited only by system memory.

Note: ObjectPAL also supports fixed-size and resizeable arrays. See the Array type for more information.

Unlike fixed-size arrays, the indexes of dynamic arrays are not integers; dynamic array indexes (also called keys) can be any valid ObjectPAL expression that evaluates to a String. Each index in a dynamic array is associated with a value.

The DynArray type includes several derived methods from the AnyType type.

Methods for the DynArray type

AnyType	**DynArray**
blank	**contains**
dataType	**empty**
isAssigned	**getKeys**
isBlank	**removeItem**
isFixedType	**size**
	view

▪

contains method

See also Example DynArray Type

Searches the indexes in a DynArray for a value.

Syntax

`contains ( const value AnyType ) Logical`

Description

contains returns True if the index of any element in a DynArray matches *value* character for character; otherwise, it returns False. **contains** is not case sensitive.

contains example

The following example uses **contains** to test whether a dynamic array index corresponds to a menu item. In this example, the form's **open** method creates a menu and assigns several values to a dynamic array. When the user selects an item from the menu, the form's **menuAction** method compares the menu selection with indexes in the DynArray. If a DynArray index is defined for the selected menu item, the **menuAction** method displays the value associated with that DynArray element; otherwise it displays the value of another element.

This code goes in the form's Var window:

```
; thisForm::Var
var
  msg DynArray[] AnyType    ; stores messages
  m1             Menu       ; menu bar
  p1             PopUpMenu  ; pop-up attached to menu item
  choice         String     ; user's menu selection
endVar
```

The code immediately following is attached to the **open** method of a form:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

    p1.addText("Time")              ; add items to the pop-up menu
    p1.addText("Date")
    p1.addText("Colors")

    m1.addPopUp("&Utilities", p1)   ; attach the pop-up to a menu bar item
    m1.show()                       ; show the menu bar

      ; Now initialize the msg dynamic array. msg Indexes correspond to
      ; the pop-up menu items generated above. msg values are values that
      ; appear in a dialog box when the user selects a menu. Note that
      ; msg does NOT contain a "Colors" index.
    msg["Time"] = time()            ; show current date for "Time" selection
    msg["Date"] = date()            ; show current date for "Date" selection
    msg["Error"] = "Sorry, this menu selection is not implemented."

endif
endMethod
```

This code is attached to the **menuAction** method of a form:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form

    choice = eventInfo.menuChoice()

    if isBlank(choice) = False then    ; if user selected a menu
      if msg.contains(choice) then     ; if selection matches an index in
                                       ; the msg dynamic array
        msgInfo(choice, msg[choice])   ; display the value of that element
      else                             ; else selection didn't match an
element
        msgStop("Stop!", msg["Error"]) ; display the value of another element
      endif
    endif

  else
    ;code here executes just for form itself
endif
endMethod
```

empty method

[See also](#) [Example](#) [DynArray Type](#)

Removes all items from a dynamic array.

Syntax

```
empty ( )
```

Description

empty removes all items from an dynamic array. The size of the DynArray becomes 0.

empty example

The following example shows how **empty** functions for a dynamic array. The code immediately following declares a dynamic array in a form's Var window. This dynamic array is global to all objects on the form.

```
; thisForm::Var
Var
  myCar DynArray[] AnyType  ; declare a dynamic array
endVar
```

The following code is attached to the **pushButton** method of the *fillButton*. When this button is pressed, the code assigns several elements of the *myCar* DynArray.

```
; fillButton::pushButton
method pushButton(var eventInfo Event)

myCar["Make"]  = "Porsche"  ; load the DynArray
myCar["Model"] = "911 sc"
myCar["Color"] = "Dark Blue"
myCar["Year"]  = 1986
  ; display myCar DynArray and indicate size in the title (4)
myCar.view("myCar size: " + String(myCar.size()))
endMethod
```

The following code is attached to the **pushButton** method of the *emptyButton* button. When this button is pressed, the code empties the *myCar* array and displays its contents.

```
; emptyButton::pushButton
method pushButton(var eventInfo Event)
myCar.empty()     ; empty the myCar DynArray

  ; display myCar DynArray and indicate size in the title (0)
myCar.view("myCar size: " + String(myCar.size()))
endMethod
```

▪

getKeys method

See also Example DynArray Type

Loads a resizeable array with indexes of an existing DynArray.

Syntax

getKeys (var ***keyNames*** Array[ ] String **)**

Description

getKeys creates the resizeable array specified in *keyNames* and assigns to the values of each element the index in the DynArray. In other words, this method stores all index values from a DynArray in a resizeable array. If *keyNames* exists, it is overwritten without asking for confirmation. Index values are sorted into the new array such that the lowest index value becomes *keyNames*[1], and so on.

getKeys example

The following example assigns several elements to the *myCar* DynArray, then uses **getKeys** to create an array that stores *myCar* indexes. The results are displayed in a **view** dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myCar DynArray[] AnyType
  ar    Array[] String
endVar

; add some elements to the DynArray
myCar["Make"]  = "Porsche"  ; load the DynArray
myCar["Model"] = "911 sc"
myCar["Color"] = "Dark Blue"
myCar["Year"]  = 1986

; now grow ar to 4 items then view the
; new array in a dialog box
myCar.getKeys(ar)
ar.view()

; displays
;  Color      (ar[1])
;  Make       (ar[2])
;  Model      (ar[3])
;  Year       (ar[4])

endMethod
```

▪

removeItem method

See also Example DynArray Type

Deletes a specified item from a DynArray.

Syntax

`removeItem (` `const` ***`value`*** `AnyType` **`)`**

Description

removeItem deletes the element (specified by its index) in *value* from a DynArray. **removeItem** is case insensitive.

▪

removeItem example

The following example concatenates two values in a dynamic array, then uses **removeItem** to remove the obsolete element.

The code immediately following is attached to a form's Var window:

```
; thisForm::Var
var
  CustInfo DynArray[] AnyType
endVar
```

This code is attached to the **pushButton** method for the *getCustInfo* button. This code loads the dynamic array with street address information. Your application might have a custom method that loads the dynamic array from a table, or from information entered by the user.

```
; getCustInfo::pushButton
method pushButton(var eventInfo Event)
  ; load the DynArray
CustInfo["Company"] = "Ultra-Fast Computers"
CustInfo["Street"]  = "1234 Able Street"
CustInfo["City"]    = "Anywhere"
CustInfo["State"]   = "Your State"
CustInfo["Zip"]     = "99444"
CustInfo["ZipExt"]  = "9344"

  ; display contents of the CustInfo Dynarray
CustInfo.view("Contents of CustInfo")
endMethod
```

In the code that follows, the value of the ZipExt element (if it exists) is concatenated to the value of the Zip element. Since the ZipExt element is no longer needed, this code removes it from the dynamic array. The following code is attached to the **pushButton** method for the *catZipExt* button.

```
; catZipExt::pushButton
method pushButton(var eventInfo Event)
if CustInfo.contains("ZipExt") then
  CustInfo["Zip"] = CustInfo["Zip"] + "-" + CustInfo["ZipExt"]
  CustInfo.removeItem("ZipExt")       ; remove obsolete element
else
  msgInfo("Once is enough", "Zip code has been concatenated")
endif
  ; display the results
CustInfo.view("Contents of CustInfo")
endMethod
```

▪

size method

See also Example DynArray Type

Returns the number of elements in a DynArray.

Syntax

```
size ( ) LongInt
```

Description

size returns the number of elements in a DynArray.

-

size example

For the following example, the **pushButton** method for *thisButton* creates a dynamic array, then displays its size in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dy DynArray[] String
endVar

dy["Name"]     = "MAST"          ; load the DynArray
dy["Business"] = "Diving"
dy["Contact"]  = "Jane Doherty"

  ; this displays "dy has 3 elements"
msgInfo("dy", "dy has " + string(dy.size()) + " elements.")
endMethod
```

view method

See also Example DynArray Type

Displays the contents of a DynArray in a dialog box

Syntax

`view ( [ const title String ] )`

Description

view list the indexes and elements of a DynArray in a modal dialog box. ObjectPAL execution suspends until the user closes this dialog box. You can specify a title for the dialog box in *title*, or you can omit *title* to display "DynArray" instead. **view** sorts the DynArray on its index before displaying the dialog box.

Unlike many other data types, DynArray values displayed in a **view** dialog box cannot be changed interactively. See AnyType::**view** for information regarding other data types and **view**.

▪

view example

For the following example, the **pushButton** method for the *thisButton* button creates a dynamic array, then displays its contents sorted in a dialog box.

```
;thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dy DynArray[] String
endVar

dy["one"] = "first"
dy["two"] = "second"
dy["three"] "third"
dy.view("This DynArray contains:")
  ; displays the following:
  ; This DynArray contains:
  ; one     first
  ; three   third
  ; two     second
endMethod
```

ErrorEvent type

See also

The ErrorEvent type provides methods you can use to get and set information about errors that occur as ObjectPAL code executes. The only built-in event method triggered by an ErrorEvent is **error**.

The ErrorEvent type includes several derived methods from the Event type.

Methods for the ErrorEvent type

Event	ErrorEvent
errorCode	**reason**
getTarget	**setReason**
isFirstTime	
isPreFilter	
isTargetSelf	
setErrorCode	

▪

User-defined error constants

See also ErrorEvent Type

You can define your own error constants, but you must keep them within a specific range. Because this range is subject to change in future versions of Paradox, ObjectPAL provides the IdRanges constants UserError and UserErrorMax to represent the minimum and maximum values allowed.

For example, suppose that you want to define two error constants, ThisError and ThatError. In a Const window, define values for your custom constants as follows:

```
Const
   ThisError = 1
   ThatError = 2
EndConst
```

Then, to use one of these constants, add it to UserError. For example,

```
method error(var eventInfo ErrorEvent)
   if eventInfo.errorCode() = UserError + ThisError then
      doSomething()
   endIf
endMethod
```

By adding UserError to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, use the value of UserErrorMax. One way to check the value is with a **message** statement:

```
message(UserErrorMax)
```

In this version of Paradox, the difference between UserError and UserErrorMax is 2046. That means the largest value you can use for an error constant is UserError + 2046. (The error code 0 is reserved to mean "no error.")

▪

reason method

See also Example ErrorEvent Type

Reports why an error occurred.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report why an ErrorEvent occurred. ObjectPAL provides ErrorReasons constants for testing the value returned by **reason**:

Note: Do not confuse **reason** with **errorCode**, which returns a value to identify an error.

▪

reason example

The following code is attached to the built-in **error** method for the form. This code reports the error code, the reason, and the message associated with the error.

```
; thisForm::error
method error(var eventInfo ErrorEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    msgInfo("Error", eventInfo.errorCode())
    if eventInfo.reason() = ErrorWarning then
      msgInfo("Warning Error", errorMessage())
    else
      msgInfo("Critical Error", errorMessage())
    endif
    disableDefault
  else
    ; code here executes just for form itself

endif
endMethod
```

▪

setReason method

See also Example ErrorEvent Type

Specifies a reason for generating an ErrorEvent.

Syntax

```
setReason ( const reasonId SmallInt )
```

Description

setReason specifies a reason for generating an ErrorEvent. This method takes an ErrorReasons constant as an argument.

setReason example

The following example creates an ErrorEvent, sets the reason to ErrorWarning, then sends the ErrorEvent to the form.

```
; sendAnError::pushButton
method pushButton(var eventInfo Event)
var
   ev  ErrorEvent
endVar
ev.setErrorCode(1)           ; set an error code of 1
                             ; (any nonzero will do)
ev.setReason(ErrorWarning) ; set the reason to ErrorWarning
thisForm.error(ev)           ; send the error to the form
endMethod
```

Event type

The Event type is the base type from which the other event types (for example, ActionEvent) are derived. Many of the methods listed here are used by the other event types as derived methods.

The following built-in event methods are triggered by Events: **open**, **close**, **setFocus**, **removeFocus**, **newValue**, and **pushButton**.

Methods for the Event type

Event

errorCode

getTarget

isFirstTime

isPreFilter

isTargetSelf

reason

setErrorCode

setReason

▪

errorCode method

See also Example Event Type
Beginner

Reports the status of an error flag.

Syntax

```
errorCode ( ) SmallInt
```

Description

errorCode returns a nonzero error code if there is an error; otherwise, **errorCode** returns 0. To test for a specific error, use the ObjectPAL Errors constants (for example, peDiskError) or a user-defined error constant. To create a list of the Error constants and the corresponding error messages, use **enumRTLErrors.** You can also use the Constants browser in the ObjectPAL IDE to browse through the list of Error constants.

errorCode example

In the following example, assume that a form contains a field object bound to the Quant field of the *Orders* table. When the field's value changes, this code executes the built-in code for this method, then checks to see if an error occurred.

```
; Quant::changeValue
method changeValue(var eventInfo ValueEvent)
   doDefault
   ; check the event to see if it has an error
   if eventInfo.errorCode() <> 0 then
      errorShow() ; Display the error message in a dialog box.
   endif
endMethod
```

▪

getTarget method

See also Example Event Type

Creates a handle to the target of an Event.

Syntax

getTarget (var ***target*** UIObject **)**

Description

getTarget returns in *target* the handle of the UIObject that was the target of the most recent Event. The target does not change as the event bubbles up the containership hierarchy.

■

getTarget example

The following example assumes that a number of fields from the Customer table are placed on a form. As the user moves from field to field, the **setFocus** method on the form identifies the target of the event, finds out if the target is a field, and, if so, changes the current field's color to light blue. This provides a more dramatic visual clue to the user than the normal highlight. The field's previous color is stored in the global variable *oldFieldColor*. When the focus is removed from the field, the form's **removeFocus** method restores the field to its original color. The previous field color is stored in a variable declared in the Var window of the form, as shown in the following code:

```
; thisForm::Var
Var
  oldFieldColor LongInt     ; to store the previous color of the field
endVar
```

The following code is attached to the **setFocus** method of the form:

```
; thisForm::setFocus
method setFocus(var eventInfo Event)
var
  targObj    UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    ; get the target
    eventInfo.getTarget(targObj)
    if targObj.Class = "Field" then  ; if it's a field, change its color
      oldFieldColor = targObj.Color  ; save old color in var global to form
      targObj.Color = LightBlue      ; highlight field on focus
    endif
  else
    ; code here executes just for form itself

endif
endMethod
```

This code is attached to the form's **removeFocus** method:

```
; thisForm::removeFocus
method removeFocus(var eventInfo Event)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    ; get the target
    eventInfo.getTarget(targObj)
    if targObj.Class = "Field" then  ; if it's a field,
      targObj.Color = oldFieldColor  ; restore color from global var
    endif
  else
    ; code here executes just for form itself

endif
endMethod
```

▪

isFirstTime method

See also Example Event Type

Reports whether the form is handling an event for the first time before dispatching it.

Syntax

isFirstTime ( ) Logical

Description

isFirstTime reports whether the form is handling an event before dispatching it to the target object, or whether the event has been dispatched and has subsequently bubbled up the containership hierarchy. This method returns True if the form is handling the event for the first time; otherwise, it returns False. Use **isFirstTime** in built-in event methods attached to the form.

▪

isFirstTime example

The following example shows how you can use **isFirstTime** with **isTargetSelf** to evaluate an event in a form-level method. This code replaces the default code for the form's **pushButton** method, which normally tests **isPreFilter**.

```
; thisForm::pushButton
method pushButton(var eventInfo Event)
var
  targObj   UIObject
endVar
; This example breaks out isFirstTime and isTargetSelf from isPreFilter.
; Three valid possibilities.
; Form's own event :
  ; isTargetSelf = True, isFirstTime = True

; Dispatched events (prefiltered events):
  ; isTargetSelf = False, isFirstTime = True

; Bubbled events (explicitly passed):
   ; isTargetSelf = False, isFirstTime = False

; For the form, isTargetSelf is never True when isFirstTime is False.

eventInfo.getTarget(targObj)    ; get the target to targObj
switch
  case eventInfo.isTargetSelf() AND eventInfo.isFirstTime() :
  ; This happens only when the form is handling its own event.
       msgInfo("Status",
               "This line will not execute for pushButton events.")

  case NOT eventInfo.isTargetSelf() AND eventInfo.isFirstTime() :
    ; This happens only when the form is dispatching an event
    ; for another object. isPrefilter returns True.

       msgInfo("Status", "Dispatching a pushButton event to "
                         + targObj.Name + ".")

  case NOT eventInfo.isTargetSelf() AND NOT eventInfo.isFirstTime() :
    ; The event has been explicitly bubbled back to the form.
    ; isPrefilter returns False.

       msgInfo("Status", "A pushButton Event " +
               "has been explicitly bubbled back to the form.")
endswitch

endMethod
```

The following code is attached to the **pushButton** method for the form's *testPassEvent* button. When the form's **pushButton** method has prefiltered the event and dispatched it back to the button, the button's **pushButton** method returns it to the form with the command **passEvent**. When the event returns to the form, the methods **isTargetSelf**, **isFirstTime**, and **isPrefilter** all return False.

```
; testPassEvent::pushButton
method pushButton(var eventInfo Event)
passEvent    ; bubble the event up the hierarchy
endMethod
```

▪

isPreFilter method

See also Example Event Type

Reports whether the form is handling an event on its own behalf.

Syntax

isPreFilter ( ) Logical

Description

isPreFilter reports whether the form is handling an event on its own behalf or on behalf of another object. It returns True only when the target is some object other than the form, and the form has not already handled this Event. **isPreFilter** is logically equivalent to the form evaluating the following statement:

```
if (NOT eventInfo.isTargetSelf()) AND eventInfo.isFirstTime()
```

This method returns True for all internal methods, and for all external methods when they first reach the form. When the external methods bubble back to the form, this method returns False. See About built-in methods for information about internal and external methods.

Note: Form methods are *not* prefiltered. In other words, when an Event occurs for the form, **isPreFilter** returns False.

isPreFilter example

See the example for **getTarget**.

▪

isTargetSelf method

See also Example Event Type

Reports whether an object is the target of an Event.

Syntax

```
isTargetSelf ( ) Logical
```

Description

isTargetSelf reports whether an object is the target of an Event. Use **isTargetSelf** in built-in event methods attached to the form.

▪

isTargetSelf example

See the example for **isFirstTime.**

▪

reason method

See also Example Event Type

Reports why an Event occurred.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report why an event occurred. The return value depends on the type of event. ObjectPAL provides the ValueReasons constants for testing the value returned by **reason** for Events. ErrorReasons constants are defined for ErrorEvents, MenuReasons constants for MenuEvents, MoveReasons constants for MoveEvents, and StatusReasons constants for StatusEvents

The **reason** method is valid for the other event types (ActionEvent, KeyEvent, MouseEvent, and ValueEvent), but it returns zero. **setReason** is also valid for ActionEvent, KeyEvent, MouseEvent, and ValueEvent, but you can use it only to set user-defined Reason constants (an advanced technique).

▪

reason example

In the following example, assume that a form contains a multirecord object bound to the *Orders* table, and that the *Ship_VIA* field is a set of radio buttons. Assume also that the form is in Edit mode. The following **newValue** method for *Ship_VIA* displays a message indicating why **newValue** was called. When the form opens, the Reason will be StartupValue.

```
; Ship_VIA::newValue
method newValue(var eventInfo Event)
; show why the newValue method was called
msgInfo("newValue reason",
    iif(eventInfo.reason() = StartupValue, "StartupValue",
    iif(eventInfo.reason() = FieldValue, "FieldValue", "EditValue")))
endMethod
```

When the user scrolls through the table or clicks the *nextRec* button, the Reason will be FieldValue.

```
; nextRec::pushButton
method pushButton(var eventInfo Event)
action(DataNextRecord)     ; this triggers a newValue for Ship_Via
                           ; with a Reason constant FieldValue
endMethod
```

When the user chooses a different radio button on Ship_VIA or clicks the *changeRadio* button, the Reason will be EditValue.

```
; changeRadio::pushButton
method pushButton(var eventInfo Event)
ORDERS.Ship_Via = "US Mail"     ; this triggers a newValue for Ship_Via
                                ; with a Reason of EditValue
endMethod
```

▪

setErrorCode method

See also Example Event Type
Beginner

Sets the error code for an Event.

Syntax
```
setErrorCode ( const errorId SmallInt )
```

Description

setErrorCode sets the error code in the event packet. If *errorId* is 0, it means "no error." Any nonzero value for *errorId* indicates an error; to indicate a specific error, use an EventErrorCodes constant or a user-defined error constant.

Calling **setErrorCode** is not the same as calling **errorLog**, which adds error information directly to the error stack. **setErrorCode** adds the error code to the current event packet; this code may or may not be added to the error stack, depending on how custom code and built-in code handles it. For more information about the event packet and the error stack, refer to the *Guide to ObjectPAL*.

▪

setErrorCode example

See the example for **errorCode.**

setReason method

See also Example Event Type

Specifies a reason for generating a move.

Syntax

```
setReason ( const reasonId SmallInt )
```

Description

setReason specifies in *reasonId* a reason for generating an Event in an object's built-in **newValue** method, where *reasonId* is a ValueReasons constant.

Note: ErrorReasons constants are defined for ErrorEvents, MenuReasons constants for MenuEvents, MoveReasons constants for MoveEvents, and StatusReasons constants for StatusEvents

See the entry for **setReason** in those sections for examples. **setReason** is also valid for ActionEvent, KeyEvent, MouseEvent, and ValueEvent, but you can use it only to set user-defined Reason constants (an advanced technique).

▪

setReason example

In the following example, assume that a form contains a multirecord object bound to the *Orders* table, and that the *Ship_VIA* field is a set of radio buttons. The following **newValue** method for *Ship_VIA* displays a message indicating why **newValue** was called.

```
; Ship_VIA::newValue
method newValue(var eventInfo Event)
; show why the newValue method was called
msgInfo("newValue reason",
        iif(eventInfo.reason() = StartupValue, "StartupValue",
        iif(eventInfo.reason() = FieldValue, "FieldValue", "EditValue")))
endMethod
```

The following code demonstrates how to set a **reason** for an event and send the event to an object.

```
; triggerValReason::pushButton
method pushButton(var eventInfo Event)
var
  ev  Event
endVar
ev.setReason(FieldValue)       ; set a reason constant for the event
ORDERS.Ship_VIA.newValue(ev)   ; send the event to the Ship_VIA field
endMethod
```

FileSystem type

Changes

FileSystem variables provide access to and information about disk files, drives, and directories. They provide a handle, a variable you can use in ObjectPAL statements to work with a directory or a file. In many cases, the first step is to use **findFirst** to see whether any information is present; this "initializes" the FileSystem variable.

Methods for the FileSystem type

FileSystem

accessRights

clearDirLock

copy

delete

deleteDir

drives

enumFileList

existDrive

findFirst

findNext

freeDiskSpace

fullName

getDir

getDrive

getFileAccessRights

getValidFileExtensions

isDir

isFile

isFixed

isRemote

isRemovable

isValidFile

makeDir

name

privDir

rename

setDir

setDirLock

setDrive

setFileAccessRights

setPrivDir

setWorkingDir
shortName
size
splitFullFileName
startUpDir
time
totalDiskSpace
windowsDir
windowsSystemDir
workingDir

Changes to FileSystem type methods

The following table lists new methods and changed methods for version 5.0:

New	Changed
clearDirLock	fullName
setDirLock	
setPrivDir	
setWorkingDir	
shortName	

The following table lists new methods for version 7.

New	Changed
shortName	None
isValidFile	

▪

accessRights method

See also Example FileSystem Type

Reports access rights (also called file attributes) of a file.

Syntax

accessRights ( ) String

Description

accessRights returns a string describing access rights, which can be one or more of the following: A, D, H, R, S, V (for archive, directory, hidden, read only, system, and volume, respectively). If the returned value is an empty string, the file has no attributes set. You must use **findFirst** before using **accessRights**.

▪

accessRights example

Checks the attributes of the file MEMO14.TXT. Calls **findFirst** to make sure the file exists, then calls **accessRights**. If the file is not marked read-only, calls Notepad to edit the file.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fileName  String
   fs        FileSystem
endVar

fileName = "c:\\pdoxwin\\myfiles\\memo14.txt"

if fs.findFirst(fileName) then

      ; if file attributes include R (read only)
   if search(fs.accessRights(), "R") > 0 then
      msgStop(fileName, "This file is marked read-only.")
   else
      ; run notepad editor for the file
      execute("NotePad.exe " + fileName)
   endIf
else
   msgStop("Error", "Can't find " + fileName)
endIf

endMethod
```

▪

clearDirLock procedure

See also Example FileSystem Type

Unlocks a specified directory.

Syntax

`clearDirLock ( const dirName String ) Logical`

Description

clearDirLock removes a directory lock placed on the directory specified in *dirName*. This method returns True if it succeeds; otherwise, it returns False.

▪

clearDirLock example

See the example for **setDirLock**.

▪

copy method

See also Example FileSystem Type

Copies a file.

Syntax

copy (const ***srcName*** String, const ***dstName*** String **)** Logical

Description

Returns True if successful in copying source file *srcName* to destination file *dstName*; otherwise, it returns False. If *dstName* exists, this method overwrites the file without asking for confirmation. This method copies only one file at a time. DOS wildcard characters are not valid with this method.

▪

copy example

Searches the current directory for the file *sourceFile*. If the file exists, copies it to a new file *destFile*.

```
; copyButton::pushButton
method pushButton(var eventInfo Event)
var
   fs           FileSystem
   sourceFile,
   destFile     String
endVar

sourceFile = "memo14.txt"
destFile = "memo14.bak"

if fs.findFirst(sourceFile) then
   if fs.copy(sourceFile, destFile) then
      message(sourceFile + " copied to " + destFile)
   else
      message("Copy failed...")
   endif
else
   msgInfo(sourceFile, "File not found.")
endIf

endMethod
```

▪

delete method

[See also](#) [Example1](#) [Example2](#) [FileSystem Type](#)

Deletes a file.

Syntax

`**delete** **(** const ***name*** String **)** Logical`

Description

Returns True if it deletes the file *name*; otherwise, returns False. Can delete only one file at a time. You cannot use DOS wildcard characters with **delete**.

▪

delete example 1

Displays a dialog box asking whether the user wants to delete the file *fileName*. If the user chooses Yes, deletes the file:

```
; delOne::pushButton
method pushButton(var eventInfo Event)
var
    fs       FileSystem
    oldFile  String
endVar

fileName = "MyText.old"

if fs.findFirst(fileName) then
   if msgYesNoCancel("Delete?", fileName) = "Yes" then
      fs.delete(fileName)
   endIf
else
   msgInfo(fileName, "File not found.")
endIf

endMethod
```

▪

delete example 2

Uses a **while** loop to delete all files in the current directory that have an extension of .OLD.

```
; delAll::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
endVar

if fs.findFirst("*.old") then
   fs.delete(fs.name())
   while fs.findNext()
      fs.delete(fs.name())
   endWhile
else
   msgInfo("*.OLD", "File not found.")
endIf

endMethod
```

▪

deleteDir method

[See also](#) [Example1](#) [Example2](#) [FileSystem Type](#)

Deletes a directory, but only if the directory is empty (contains no files).

Syntax

deleteDir (const ***name*** String **)** Logical

Description

Returns True if successful in deleting the directory *name*; otherwise, returns False. Does not prompt for confirmation.

▪

deleteDir example 1

Deletes the directory C:\DOS. If not successful (for example, because the directory is not empty), displays an error message.

```
; delDOS::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
endVar

if fs.findFirst("c:\\dos") then
   if not fs.deleteDir("c:\\dos") then
      msgStop("Error", "Could not delete directory.")
   endIf
endIf

endMethod
```

In the following, **enumFileList** checks whether the directory C:\SCAN\SUBSCAN is empty. If so, it creates an array containing one item (the directory name), and **deleteDir** deletes the directory:

```
; delDir1::pushButton
method pushButton(var eventInfo event)
var
   fs  FileSystem
   fileNames Array[] String
endVar

fs.enumFileList("c:\\scan\\subscan", fileNames)

; compare size to 1 because directory has no filespec
if fileNames.size() = 1 then
   fs.deleteDir("c:\\scan\\subscan")
else
   msgStop("Stop", "Directory is not empty.")
endIf

endMethod
```

▪

deleteDir example 2

First, **enumFileList** creates an array containing two items, one each for the current directory and its parent directory (because of the *.* file specification at the end of the path). Then **deleteDir** deletes the directory C:\SCAN\SUBSCAN:

```
; delDir2::pushButton
method pushButton(var eventInfo event)
var
   fs  FileSystem
   fileNames Array[] String
endVar

fs.enumFileList("c:\\scan\\subscan\\*.*", fileNames)

; compare size to 2 because directory has the *.* filespec
if fileNames.size() = 2 then ; size = 2 because of *.* filespec
   fs.deleteDir("c:\\scan\\subscan")
else
   msgStop("Stop", "Directory is not empty.")
endIf

endMethod
```

▪

drives method

See also Example FileSystem Type

Returns the letters of the drives attached to the system and known to Windows.

Syntax

drives ( ) String

Description

drives returns a string containing only the letters (no colons) of the drives attached to the system and known to Windows.

drives example

Displays a dialog box listing the ID letters of the drives attached to the system:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
endVar

; this displays a list of attached drives
; example:  ABCHJKXY
msgInfo("Drives", fs.drives())

endMethod
```

▪

enumFileList method

See also Example FileSystem Type

Lists information about files.

Syntax

```
1. enumFileList ( const fileSpec String, var arrayName Array[ ] String )
2. enumFileList ( const fileSpec String, const tableName String )
```

Description

enumFileList lists information about files matching the criteria in *fileSpec*. If *fileSpec* is *.*, the array or table includes records for the current directory (.) and the parent directory (..).

In syntax 1, writes data to the array *arrayName,* which you must declare before calling this method. The resulting array contains file names and extensions only, not paths.

In syntax 2, writes data to the table *tableName*. If the table does not exist, creates it automatically and enumerates the file list. If *tableName* does not specify a path, creates the table in :WORK:. If the table exists and is open, appends data to it; if it is closed, overwrites its data.

Here is the structure of the table:

Field name	Type & size		Description
Name	Alpha	255	File name (and extension, if any).
Size	Numeric		File size in bytes.
Attributes	Alpha	10	DOS file attributes.
Date	Alpha	10	Date of last modification.
Time	Alpha	10	Time of last modification.

Lists file names in the same order as the directory▪not necessarily in alphabetical order.

▪

enumFileList example

Demonstrates both syntaxes of **enumFileList**. First, **enumFileList** searches the specified directory for forms and uses syntax 1 to create an array of file names, which is displayed in a pop-up menu. Then, **enumFileList** uses syntax 2 to create a table of information on the files and displays it in a Table window.

```
; demoButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   formDir,  theForm String
   formNames Array[] String
   tv  tableView
   p  PopUpMenu
endVar

formDir = "C:\\pdoxwin\\sample\\*.f?l"

if fs.findFirst(formDir) then           ; if one *.f?l is found
   fs.enumFileList(formDir, formNames)  ; create an array of *.f?l files
   p.addArray(formNames)                ; show the array in a pop-up menu
   theForm = p.show()                   ; display a pop-up menu of filenames
endIf

if fs.findFirst(formDir) then           ; if one *.f?l is found
   fs.enumFileList(formDir, "forms.db") ; create FORMS.DB listing *.f?l files
   tv.open("forms.db")                  ; display FORMS.DB table
endIf

endMethod
```

▪

existDrive method

See also Example FileSystem Type

Reports whether a drive is attached to the system.

Syntax

```
existDrive ( const driveLetter String ) Logical
```

Description

existDrive returns True if *driveLetter* is attached to the system; otherwise, it returns False. You specify *driveLetter* with a letter ("C") or a letter and a colon ("C:").

▪

existDrive example

Calls **existDrive** to check whether drive P exists; if so, **setDrive** makes it the default drive.

```
; checkDrive::pushButton
method pushButton(var eventInfo Event)
var
   fs         FileSystem
   driveName  String
endVar

driveName = "P"

if fs.existDrive(driveName) then
   fs.setDrive(driveName)
else
   msgStop("Stop", "Drive " + driveName + " is not attached.")
endIf

endMethod
```

■

findFirst method

See also		Example		FileSystem Type

Searches a file system for a file name.

Syntax

`findFirst ( ``const `***`pattern`***` String `**`)`**` Logical`

Description

findFirst returns True if a file is found whose name matches *pattern*; otherwise, it returns False. *pattern* may contain the DOS wildcard characters * and ?, as used with the DOS command DIR. Examples of pattern include:

- "C:*.*"
- "..\\myDir*.*"
- "*.txt"
- "fr*.db?"

Use **findFirst** to check whether a file or directory exists, and to initialize a FileSystem variable before calling another FileSystem method or procedure. You must fully qualify **findFirst** calls to other than the current default drive or path, unless you reset the default drive and path with the **setDir** method.

Under Windows 95, **findFirst** will also find the 8.3 format of the file name that exists in the file system for long filenames.

Note: **findFirst** finds file and directory names in the order they're listed in the directory, which is not necessarily alphabetical. The first value returned by **findFirst** depends on the path and file specification.

▪

findFirst example

The following example demonstrates how **findFirst** behaves depending on the file specification in *pattern*:

```
; buttonOne::pushButton
method pushButton(var eventInfo Event)
var
   fs FileSystem
endVar

; Search in the root directory for a file
; or directory named PDOXWIN.
if fs.findFirst("c:\\pdoxwin") then
   ; this displays PDOXWIN (findFirst finds the directory)
   msgInfo("Pattern: c:\\pdoxwin", "Name: " + fs.name())
else
   errorShow()
endIf

; >>INVALID PATTERN CAUSES AN ERROR!! <<
if fs.findFirst("c:\\pdoxwin\\") then
   message("This message never displays.")
else
   errorShow("Invalid pattern: c:\\pdoxwin\\")
endIf

; Search in the PDOXWIN directory for
; any file or directory.
if fs.findFirst("c:\\pdoxwin\\*.*") then
   ; This displays one dot (.) because the
   ; first file in a directory is a single dot (.).
   msgInfo("Pattern: c:\\pdoxwin\\*.*", "Name: " + fs.name())
else
   errorShow()
endIf

endmethod
```

▪

findNext method

[See also](#) [Example1](#) [Example2](#) [FileSystem Type](#)

Searches a file system for multiple instances of a file name.

Syntax

findNext (** [const ***fileSpec* String] **)** Logical

Description

After **findFirst** succeeds, **findNext** searches for the next file whose name matches the pattern. **findNext** returns True if successful; otherwise, it returns False.

As a shortcut, you can use the optional argument *fileSpec* to specify a path and file specification. If you do, the call to **findFirst** is unnecessary.

▪

findNext example 1

The following example calls **findNext** to fill a list with the names of the tables in the current directory. The example assumes that a field displayed as a drop-down list has already been placed in the form. The code is attached to the built-in **open** method of list object contained by the field object.

```
; tablesFld.listObj::open
method open(var eventInfo Event)
var
   fs FileSystem
endVar

doDefault

;   This while loop fills the list in the drop-down edit
; box with *.db files in the default sample directory
while fs.findNext("c:\\pdoxwin\\sample\\*.db")
   self.list.selection =
   self.list.selection + 1
   self.list.value = fs.name()
endWhile
endMethod
```

▪

findNext example 2

The following example uses **findNext** with a file specification as an argument and displays a pop-up menu listing the files in the C:\PDOXWIN directory:

```
; editText::pushButton
method pushButton(var eventInfo Event)
var
   fs      FileSystem
   p       PopUpMenu
   choice  String
endVar

; search for *.txt files in the PDOXWIN directory
; then add their names to a pop-up menu
while fs.findNext("c:\\pdoxwin\\*.txt")
   p.addText(fs.name())
endWhile

choice = p.show()                    ; show the pop-up menu
if not choice.isBlank() then         ; if user selected a file
   execute("Notepad.exe " + choice)  ; edit the file in Notepad
endif

endMethod
```

▪

freeDiskSpace method

See also Example1 Example2 FileSystem Type

Returns the amount of free space on a drive.

Syntax

```
freeDiskSpace ( const driveLetter String ) LongInt
```

Description

freeDiskSpace returns the number of bytes available on drive *driveLetter*. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

freeDiskSpace example 1

The following example displays a dialog box listing the number of bytes available on drive C:

```
; showCSpace::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
endVar

msgInfo("Free bytes on drive C:", fs.freeDiskSpace("C"))

endMethod
```

freeDiskSpace example 2

The following example compares the size of the file MEMO14.TXT and the amount of space available on the current drive. If there's enough space, the code calls **copy** to copy the file.

```
; copyFile::pushButton
method pushButton(var eventInfo Event)
   var
      fs           FileSystem
      stDrive      String
      liFileSize,
      liFreeSpace  LongInt
      dyFileInfo   DynArray[] String
   endVar

   if fs.findFirst(":WORK:memo14.txt") then
      liFileSize = fs.size()
      splitFullFileName(workingDir(), dyFileInfo)
      stDrive = dyFileInfo["DRIVE"]
      liFreeSpace = fs.freeDiskSpace(stDrive)
   else
      msgStop("MEMO14.TXT", "File not found.")
      return
   endIf

   if liFreeSpace > liFileSize then
      fs.copy("memo14.txt", "memo14.bak")
      message("File copied successfully.")
   else
      msgStop("Copy", "Not enough disk space to copy file.")
   endIf

endMethod
```

▪

fullName method/procedure

See also Example FileSystem Type Changes

Returns the full path to a file.

Syntax

```
1. (Method) fullName ( ) String
2. (Procedure) fullName ( const fileName String ) String
```

Description

In syntax 1, after a successful **findFirst** or **findNext**, **fullName** returns the full path of the found file. Use this method with **splitFullFileName** to analyze the components of a file name.

Syntax 2 (added in version 5.0) operates on a file name, expanding or translating aliases or relative directory operators and returning the expanded string. For example, suppose :WORK: is defined as C:\PDOXWIN\FORMS. Given the string ":WORK:myForm.fsl" syntax 2 would return "C:\PDOXWIN\FORMS\myForm.fsl".

Changes to fullName method

The following changed for version 5.0:

Syntax 2. (Procedure) **fullName (** const ***fileName*** String **)** String

Syntax 2 operates on a file name, expanding or translating aliases or relative directory operators and returning the expanded string. For example, suppose :WORK: is defined as C:\PDOXWIN\FORMS. Given the string ":WORK:myForm.fsl" syntax 2 returns "C:\PDOXWIN\FORMS\myForm.fsl".

■

fullName example

Calls **fullName** to get the full name of the first form listed in the current directory. Then calls **splitFullFileName** to split the name into its component parts and store them in a DynArray. Then calls **view** to display the DynArray.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   splitName DynArray[] String
   fullFileName  String
endVar

; if the customer.db file is in the sample directory
if fs.findFirst("c:\\pdoxwin\\sample\\customer.db") then

   ; store the full file name to a variable
   fullFileName = fs.fullName()

   ; split file name into parts and store them in a DynArray
   splitFullFileName(fullFileName, splitName)

   ; display the component parts
   splitName.view("Split name")
endIf

endMethod
```

▪

getDir method

[See also] [Example] [FileSystem Type]

Returns the directory path to which the FileSystem variable is pointing.

Syntax

getDir ( ) String

Description

Returns a string representing the path of the directory path to which the FileSystem variable is pointing. (Use **setDir** to make a FileSystem variable point to a specified directory.) Does not include the drive letter▪use **getDrive** for that.

getDir example

Gets the path of the directory to which the FileSystem variable is pointing, and compares it with a path. If they don't match, calls **setDir** to change the directory.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   st  String
endVar

   st = "c:\\pdoxwin\\myforms"

   if fs.getDir() <> st then
      fs.setDir(st)
   endIf
endMethod
```

getDrive method

See also Example FileSystem Type

Returns the drive letter or alias pointed to by the FileSystem variable.

Syntax

```
getDrive ( ) String
```

Description

Returns a string representing the drive letter or alias pointed to by the FileSystem variable.

▪

getDrive example

Calls **getDrive** to return the alias of the working directory. Then sets the default drive to H and calls **getDrive** again to confirm the change.

```
; setH::pushButton
method pushButton(var eventInfo Event)
var
   fs         FileSystem
   newDrive   String
endVar

msgInfo("Default drive", fs.getDrive())       ; Displays :WORK:

newDrive = "H"

if fs.existDrive(newDrive) then
   if fs.setDrive(newDrive) then
      msgInfo("Default drive", fs.getDrive())   ; Displays H:
   else
      msgStop(newDrive, "Could not set drive.")
   endIf
else
   msgStop(newDrive, "Drive is not attached.")
endIf

endMethod
```

▪

getFileAccessRights procedure

See also   Example   FileSystem Type

Reports access rights (also called file attributes) of a file.

Syntax

getFileAccessRights **(** const ***fileName*** String **)** String

Description

Returns a string describing access rights of a file. Return values can be one or more of the following: A, D, H, R, S, V (for archive, directory, hidden, read only, system, and volume, respectively). If the returned value is an empty string, the file has no attributes set. Similar to the **accessRights** method; does not, however, require you to call the **findFirst** method first.

getFileAccessRights example

Displays the file attributes for the file C:\CONFIG.SYS in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fileName String
endVar

fileName = "C:\\CONFIG.SYS"

msgInfo(fileName, getFileAccessRights(fileName))

endMethod
```

▪

getValidFileExtensions procedure

See also Example FileSystem Type

Returns the valid file extensions for a specified object.

Syntax

getValidFileExtensions (**const ***objectType* String **)** String

Description

Returns a string containing the valid file extensions for the object specified in **objectType**, which is a Form, Library, Report, or Script.

getValidFileExtensions example

Displays a dialog box listing the valid file extensions for forms:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fx String
endVar

fx = getValidFileExtensions("Form")
msgInfo("Form file extensions:", fx)    ; displays fsl fdl

endMethod
```

▪

isDir procedure

See also   Example   FileSystem Type

Reports whether a specified string represents the name of a directory.

Syntax

isDir (const ***dirName*** String **)** Logical

Description

Returns True if *dirName* is a valid directory name; otherwise returns False.

▪

isDir example

Calls **isDir** to make sure that the directory specified by the variable *newDir* is valid. If so, calls **setDir to** make *newDir* the default directory. In this example, the value of *newDir* is hard coded, but in practice, it could be supplied by the user, read from a table, or come from another source.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs      FileSystem
   newDir  String
endVar

newDir = "C:\\pdoxwin\\diveplan"
if isDir(newDir) then
   fs.setDir(newDir)
   msgInfo("Current directory", fs.getDir())
else
   msgStop(newDir, "Directory does not exist.")
endIf

endMethod
```

isFile procedure

See also Example1 Example2 FileSystem Type

Reports whether a specified string is the name of a file in the current file system.

Syntax

isFile (const ***fileName*** String **)** Logical

Description

Returns True if *fileName* is a file in the current file system; otherwise returns False.

isFile example 1

Calls **isFile** and displays messages reporting whether the file specifications represent actual files.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs FileSystem
endVar

message(isFile("c:\\dos\\chkdsk.exe")) ; displays True
sleep(1500)
message(isFile("c:\\dos\\MyXFilex.ext"))   ; displays False
sleep(1500)

endMethod
```

▪

isFile example 2

Prompts the user to enter the full path and file name of a file to delete. Calls **isFile** to test whether the file exists; if so, calls **delete** to delete it:

```
; buttonOne::pushButton
method pushButton(var eventInfo Event)
var
   fs        FileSystem
   fileName  String
endVar

fileName = "Enter full path and filename here."
fileName.view("Delete a file")

if isFile(fileName) then         ; if the specified file exists
   fs.delete(fileName)            ; delete the file
   message("File deleted.")
else
   msgStop(fileName, "File not found.")
endIf

endMethod
```

▪

isFixed method

See also Example FileSystem Type

Reports whether a drive is fixed.

Syntax

`isFixed ( const driveLetter String ) Logical`

Description

Returns True if *driveLetter* represents a fixed (not removable or network) drive; otherwise returns False.
You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

isFixed example

In the following example, drive C is the user's local hard disk, and drive H is a network drive:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
endVar

msgInfo("Is drive C fixed?", fs.isFixed("C"))  ; displays True
msgInfo("Is drive H fixed?", fs.isFixed("H"))  ; displays False

endMethod
```

isRemote method

See also   Example   FileSystem Type

Reports whether a drive is remote (a network drive).

Syntax

```
isRemote ( const driveLetter String ) Logical
```

description

Returns True if *driveLetter* represents a remote (network) drive; otherwise returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

isRemote example

In the following example, drive H is a network (remote) drive. This code calls **existDrive** to make sure drive H is attached, then calls **isRemote** to find out whether drive H is a network drive.

```
var
   h FileSystem
endVar
if h.existDrive("h") then ; if drive H is attached
   if h.isRemote("h") then
      msgInfo("Drive H: ", "Remote Drive")
   else
      msgInfo("Drive H:", "Not a Remote Drive.")
   endIf
else
   msgStop("Drive H", "Drive is not attached.")
endIf
```

▪

isRemovable method

[See also] [Example] [FileSystem Type]

Reports whether a drive is removable.

Syntax

isRemovable (**const ***driveLetter* String **)** Logical

Description

Returns True if *driveLetter* represents a removable drive; otherwise returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

isRemovable example

In the following example, drive D is a removable drive. This code calls **existDrive** to make sure drive D is attached, then calls **isRemovable** to find out whether drive D is a removable drive:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   s   String
endVar

if fs.existDrive("D:") then ; if drive D is attached
   if fs.isRemovable("D") then
      msgInfo("Drive D: ", "Removable Drive")
   else
      msgInfo("Drive D:", "Not a Removable Drive.")
   endIf
endIf

endMethod
```

▪

isValidFile procedure

[See also] [Example] [FileSystem Type]

Checks if a filename is valid.

Syntax

`isValidFile ( const fileName String ) Logical`

Description

isValidFile checks if the filename is valid for the file system. Use **isValidFile** to see if long file names are supported on a specific volume. This procedure returns True if the file is valid.

isValidFile example

This example uses the view dialog to ask for a new file name. **isValidFile** is used to check if the file is valid for the volume so that it can be copied to that volume.

```
proc copyNewFile( origFileName  String )
var
   newFile string
endVar

newFile.view()

if isValidFile( newFile ) then
   copy( origFileName, newFile )
else
   msgInfo( "Error", "This is not a valid file name" )
endif

endProc
```

▪

makeDir method

See also Example FileSystem Type

Creates a new directory.

Syntax

`makeDir ( const name String ) Logical`

Description

Creates all directories and subdirectories specified in *name*. Returns True if successful in creating *name* (or if the directory already exists); otherwise returns False.

▪

makeDir example

Tries to create a new directory on drive C, and displays a dialog box to report success or failure.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   l   Logical
endVar

; this creates \New and \New\Directory etc...
l = fs.makeDir("C:\\New\\Directory\\Tree")

msgInfo("Status", iif(l, "New directory created", "makeDir Failure"))

endMethod
```

▪

name method

See also   Example   FileSystem Type

Returns the name of a file.

Syntax

`name ( ) String`

Description

After a successful **findFirst** or **findNext**, returns the name of the file whose name matches the pattern.

▪

name example

Calls **findFirst** and **findNext** to find the tables in the current directory, then calls **name** to create a pop-up menu listing the file names.

```
; showName::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   p  PopUpMenu
   tv  TableView
   choice, path  String
endVar

if fs.findFirst("*.db") then    ; if a *.db file exists
   p.addStaticText("Tables")     ; create a pop-up menu
   p.addSeparator()
   p.addText(fs.name())          ; use file names in pop-up
   while fs.findNext()
      p.addText(fs.name())
   endWhile
   choice = p.show()             ; show the menu
   if not choice.isBlank() then  ; if user selected a table
      tv.open(choice)             ; display the selected table
   endif
endIf

endMethod
```

privDir procedure

See also Example FileSystem Type

Returns the name of the user's private directory.

Syntax

```
privDir ( ) String
```

Description

Returns a string containing the full DOS path (including the drive ID letter) of the user's private directory.

Each user must have a private directory where temporary tables are stored. It can be on a network or a local drive. You use **setPrivDir** to specify the path to the private directory.

▪

privDir example

Calls **privDir** to display the path to :PRIV: in the status bar.

```
method pushButton(var eventInfo Event)
   message("Y our private directory is: ", privDir())
endMethod
```

▪

rename method

See also   Example   FileSystem Type

Renames a file.

Syntax

`rename ( ``const `***`oldName,`***` String `***`newName`***` String `**`)`**` Logical`

Description

Changes the name of file *oldName* to *newName*. If *newName* is used by another file, the method fails; it does not overwrite the existing file. Returns True if successful; otherwise returns False. **rename** is independent of **findFirst** and **findLast**.

▪

rename example

Searches the current directory for the file specified in the variable *oldName*. If it exists, calls **rename** to rename it. A dialog box appears to report any errors.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   oldName, newName  String
endVar

oldName = "memo14.txt"
newName = "memo14.bak"

if fs.findFirst(oldName) then
   if not fs.rename(oldName, newName) then
      msgStop("Could not rename file", newName + " already exists.")
   endIf
else
   msgStop(oldName, "File not found.")
endIf

endMethod
```

▪

setDir method

See also Example FileSystem Type

Sets the directory path for a FileSystem variable.

Syntax

```
setDir ( const name String ) Logical
```

Description

Sets the directory path to *name* for a FileSystem variable. Compare **setDir** to **setDrive**, which sets the default drive.

▪

setDir example

Calls **isDir** to check whether the directory *newDir* is valid. If so, calls **setDir** to make *newDir* the default directory.

```
method pushButton(var eventInfo Event)
var
   fs      FileSystem
   newDir  String
endVar

   newDir = "c:\\pdoxwin\\mine\\zap"

   if isDir(newDir) then
      fs.setDir(newDir)
   else
      msgStop(newDir, "Not a valid directory.")
   endIf

   message(fs.getDir()) ; displays \pdoxwin\mine\zap
endMethod
```

setDirLock procedure

See also Example FileSystem Type

Locks a specified directory.

Syntax

`setDirLock ( const dirName String ) Logical`

Description

Locks the directory *dirName*. Returns True if successful; otherwise returns False.

A directory lock makes the directory read-only, which prevents Paradox from attempting to read from or write to a lock file in that directory. A directory lock is required for Paradox to access data from a CD-ROM drive, and can improve performance on network drives and local drives. A lock is not be respected on a local drive if Local Share is off.

▪

setDirLock example

Calls **setDirLock** to make a network drive read-only when the form opens, and calls **clearDirLock** to remove the lock when the form closes.

The following code is attached to the form's built-in **open** method.

```
method open(var eventInfo Event)
   var
      h FileSystem
   endVar

   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:

   else
      ;// This code executes only for the form:
      if h.existDrive("h") then ; if drive H is attached
         if h.isRemote("h") then
            setDirLock("h")
            message("Drive H: locked.")
         else
            msgStop("Drive H:", "Not a Remote Drive.")
            return
         endIf
      else
         msgStop("Drive H:", "Drive is not attached.")
         return
      endIf

   endIf
endMethod
```

The following code is attached to the form's built-in **close** method.

```
method close(var eventInfo Event)

   var
      h FileSystem
   endVar

   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:

   else
      ;// This code executes only for the form:
      if h.existDrive("h") then ; if drive H is attached
         if h.isRemote("h") then
            clearDirLock("h")
            message("Drive H: unlocked.")
         else
            msgStop("Drive H:", "Not a Remote Drive.")
            return
         endIf
      else
         msgStop("Drive H:", "Drive is not attached.")
         return
      endIf

   endIf

endMethod
```

▪

setDrive method

See also Example1 Example2 FileSystem Type

Makes a specified drive the default drive.

Syntax

setDrive (** `const` ***name* `String` **)** `Logical`

Description

setDrive sets the default drive to *name.* Returns True if successful; otherwise returns False. You specify *name* with a letter ("C"), a letter and a colon ("C:"), or an alias (":MAST:").

setDrive example 1

Calls **view**, defined for the String type, to display a dialog box and prompt the user for input. If the user enters the ID letter of a valid drive, calls **setDrive to** make the ID letter the default drive.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs         FileSystem
   newDrive   String
endVar

newDrive = "Enter drive ID or alias here."
newDrive.view("Change default drive.") ; prompt user for input

if fs.existDrive(newDrive) then
   fs.setDrive(newDrive)
else
   msgStop(newDrive, "Drive not available.")
endIf

endMethod
```

▪

setDrive example 2

Shows how to use an alias with **setDrive**. (Assumes that the alias :MAST: has already been defined.)

```
; setDrive::pushButton
method pushButton(var eventInfo Event)
var
   fs FileSystem
endVar

fs.setDrive(":MAST:")

endMethod
```

▪

setFileAccessRights procedure

See also Example FileSystem Type

Sets access rights (also called attributes) of a file.

Syntax

setFileAccessRights (const ***fileName*** String, const ***rights*** String **)** Logical

Description

Sets the access rights of *fileName* to those specified in *rights,* which is a string that contains one or more or the following: A, D, H, R, S, V (for archive, directory, hidden, read only, system, and volume, respectively). If *rights* is an empty string (""), removes all access rights settings for *fileName*. You don't have to declare a FileSystem variable (or use the **findFirst** method) before calling **setFileAccessRights**.

setFileAccessRights example

Sets file access rights for C:\CONFIG.SYS to read only ("R") and hidden ("H").

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fileName  String
endVar

fileName = "C:\\CONFIG.SYS"

; set file attribute for CONFIG.SYS to read only and hidden
if setFileAccessRights(fileName, "RH") then
   ; if successful, display a message with the current attributes
   message (fileName + " attributes set to " +
            getFileAccessRights(fileName))
else
   ; otherwise, the procedure failed
   message("Can't set file attributes for " + fileName)
endif

endMethod
```

▪

setPrivDir procedure

See also Example1 Example2 FileSystem Type

Sets or changes the private directory (:PRIV:).

Syntax

```
setPrivDir ( const path String ) Logical
```

Description

Sets the path to the current private directory (:PRIV:) to *path*. Returns True if successful; otherwise returns False. The following table gives examples of valid values for *path*:

Value of *path*	Example
Directory name	ORDERS
Full path	C:\\PDOXWIN\\APPS\\ORDERS\\
Relative path	..\\..\\ORDERS
Alias	:ORDERS:

Paradox closes all its open windows (prompting you to save documents, as needed) and frees all locks before setting the private directory. Therefore, **setPrivDir** does not take effect until all ObjectPAL code has finished executing. You can keep a form open by adding code to its built-in **menuAction** method to trap for the MenuChangingPriv menu command (see the example for details). If you do so, save any documents that need saving before changing the working directory. **setPrivDir** returns True if successful; otherwise returns False.

ObjectPAL provides the following MenuCommands constants for handling changes to the private directory:

Constant	Description
MenuFilePrivateDir	Issued when the user chooses Edit|Preferences|Database:Private Directory from the Paradox menu. Trap for this constant to prevent the user from changing the private directory.
MenuChangingPriv	Issued just before :PRIV: changes. Trap for this constant to keep a form open when changing the private directory.
MenuChangedPriv	Issued just after :PRIV: changes. Trap for this constant to find out when the private directory has changed.

ObjectPAL also provides the constant MenuFileWorkingDir, issued when the user chooses Edit|Preferences|Database:Working Directory from the built-in Paradox menu.

▪

setPrivDir example 1

Handles a menu choice to change the private directory and the resulting menu commands generated by Paradox. When you choose Edit|Preferences|Database:Private Directory from the default Paradox menu, calls **disableDefault** to block the default behavior, preventing Paradox from displaying the Set Private Dir dialog box. Next, tests the value of a Logical variable *okToChangePriv* (declared and assigned elsewhere). If *okToChangePriv* is True, calls **setPrivDir** to set :PRIV: behind the scenes.

Also handles the MenuChangingPriv menu command, issued by Paradox just before it changes the private directory. **setErrorCode** sets the error code to a nonzero value, which keeps this form open when :PRIV: changes. Responds to the MenuChangedPriv menu command, issued by Paradox just after it changes the private directory.

```
method menuAction(var eventInfo MenuEvent)

const
   kKeepFormOpen = UserMenu  ; UserMenu is an ObjectPAL constant.
endConst                     ; Any nonzero value keeps the form open.

   ; In a real app you'd declare and assign this variable elsewhere.
   okToChangePriv = True

   switch
      case eventInfo.id() = MenuFilePrivateDir :
         disableDefault                  ; Block the default behavior.
         if okToChangePriv then
            setPrivDir("c:\\pdx\\mine") ; Set :PRIV: to hard-coded path.
         else
            return
         endIf

      case eventInfo.id() = MenuChangingPriv :
         eventInfo.setErrorCode(kKeepFormOpen)

      case eventInfo.id() = MenuChangedPriv :
         ; You may want to take some action after changing :PRIV:.
         ; This example just displays the new path.
         message(privDir())
         sleep(1000)

      otherwise : doDefault
   endSwitch
endMethod
```

▪

setPrivDir example 2

The **open** and the **menuAction** methods of a form set the private directory before the form opens. In the form's built-in **open** method, **setPrivDir** changes the current private directory to the same directory as the form. The ObjectPAL code in the **menuAction** prevents the form from closing during the change.

The following code is attached to the form's built-in **open** method. It gets the form's file name and uses it to set the private directory.

```
;frm1 :: open
method open(var eventInfo Event)
   var
      f                Form
      dynPath   DynArray[] String
   endVar

   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:
      f.attach()
      splitFullFileName(f.getFileName(), dynPath)
      setPrivDir(dynPath["Drive"] + dynPath["Path"])
   else
      ;// This code executes only for the form:
   endIf
endMethod
```

The following code is attached to the form's built-in **menuAction** method. It keeps the form open after changing the private directory.

```
;frm1 :: menuAction
method menuAction(var eventInfo MenuEvent)
const
   kKeepFormOpen = UserMenu    ; UserMenu is an ObjectPAL constant.
endConst                  ; Any nonzero value keeps the form open.

   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:
      if eventInfo.id() = MenuChangingPriv then
         eventInfo.setErrorCode(kKeepFormOpen)
      endIf
   else
      ;// This code executes only for the form:
   endIf
endMethod
```

▪

setWorkingDir procedure

See also Example1 Example2 FileSystem Type

Sets or changes the working directory (:WORK:).

Syntax

```
setWorkingDir ( const path String ) Logical
```

Description

Sets the path to the current working directory (:WORK:) to *path*. The following table gives examples of valid values for *path*:

Value of *path*	Example
Directory name	ORDERS
Full path	C:\\PDOXWIN\\APPS\\ORDERS\\
Relative path	..\\..\\ORDERS
Alias	:ORDERS:

By default, Paradox closes all open windows before setting the working directory, and prompts you to save changed documents. Therefore, **setWorkingDir** does not take effect until all ObjectPAL code has executed. You keep a form open by adding code to its built-in **menuAction** method to trap for the MenuChangingWork menu command. (See the example for details.) If you do so, save any documents before changing the working directory. **setWorkingDir** returns True if successful; otherwise, it returns False.

ObjectPAL provides the following MenuCommands constants for handling changes to the working directory:

Constant	Description
MenuFileWorkingDir	Issued when the user chooses Edit|Preferences|Database:Working Directory from the built-in Paradox menu. Trap for this constant to prevent the user from changing the working directory.
MenuChangingWork	Issued just before :WORK: changes. Trap for this constant to keep a form open when changing the working directory.
MenuChangedWork	Issued just after :WORK: changes. Trap for this constant to find out when the working directory has changed.

ObjectPAL also provides the constant MenuFilePrivateDir, issued when the user chooses Edit|Preferences|Database:Private Directory from the built-in Paradox menu.

▪

setWorkingDir example 1

Handles a menu choice to change the working directory, and the resulting menu commands generated by Paradox. When you choose Edit|Preferences|Database:Working Directory from the default Paradox menu, calls **disableDefault** to block the default behavior, thus preventing Paradox from displaying the Set Working Dir dialog box. Next, tests the value of a Logical variable *okToChangeWork* (declared and assigned elsewhere). If *okToChangeWork* is True, calls **setWorkingDir** to set :WORK: behind the scenes.

Also handles the MenuChangingWork menu command, issued by Paradox just before it changes the working directory. The call to **setErrorCode** sets the error code to a nonzero value, which keeps this form open when :WORK: changes. Responds to the MenuChangedWork menu command, issued by Paradox just after it changes the working directory.

```
method menuAction(var eventInfo MenuEvent)
const
   kKeepFormOpen = UserMenu   ; UserMenu is an ObjectPAL constant.
endConst                ; Any nonzero value keeps the form open.

   ; In a real app you'd declare and assign this variable elsewhere.
   okToChangeWork = True

   switch
      case eventInfo.id() = MenuFileWorkingDir :
         disableDefault                 ; Block the default behavior.
         if okToChangeWork then
            setWorkingDir("c:\\pdx\\mine") ; Set :WORK: to hard-coded path.
         else
            return
         endIf

      case eventInfo.id() = MenuChangingWork :
         eventInfo.setErrorCode(kKeepFormOpen)

      case eventInfo.id() = MenuChangedWork :
         ; You may want to take some action after changing :WORK:.
         ; This example just displays the new path.
         message(workingDir())
         sleep(1000)

      otherwise : doDefault
   endSwitch
endMethod
```

setWorkingDir example 2

A form's **open** and **menuAction** methods set the working directory before the form opens. In the form's built-in **open** method, **setWorkingDir** changes the current working directory to the same directory as the form. The ObjectPAL code in the **menuAction** prevents the form from closing during the change.

The following code is attached to the form's built-in **open** method. It gets the form's file name and uses it to set the working directory.

```
;frm1 :: open
method open(var eventInfo Event)
var
   f                   Form
   dynPath   DynArray[]  String
endVar

if eventInfo.isPreFilter() then
   ;// This code executes for each object on the form:
else
   ;// This code executes only for the form:
   f.attach()
   splitFullFileName(f.getFileName(), dynPath)
   setWorkingDir(dynPath["Drive"] + dynPath["Path"])
endIf

endMethod
```

The following code is attached to the form's built-in **menuAction** method. It keeps the form open after changing the working directory.

```
;frm1 :: menuAction
method menuAction(var eventInfo MenuEvent)
const
   kKeepFormOpen = UserMenu   ; UserMenu is an ObjectPAL constant.
endConst                      ; Any nonzero value keeps the form open.

   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:
      if eventInfo.id() = MenuChangingWork then
         eventInfo.setErrorCode(kKeepFormOpen)
      endIf
   else
      ;// This code executes only for the form:
   endIf
endMethod
```

shortName method

See also Example FileSystem Type

Returns the short name of a file.

Syntax

shortName ( ) `String`

Description

After a successful findFirst or findNext, **shortName** returns the short name of the file whose name matches the pattern. The short name is the 8.3 name of the file that is stored in the file system.

shortName example

Calls findFirst and findNext to find the tables in the current directory, then calls shortName to create a pop-up menu listing the file names.

```
; showName::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   p  PopUpMenu
   tv  TableView
   choice, path  String
endVar

if fs.findFirst("*.db") then    ; if a *.db file exists
   p.addStaticText("Tables")     ; create a pop-up menu
   p.addSeparator()
   p.addText(fs.shortName())          ; use file names in pop-up
   while fs.findNext()
      p.addText(fs.shortName())
   endWhile
   choice = p.show()             ; show the menu
   if not choice.isBlank() then  ; if user selected a table
      tv.open(choice)             ; display the selected table
   endif
endIf

endMethod
```

▪

size method

See also Example FileSystem Type

Returns the size of a file.

Syntax

size ( ) LongInt

Description

Returns the number of bytes in a found file after a successful **findFirst** or **findNext**.

size example

Creates a DynArray containing the file names and sizes of the Paradox tables in the current directory. The call to **view**, defined for the DynArray type, displays the information in a dialog box.

```
; demoButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  da DynArray[] LongInt
endVar

if fs.findFirst("*.db") then
  da[fs.name()] = fs.size()
  while fs.findNext()
    da[fs.name()] = fs.size()
  endWhile
  da.view("Names and sizes")
else
  msgStop("*.db", "file not found.")
endIf

endMethod
```

▪

splitFullFileName procedure

See also Example1 Example2 Example3 FileSystem Type

Breaks a full path name into its component parts.

Syntax

```
1. splitFullFileName ( const fullFileName String, var components DynArray[ ] String )
2. splitFullFileName ( const fullFileName String, var driveName String, var pathName String, var fileName String, var extensionName String )
```

Description

Divides a full file path (typically obtained using **fullName**) into its component parts. Does not return the values directly, but assigns them to variables you declare and pass as arguments.

Syntax 1 assigns the results to a DynArray that you must declare and pass as an argument. The DynArray has the following keys: DRIVE, PATH, NAME, and EXT.

Syntax 2 assigns the results to four String variables that you must declare and pass as arguments.

With both syntaxes, the file path components include colons, periods, slashes, and backslashes, as appropriate. For example, given a full file name of C:\PDOXWIN\FORMS\ORDERS.FSL, **splitFullFileName** assigns values as follows:

DRIVE = C:, PATH = \PDOXWIN\FORMS\ NAME = ORDERS, and EXT = .FSL.

The DRIVE variable (or key) stores everything up to and including the last colon in the file name. If the file name includes an alias, the alias is assigned to DRIVE. If the file name does not include a drive or an alias, an empty string is assigned to DRIVE.

The PATH variable (or key) stores everything following the drive, up to and including the last backslash or slash. If the file name does not include a path, an empty string is assigned to PATH. If a directory name in the path includes an extension, it is included.

The NAME variable (or key) stores everything following the path, up to but not including the last period that separates a file name from its extension. If the file name does not include a name, an empty string is assigned to NAME.

The EXT variable (or key) stores everything following the name, including the last period. If the file name does not include an extension, an empty string is assigned to EXT.

splitFullFileName example 1

Calls **fullName** to get the full name of the first form listed in the current directory. Then calls **splitFullFileName** to split the name into its component parts and store them in a DynArray; then calls **view** to display the DynArray.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  splitName DynArray[] anytype
  fullFileName String
endVar

; if the customer.db file is in the sample directory
if fs.findFirst("c:\\pdoxwin\\sample\\customer.db") then

  ; store the full file name to a variable
  fullFileName = fs.fullName()

  ; split file name into parts and store them in a DynArray
  splitFullFileName(fullFileName, splitName)

  ; display the component parts
  splitName.view("Split name")
endIf

endMethod
```

▪

splitFullFileName example 2

Calls **splitFullFileName** to split the full name of a form into its component parts, then displays the path and the file name (without an extension) in dialog boxes.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs FileSystem
   driveName, pathName, fileName, extName String
endVar

if fs.findFirst("*.fsl") then
   splitFullFileName(fs.fullName, driveName, pathName, fileName, extName)
   pathName.view("Path name") ; displays the path
   fileName.view("File name") ; displays the filename (no extension)
endIf

endMethod
```

splitFullFileName example 3

Displays a dialog box and prompts you to enter a filename. **splitFullFileName** splits the filename into its component parts, then displays the parts in dialog boxes. Notice how **splitFullFileName** deals with aliases, incomplete or complex file names, and the like.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
   var
      stTestFileName,
      stPrompt,
      stDrive,
      stPath,
      stName,
      stExt         String
      dyFileName   DynArray[] String
   endVar

   stPrompt = "Enter a file name here."
   stTestFileName = stPrompt

   stTestFileName.view("Enter a file name to split:")

   if stTestFileName = stPrompt then
      ; User closed the dialog box without clicking OK,
      ; or clicked OK without typing a value.
      return
   else
      ; User typed a value and clicked OK.
      splitFullFileName(stTestFileName, dyFileName)
      dyFileName.view("DynArray")

      splitFullFileName(stTestFileName, stDrive, stPath, stName, stExt)
      stDrive.view("Drive")
      stPath.view("Path")
      stName.view("Name")
      stExt.view("Ext")
   endIf
endMethod
```

▪

startUpDir procedure

See also Example FileSystem Type

Returns a string containing the path to the user's start-up directory.

Syntax

```
startUpDir ( ) String
```

Description

Returns a string containing the full path (including the drive ID letter) of the start-up directory, from which Paradox was started.

startUpDir example

Displays a dialog box listing the path to the directory from which Paradox started:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

msgInfo("Start-up directory", startUpDir())

endMethod
```

▪

time method

See also   Example   FileSystem Type

Returns the time and date a file was last modified.

Syntax

time ( ) `DateTime`

Description

Returns a DateTime value representing the time and date of the last modification to a file.

▪

time example

Calls **time** to get the time and date of the most recent change to the *Customer* table. Then compares the modification date with today's date and report the results.

```
method pushButton(var eventInfo Event)
   var
      fs FileSystem
   endVar

   if fs.findFirst("customer.db") then
      if fs.time() < DateTime(today()) then
         message("old version")
      else
         message("new version")
      endif
   endIf
endMethod
```

▪

totalDiskSpace method

See also Example FileSystem Type

Returns the capacity of a drive.

Syntax

totalDiskSpace (const ***driveLetter*** String **)** LongInt

Description

Returns the total number of bytes *driveLetter* (specified by a letter ("C") or a letter and a colon ("C:")), can hold.

▪

totalDiskSpace example

Calls **totalDiskSpace** and **freeDiskSpace** to calculate the amount of space in use. Stores the information in a DynArray, then calls the **view** method defined for the DynArray type to display the information in a dialog box.

```
; spaceUsed::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  da DynArray[] LongInt
endVar

da["Total space"] = fs.totalDiskSpace("C")
da["Free space"] = fs.freeDiskSpace("C")
da["Space in use"] = da["Total space"] - da["Free space"]
da.view("Drive C")

endMethod
```

windowsDir procedure

See also Example FileSystem Type

Returns the path to the WINDOWS directory.

Syntax

windowsDir ( ) `String`

Description

Returns the path to the WINDOWS directory.

▪

windowsDir example

Reads the file WIN.INI from drive B and copies it to the WINDOWS directory on the default drive:

```
; copyWinIni::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   fileName, destName  String
endVar

fileName = "\\win.ini"

fs.setDrive("B")
if fs.findFirst(fileName) then
   destName = windowsDir() + fileName
   fs.copy(fileName, destName)
endIf

endMethod
```

windowsSystemDir procedure

See also Example FileSystem Type

Returns the path to the Windows system directory.

Syntax

windowsSystemDir ( ) `String`

Description

Returns the path to the Windows system directory.

windowsSystemDir example

Reads the file SPECIAL.DRV from drive B and copies it to the Windows system directory on the default drive:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   fs  FileSystem
   fileName, destName  String
endVar

fileName = "\\special.drv"

fs.setDrive("B")
if fs.findFirst(fileName) then
   destName = windowsSystemDir() + fileName
   fs.copy(fileName, destName)
endIf

endMethod
```

▪

workingDir procedure

See also   Example   FileSystem Type

Returns the name of the current working directory.

Syntax

`workingDir ( )` `String`

Description

Returns the name (including the path) of the current working directory (:WORK:).

workingDir example

Displays a message containing the path to the current working directory:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

message("Working directory is: " + workingDir())

endMethod
```

■

Disk errors

See also

When a method fails because of a disk error, the error code constant is peDiskError, and the error message is, "A disk error occurred: " plus one of the following strings:

"Invalid function number."

"The file could not be found."

"The directory path could not be found."

"No file handle available."

"Access to this file is denied. It is read only or a directory."

"Invalid handle."

"Memory control blocks have been damaged."

"Insufficient memory to allocate file structures."

"Invalid memory block address."

"Invalid environment."

"Invalid format."

"Invalid file access byte."

"Invalid data."

"Invalid drive."

"Cannot remove the current directory."

"Not the same device."

"No more files match the wildcard specification."

"Cannot write to a write-protected disk."

"Unknown unit."

"The drive is not ready."

"Command is not recognized."

"Checksum error (Bad CRC)."

"Invalid request structure length."

"File seek error."

"Unknown media type."

"Sector not found."

"Out of paper."

"An error occurred while trying to write to the disk."

"An error occurred while trying to read from the disk."

"General DOS error."

"File sharing violation."

"File lock violation."

"Invalid disk change."

"File control blocks unavailable."

"Sharing buffer overflow."

"Bad code page."

"Handle EOF."
"The disk is full."
"Device is not supported."
"Device is not listening."
"Duplicate name."
"Invalid network path."
"The network is busy."
"The device does not exist."
"Too many commands."
"Adapter error."
"Invalid network response."
"Network error."
"Adapter is incompatible."
"The print queue is full."
"Out of spool space."
"Print job was canceled."
"The network name was deleted."
"Your access to the network is denied."
"Invalid device type."
"Invalid network name."
"Too many names."
"Too many sessions."
"Sharing pause."
"Request not accepted."
"Redirection pause."
"The file already exists."
"Duplicate file control blocks."
"Cannot create the specified directory."
"DOS critical error."
"Out of structures. Cannot perform operation."
"Drive is already assigned."
"Invalid password."
"Invalid parameter."
"Network write error."
"Comp command is not loaded."
"The mode specification is invalid."
"Cannot write to the file because it was opened in read-only mode."

▪

Form type

Changes ▪

A Form variable provides a handle for working with a Paradox form. Form type methods let you

- Load a form in a Form Design window and save a design
- Open and close a form
- Attach to an open form
- Work with tables in a data model
- Work with table aliases
- Enumerate object names, properties, and source code for methods
- Determine and change the position of a form, as well as maximize or minimize the form
- Send events to a form, such as a **mouseUp** or **keyPhysical**
- Get and set methods for a form

The Form type is the base type from which the other Display manager types (for example, Report) are derived. Many of the methods listed in this section are used by the Application, Report, and TableView types, too.

Methods in the Form type

Form
action
attach
bringToTop
close
create
delayScreenUpdates
deliver
design
disableBreakMessage
disablePreviousError
dmAddTable
dmAttach
dmBuildQueryString
dmEnumLinkFields
dmGet
dmGetProperty
dmHasTable
dmLinkToFields
dmLinkToIndex
dmPut
dmRemoveTable
dmResync
dmSetProperty

dmUnlink
enumDataModel
enumSource
enumSourceToFile
enumTableLinks
enumUIObjectNames
enumUIObjectProperties
formCaller
formReturn
getFileName
getPosition
getProtoProperty
getSelectedObjects
getStyleSheet
getTitle
hide
hideToolbar
isCompileWithDebug
isDesign
isMaximized
isMinimized
isToolbarShowing
isVisible
keyChar
keyPhysical
load
maximize
menuAction
methodDelete
methodGet
methodSet
minimize
mouseDouble
mouseDown
mouseEnter
mouseExit
mouseMove
mouseRightDouble
mouseRightDown
mouseRightUp

mouseUp
moveToPage
open
openAsDialog
postAction
run
save
saveStyleSheet
selectCurrentTool
setCompileWithDebug
setIcon
setMenu
setPosition
setProtoProperty
setSelectedObjects
setStyleSheet
setTitle
show
showToolbar
wait
windowClientHandle
windowHandle

Changes to Form type methods

The following table lists new methods and methods that were changed for version 7.

New	Changed
isCompileWithDebug	
setCompileWithDebug	
setIcon	

The following table lists new methods and methods that were changed for version 5.0. All methods with "SpeedBar" in their names were changed to use "Toolbar" in version 5.0. For example, **hideSpeedBar** was replaced by **hideToolbar**. ObjectPAL code that uses the old names will compile and execute as before, but you should use the new names whenever possible.

New	Changed
dmAttach	dmGet
dmBuildQueryString	dmGetProperty
dmEnumLinkFields	dmHasTable
dmResync	dmLinkToFields
enumDataModel	dmLinkToIndex
getProtoProperty	dmPut
getSelectedObjects	dmRemoveTable
getStyleSheet	dmSetProperty
hideToolbar	dmUnlink
isDesign	enumTableLinks
isToolbarShowing	
load	
saveStyleSheet	
selectCurrentTool	
setMenu	
setProtoProperty	
setSelectedObjects	
setStyleSheet	
showToolbar	

▪

action method/procedure

See also Example Form Type

Performs an action command.

Syntax

`action ( const actionId SmallInt ) Logical`

Description

action performs the action represented by the constant *actionId*, where *actionId* is a constant in one of the following action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use **action** to send a user-defined action constant to a built-in **action** method. User-defined action constants are simply integers that don't interfere with any of ObjectPAL's constants. You can use them to signal other parts of an application. For instance, assume that the Const window for a form declares a constant named *myAction*. In the built-in **action** method for a page on the form, you might check the value of every incoming ActionEvent (with the **id** method); if the value is equal to *myAction*, you can respond to that action accordingly. Paradox's default response for user-defined action constants is simply to pass the action to the **action** method.

This **action** method is distinct from the built-in **action** method for a form or for any other UIObject. The built-in **action** method for an object responds to an action event; this method causes an ActionEvent.

Note: When you call the **action** method as a procedure, the form dispatches it to the object represented by *self*. The event bubbles through the containership hierarchy until the event either reaches an object that can handle the action or the event reaches the form. If the event reaches the form, and the action is a data action, the form sends the event to the master table for the form.

▪

action example

In the following example, a form named *Sitenote* contains field objects bound to the *Sites* table. The current form contains a button named *openEditSites*; the **pushButton** method for *openEditSites* opens *Sitenote*, starts Edit mode, and waits for *Sitenote* to be closed:

```
; openEditSites::pushButton
method pushButton(var eventInfo Event)
var
   siteForm  Form
endVar
siteForm.open("Sitenote.fsl")  ; open Sitenote
siteForm.action(DataBeginEdit) ; start Edit mode on siteForm
message("To return, close Sitenote form.")
siteForm.wait()                ; this form will be inactive until
                               ; Sitenote returns
siteForm.close()               ; this form must close Sitenote
endMethod
```

attach method

See also Example Form Type

Associates a Form variable with an open form.

Syntax

```
attach ( [ const formTitle String ] ) Logical
```

Description

attach associates a Form variable with an open form. You can use *formTitle* to specify a form's current title, or you can omit *formTitle* to attach to the form where **attach** is executing. This method returns True if it succeeds; otherwise, it returns False.

Note: The argument *formTitle* specifies a form's title as displayed in the title bar (for example, Orders), not the form's file name or UIObject name. You can specify a form's title interactively by right-clicking the form's title bar, choosing Window Style and entering a value in the Window Style dialog box. You can specify a title in ObjectPAL by setting a form's Title property, or by calling **setTitle**.

▪

attach example

In the following example, a form has two buttons: *openSites* and *attachToSites*. The **pushButton** method for *openSites* opens the *Sitenote* form. The **pushButton** method for *attachToSites* attaches the form variable *sitesForm* to the open form by way of the form's current title. In this case, the form title wasn't changed, so *attachToSites* can attach to *Sitenote* using the default title. Once attached, the **pushButton** method uses the *sitesForm* handle to minimize, maximize, and restore *Sitenote*.

The following code is attached to the **pushButton** method for *openSites*:

```
; openSites::pushButton
method pushButton(var eventInfo Event)
var
   sitesForm Form
endVar
sitesForm.open("Sitenote")
siteForm.Title = "Notes" ; Set the form's title.

endMethod
```

This code is attached to the **pushButton** method for *attachToSites*:

```
; attachToSites::pushButton
method pushButton(var eventInfo Event)
var
   sitesForm  Form
endVar

; Attach to Sitenote by its title (Notes).
; Note that this won't work:   sitesForm.attach("Sitenote")
if not sitesForm.attach("Notes") then
      errorShow()
      return
endIf

; cycle through sizes
sitesForm.minimize()                ; minimize the form
sleep(2000)                         ; pause
sitesForm.maximize()                ; maximize the form
sleep(2000)                         ; pause
sitesForm.show()                    ; restore to original size
endMethod
```

■

bringToTop method/procedure

See also   Example   Form Type
Beginner

Brings the window to the top of the display stack and makes it active.

Syntax
```
bringToTop ( )
```

Description
When several windows are displayed they seem to overlap, giving an appearance of layers. Use **bringToTop** to display a window at the top of the stack, not overlapped by any other windows. **bringToTop** makes a form the active window.

If a hide statement has made a form invisible, **bringToTop** makes it visible again.

▪

bringToTop example

In the following example, the **pushButton** method for a button named *openSeveral* opens the *Sitenote* form, then opens a table window for the *Orders* table. The table window, *orderTV*, opens over the *Sitenote* form, *siteForm*. The method pauses for a few seconds, then makes *siteForm* the topmost layer:

```
; openSeveral::pushButton
method pushButton(var eventInfo Event)
var
   siteForm  Form
   orderTV   TableView
endVar
siteForm.open("Sitenote.fsl")   ; opens Sitenote form
orderTV.open("orders")          ; opens Orders over Sitenote
message("About to make the Sitenote form the highest layer.")
beep()
sleep(5000)                    ; pause
siteForm.bringToTop()          ; make Sitenote highest layer

endMethod
```

▪

close method/procedure

[See also](#) [Example](#) [Form Type](#)
Beginner

Closes a window.

Syntax

```
1. ( Method ) close ( )
2. ( Procedure ) close ( [ const returnValue AnyType ] )
```

Description

close closes a window as if the user has chosen Close from the Control menu.

close example

The following example demonstrates using **close** to return a value to a form that called it with **wait**. Assume a form contains a button called *btn1*. A second form contains two buttons called *btnReturnOK* and *btnReturnCancel*. The first form opens the second form and waits for one of three values: OK, Cancel, or False. OK and Cancel are returned from the two buttons on the second form (see code below) and False is returned if the user closes the second form without pressing a button. The first form processes the users selection in a **switch** statement that calls one of three custom methods (assumed to be defined elsewhere).

The following code is attached to the button *btn1* in the first (calling) form.

```
;frm1.btn1 :: pushButton
method pushButton(var eventInfo Event)
   var
      f   Form        ;Declare form variable.
      s   String      ;Declare string value.
   endVar

   f.open("wait2")      ;Open form that will return string.
   s = string(f.wait())      ;Wait for value from other form.
   s.view("Returned value")   ;View returned value.

   ;Process returned value using custom methods defined elsewhere.
   switch
      case s = "OK"    : cmOK()       ;User pressed the OK button.
      case s = "Cancel": cmCancel();User pressed the Cancel button.
      case s = "False" : cmNone()  ;User closed form, no button pressed.
   endSwitch
endmethod
```

The following code is attached to the button *btnReturnOK* in the second (called) form.

```
;frm2.btnReturnOK :: pushButton
method pushButton(var eventInfo Event)
   close("OK")         ;Close & return OK.
endmethod
```

The following code is attached to the button *btnReturnCancel* in the second (called) form.

```
;frm2.btnReturnCancel :: pushButton
method pushButton(var eventInfo Event)
   close("Cancel")      ;Close & return Cancel.
endmethod
```

create method

See also Example Form Type

Creates a blank form in a Form Design window.

Syntax

```
create ( ) Logical
```

Description

create creates a blank form and leaves it in a Form Design window. You can use the UIObject type methods **create** and **methodSet** to place objects in the new form and attach methods to them. You can attach methods to the form using the Form type method **methodSet.** Use the Form type method **run** to open the form in a Form window.

▪

create example

In the following example, the **pushButton** method for a button named *createAForm* creates a new form with the **create** method and sets the value of the new form's **mouseUp** method with **setMethod**. The **pushButton** method for *createAForm* then saves the new form to a file named NEWHELLO.FSL, runs the form, and calls the new form's **mouseUp** method (supplying the correct arguments). The **mouseUp** method for the *Newhello* form opens a dialog box that displays "Hello". Once the dialog box is closed (by the user), the **pushButton** method for *createAForm* closes the *Newhello* form.

```
; createAForm::pushButton
method pushButton(var eventInfo Event)
var
   newForm Form
endVar
newForm.create()               ; create a new blank form (a Form Design
window)
newForm.methodSet("mouseUp",   ; set the mouseUp method for the form
"method mouseUp(var eventInfo MouseEvent)
msgInfo(\"Greetings\", \"Hello\")
endMethod")                    ; backslashes delimit embedded quotes
newForm.save("newhello")       ; save the form
newForm.run()                  ; run the new form (View Data window)
                               ; call the mouseUp method for the form
newForm.mouseUp(100, 100, LeftButton )  ; dialog box displays "Hello"
newForm.close()                ; close the form
endMethod
```

delayScreenUpdates procedure

See also Example Form Type

Turns delayed screen updates on or off.

Syntax

```
delayScreenUpdates ( const yesNo Logical )
```

Description

delayScreenUpdates postpones or enables redrawing areas of the screen. You must specify Yes or No in *yesNo*. Specifying Yes delays screen updates (redraws) until the system yields or is idle. This can increase performance in operations that frequently refresh the display (for example, when using ObjectPAL to add items to a list). Specifying No allows screen updates to occur without delay.

For some operations, you won't notice a difference when **delayScreenUpdates** is set to Yes. This is especially true if the application is running on a fast machine.

delayScreenUpdates example

The following two methods override the **pushButton** methods for their respective buttons. The *drawOneByOne* button draws a number of boxes without changing **delayScreenUpdates**. The *drawAllAtOnce* button draws the same number of boxes, to a different location, but first sets **delayScreenUpdates** to Yes. If you run this code, you'll see the boxes created by *drawOneByOne* appear one at a time, but still rapidly. The boxes created by *drawAllAtOnce* are created behind the scenes▪which causes a short pause

▪then appear all at the same time.

```
; drawOneByOne::pushButton
method pushButton(var eventInfo Event)
var
  ui UIObject
endVar

; delayScreenUpdates(No) is the default
; Create and display a set of boxes, showing them as
; they're created.
for i from 750 to 2550 step 300
  for j from 750 to 2550 step 300
    ui.create(boxTool, i, j, 150, 150)
    ui.Color = Blue
    ui.Visible = Yes
  endfor
endfor
endMethod
```

The *drawAllAtOnce* button on the same form creates the same number of boxes, but does so with **delayScreenUpdates** set to Yes. On very fast machines, you still may not be able to see the difference.

```
; drawAllAtOnce::pushButton
method pushButton(var eventInfo Event)
var
  ui UIObject
endVar

delayScreenUpdates(Yes)
; This code will create all boxes, then display
; them all at once.
for i from 4950 to 6750 step 300
  for j from 750 to 2550 step 300
    ui.create(boxTool, i, j, 150, 150)
    ui.Color = Red
    ui.Visible = Yes
  endfor
endfor
; reset to default
delayScreenUpdates(No)

endMethod
```

▪

deliver method

See also Example Form Type

Delivers a form.

Syntax

```
deliver ( ) Logical
```

Description

deliver behaves like File|Deliver. This method saves a copy of a form with an .FDL extension, which prevents users from editing the form in the Form Design window. Users can open the form only in a Form window. Switching to the Form Design window on an open, delivered form is also prohibited.

Paradox opens saved forms before delivered forms with the same name. For example, suppose the working directory contains ORDERS.FSL (a saved form) and ORDERS.FDL (a delivered form). The following statement opens the saved form, ORDERS.FSL.

```
ordersForm.open("ORDERS") ; Opens :WORK:ORDERS.FSL.
```

To specify a delivered form, include the .FDL extension. For example,

```
ordersForm.open("ORDERS.FDL") ; Opens the delivered form.
```

deliver example

In the following example, the *createDeliver* button creates a new form, saves it to the name *Newhello*, then delivers it (which saves a version as NEWHELLO.FDL). When the method attempts to load the form in a Form Design window, load returns False, because a delivered form can't be loaded in a Form Design window.

```
; createDeliver::pushButton
method pushButton(var eventInfo Event)
var
   newForm Form
endVar
newForm.create()              ; create a new blank form (a Form Design
window)
newForm.save("newhello")      ; save the form
newForm.deliver()             ; deliver the newly created form
newForm.close()               ; close the form
if NOT newForm.load("newhello.fdl") then  ; load will return False
   errorShow("Can't load a delivered form.")
endif
endMethod
```

▪

design method

See also Example Form Type

Switches a form from the Form window to the Form Design window.

Syntax

```
design ( ) Logical
```

Description

design switches a form from the Form window to the Form Design window. This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL). For more information about saving and delivering forms, refer to the *Guide to ObjectPAL*.

Use **run** to switch from the Form Design window to the Form window.

Note: Some form actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

▪

design example

This example uses a custom procedure to force a form (specified by its title) into design mode.

```
proc forceDesign(const foTemp Form) Logical
if foTemp.isDesign() then
   return True
else
   return foTemp.design()
endIf
endProc
```

▪

disableBreakMessage procedure

See also Example Form Type

Prevents program interruption by Ctrl+Break.

Syntax

```
disableBreakMessage ( const yesNo Logical ) Logical
```

Description

disableBreakMessage lets you prevent or allow the user to interrupt a running program with Ctrl+Break.

disableBreakMessage example

In the following example, assume a form contains a table frame bound to the *Orders* table. The following code prevents the loop from being interrupted by a Ctrl+Break.

```
; throughTable::pushButton
method pushButton(var eventInfo Event)
; just a loop to test Ctrl-breaking out of
disableBreakMessage(Yes)     ; don't allow a Ctrl+Break
while NOT ORDERS.atLast()
  ORDERS.action(DataNextRecord)
endwhile
endMethod
```

▪

disablePreviousError procedure

[See also](#) [Example](#) [System Type](#)

Specifies whether you have access to the Previous Error dialog box.

Syntax

`disablePreviousError ( const yesNo Logical ) Logical`

Description

By default, when you move the mouse pointer over the status bar, the pointer changes shape; you can then click the status bar to display the Previous Error dialog box (if error information is available). If *yesNo* is Yes (or True), prevents this behavior; if No (or False), restores the default behavior.

Returns True if successful; otherwise, returns False. This setting remains in effect (and affects all forms) as long as Paradox is running. The default behavior is restored the next time you start Paradox.

disablePreviousError example

Uses **disablePreviousError** in a script named *InitApp* to prevent user access to the Previous Error dialog box.

```
; InitApp::run
method run(var eventInfo Event)
   disablePreviousError(Yes)
   openMainForm() ; Call a custom method to open the main application form.
endMethod
```

dmAddTable method/procedure

See also Example Form Type

Adds a table to a form's data model.

Syntax

```
dmAddTable ( const tableName String ) Logical
```

Description

dmAddTable adds the table *tableName* to a form's data model, where *tableName* is a valid table name. This method returns True if it succeeds; otherwise, it returns False.

dmAddTable example

In the following example, a form contains a button named *toggleSites*, and a list field named *showSiteNames*. The list data for the *showSiteNames* field is set with the DataSource property of its list object, *ListNames*. The **pushButton** method for *toggleSites* checks to see if the *Sites* table is in the data model for the form. If so, the reference to *Sites* is removed from the DataSource property of *ListNames*, then *Sites* is removed from the data model. Otherwise, the *Sites* table is added to the data model, and the DataSource property of *ListNames* is set to the *Site Name* field of *Sites*.

This is the code for the **pushButton** method of *toggleSites*:

```
; toggleSites::pushButton
method pushButton(var eventInfo Event)
; toggle Sites.db in and out of the data model
if dmHasTable("Sites") then    ; is Sites in data model?
  ; if so, remove dependencies, then remove table
  ; remove Sites as source from showSiteNames.ListNames
  showSiteNames.ListNames.DataSource = ""
  showSiteNames.Visible = False
  ; remove Sites from the data model
  dmRemoveTable("Sites")
  whichTable = ""
else
  ; if not already in data model, then add Sites
  dmAddTable("Sites")
  ; set the data for the list from the Sites table
  showSiteNames.ListNames.DataSource = "[Sites.Site Name]"
  showSiteNames.Visible = True
  whichTable = "Sites"
endIf

endMethod
```

▪

dmAttach method/procedure

See also Example Form Type

Associates a TCursor variable with a table in the form's data model.

Syntax

```
dmAttach ( tc TCursor, const tableName String ) Logical
```

Description

dmAttach associates the TCursor variable *tc* with the table *tableName* in the form's data model, where *tableName* is either a valid table name or a table alias. This method returns True if it succeeds; otherwise it returns False.

dmAttach example

The following example demonstrates how to use **dmAttach** and **dmResync** to keep two forms synchronized. Both forms have the *Customer* table in their data models. When the user moves from the first form *frm1* to the second form *frm2*, a form variable *f* is used to attach back to the first form and **dmAttach** is used to attach to the appropriate table in its data model. Finally, **dmResync** is used to move to the same record as the first form.

```
;Frm2.pge1 :: setFocus
method setFocus(var eventInfo Event)
var
   f   Form         ;Declare a form variable.
   tc   TCursor   ;Declare a TCursor variable.
endVar

if f.attach("dmAttach2") then   ;Attach to other form.
   f.dmAttach(tc, "Customer.db")   ;Attach tc to a table in the
   dmResync("Customer.db", tc)   ;data model of the other form.
                              ;Then sync the two forms.
endIf
endMethod
```

▪

dmBuildQueryString method/procedure

See also Example1 Example2 Form Type

Builds a query string based on the data model of a form.

Syntax

`dmBuildQueryString ( var queryString String ) Logical`

Description

dmBuildQueryString builds a query string *queryString* based on the data model of a form. The query built by **dmBuildQueryString** creates checked example elements for all the link fields in the data model. The form's data model must have a linked table. **dmBuildQueryString** returns True if it is successful; otherwise, it returns False.

▪

dmBuildQueryString example 1

For the following example, assume a data model has the *Customer* and *Orders* tables linked on the CustomerNo field. The following code displays a query string based on that data model.

```
method pushButton(var eventInfo Event)
   var
      stQBE String
   endVar
   dmBuildQueryString(stQBE)
   stQBE.view("Query String")

{
Displays the following string:
Query

::C:\PDOXWIN\BLDNOTES\CUSTOMER.DB|CustomerNo   |
                                 |Check _join1!|

::C:\PDOXWIN\BLDNOTES\ORDERS.DB|CustomerNo  |
                               |Check _join1|

EndQuery
}
endMethod
```

dmBuildQueryString example 2

In the following example, suppose a form contains a button named *btnDMQuery*. The **pushButton** method for *btnDMQuery* uses **dmBuildQueryString** as a procedure to generate a query string in *s*. Then **readFromString** is called to assign the string to a Query variable. The method then runs the query and opens a Table window for the *Answer* table.

```
;btnDMQuery :: pushButton
method pushButton(var eventInfo Event)
   var
      s    String
      tv    TableView
      qVar    Query
   endVar

   dmBuildQueryString(s)
   qVar.readFromString(s)
   if qVar.executeQBE() then
      tv.open(":PRIV:ANSWER.DB")
   else
      errorShow()
      return
   endIf
endMethod
```

▪

dmEnumLinkFields method/procedure

See also Example Form Type

Lists the fields that link two tables.

Syntax

`dmEnumLinkFields ( var masterTable String, var masterFields Array[ ] String, const detailTable String, var detailFields Array[ ] String, var detailIndex String ) Logical`

Description

dmEnumLinkFields lists the fields that link the tables named in *masterTable* and *detailTable*. You must supply a table name or table alias for *detailTable*, and this method assigns values to the other variables (passed as arguments) as follows:

Variable	Assigned value
masterTable	The name of the master table. Blank if the table specified in *detailTable* has no master table.
masterFields	Names of the linking fields in the master table. Blank if the table specified in *detailTable* has no master table.
detailFields	Names of the linking fields in the detail table. Blank if the table specified in *detailTable* has no master table. If the detail table is a dBASE table and uses an expression index, the expression is returned in angled brackets. Examples: <FIRSTNAME + LASTNAME> means an expression index based on the fields named FIRSTNAME and LASTNAME; <FIRSTNAME + LASTNAME;QTY > 1> means an expression index based on the fields named FIRSTNAME and LASTNAME with QTY > 1 as a subset condition.
indexName	Name of the index used by the detail table. Blank if the table specified in *detailTable* is not using an index. If the detail table is a dBASE table, you can use **dmGetProperty** to get the associated tag name, if any.

The tables must already be in the specified data model. This method returns True if successful; otherwise, it returns False.

dmEnumLinkFields example

In the following example, assume that a form's data model links the *Customer* and *Orders* tables on the CustomerNo field, with the *Orders* table as the detail table. The tables do not use secondary indexes.

```
method pushButton(Var eventInfo Event)
   var
      mAr, dAr Array[] String
      m, d, inx String
   endVar

   d = "orders"
   dmEnumLinkFields(m, mAr, d, dAr, inx)
   m.view("Master table name")     ; Displays CUSTOMER.DB
   mAr.view("Master link fields") ; Displays Customer No
   d.view("Detail table name")     ; Displays Orders
   dAr.view("Detail link fields") ; Displays Customer No
   inx.view("Index name")          ; Displays Customer No
endMethod
```

▪

dmGet method/procedure

See also Example Form Type

Retrieves a field value from a table in the data model.

Syntax

`dmGet ( const tableName String, const fieldName String, var datum AnyType ) Logical`

Description

dmGet provides access to table data in the form's data model. **dmGet** writes to *datum* a field value from a specified table. The table specified by *tableName* must be the name or table alias of a table in the form's data model (support for table aliases was added in version 5.0). *fieldName* must be a field in *tableName*.

▪

dmGet example

In the following example, a form contains a table frame bound to the *Sites* table. The table frame contains only two fields: Site No and Site Name. The **pushButton** method for a button named *getHighlight* uses **dmGet** to find the value of the Site Highlight field for the current record. The method then displays the Site Highlight value in a dialog box and asks the user whether to change the value. If the user answers "Yes" in the dialog box, the method shows the original value for Site Highlight in a dialog box and prompts the user for a new value. The method then uses **dmPut** to write the changed value back to the *Sites* table:

```
; getHighlight::pushButton
method pushButton(var eventInfo Event)
var
  siteHighlight AnyType
  qAnswer       String
endVar
; get the value in the Site Highlight field for the current record
if dmGet("Sites", "Site Highlight", siteHighlight) then
 ; show the highlight and ask the user whether to change it
 qAnswer = msgQuestion("Change Highlight?",
              "At site " + SITES.Site_Name +
              " the highlight is " +
              String(siteHighlight) + ". Change highlight?")
 if qAnswer = "Yes" then
   ; check for Edit mode
   if thisForm.Editing <> True then
     action(DataBeginEdit)
   endif
   ; ask user to replace existing highlight value in View dialog box
   siteHighlight.view("Enter a new highlight:")
   ; write the changed highlight back to the Site Highlight field
   dmPut("Sites", "Site Highlight", siteHighlight)
 endif
else
  msgStop("Sorry", "Couldn't find the highlight for this site.")
endif
endMethod
```

For information on table aliases, see Table Aliases in the Paradox User's Guide help.

▪

dmGetProperty method/procedure

See also Example Form Type

Returns the value of a specified table property.

Syntax

```
1. dmGetProperty ( const tableName String, const propertyName String )
AnyType
2. dmGetProperty ( const tableName String, const propertyName String, var
value AnyType ) Logical
```

Description

Returns the value of a property *propertyName* of the table *tableName* in the specified data model. The value of *tableName* must be a valid table name or a table alias (support for table aliases was added in version 5.0).

Its return value depends on the value of *propertyName* that you supply from the following:

This value	Returns
AutoAppend	True if AUTO APPEND is set to True for the table. Otherwise, returns False.
Editing	True when a form is in Edit mode, or a field object is active and being edited. Otherwise, returns False.
Flyaway	True when a record has moved to its sorted position in a table. Otherwise, returns False.
FullName	The full file name (as a string, including path or alias) of the table. (Added in version 5.0.)
Index	The name of the index (as a string) that is currently used to view the table. For a child table, it returns the name of the index chosen in the link diagram. For a master table or unlinked table, it returns the setting of ORDER/RANGE. It returns an empty string when the primary key is used.
Inserting	True when a record is being inserted anywhere in a form. Otherwise, returns False.
LinkType	A string describing the way the table relates to its master: "None", "One-to-one", or "One-to-many".
Locked	True when the table bound to a design object is locked. Otherwise, returns False.
Name	The table's table alias (as a string) if it exists; otherwise, returns an empty string. (Added in version 5.0.)
Next	The name (as a string) of the next object in the same container.
One-to-many	The name (as a string) of the first detail table linked 1:M to this table.
One-to-one	The name (as a string) of the first detail table linked 1:1 to this table.
Parent	The table name (as a string) of this table's master in the data model.
Readonly	True if READONLY is set to True for the table. Otherwise, returns False.
Refresh	True when data displayed onscreen is being changed, either across a network, by an ObjectPAL statement, or user action. Otherwise, returns False.
StrictTranslation	True if STRICT TRANSLATION is set to True for the table. Otherwise, returns False.
TagName	The tag name (as a string) for the current dBASE index (if any). Otherwise, it returns an empty string.

Touched	True when the user has made changes to data not yet committed.

Syntax 1 returns the property value directly.

Syntax 2 (added in version 5.0) assigns the value to *value*, an AnyType variable that you declare and pass as an argument. Syntax 2 returns True if the method succeeds; otherwise, it returns False.

For both syntaxes, **dmGetProperty** returns False if *tableName* is not in the data model, or if the value of *propertyName* is not one of the strings listed above. (This feature was added in version 5.0.)

The value of *tableName* must be a valid table name or a table alias (support for table aliases was added in version 5.0).

If *propertyValue* = "Name" this method returns the table's table alias (as a string) if it exists; otherwise, it returns an empty string. (Added in version 5.0.)

If *propertyValue* = "FullName" this method returns the full file name (including path or alias) of the table. (Added in version 5.0.)

dmGetProperty example

The following example sets a table's Auto Append property to False if the table isn't read-only, then checks to see if the table has a one-to-many link to another table. If it does, the read-only setting of the parent table is set to the same read-only setting as the detail (subject) table.

```
method UpdateProperties()

if dmGetProperty(subject.tableName, "ReadOnly") <> True then
  dmSetProperty(subject.tableName, "AutoAppend", False)
endif

if dmGetProperty(subject.tableName, "LinkType") = "One-to-many" then
  dmSetProperty(dmGetProperty(subject.tableName,"Parent"),"ReadOnly",
                dmGetProperty(subject.tableName, "ReadOnly"))
endif

endMethod
```

For information on table aliases, see Table Aliases in the Paradox User's Guide help.

■

dmHasTable method/procedure

See also Example Form Type

Reports whether a table is part of the data model of a form.

Syntax

`dmHasTable ( ``const `***`tableName`***` String `**`)`**` Logical`

Description

dmHasTable reports whether *tableName* is a table associated with a form, where *tableName* is a valid table name or table alias (support for table aliases was added in version 5.0).

▪

dmHasTable example

See the example for **dmAddTable** for an illustration of how to use **dmHasTable** as a procedure.

The following example shows how **dmHasTable** is used as a method. The **pushButton** method for a button named *isStockInDM* works with the form specified by the variable *thatForm*. This method opens the *Ordentry* form, then checks to see if the *Stock* table is in *thatForm's* data model. If not, the *Stock* table is added to the data model for *thatForm*:

```
; isStockInDM::pushButton
method pushButton(var eventInfo Event)
var
  thatForm Form
endVar
thatForm.load("Ordentry")                    ; open ORDENTRY form
if not thatForm.dmHasTable("stock") then     ; is Stock in data model
  msgInfo("Status", "Adding Stock to data model for form.")
  thatForm.dmAddTable("stock")               ; if not, add it
  thatForm.save()
else
  msgInfo("Status", "Stock is already in data model for form.")
endif
thatForm.close()
endMethod
```

For information on table aliases, see Table Aliases in the Paradox User's Guide help.

dmLinkToFields method/procedure

See also Example1 Example2 Form Type

Links two tables in a data model based on lists of field names.

Syntax

```
dmLinkToFields ( const masterTable String, const masterFields Array[ ]
String, const detailTable String, const detailFields Array[ ] String )
Logical
```

Description

dmLinkToFields links the tables specified in *masterTable* and *detailTable* on the field names listed in *masterFields* and *detailFields* (resizeable arrays of strings). The values of *masterTable* and *detailTable* can be table names or table aliases (support for table aliases was added in version 5.0). The tables must already be in the specified data model.

The linking fields cannot be any of the following types: Binary, Byte, Formatted Memo, Graphic, Logical, Memo, or OLE. This method returns True if successful; otherwise, it returns False. If detail table does not have an index that matches the fields in *detailFields*, it returns False.

dmLinkToFields example 1

The following example creates a form, adds the Customer and Orders tables to the new specified data model, and calls **dmLinkToFields** to link the tables. Then it creates some field objects and a table frame and binds them to the tables. Finally, this code runs the new form so you can see the results.

This code specifies the names of the fields to link; you could leave this to Paradox, but Paradox default linking may not give the results you expect.

```
method pushButton(var eventInfo Event)
   var
      masterTC, detailTC      TCursor
      newForm                 Form
      masterFieldsAr,
      detailFieldsAr,
      keyFieldsAr               Array[] String
      badKeyTypesAr            Array[5] String
      masterName,
      detailName,
      keyFieldName,
      newFormName             String
      newField,
      newTFrame               UIObject
      x, y, w, h, offset      LongInt
      i                       SmallInt
   endVar

   ; initialize variables
   masterName = "customer.db"
   detailName = "orders.db"
   newFormName = "custOrd.fsl"

   badKeyTypesAr[1] = "MEMO"       ; types not allowed as key fields
   badKeyTypesAr[2] = "FMTMEMO"
   badKeyTypesAr[3] = "BINARYBLOB"
   badKeyTypesAr[4] = "GRAPHIC"
   badKeyTypesAr[5] = "OLEOBJ"
   badKeyTypesAr[6] = "LOGICAL"
   badKeyTypesAr[7] = "BYTES"

   masterTC.open(masterName)
   masterTC.enumFieldNames(masterFieldsAr)

   detailTC.open(detailName)
   detailTC.enumFieldNames(detailFieldsAr)

   ; specify the key field(s)
   keyFieldName = "Customer No"

   ; make sure key field type is valid
   if badKeyTypesAr.contains(masterTC.fieldType(keyFieldName)) or
      badKeyTypesAr.contains(detailTC.fieldType(keyFieldName)) then
      msgStop("Invalid key field type:",
              keyFieldName + " in\n" +
              masterName + " or\n" + detailName)
      return
   else
      keyFieldsAr.grow(1)
      keyFieldsAr[1] = keyFieldName
   endIf

   ; create the form
   newForm.create()
   newForm.dmAddTable(masterName)
   newForm.dmAddTable(detailName)
```

```
   if newForm.dmLinkToFields(masterName, keyFieldsAr,
                             detailName, keyFieldsAr) then

   ; place objects in the form

      x = 100
      y = 100
      w = 2880
      h = 360
      offset = 10

   ; create field objects bound to master table
      for i from 1 to masterFieldsAr.size()
         newField.create(FieldTool, x, y, w, h, newForm)
         y = y + h + offset
         newField.TableName = masterName
         newField.FieldName = masterFieldsAr[i]
         newField.Visible = Yes
      endFor

   ; create a table frame bound to detail table
      newTFrame.create(TableFrameTool, x, y, w, 8 * h, newForm)
      newTFrame.TableName = detailName
      newTFrame.Visible = Yes

   ; save the form and run it
      newForm.save(newFormName)
      newForm.run()

   else

      errorShow("Link failed")
   endIf

endMethod
```

▪

dmLinkToFields example 2

The following example shows how to use **dmLinkToFields** to link three tables 1:M:M. Like the previous example, this code specifies which fields to link.

```
method pushButton(var eventInfo Event)
   var
      firstTable,
      secondTable,
      thirdTable      String
      firstKeyAr,
      secondKeyAr,
      thirdKeyAr      Array[] String
      newForm         Form
   endVar

   ; initialize variables
   firstTable = "customer.db"
   secondTable = "orders.db"
   thirdTable = "lineitem.db"

   firstKeyAr.grow(1)
   firstKeyAr[1] = "Customer No"
   secondKeyAr.grow(1)
   secondKeyAr[1] = "Customer No"
   ; thirdKeyAr is initialized below, after 1st link

   ; create the form
   newForm.create()

   newForm.dmAddTable(firstTable)
   newForm.dmAddTable(secondTable)
   newForm.dmAddTable(thirdTable)

   ; 1st link
   if newForm.dmLinkToFields(firstTable, firstKeyAr,
                       secondTable, secondKeyAr) then

      ; initialize arrays for 2nd link
      secondKeyAr[1] = "Order No"

      thirdKeyAr.grow(1)
      thirdKeyAr[1]  = "Order No"

      ; 2nd link
      if newForm.dmLinkToFields(secondTable, secondKeyAr,
                       thirdTable, thirdKeyAr) then

         {Code to create UIObjects in new form could go here.}

         newForm.save("ordentry.fsl")

      else
         errorShow("2:3 link failed.")
      endIf

   else
      errorShow("1:2 link failed.")
   endIf

endMethod
```

▪

dmLinkToIndex method/procedure

See also Example Form Type

Links two tables in the form's data model based on a list of field names and an index name.

Syntax

```
dmLinkToIndex ( const masterTable String, const masterFields Array[ ] String,
const detailTable String, const detailIndex String ) Logical
```

Description

Links the tables specified in *masterTable* and *detailTable* on the field names listed in *masterFields* and the index specified in *detailIndex*. You can specify a Paradox table's primary index by assigning an empty string to *detailIndex*.

The values of *masterTable* and *detailTable* can be table names or table aliases (support for table aliases was added in version 5.0). The tables must already be in the specified data model. This method returns True if successful; otherwise, it returns False.

The linking fields cannot be any of the following types: Binary, Bytes, Formatted Memo, Graphic, Logical, Memo, or OLE.

dmLinkToIndex example

The following example creates a form, adds the Customer and Orders tables to the new specified data model, and calls **dmLinkToIndex** to link the tables. Then it creates some field objects and a table frame and binds them to the tables. Finally, this code runs the new form so you can see the results.

```
method pushButton(var eventInfo Event)
   var
      masterTC, detailTC      TCursor
      newForm                 Form
      masterFieldsAr,
      detailFieldsAr,
      masterKeysAr,
      detailKeysAr            Array[] String
      masterName,
      detailName,
      detailIndexName,
      newFormName             String
      newField,
      newTFrame               UIObject
      x, y, w, h, offset      LongInt
      i                       SmallInt
   endVar

   ; Initialize variables
   detailIndexName = "Customer No"
   newFormName = "idxDemo"
   masterName = "customer.db"
   detailName = "orders.db"

   masterTC.open(masterName)
   masterTC.enumFieldNames(masterFieldsAr)
   masterTC.enumFieldNamesInIndex(masterKeysAr)

   detailTC.open(detailName)
   detailTC.enumFieldNames(detailFieldsAr)


   ; create the form
   newForm.create()
   newForm.dmAddTable(masterName)
   newForm.dmAddTable(detailName)

   if newForm.dmLinkToIndex(masterName, masterKeysAr,
                        detailName, detailIndexName) then

      x = 100
      y = 100
      w = 2880
      h = 360
      offset = 10

      for i from 1 to masterFieldsAr.size()
         newField.create(FieldTool, x, y, w, h, newForm)
         y = y + h + offset
         newField.TableName = masterName
         newField.FieldName = masterFieldsAr[i]
         newField.Visible = Yes
      endFor

      newTFrame.create(TableFrameTool, x, y, w, 8 * h, newForm)
      newTFrame.TableName = detailName
      newTFrame.Visible = Yes
```

```
      newForm.save(newFormName)
      newForm.run()

   else

      errorShow("Link failed")

   endIf

endMethod
```

For information on table aliases, see Table Aliases in the Paradox User's Guide help.

▪

dmPut method/procedure

See also Example Form Type

Writes data to a table in the data model.

Syntax

```
dmPut ( const tableName String, const fieldName String, const datum AnyType )
Logical
```

Description

dmPut provides access to table data in the data model. **dmPut** writes *datum* to a field in a specified table. The value of *tableName* can be a table name or a table alias (support for table aliases was added in version 5.0). The table specified by *tableName* must be one of the tables in the specified data model. *fieldName* must be a field in *tableName*. This method returns True if it succeeds; otherwise, it returns False.

▪

dmPut example

See the example for **dmGet**.

▪

dmRemoveTable method/procedure

See also Example Form Type

Removes a table from the form's data model.

Syntax

`dmRemoveTable ( const tableName String ) Logical`

Description

dmRemoveTable removes *tableName* from a form's data model. The value of *tableName* can be a table name or a table alias (support for table aliases was added in version 5.0). Any objects on the form that depend on the table will be undefined when the table is removed. If any UIObjects in the form are bound to the table, **dmRemoveTable** fails. It returns True if it succeeds; otherwise, it returns False.

▪

dmRemoveTable example

See the example for **dmAddTable**.

For information on table aliases, see Table Aliases in the Paradox User's Guide help.

dmResync method/procedure

See also Example Form Type

Resynchronizes a table in the form's data model to a TCursor.

Syntax

```
dmResync ( const tableName String, var tc TCursor ) Logical
```

Description

dmResync synchronizes a specified table in a data model with the TCursor *tc*. The value of *tableName* can be a table name or a table alias (support for table aliases was added in version 5.0).

When you resynchronize a table to a TCursor, the table's filter, index, and current record position will be changed to those of the TCursor. (For dBASE tables, the table will also take the Show Deleted setting of the TCursor.) This method works on forms in design mode or run mode.

Note: **dmResync** only works when the TCursor is associated with the table in the data model. However, the table does not have to be displayed in the form.

dmResync example

This example shows how to use **dmResync** with the DataSource property to add items to a drop-down edit list. First, it shows how to use DataSource alone, which fills a list with values from a specified field (column) of a table. Then it shows how to use a TCursor and **dmResync** to fill a list with a specified subset of those values.

A field displayed as a drop-down edit list is a compound object: the field object (which displays the field value) contains a list object (which contains the items in the list). In a form, the list object is represented by the down-arrow (the arrow you click to display the list). See the *Guide to ObjectPAL* for more details about lists and list objects.

The usual place to attach list-building code is the list object's built-in **open** method, but you can attach the code to other methods, or even to other objects (as shown in the second part of this example).

Suppose a form contains a field object displayed as a drop-down edit list. The field object is bound to the ShipVia field of the *Orders* table. The following code is attached to the built-in **open** method of the list object (not the field object) named *shipViaList*. It fills the list with all the values in the ShippingCo field of the *Shippers* table in the working directory.

```
; shipViaList::open
; Full containership path: form.page.ShipVia.shipViaList
method open (var eventInfo Event)
   doDefault
   ; Fills list with all values in ShippingCo field of Shippers table.
   self.DataSource = "[Shippers.ShippingCo]"
endMethod
```

The following code uses **dmResync** to filter the list depending on the value of another field. The premise here is that certain shipping methods are less expensive (and so more desirable) in certain parts of the country. So, when the user changes the value of the *State* field, this code updates the items in the list of shippers.

```
; State::changeValue
method changeValue (var eventInfo ValueEvent)
   var
      tcShippers   TCursor
      stStateCode,
      stFldName,
      stDmTbName   String
      dyCriteria   DynArray[] AnyType
   endVar

   doDefault ; Execute the built-in code to commit the field value.
   if eventInfo.errorCode() <> 0 then
      return ; If there's an error, exit the method.
   endIf

   stStateCode = self.Value  ; Get the value of the State field.
   stFldName   = "State"     ; Filter on the State field.
   stDmTbName  = "Shippers"

   dyCriteria[stFldName] = stStateCode

   ; Associate a TCursor with a table in the form's data model.
   dmAttach(tcShippers, stDmTbName)

   tcShippers.setGenFilter(dyCriteria) ; Set a filter on the TCursor.
   ; You could also set an index, etc.

   ; Synchronize the table in the data model with the TCursor.
   ; The table takes the filter from the TCursor.
   dmResync(stDmTbName, tcShippers)

   ; Now the list displays only the shippers for the specified state.
   ShipVia.shipViaList.DataSource = "[Shippers.ShippingCo]"

endMethod
```

For information on table aliases, see Table Aliases in the Paradox User's Guide help.

▪

dmSetProperty method/procedure

See also Example Form Type

Sets the value of a specified table property.

Syntax

```
dmSetProperty ( const tableName String, const propertyName String, value
AnyType) Logical
```

Description

dmSetProperty lets you change the value of a property, specified by *propertyName*, associated with the table specified in *tableName* and found in the specified data model.

The value of *tableName* can be a table name or a table alias (support for table aliases was added in version 5.0). The value of *propertyName* is one of the following properties:

AutoAppend	Set *propertyValue* to True to set AUTO APPEND ON for the table. Otherwise, set it to False.
Name	The value of *propertyValue* specifies the table's table alias as a string. The operation fails if the table alias is already in use. (Added in version 5.0.)
ReadOnly	Set *propertyValue* to True if READONLY should be True for the table. Otherwise, set it to False.
StrictTranslation	Set *propertyValue* to True if STRICT TRANSLATION should be True for the table. Otherwise, set it to False.
Touched	Set *propertyValue* to True when the user has made changes not yet committed.

dmSetProperty example

See the example for **dmGetProperty**.

For information on table aliases, see Table Aliases in the Paradox User's Guide help.

dmUnlink method/procedure

See also　　Example　　Form Type

Unlinks two tables in the form's data model.

Syntax

```
dmUnlink ( const masterTable String, const detailTable String ) Logical
```

Description

dmUnlink unlinks the tables specified in *masterTable* and *detailTable*. *masterTable* must refer to the master table in the link, and *detailTable* must refer to the detail table in the link. The values of *masterTable* and *detailTable* can be table names or table aliases (support for table aliases was added in version 5.0).

This method fails if the tables are not in the specified data model; it also fails if they are in the specified data model but not linked.

This method returns True if successful; otherwise, it returns False.

▪

dmUnlink example

```
method pushButton(var eventInfo Event)

   var
      theForm            Form
      masterTable,
      oldDetailTable,
      newDetailTable,
      oldFormName,
      newFormName         String
      newKeysAr         Array[] String
   endVar

   ; initialize variables
   oldFormName = "custOrd"
   newFormName = "newOrd"

   masterTable    = "CUSTOMER"
   oldDetailTable = "ORDERS"
   newDetailTable = "NEW_ord"

   newKeysAr.grow(1)
   newKeysAr[1] = "Customer No"

   ; load the form and change the data model
   theForm.load(oldFormName)

   if theForm.dmHasTable(masterTable) and
      theForm.dmHasTable(oldDetailTable) then

      theForm.dmAddTable(newDetailTable)
      theForm.dmUnlink(masterTable, oldDetailTable)

      theForm.dmLinkToFields(masterTable, newKeysAr,
                             newDetailTable, newKeysAr)

      theForm.ORDERS.TableName = newDetailTable

      theForm.dmRemoveTable(oldDetailTable)
      theForm.save(newFormName)

   else
      errorShow()
   endIf

endMethod
```

For information on table aliases, see Table Aliases in the Paradox User's Guide help.

▪

enumDataModel method/procedure

See also Example Form Type

Lists the tables in the form's data model.

Syntax

```
enumDataModel ( const tableName String ) Logical
```

Description

enumDataModel creates a Paradox table listing information about the tables in the form's data model. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of the created table is

Field Name	Type	Description
TableName	A128	Table alias, if it exists, or file name of the table (without file extension)
PropertyName	A64	A property name
PropertyValue	A255	Value of the corresponding property

■

enumDataModel example

In the following example, a form contains a button named *enumerateDataModel*. The **pushButton** method for *enumerateDataModel* uses **enumDataModel** as a procedure to enumerate the properties of all the tables in the data model for the current form to a table called DMORDERS.DB. The method then opens a table window for the *DMOrders* table.

```
;enumerateDataModel::pushButton
method pushButton(var eventInfo Event)
   var
      tv   TableView
   endVar

   enumDataModel("dmOrders.db")
   tv.open("dmOrders.db")
endMethod
```

Property Names for Form::enumDataModel

Property	Description
AutoAppend	Returns True if AUTO APPEND is set to True for the table. Otherwise, returns False.
FullName	Returns the full file name (including path or alias) of the table.
Index	Returns the name of the index (as a string) that is currently used to view the table. For a child table, it returns the name of the index chosen in the link diagram. For a master table or unlinked table, it returns the setting of ORDER/RANGE. It returns an empty string when the primary key is used.
LinkFields	Returns a comma-separated list of fields that define the link. If the detail table is a dBASE table and uses an expression index, the expression is returned in angled brackets. Examples: <FIRSTNAME + LASTNAME> means an expression index based on the fields named FIRSTNAME and LASTNAME; <FIRSTNAME + LASTNAME;QTY > 1> means an expression index based on the fields named FIRSTNAME and LASTNAME with QTY > 1 as a subset condition.
LinkType	Returns a string describing the way the table relates to its master: "None", "One-to-one", or "One-to-many".
Name	Returns the table's table alias (as a string) if it exists; otherwise, returns an empty string.
Next	Returns the name (as a string) of the next object in the same container.
One-to-many	Returns the name (as a string) of the first detail table linked 1:M to this table.
One-to-one	Returns the name (as a string) of the first detail table linked 1:1 to this table.
Parent	Returns the table name (as a string) of this table's master in the data model. For example, in a CUSTOMER->>BOOKORD form, dmGetProperty("BOOKORD","PARENT") = "CUSTOMER.DB". If the table has no master, an empty string is returned.
Readonly	Returns True if READONLY is set to True for the table. Otherwise, returns False.
StrictTranslation	Returns True if STRICT TRANSLATION is set to True for the table. Otherwise, returns False.
TagName	Returns the tag name (as a string) for the current dBASE index (if any). Otherwise, it returns an empty string.

▪

enumSource method/procedure

See also   Example   Form Type

Creates a table listing the methods for each object in a form.

Syntax

`enumSource ( const tableName String [ , const recurse Logical ] ) Logical`

Description

enumSource creates a Paradox table listing every object you have written a method for, along with the ObjectPAL source code for the method. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of the created table is

Field name	Type	Size
Object	A	128
MethodName	A	128
Source	M	64

The Object field contains the full path name of the object.

If *recurse* is False, this method returns the method definitions for the form only. To include the source code of methods for all objects contained by the form, *recurse* must be True.

▪

enumSource example

In the following example, a form contains a button named *getSource*. The **pushButton** method for *getSource* uses **enumSource** as a procedure to enumerate the source code for the current form to a table named TEMPSORC.DB. The method then opens a table window for the *Tempsorc* table and waits for the user to close it. Then the method opens the *Sitenote* form to *siteForm*, uses enumSource as a method to write the source code for *siteForm* to a table named SITESORC.DB, and views the table:

```
; getSource::pushButton
method pushButton(var eventInfo Event)
var
  siteForm   Form
  tempTable  TableView
endVar

siteForm.open("Sitenote.fsl")        ; open another form

; write source for siteForm to SITESORC.DB
siteForm.enumSource("sitesorc.db", True)
siteForm.close()                  ; close the form
tempTable.open("sitesorc.db")     ; view the new table
tempTable.wait()                  ; wait for the user to close
                                  ; the table
endMethod
```

■

enumSourceToFile method/procedure

See also Example Form Type

Creates a file listing the methods for each object in a form.

Syntax

```
enumSourceToFile ( const fileName String [ , const recurse Logical ] )
Logical
```

Description

enumSourceToFile creates a text file listing every object you've written a method for, along with the ObjectPAL source code for the method. Use the argument *fileName* to specify a name for the file. If *fileName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *fileName*; if no alias or path is specified, Paradox creates *fileName* in the working directory (:WORK:).

If *recurse* is False, this method returns the method definitions for the form only. To include the source code of methods for all objects contained by the form, *recurse* must be True.

▪

enumSourceToFile example

The following code is attached to the **pushButton** method for a button named *getSourceToFile*. This method writes all the source code for the current form to TEMPSORC.TXT. The method then opens the *Sitenote* form and writes all the code for that form to a file named SITESORC.TXT:

```
; getSourceToFile::pushButton
method pushButton(var eventInfo Event)
var
  siteForm   Form
endVar
enumSourceToFile("tempsorc.txt", True) ; writes all source for the
                                       ; current form to TEMPSORC.TXT

siteForm.open("Sitenote.fsl")              ; open another form
; write source for siteForm to SITESORC.TXT
siteForm.enumSourceToFile("sitesorc.txt", True)
siteForm.close()                       ; close the form
endMethod
```

enumTableLinks method/procedure

See also Example Form Type

Creates a table listing the tables linked in a form.

Syntax

```
enumTableLinks ( const tableName String ) Logical
```

Description

enumTableLinks creates a Paradox table listing the names of tables linked in a form and the types of links. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

This method creates a table containing one record for each table in the data model. The structure of the table (which changed in version 5.0) is

Field name	Type	Description
Table	A255*	Table name, without alias, path, or extension (for example, ORDERS).
Parent	A255*	Name of parent table, or blank if table has no parent.
LinkType	A24*	Type of link between table and parent table: None, One-to-many, or One-to-one.

enumTableLinks example

In the following example, the **pushButton** method for a button named *showTableLinks* writes table links for the current form to a table named TEMPLINK.DB. The method then opens the *Sitenote* form, and writes the table links for that form to a table named SITENOTE.DB:

```
; showTableLinks::pushButton
method pushButton(var eventInfo Event)
var
  siteForm  Form
  tempTable TableView
endVar
enumTableLinks("templink.db")           ; lists links to current form
tempTable.open("templink")
tempTable.wait()
siteForm.open("Sitenote.fsl")
siteForm.enumTableLinks("Sitenote.db") ; lists links to siteForm
siteForm.close()
tempTable.open("Sitenote.db")
tempTable.wait()
tempTable.close()
endMethod
```

▪

enumUIObjectNames method

See also   Example   Form Type

Creates a table listing the UIObjects contained in a form.

Syntax

```
enumUIObjectNames ( const tableName String ) Logical
```

Description

enumUIObjectNames creates a Paradox table listing the name and type of each object contained in a form. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of tableName is

Field Name	Type	Size
ObjectName	A	128
ObjectClass	A	32

Note: ObjectName includes the entire path name of the object.

enumUIObjectNames example

In the following example, the **pushButton** method for a button named *getObjectNames* opens the *Sitenote* form and enumerates all the object names on the form to a table named *Siteobjs*. The method then opens the *Siteobjs* table and waits for the user to close it:

```
; getObjectNames::pushButton
method pushButton(var eventInfo Event)
var
  siteForm  Form
  tempTable TableView
endVar
if siteForm.open("Sitenote.fsl") then            ; open the form
  siteForm.enumUIObjectNames("siteobjs.db") ; write object names
                                            ; SITEOBJS.DB
  siteForm.close()                          ; close the form
  tempTable.open("siteobjs")                ; open the new table
  tempTable.wait()                          ; wait for return
  tempTable.close()                         ; close after return
endIf
endMethod
```

enumUIObjectProperties method

See also Example Form Type

Lists the properties of each UIObject contained in a form.

Syntax

`enumUIObjectProperties ( const tableName String ) Logical`

Description

enumUIObjectProperties creates a Paradox table listing the name, property name, and property value of each object contained in a form. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails.

The structure of *tableName* is:

Field name	Type	Size
ObjectName	A	128
PropertyName	A	64
PropertyType	A	48
PropertyValue	A	255

enumUIObjectProperties example

In the following example, the **pushButton** method for a button named *getProps* writes the properties for all objects contained by the current form to a table named *Tempprop*:

```
; getProperties::pushButton
method pushButton(var eventInfo Event)
var
  siteForm  Form
  tempTable TableView
endVar
if siteForm.open("Sitenote.fsl") then
  message("Enumerating properties to Siteprop table.")
  siteForm.enumUIObjectProperties("siteProp.db")
  tempTable.open("siteprop")
  message("Close the table to continue.")
  tempTable.wait()
  tempTable.close()
endIf
; to enumerate objects for current form, use the UIObject
; type method enumUIObjectProperties
; thisForm is the object ID for current form
message("Enumerating properties to Tempprop table.")
   enumUIObjectProperties("tempprop.db")
tempTable.open("tempprop")
message("Close the table to continue.")
tempTable.wait()
tempTable.close()
endMethod
```

▪

formCaller procedure

See also Example Form Type

Creates a handle to the calling form.

Syntax

`formCaller ( var caller Form ) Logical`

Description

formCaller assigns the handle of the current form's calling form to *caller*, if the form is in a wait. If the current form was not opened by another form, and the form that opened the current form is not waiting upon the current form, the method returns False, and *caller* is unassigned.

formCaller example

In the following example, the **pushButton** method for *whoCalledMe* finds out which form called the current form:

```
; callOtherForm::pushButton  (calling form)
method pushButton(var eventInfo Event)
var
  siteForm  Form
endVar
siteForm.open("sitenote.fsl")  ; open siteForm
siteForm.wait()                ; wait for siteForm to return
siteForm.close()               ; close siteForm
endMethod
```

This is the code for *whoCalledMe* on the current form.

```
; whoCalledMe::pushButton
method pushButton(var eventInfo Event)
var
  myCaller     Form
  callerTitle  AnyType
endVar
if formCaller(myCaller) then        ; try to get a handle to
                                    ; the calling form
  callerTitle = myCaller.getTitle() ; get the form's title
  msgInfo("FYI", "I was called by: \n" + callerTitle)
endif
formReturn()
endMethod
```

▪

formReturn procedure

See also Example Form Type

Returns control to a suspended method.

Syntax

```
formReturn ( [ const returnValue AnyType ] )
```

Description

When one form opens another form and calls **wait**, the first form suspends ObjectPAL execution (in effect, yielding to the second form) until the second form returns control by calling **formReturn**. You can choose to return a value to the first form in *returnValue*. You can also use **formReturn** to return control (and a value) from a script.

formReturn posts a message to the Windows message queue, so ObjectPAL statements that follow **formReturn** will execute before the form returns control.

If no other form is waiting for the current form, **formReturn** closes the current form. If a form is waiting for the current form, **formReturn** does not close the current form.

formReturn example

The following example consists of three methods. The **pushButton** method for *openDialog* opens another form as a dialog box and waits for it to return a value. The other two methods are attached to buttons in the dialog box form. They use **formReturn** to return control and values to the calling form. Note that the calling form must call **close** to close the dialog box; the call to **formReturn** does not close it.

```
; openDialog::pushButton
method pushButton(var eventInfo Event)
var
  dlgForm     Form
  whichButton String
endVar
if dlgForm.openAsDialog("foforet2", WinStyleDefault,
                        1440, 1440, 7200, 5760) then
  ; waits until dlgForm calls formReturn or is closed
  ; returned value is stored to whichButton
  whichButton = String(dlgForm.wait())
  dlgForm.close()
  ; return value is cast to a String so that it will be correct
  ; type even if user closes dialog box from the system menu
  msgInfo("Button pressed", whichButton)
else
  msgStop("Stop", "Couldn't open the form.")
endIf
endMethod
```

This method is attached to **pushButton** method for *OKButton* in *dlgForm.* It returns a value of "OK" when it returns control to the method that called wait:

```
; OKButton::pushButton
method pushButton(var eventInfo Event)
formReturn("OK")      ; return "OK" to calling form
endMethod
```

This method is attached to *cancelButton* in *dlgForm.* It returns a value of "Cancel" when it returns control to the method that called **wait**. The **message** statement that follows the call to **formReturn** is not required; it is included here to show that statements following a call to **formReturn** execute before control is returned to the calling form.

```
; cancelButton::pushButton
method pushButton(var eventInfo Event)
formReturn("Cancel")  ; return "Cancel" to calling form
message("Cancel")     ; This statement will execute.
endMethod
```

■

getFileName method/procedure

See also Example Form Type

Returns the path, file name, and extension of the associated form.

Syntax

```
getFileName ( ) String
```

Description

As a method, **getFileName** returns the path, file name, and extension of the form associated with a Form variable. As a procedure, it returns the path, file name, and extension of the current form.

Compare this method to **getTitle,** which returns the text in a Form window's title bar.

getFileName example

The following example displays the file name of the current form in the status bar.

```
method pushButton(var eventInfo Event)
   message(getFileName())
endMethod
```

▪

getPosition method/procedure

See also   Example   Form Type

Reports the position of a window onscreen.

Syntax

```
getPosition ( var x LongInt, var y LongInt, var w LongInt, var h LongInt )
```

Description

getPosition gets the position of a window relative to the Paradox Desktop. The arguments *x* and *y* contain the horizontal and vertical coordinates of the upper left corner of the form (in twips), and *w* and *h* contain the width and height (in twips).

To ObjectPAL, the screen is a two-dimensional grid, with the origin (0, 0) at the upper left corner of an object's container, positive x-values extending to the right, and positive y-values extending down.

For dialog boxes and for the Paradox Desktop application, the position is given relative to the entire screen; for forms, reports, and table windows, the position is given relative to the Paradox Desktop.

getPosition example

In the following example, the **pushButton** method for *moveOtherForm* opens a form and gets its position. The method then opens a second instance of the same form and sets its position so that no part of the second form overlaps the first:

```
; moveOtherForm::pushButton
method pushButton(var eventInfo Event)
var
  siteFormOne,
  siteFormTwo    Form
  x, y, w, h     LongInt
endVar
if siteFormOne.open("Sitenote") then
  siteFormOne.getPosition(x, y, w, h)
  siteFormTwo.open("Sitenote.fsl")    ; open another instance
  ; set position so that no part overlaps other instance
  siteFormTwo.setPosition(x + w, y + h, w, h)
endif
endMethod
```

▪

getProtoProperty method/procedure

See also Example Form Type

Reports the value of a specified property of a prototype object.

Syntax

```
getProtoProperty ( const objectType SmallInt, propertyName String ) AnyType
```

Description

getProtoProperty returns the value of the property specified in *propertyName* of the prototype object specified in *objectType*. To specify *objectType*, use one of the UIObjectTypes constants. If called as a method, **getProtoProperty** operates on prototype objects in the style sheet of the specified form. If called as a procedure, it uses the style sheet of the current form.

getProtoProperty example

This example uses **getProtoProperty** to store the current default color for the box tool. Next, it specifies a new box color and creates three new boxes. Then it restores the default box color.

```
   const
      kOneInch = 1440 ; One inch = 1,440 twips.
   endConst
method mouseClick(var eventInfo MouseEvent)
   var
      uiRedBox       UIObject
      thisForm       Form
      liDefaultBoxColor    LongInt
   endVar
   thisForm.attach() ; Get a handle to this form.

   ; Get current default color.
   liDefaultBoxColor = thisForm.getProtoProperty(BoxTool, "Color")

   ; Set box color and create 3 boxes using new prototype.
   thisForm.setProtoProperty(BoxTool, "Color", Red)
   uiRedBox.create(BoxTool, kOneInch, kOneInch, kOneInch, kOneInch)
   uiRedBox.Visible = Yes
   uiRedBox.create(BoxTool, 2 * kOneInch, kOneInch, kOneInch, kOneInch)
   uiRedBox.Visible = Yes
   uiRedBox.create(BoxTool, 3 * kOneInch, kOneInch, kOneInch, kOneInch)
   uiRedBox.Visible = Yes

   ; Restore the default box color.
   thisForm.setProtoProperty(BoxTool, "Color", liDefaultBoxColor)
endMethod
```

▪

getSelectedObjects method/procedure

See also Example Form Type

Creates an array listing the selected objects in a form.

Syntax

```
getSelectedObjects ( var objects Array[ ] UIObject ) SmallInt
```

Description

getSelectedObjects creates an array *objects* listing the selected objects of a form, and returns the number of objects selected. This procedure is useful for creating routines that manipulate objects on forms in design mode.

▪

getSelectedObjects example

This example creates a form that contains three boxes. Next, it selects two of the boxes, displays their names in a dialog box, and sets their color to blue.

```
;btnObjectsSelected :: pushButton
const
   kOneInch = 1440 ; One inch = 1,440 twips.
endConst

method pushButton(var eventInfo Event)
   var
      foTemp   Form
      arObjects   Array[] UIObject
      arObjNames   Array[] String
      uiVar      UIObject
      si,
      siSelObj   SmallInt
      stBoxName   String

   endVar

   foTemp.create()

   ; Create 3 boxes.
   for si from 1 to 3
      uiVar.create(BoxTool, si * kOneInch, si*kOneInch,
                   kOneInch, kOneInch, foTemp)
      uiVar.Name = "Box" + String(si)
      uiVar.Visible = Yes
   endFor

   ; Select Box2 and Box3 by setting the Select property.
   for si from 2 to 3
      stBoxName = "Box" + String(si)
      uiVar.attach(foTemp.(stBoxName))
      uiVar.Select = Yes
   endFor

   ; Get the selected objects.
   siSelObj = foTemp.getSelectedObjects(arObjects)
   siSelObj.view("Number of selected objects:")

   ; Get the names of the selected objects.
   arObjNames.setSize(siSelObj)
   for si from 1 to siSelObj
      uiVar.attach(arObjects[si])
      arObjNames[si] = uiVar.Name
   endFor
   arObjNames.view("Names of selected objects:")

   ; Change the color of the selected objects.
   for si from 1 to arObjects.size()
      uiVar.attach(arObjects[si])
      uiVar.Color = Blue
   endFor

   foTemp.close()
endMethod
```

▪

getStyleSheet method/procedure

See also Example Form Type

Returns the name of a form's style sheet.

Syntax

```
getStyleSheet ( ) String
```

Description

getStyleSheet returns the file name of a form's style sheet. If the style sheet is in :WORK:, **getStyleSheet** returns the file name and extension, if any (for example, BORLAND.FT). Otherwise, **getStyleSheet** returns the full path (for example, C:\PDOXWIN\BORLAND.FT).

If called as a method, **getStyleSheet** returns the file name of the style sheet of the specified form. If called as a procedure, it uses the style sheet of the current form.

It returns the name of the style sheet used by the specified form, which may be different from the Paradox system style sheet. To get the name of the default screen style sheet, call the **getDefaultScreenStyleSheet** procedure defined for the System type. To get the name of the default printer style sheet, call the **getDefaultPrinterStyleSheet** procedure defined for the System type.

▪

getStyleSheet example

See the example for **setStyleSheet**.

■

getTitle method/procedure

See also Example Form Type

Returns the text of the window title bar.

Syntax

```
getTitle ( ) String
```

Description

getTitle returns the text in the title bar of the window containing the object.

getTitle example

In the following example, the **pushButton** method for *showTitle* opens a form, gets the new form's title and displays the title in a dialog box. This method then switches the open form to the Form Design window and retrieves its title again:

```
; showTitle::pushButton
method pushButton(var eventInfo Event)
var
   siteForm  Form
   titleText String
endVar
siteForm.open("Sitenote.fsl")
titleText = siteForm.getTitle() ; reads window title into titleText
msgInfo("Title:", titleText)    ; displays "Form : SITENOTE.FSL"
siteForm.design()               ; switch to the Form Design window
sleep()                         ; yield!
titleText = siteForm.getTitle() ; get the Form Design window title
msgInfo("Title:", titleText)    ; displays "Form Design: SITENOTE.FSL"
siteForm.close()
endMethod
```

hide method/procedure

See also Example Form Type
Beginner

Makes a window invisible.

Syntax
`hide ( )`

Description
hide makes a window invisible but doesn't close it.

hide example

In the following example, the **pushButton** method for *hideForm* opens a form, hides it, then shows it:

```
; hideForm::pushButton
method pushButton(var eventInfo Event)
var
  siteForm Form
endVar
siteForm.open("Sitenote.fsl")         ; displays Sitenote form
siteForm.hide()                     ; makes form invisible
siteForm.action(DataEnd)            ; move to the end of the table
siteForm.action(DataBeginEdit)      ; start edit mode
siteForm.action(DataInsertRecord) ; insert a new, blank record
if NOT siteForm.isVisible() then
  msgInfo("Status", "It's hidden.")
endif
message("Come out, come out, wherever you are!")
siteForm.show()                     ; make form visible again
if siteForm.isVisible() then
  msgInfo("Status", "It's visible.")
endif
endMethod
```

hideToolbar procedure

[See also](#) [Example](#) [Form Type](#)

Makes the standard Toolbar invisible.

Syntax

```
hideToolbar ( )
```

Description

hideToolbar removes the standard Toolbar from the Desktop. You must call **showToolbar** to restore it.

hideToolbar example

In the following example, the **pushButton** method for the *toggleToolbar* button checks whether the Toolbar is showing. If the Toolbar is visible, this method hides it; if the Toolbar isn't visible, this method shows it:

```
; toggleToolbar::pushButton
method pushButton(var eventInfo Event)
if isToolbarShowing() then    ; if Toolbar is off
   hideToolbar()              ; hide it
else                          ; otherwise
   showToolbar()              ; show it
endif
endMethod
```

isCompileWithDebug method

See also Example Form Type

Reports the status of the compile with debug setting

Syntax

`isCompileWithDebug ( )` `Logical`

Description

isCompileWithDebug reports the status of the compile with debug setting that can be set interactively during form design. **isCompileWithDebug** returns True if Compile With Debug is set in the form.

isCompileWithDebug example

In the following example, the central form of a management system has two buttons: getCompileStatus and setCompileStatus. The pushButton method of each button opens the Windows 95 file browser dialog to allow a user to select the file which will be examined/manipulated. Each method analyzes the fileName selected to determine which fileType and then opens it under the appropriate object type.

The following code is attached to the pushButton method for getCompileStatus:

```
; getCompileStatus::pushButton
method pushButton(var eventInfo Event)
var
   theForm  Form         ;// Object variable for forms
   theLibrary  Library      ;// Object variable for forms
   theScript  Script      ;// Object variable for forms
   fbi  FileBrowserInfo      ;// File Browser information structure
   selectedFile  String      ;// FileName selected by user
   fileType  String      ;// File type of file selected by user
   status  Logical      ;// Debug status of the selected file
endVar
         ;//Set allowable file types: Forms, Libraries, and Scripts
   fbi.AllowableTypes = fbForm + fbLibrary + fbScript
   if fileBrowser(selectedFile, fbi) then
      ;// The user selected a file
     fileType = upper(substr(selectedFile, selectedFile.size() - 2, 3))
      switch
         case fileType = "FSL" :
            ;// Load the Form
            theform.load(fbi.Drive + fbi.Path + selectedFile)
            ;// Determine its status
            status = theForm.isCompileWithDebug()
            ;// Close the Form
            theForm.close()

         case fileType = "LSL" :
            ;// Load the Library
            theLibrary.load(fbi.Drive + fbi.Path + selectedFile)
            ;// Determine its status
            status = theLibrary.isCompileWithDebug()
            ;// Close the Library
            theLibrary.close()

         case fileType = "SSL" :
            ;// Load the Script
            theScript.load(fbi.Drive + fbi.Path + selectedFile)
            ;// Determine its status
            status = theScript.isCompileWithDebug()
            ;// Close the Script
            theScript.close()
      endSwitch
      ;// Inform the user
      msgInfo(selectedFile + " compiled with Debug information?", status)
   else
      ;// The user didnt select a file
      msgInfo("No file selected", "Please try again.")
   endIf
endMethod
```

isDesign method/procedure

See also Example Form Type

Reports whether a form is displayed in a Form Design window.

Syntax

```
isDesign ( ) Logical
```

Description

isDesign returns True if a form is displayed in a Form Design window; otherwise, it returns False.

isDesign example

In the following example, **enumFormNames** is used to populate an array *ar* with the names of the open forms. Then a **for** loop steps through the array and saves the form if it is in design mode.

```
;btnSaveForms :: pushButton
method pushButton(var eventInfo Event)
   var
      ar               Array[] AnyType
      siCounter   SmallInt
      f                   Form
   endVar

   enumFormNames(ar)

   for siCounter from 1 to ar.size()
      f.attach(ar[siCounter])
      if f.getFileName() = "" then
         msgStop("Warning", "At least one form is a new form.")
      else
         if f.isDesign() then
            f.save()
         endIf
      endIf
   endFor
endMethod
```

▪

isMaximized method/procedure

See also   Example   Form Type

Reports whether a window is displayed at its maximum size.

Syntax

`isMaximized ( ) Logical`

Description

isMaximized returns True if a form is displayed full screen; otherwise, it returns False.

▪

isMaximized example

In the following example, the **pushButton** method for the *cycleSize* button (on the current form) opens or attaches to the *Sitenote* form with the variable *siteForm*. If *siteForm* is maximized, this method minimizes it. If *siteForm* is minimized, this method restores it to its previous size with the show method. If *siteForm* is neither maximized nor minimized, this method maximizes it:

```
; cycleSize::pushButton
method pushButton(var eventInfo Event)
var
  siteForm Form
endVar
; try attaching to form, since it might be open
if NOT siteForm.attach("Form : SITENOTE.FSL") then
  ; if attaching fails, try opening the form
  if NOT siteForm.open("sitenote.fsl") then
    msgStop("Failed", "Couldn't open Sitenote.")
    return     ; if open fails, give up
  endif
endif

; if we reach this point, we have a good form handle
switch
  case isMaximized()  :                    ; if forms are maximized
    msgInfo("Status", "Siteform is maximized.")
    siteForm.show()                        ; restore size
  case siteForm.isMinimized() :            ; if form is minimized
    msgInfo("Status", "Siteform is minimized.")
    siteForm.maximize()
  case NOT (siteForm.isMaximized() OR siteForm.isMinimized()):
    msgInfo("Status", "Siteform is neither minimized or maximized.")
    siteForm.minimize()                             ; minimize
  otherwise :
    msgStop("Stop", "Unable to change size of Siteform.")
endswitch
endMethod
```

■

isMinimized method/procedure

See also Example Form Type

Reports whether a window is displayed as an icon.

Syntax

```
isMinimized ( ) Logical
```

Description

isMinimized returns True if a form is displayed as an icon; otherwise, it returns False.

▪

isMinimized example

See the example for **isMaximized.**

▪

isToolbarShowing procedure

See also Example Form Type

Reports whether the standard Toolbar is visible.

Syntax

`isToolbarShowing ( ) Logical`

Description

isToolbarShowing returns True if the standard Toolbar is visible; otherwise, it returns False.

▪

isToolbarShowing example

See the example for **hideToolbar**.

▪

isVisible method/procedure

See also Example Form Type

Reports whether any part of a window is displayed.

Syntax

```
isVisible ( ) Logical
```

Description

isVisible returns True if any part of a window is displayed (not hidden); otherwise, it returns False.

isVisible example

In the following example, the **pushButton** method for the *siteToTop* button attempts to attach to an open form. If the attach is successful, the method checks to see if the form is visible. If the form is visible, the method makes it the topmost window:

```
; siteToTop::pushButton
method pushButton(var eventInfo Event)
var
  siteForm Form
endVar
; if form is on desktop
if siteForm.attach("Form : SITENOTE.FSL") then
  if siteForm.isVisible() then     ; if form is visible
    siteForm.bringToTop()          ; make it the topmost layer
  else
    msgStop("Sorry", "Can't see Sitenote form.")
  endif
endif
endMethod
```

▪

keyChar method

See also Example Form Type

Sends an event to a form's **keyChar** method.

Syntax

```
1. keyChar ( const aChar SmallInt, const vChar SmallInt, const state SmallInt
) Logical
2. keyChar ( const characters String [ , const state SmallInt ] ) Logical
```

Description

keyChar sends an event to a form's **keyChar** method. For syntax 1, you must specify the ANSI character code in *aChar*, the virtual key code in *vChar*, and the keyboard state in *state* (using KeyboardStates constants). For syntax 2, you can specify a string of one or more characters and, optionally, use KeyBoardStates constants to specify a keyboard state.

▪

keyChar example

In the following example, a form named *Otherfrm* is already open, and it contains one field named *fieldOne*. The form-level keyChar method for *Otherfrm* echoes characters to *fieldOne*. The **pushButton** method of a button named *callOtherKeyC* on the current form attaches to *Otherfrm* as *otherForm* and calls the keyChar method for *otherForm*, passing it a string. This is the code for the **pushButton** method for *callOtherKeyC* on the current form:

```
; callOtherKeyC::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; attach to the other form (assumes it's open)
if otherForm.attach("Form : OTHERFRM.FSL") then
  otherForm.keyChar("Hi! ")    ; send a string
else
  msgStop("Error", "The other form is not available.")
endif
endMethod
```

This code is attached to *Otherfrm*'s form-level **keyChar** method:

```
; thisForm::keyChar  (OTHERFRM.FSL)
method keyChar(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; send the key on to fieldOne
    msgInfo("Status", "Executing Otherfrm's keychar.")
    fieldOne.keyChar(eventInfo.char())
endif
endMethod
```

▪

keyPhysical method

See also Example Form Type

Sends an event to a form's **keyPhysical** method.

Syntax

`keyPhysical ( const aChar SmallInt, const vChar SmallInt, const state SmallInt ) Logical`

Description

keyPhysical sends an event to a form's **keyPhysical** method. You must specify the ANSI character code in *aChar*, the virtual key code in *vChar*, and the keyboard state in *state* (using KeyboardStates constants).

▪

keyPhysical example

In the following example, a form named *OtherFr2* is already open, and it contains one field named *fieldOneThere*. The form-level **keyPhysical** method for *Otherfrm* echoes characters to *fieldOneThere*. The **keyPhysical** method of a field named *fieldOneHere* on the current form attaches to *Otherfrm* as *otherForm*. The method then calls the **keyPhysical** method for *otherForm*, passing it the ANSI code of the character or keypress, the virtual ANSI code of the character or keypress, and the keyboard state. This is the code for the **keyPhysical** method for *fieldOneHere* on the current form:

```
; fieldOneHere::keyPhysical    (current form)
method keyPhysical(var eventInfo KeyEvent)
var
  otherForm Form
endVar
; attach to the other form (assumes it's open)
if otherForm.attach("Form : OTHERFR2.FSL") then
  ; switch statement sorts out keyBoardState
  switch
    case eventInfo.isShiftKeyDown() :
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharCode(), Shift)
    case eventInfo.isAltKeyDown() :
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharCode(),
                            Alt)
    case eventInfo.isControlKeyDown() :
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharCode(),
                            Control)
    otherwise:
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharCode(),
                            0)

  endSwitch
else
  msgStop("Error", "The other form is not available.")
endif
endMethod
```

The following code is attached to the **keyPhysical** method for *otherForm*:

```
; thisForm::keyPhysical   (OTHERFRM)
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    ; pass keyPhysical on to fieldOneThere
    ; switch statement sorts out keyBoardState
    switch
      case eventInfo.isShiftKeyDown() :
        fieldOneThere.keyPhysical(eventInfo.charAnsiCode(),
                                  eventInfo.vCharCode(), Shift)
      case eventInfo.isAltKeyDown() :
        fieldOneThere.keyPhysical(eventInfo.charAnsiCode(),
                                  eventInfo.vCharCode(), Alt)
      case eventInfo.isControlKeyDown() :
        fieldOneThere.keyPhysical(eventInfo.charAnsiCode(),
                                  eventInfo.vCharCode(), Control)
      otherwise :
        fieldOneThere.keyPhysical(eventInfo.charAnsiCode(),
                                  eventInfo.vCharCode(), 0)
    endSwitch
endif
endMethod
```

▪

load method

See also Example Form Type

Opens a form in the Form Design window.

Syntax

`load ( const formName String, [const windowStyle LongInt [ , const x LongInt, const y LongInt, const w LongInt, const h LongInt ] ] ) Logical`

Description

load opens *formName* in the Form Design window. You have the option to specify in *windowStyle* a WindowStyles constant (or combination of constants). You also have the option to specify (in twips) the window's size and position: arguments *x* and *y* specify the position of the upper left corner, arguments *w* and *h* specify the width and height, respectively. Both of these options were added in version 5.0. This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL).

Compare this method to **open**, which opens a form in the Form window. To switch from the Form Design window to the Form window, use **run.** To switch from the Form window to the Form Design window, use **design.**

In either the Form Design window or the Form window, you can use UIObject type methods **create** and **methodSet** to place objects in the new form and attach methods to them. However, if you create objects while the form is in the Form window, the newly created objects will not automatically be saved when the form is closed.

Note: Some form actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** procedure in the System type for more information.

▪

load example

In the following example, the **pushButton** method for a button named *drawABox* loads the *Sitenote* form in a Form Design window. The method then sets the position of the form, creates a small box, names the box *newBox*, and sets its color to Blue. In the Form window, the box won't be visible; by default, the Visible property of objects created in this manner is False.

```
; drawABox::pushButton
method pushButton(var eventInfo Event)
var
   myForm Form
   newObj UIObject
endVar
; open Sitenote in a Form Design window
if myForm.load("Sitenote.fsl") then
   myForm.setPosition(720, 720, 1440*6, 1440*5)  ; 6" by 5"
   newObj.create(BoxTool, 1440, 1440*3, 360, 360, myForm)
   newObj.name = "newBox"
   newObj.color = Blue
else
   msgStop("Stop", "Couldn't load the form.")
endIf
endMethod
```

■

maximize method/procedure

See also Example Form Type
Beginner

Maximizes a window.

Syntax

```
maximize ( )
```

Description

maximize displays a window at its full size. Calling this method is equivalent to choosing Maximize from the Control menu.

▪

maximize example

In the following example, the **pushButton** method for the *goSites* button opens the *Sitenote* form (assumed to be in the current database), minimizes the current form, then waits for a response. If *Sitenote* returns "OK", this method maximizes the current form; otherwise, it restores the current form to its previous size:

```
; goSites::pushButton
method pushButton(var eventInfo Event)
var
  siteForm     Form
  returnString String
endVar
; open the Sitenote form, minimize self (this form), then wait
siteForm.open("Sitenote")
minimize()
returnString = String(siteForm.wait())
; if siteForm returned "OK", then maximize--otherwise, restore
if returnString = "OK" then
  maximize()
  siteForm.close()
else
  show()
  siteForm.close()
endif
endMethod
```

This code is attached to a button named *OKButton* on *Sitenote*:

```
; OKButton::pushButton
method pushButton(var eventInfo Event)
formReturn("OK")    ; return the string "OK" to the calling form
endMethod
```

▪

menuAction method/procedure

See also Example Form Type

Sends an event to a form's **menuAction** method.

Syntax

`menuAction` `(` `const` ***`action`*** `SmallInt` **`)`** `Logical`

Description

menuAction constructs a MenuEvent and calls a specified form's **menuAction** method. *action* is one of the MenuCommand constants, or a user-defined menu constant.

Note: You can't use **menuAction** to send a menu command constant that is equivalent to a File|New menu choice (for example, File|New|Form) or a File|Open menu choice (for example, File|Open|Table). To simulate these choices, call the appropriate ObjectPAL method (for example, Form::**create** or TableView::**open**). This feature was changed in version 5.0; previous versions were more restrictive.

▪

menuAction example

In the following example, the *sendATile* button on the current form opens the *Sitenote* form and sends it a MenuWindowTile action.

```
; sendATile::pushButton
method pushButton(var eventInfo Event)
var
  siteForm  Form
endVar
if siteForm.open("Sitenote.fsl") then
  siteForm.menuAction(MenuWindowTile)
endif
endMethod
```

▪

methodDelete method

See also Example Form Type

Deletes a form-level method from a form.

Syntax

`methodDelete ( const methodName String ) Logical`

Description

methodDelete deletes a built-in or custom method specified in *methodName* from a form. You can also specify "Var", "Proc", "Uses", or "Const" in *methodName* to clear the Var, Proc, Uses, or Const window of a form. If *methodName* is a built-in event method, the built-in behavior for that method is restored.

This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL).

methodDelete example

In the following example, two forms are on the desktop in a Form Design window: *Otherone* and *Othertwo*. The **pushButton** method for a button named *moveMethod* (on the current form) moves a method from *Otherone* to *Othertwo*:

```
; moveMethod::pushButton
method pushButton(var eventInfo Event)
var
  tempFormSrc,
  tempFormDest    Form
  transMethod String
endVar
; try to attach to both the source and the destination form
; assume source and destination are on the desktop in a Form Design window
if tempFormSrc.attach("Form Design : OTHERONE.FSL") AND
   tempFormDest.attach("Form Design : OTHERTWO.FSL") then
  ; get definition for source form's mouseRightUp, then delete
  transMethod = tempFormSrc.methodGet("mouseRightUp")
  tempFormSrc.methodDelete("mouseRightUp")
  ; copy the method to the destination form mouseRightUp
  tempFormDest.methodSet("mouseRightUp", transMethod)
else
  msgStop("Error", "Couldn't attach to source and destination forms.")
endif
endMethod
```

▪

methodGet method

See also Example Form Type

Gets a form-level method.

Syntax

`methodGet (const methodName String ) String`

Description

methodGet gets the text of the built-in or custom form-level method specified in *methodName* attached to a form. You can also specify "Var", "Const", "Uses", or "Proc" to get the contents of the Var, Const, Uses, or Proc window of a form.

This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL).

-

methodGet example

See the example for **methodDelete**.

▪

methodSet method

See also Example Form Type

Sets the definition of a method attached to a form.

Syntax

```
methodSet (const methodName String, const methodText String ) Logical
```

Description

methodSet writes the text in *methodText* to the built-in or custom form-level method *methodName*, overwriting any existing method definition. You can also specify "Var", "Const", "Uses", or "Proc" to set the contents of the Var, Const, Uses, or Proc window of a form.

This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL).

methodSet example

See the example for **methodDelete**.

▪

minimize method/procedure

See also   Example   Form Type

Minimizes a window.

Syntax

```
minimize ( )
```

Description

minimize displays a window as an icon. Calling this method is equivalent to choosing Minimize from the Control menu.

▪

minimize example

See the example for **maximize**.

▪

mouseDouble method

See also Example Form Type

Sends an event to a form's **mouseDouble** method.

Syntax

```
mouseDouble ( const x LongInt, const y LongInt, const state SmallInt )
Logical
```

Description

mouseDouble constructs a MouseEvent and sends it to a form's **mouseDouble** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

▪

mouseDouble example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseDouble* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseDouble** method for *otherForm*:

```
; sendMouseDouble::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseDouble to target form at coordinates 1000, 1000
  otherForm.mouseDouble(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseDouble** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseDouble  (OTHERMSE)
method mouseDouble(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseDouble"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

▪

mouseDown method

See also Example Form Type

Sends an event to a form's **mouseDown** method.

Syntax

```
mouseDown ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseDown constructs an event and sends it to a form's **mouseDown** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

▪

mouseDown example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseDown* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseDown** method for *otherForm*:

```
; sendMouseDown::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseDown to target form at coordinates 1000, 1000
  otherForm.mouseDown(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseDown** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseDown  (OTHERMSE)
method mouseDown(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseDown"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

▪

mouseEnter method

See also Example Form Type

Sends an event to a form's **mouseEnter** method.

Syntax

```
mouseEnter ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseEnter constructs a MouseEvent and sends it to a form's **mouseEnter** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

▪

mouseEnter example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseEnter* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseEnter** method for *otherForm*:

```
; sendMouseEnter::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseEnter to target form at coordinates 1000, 1000
  otherForm.mouseEnter (1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseEnter** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseEnter  (Othermse)
method mouseEnter(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseEnter"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

▪

mouseExit method

See also Example Form Type

Sends an event to a form's **mouseExit** method.

Syntax

```
mouseExit ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseExit constructs a MouseEvent and sends it to a form's **mouseExit** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

▪

mouseExit example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseExit* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseExit** method for *otherForm*:

```
; sendMouseExit::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseExit to target form at coordinates 1000, 1000
  otherForm.mouseExit(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseExit** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseExit  (Othermse)
method mouseExit(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseExit"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

▪

mouseMove method

See also Example Form Type

Sends an event to a form's **mouseMove** method.

Syntax

mouseMove **(** const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt **)** Logical

Description

mouseMove constructs an event and sends it to a form's **mouseMove** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

■

mouseMove example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseMove* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseMove** method for *otherForm*:

```
; sendMouseMove::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseMove to target form at coordinates 1000, 1000
  otherForm.mouseMove(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseMove** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseMove  (Othermse)
method mouseMove(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseMove"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

▪

mouseRightDouble method

See also Example Form Type

Sends an event to a form's **mouseRightDouble** method.

Syntax

```
mouseRightDouble (const x LongInt, const y LongInt, const state SmallInt )
Logical
```

Description

mouseRightDouble constructs a MouseEvent and sends it to a form's **mouseRightDouble** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

▪

mouseRightDouble example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named send*MouseRightDouble* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseRightDouble** method for *otherForm*:

```
; mouseRightDouble::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseRightDouble to target form at coordinates 1000, 1000
  otherForm.mouseRightDouble(1000, 1000, RightButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseRightDouble** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseRightDouble  (Othermse)
method mouseRightDouble(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseRightDouble"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

mouseRightDown method

See also Example Form Type

Sends an event to a form's **mouseRightDown** method.

Syntax

```
mouseRightDown ( const x LongInt, const y LongInt, const state SmallInt )
Logical
```

Description

mouseRightDown constructs a MouseEvent and sends it to a form's **mouseRightDown** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

▪

mouseRightDown example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseRightDown* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseRightDown** method for *otherForm*:

```
; mouseRightDown::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseRightDown to target form at coordinates 1000, 1000
  otherForm.mouseRightDown(1000, 1000, RightButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseRightDown** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseRightDown  (Othermse)
method mouseRightDown(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseRightDown"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

▪

mouseRightUp method

See also Example Form Type

Sends an event to a form's **mouseRightUp** method.

Syntax

mouseRightUp (**const ***x* LongInt, const ***y*** LongInt, const ***state*** SmallInt **)**
Logical

Description

mouseRightUp constructs a MouseEvent and sends it to a form's **mouseRightUp** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

▪

mouseRightUp example

In the following example, assume the form *Othermse* is already open. The **pushButton** method for a button named *sendMouseRightUp* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseRightUp** method for *otherForm*:

```
; mouseRightUp::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseRightUp to target form at coordinates 1000, 1000
  otherForm.mouseRightUp(1000, 1000, RightButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseRightUp** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseRightUp  (Othermse)
method mouseRightUp(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseRightUp"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

▪

mouseUp method

See also Example Form Type

Sends an event to a form's **mouseUp** method.

Syntax

`mouseUp ( const x LongInt, const y LongInt, const state SmallInt ) Logical`

Description

mouseUp constructs a MouseEvent and sends it to a form's **mouseUp** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state using KeyBoardStates constants.

▪

mouseUp example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseUp* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseUp** method for *otherForm*:

```
; sendMouseUp::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseUp to target form at coordinates 1000, 1000
  otherForm.mouseUp(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endMethod
```

This code is attached to the **mouseUp** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseUp  (Othermse)
method mouseUp(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseUp"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endMethod
```

▪

moveTo method

See also Example Form Type

Moves to a form.

Syntax

```
moveTo ( [const objectName String] ) Logical
```

Description

moveTo moves the focus to a form. Optionally, it moves to the object specified in *objectName*.

moveTo example

In the following example, a form named *Sitenote* is already open in the Form window on the desktop. the **pushButton** method for the *goToSites* button in the current form attaches the variable *otherForm* to *Sitenote*, determines if *otherForm* is visible, and, if so, moves to *otherForm*. if *otherForm* is not visible, the method uses **show** to display the form at its default size (**show** also moves the focus to the target form):

```
; goToSites::pushButton
method pushButton(var eventInfo Event)
var
  otherForm  Form
endVar
; assume that Sitenote form is already open
if otherForm.attach("Form : SITENOTE.FSL") then
  if otherForm.isVisible() then
    otherForm.moveTo()       ; if form is visible, move to it
  else
    otherForm.show()         ; otherwise, make it visible
  endif
else
  msgStop("Stop", "Couldn't find form.")
endif
endMethod
```

moveToPage method/procedure

See also Example Form Type

Displays a specified page of a form.

Syntax

```
moveToPage ( const pageNumber SmallInt ) Logical
```

Description

moveToPage displays the page of a form specified in *pageNumber*. *pageNumber* can be an integer variable or an integer constant, but it can't be an object ID. To move to a page by its object ID, use the **moveTo** method from the UIObject type.

▪

moveToPage example

In the following example, the current form has two pages. The *Sitenote* form exists in the working directory and has four pages. The **pushButton** method for *pageThruSites* (on the current form) first moves to the second page of the current form. Then the method opens the *Sitenote* form to the *otherForm* variable, and pages through *otherForm*:

```
; pageThruSites::pushButton
method pushButton(var eventInfo Event)
const
   BillingInfo = SmallInt(4)
endConst
var
   myForm, otherForm  Form
   somePage           SmallInt
endVar
moveToPage(2)                       ; moves to page 2 on this form
if otherForm.open("Sitenote.fsl") then  ; opens to first page
  sleep(2000)                       ; pause
  otherForm.moveToPage(2)           ; moves to page 2 of SiteNote
  sleep(2000)
  somePage = 3
  otherForm.moveToPage(somePage)    ; moves to page 3
  sleep(2000)
  otherForm.moveToPage(BillingInfo) ; moves to page 4
  sleep(2000)
endIf
endMethod
```

▪

open method

See also Example1 Example2 Form Type
Beginner

Opens a window.

Syntax

```
1. open ( const formName String [ , const windowStyle LongInt ] ) Logical
2. open ( const formName String, const windowStyle LongInt, const x SmallInt,
const y SmallInt, const w SmallInt, const h SmallInt ) Logical
3. open ( const openInfo FormOpenInfo ) Logical
```

Description

open displays the form specified in *formName*. The form is opened in a Form window. The optional arguments *x* and *y* specify the location of the upper left corner of the form (in twips), *w* and *h* specify the width and height (in twips), and *windowStyle* specifies display attributes using WindowStyles constants. You can specify more than one window style element by adding the constants together. For example, the following code opens a form and specifies both vertical and horizontal scroll bars:

```
theForm.open("sales", WinStyleDefault + WinStyleVScroll + WinStyleHScroll)
```

Compare this method with **load,** which opens a form in a Form Design window.

Syntax 3 lets you specify form settings from *openInfo*, a record of type FormOpenInfo. The predefined FormOpenInfo record, an instance of the Record Type, has the following structure:

```
x, y, w, h     LongInt ;position and size of the form
name           String  ;name of form to open
masterTable    String  ;new master table name
queryString    String  ;query to run (actual query string)
winStyle       LongInt ;window style constant(s)
```

You can use the *masterTable* member to specify a different master table for the form (this is similar to choosing a different table for a form when you open the form from the Open Form dialog box). Alternatively, you can specify a query string in the *queryString* member. Paradox executes the query and opens the form; the result of the query is the master table.

Paradox opens saved forms before delivered forms with the same name. For example, suppose the working directory contains ORDERS.FSL (a saved form) and ORDERS.FDL (a delivered form). The following statement opens the saved form, ORDERS.FSL.

```
ordersForm.open("ORDERS") ; Opens :WORK:ORDERS.FSL.
```

To specify a delivered form, include the .FDL extension. For example,

```
ordersForm.open("ORDERS.FDL") ; Opens the delivered form.
```

Note: Some form actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** procedure in the System type for more information.

▪

open example 1

In the following example, the **keyPhysical** method for a field named *fieldOne* tests all key events. When the user presses F1, the form HELPFORM opens. The **keyPhysical** method opens a form from the current directory:

```
; fieldOne::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  helpForm Form
endVar
message(eventInfo.vChar())
if eventInfo.vChar() = "VK_F1" then
  helpForm.open("helpform", WinStyleDefault,
                 720, 720, 1440 * 2, 1440 * 4)
  disableDefault
endIf

endMethod
```

■

open example 2

The following example works like the previous example, except that it uses a FormOpenInfo record to set the characteristics of the form to be opened.

```
; fieldOne::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  openHelpForm FormOpenInfo   ; a predeclared record type
  helpForm     Form
endVar
message(eventInfo.vChar())
if eventInfo.vChar() = "VK_F1" then
  openHelpForm.x = 720
  openHelpForm.y = 720
  openHelpForm.w = 2 * 1440
  openHelpForm.h = 4 * 1440
  openHelpForm.name = "helpform"
  helpForm.open(openHelpForm)
  disableDefault
endIf
endMethod
```

■

openAsDialog method

See also Example Form Type

Opens a Form window as a dialog box.

Syntax

```
1. openAsDialog ( const formName [ , const windowStyle LongInt] ) Logical
2. openAsDialog ( const formName String, const windowStyle LongInt, const x SmallInt, const y SmallInt, const w SmallInt, const h SmallInt ) Logical
3. openAsDialog ( const openInfo FormOpenInfo ) Logical
```

Description

openAsDialog opens the form *formName* and displays it on top of any other open windows. The form is in the Form window. *formName* is always on top, whether it's active or not. The optional arguments *x* and *y* specify the upper left corner of the window (in twips), *w* and *h* specify the width and height (in twips), and *windowStyle* specifies display attributes using WindowStyles constants. You can specify more than one window style element by adding the constants. For example, the following code opens a form and specifies both vertical and horizontal scroll bars:

```
theForm.openAsDialog("sales", WinStyleDefault + WinStyleVScroll +
WinStyleHScroll)
```

Syntax 3 lets you specify form settings from *openInfo*, a record of type FormOpenInfo. The FormOpenInfo record type is predeclared and has the following structure:

```
x, y, w, h       LongInt  ; position and size of the form
name             String   ; name of form to open
masterTable      String   ; master table name
queryString      String   ; run this query
```

▪

openAsDialog example

In the following example, the **keyPhysical** method for a field named *fieldOne* tests all key events. When the user presses F1, the form HELPFORM opens. The **keyPhysical** method opens a form as a dialog box:

```
; fieldOne::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  helpForm Form
endVar
; if user presses F1, open a help dialog box
if eventInfo.vChar() = "VK_F1" then

  helpForm.openAsDialog("helpform", WinStyleDefault,
                        720, 720, 1440 * 4, 1440 * 3)

  helpForm.setTitle("Application Help")
  helpForm.wait()
  helpForm.close()
  disableDefault                        ; don't call Help system
endIf
endMethod
```

▪

postAction method

See also   Example   Form Type

Posts an action to an action queue for delayed execution.

Syntax

```
postAction ( const actionId SmallInt )
```

Description

postAction works like **action**, except that the action is not executed immediately. Instead, the action specified by *actionID* is posted to an action queue at the time of the method call; Paradox waits until a yield occurs (for example, by the current method completing execution or by a call to **sleep**).

The value of *actionID* can be a user-defined action constant or a constant from one of the following Action classes:

ActionDataCommands

ActionEditCommands

ActionFieldCommands

ActionMoveCommands

ActionSelectCommands

postAction example

In the following example, the **pushButton** method for *openSitesNew* opens the *Sitenote* form to the variable *otherForm*. The method then posts three actions to *otherForm*, and displays a message in a dialog box. The actions specified by **postAction** occur when Paradox yields:

```
; openSitesNew::pushButton
method pushButton(var eventInfo Event)
; otherForm variable is global to form--stays in scope after method ends
if otherForm.open("Sitenote.fsl") then
  ; these actions will not execute until after this method ends
  otherForm.postAction(DataEnd)          ; move to the last record
  otherForm.postAction(DataBeginEdit)    ; start Edit mode
  otherForm.postAction(DataInsertRecord) ; insert a new blank record
  msgInfo("Status", "About to perform posted actions. Watch closely.")
else
  msgStop("Stopped", "Could not open form.")
endif
endMethod
```

▪

run method

See also　　Example　　Form Type

Switches a form from the Form Design window to the Form window.

Syntax

`run ( ) Logical`

Description

run switches a form from the Form Design window to the Form window. This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL). For more information about saving and delivering forms, refer to the *Guide to ObjectPAL*.

To switch from the Form window to the Form Design window, use **design**.

Note: Some form actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

run example

The following example opens the *Sitenote* form in a Form Design window, deletes the **pushButton** method from the form, then runs the form. Assume that the *Sitenote* form is in the current directory. This code is attached to the **pushButton** code for *delPushButton*:

```
; delPushButton::pushButton
method pushButton(var eventInfo Event)
var
  otherForm  Form
endVar
; load the Sitenote form, delete the pushButton
; method, then run the form
if otherForm.load("Sitenote") then
  otherForm.methodDelete("pushButton")
  otherForm.run()
endif               ; won't be permanent
endMethod
```

▪

save method

See also Example Form Type

Saves a form to disk.

Syntax

```
save ( [ const newFormName String ] ) Logical
```

Description

save writes a form to disk in the user's current working directory. This method works only when the form is in a Form Design window.

The *newFormName* argument specifies the name for the form. If the form already has a name, Paradox saves it using that name. If you omit *newFormName* and the form doesn't have a name already, this method returns an error.

▪

save example

See the example for **create**.

saveStyleSheet method

See also Example Form Type

Saves a style sheet.

Syntax

```
saveStyleSheet ( const fileName String, const overWrite Logical ) Logical
```

Description

saveStyleSheet saves a style sheet to the file specified in *fileName*. If *fileName* does not specify a full path for the style sheet file, this method saves it to :WORK:.

The value of *overWrite* specifies what to do if the file already exists. If *overWrite* is True and the file exists, Paradox overwrites the file without asking for confirmation. If *overWrite* is False and the file exists, the file is not saved. This method returns True if it saves the file; otherwise, it returns False.

saveStyleSheet saves the form's current style sheet, including any changes made interactively or using ObjectPAL. If called as a method, **saveStyleSheet** operates on the specified form. If called as a procedure, it operates on the current form. It returns True if it succeeds; otherwise, it returns False.

saveStyleSheet example

This example sets the frame style of field objects and text objects, then saves the form's style sheet to a file named IN3DFRAM.FT. If the file exists, it is overwritten.

```
const
   kOverWrite = Yes
endConst

method mouseClick(var eventInfo MouseEvent)
   var
      thisForm Form
   endVar

   thisForm.attach()
   thisForm.setProtoProperty(FieldTool, "Frame.Style", Inside3DFrame)
   thisForm.setProtoProperty(TextTool, "Frame.Style", Inside3DFrame)
   thisForm.saveStyleSheet("in3dfram.ft", kOverWrite)
endmethod
```

▪

selectCurrentTool method

See also Example Form Type

Specifies a Toolbar tool to use.

Syntax

```
selectCurrentTool ( objType SmallInt ) Logical
```

Description

selectCurrentTool specifies a Toolbar tool to use, where *objType* is one of the UIObjectTypes constants. When used with a form in the Form Design window, this method makes the specified tool active and sets the mouse shape accordingly.

selectCurrentTool example

This example creates a form and sets the current tool to the Field tool.

```
method pushButton(var eventInfo Event)
   var
      foTest   Form
   endVar

   foTest.create()
   foTest.selectCurrentTool(FieldTool)
   msgInfo("Next step:",
          "Click and drag to draw a field object.")
endMethod
```

▪

setCompileWithDebug method

See also	Example	Form Type

Sets compile with debug.

Syntax

`setCompileWithDebug ( const YesNo Logical ) Logical`

Description

setCompileWithDebug sets the compile with debug flag to true or false. This is the same as setting compile with debug interactively in form design. **setCompileWithDebug** returns True if successful and returns False if unsuccessful.

▪

setCompileWithDebug example

In the following example, the central form of a management system has two buttons: getCompileStatus and setCompileStatus. The pushButton method of each button opens the Windows 95 file browser dialog to allow a user to select the file which will be examined/manipulated. Each method analyzes the fileName selected to determine which fileType and then opens it under the appropriate object type.

The following code is attached to the pushButton method for setCompileStatus:

```
; setCompileStatus::pushButton
method pushButton(var eventInfo Event)
var
   theForm       Form             ;// Object variable for forms
   theLibrary    Library          ;// Object variable for forms
   theScript     Script           ;// Object variable for forms
   fbi           FileBrowserInfo  ;// File Browser information structure
   selectedFile  String           ;// FileName selected by user
   fileType      String           ;// File type of file
                                  ;// selected by user
   status        Logical          ;// Debug status of the selected file
   toggle        String           ;// User choice for
endVar

      ;//Set allowable file types: Forms, Libraries, and Scripts
   fbi.AllowableTypes = fbForm + fbLibrary + fbScript
   if fileBrowser(selectedFile, fbi) then
      ;// The user selected a file
      fileType = upper(substr(selectedFile, selectedFile.size() - 2, 3))
      switch
         case fileType = FSL :
            ;// Load the Form
            theform.load(fbi.Drive + fbi.Path + selectedFile)
            ;// Determine its status
            status = theForm.isCompileWithDebug()
            toggle = msgYesNoCancel (Select a choice, selectedFile
                                 + iif(status,  is ,  is not ) +
                       "compiled with Debug information - toggle?")
            switch
               case toggle = Yes :
                  ;// Toggle status
                  theForm.setCompileWithDebug(NOT(status))
                  ;// Save the change
                  theForm.save()
                  msgInfo(User Notification,
                          Toggle of Debug State Completed.)
               case toggle = No or toggle = Cancel :
                  msgInfo(User Notification,
                          Toggle of Debug State Canceled.)
            endSwitch
            ;// Close the Form
            theForm.close()

         case fileType = "LSL" :
            ;// Load the Library
            theLibrary.load(fbi.Drive + fbi.Path + selectedFile)
            ;// Determine its status
            status = theLibrary.isCompileWithDebug()
            toggle = msgYesNoCancel (Select a choice, selectedFile
                                 + iif(status,  is ,  is not )
                       + "compiled with Debug information - toggle?")
            switch
               case toggle = Yes :
                  ;// Toggle status
                  theLibrary.setCompileWithDebug(NOT(status))
                  ;// Save the change
                  theLibrary.save()
```

```
                  msgInfo(User Notification,
                          Toggle of Debug State Completed.)
               case toggle = No or toggle = Cancel :
                  msgInfo(User Notification,
                          Toggle of Debug State Canceled.)
            endSwitch
            ;// Close the Library
            theLibrary.close()

         case fileType = "SSL" :
            ;// Load the Script
            theScript.load(fbi.Drive + fbi.Path + selectedFile)
            ;// Determine its status
            status = theScript.isCompileWithDebug()
            toggle = msgYesNoCancel (Select a choice, selectedFile
                                  + iif(status,  is ,  is not )
                           + "compiled with Debug information - toggle?")
            switch
               case toggle = Yes :
                  ;// Toggle status
                  theScript.setCompileWithDebug(NOT(status))
                  ;// Save the change
                  theScript.save()
                  msgInfo(User Notification,
                          Toggle of Debug State Completed.)
               case toggle = No or toggle = Cancel :
                  msgInfo(User Notification,
                          Toggle of Debug State Canceled.)
            endSwitch
            ;// Close the Script
            theScript.close()
      endSwitch
      ;// Inform the user
      msgInfo(selectedFile
            + " compiled with Debug information?",
                 status)
   else
      ;// The user didnt select a file
      msgInfo("No file selected", "Please try again")
   endIf
endMethod
```

▪

setIcon method/procedure

See also Example Form Type

Specifies the icon to be used with a form, report, or desktop.

Syntax

`setIcon ( ``const `***`fileName`***` String `**`)`**` Logical`

Description

setIcon specifies the icon to be used with a form, report, or desktop. The file specified with *fileName* must be a valid icon file and the file's name must have an extension of .ICO. **setIcon** returns True if successful and returns False if unsuccessful.

After you set the icon for a form, all the forms on the desktop will change to the new icon and any form opened will be set to the new icon.

setIcon example

The following example sets the file, DOCFILE.ICO as the icon.

```
method init ( var eventInfo Event )
   setIcon ( "i:\\resource\\docfile.ico" )
endMethod
```

▪

setMenu method

See also Example Form Type

Associates a menu with a form.

Syntax

```
setMenu ( const menuVar Menu )
```

Description

setMenu associates the menu specified in *menuVar* with a form. This method performs the same function as Menu::**show**, and adds the following features:

- When the form gets focus, Paradox displays the associated menu.
- Actions resulting from choices from that menu are sent to that form.

setMenu example

The following example is a script. It opens a form, builds a simple menu, then uses **setMenu** to assign the menu to the form.

```
method run(var eventInfo Event)
   var
      foOrders   Form
      muOrderForm   Menu
      puFormFile   PopUpMenu
   endVar

; Build a menu for the form.
   foOrders.open("orders")

; Setting the StandardMenu property to False
; (either in ObjectPAL code or interactively)
; can reduce flicker when changing menus.
   foOrders.StandardMenu = False

   puFormFile.addText("&New Form", MenuEnabled, MenuFormNew)
   puFormFile.addText("&Open Form", MenuEnabled, MenuFormOpen)
   puFormFile.addText("&Exit", MenuEnabled, MenuFileExit)

   muOrderForm.addPopUp("&File", puFormFile)

   foOrders.setMenu(muOrderForm)

endMethod
```

▪

setPosition method/procedure

See also		Example		Form Type

Positions a window onscreen.

Syntax

```
setPosition ( const x LongInt, const y LongInt, const w LongInt, const h
LongInt )
```

Description

setPosition positions a window onscreen. The arguments *x* and *y* specify the coordinates of the upper left corner of the form (in twips), and *w* and *h* specify the width and height (in twips).

To ObjectPAL, the screen is a two-dimensional grid, with the origin (0, 0) at the upper left corner of an object's container, positive x-values extending to the right, and positive y-values extending down.

For dialog boxes and for the Paradox Desktop application, the position is given relative to the entire screen; for forms, reports, and table windows, the position is given relative to the Paradox Desktop.

▪

setPosition example

See the example for **getPosition**.

▪

setProtoProperty method/procedure

See also Example Form Type

Sets the value of a specified property of a prototype object.

Syntax

`setProtoProperty ( ``const `***`objectType`***` SmallInt, `***`propertyName`***` String, `***`value`***
`AnyType `**`)`**` Logical`

Description

setProtoProperty sets the property specified in *propertyName* of the prototype object specified in *objectType* to the value specified in *value*. To specify *objectType*, use one of the UIObjectTypes constants. If called as a method, **setProtoProperty** operates on prototype objects in the style sheet of the specified form. If called as a procedure, it uses the style sheet of the current form.

Changes to the style sheet are not saved automatically. You must either save the style sheet interactively or call **saveStyleSheet**.

▪

setProtoProperty example

See the example for **saveStyleSheet.**

▪

setSelectedObjects method

See also Example Form Type

Selects specified objects in a form.

Syntax

```
setSelectedObjects ( objects Array[ ] UIObject, const yesNo Logical )
```

Description

setSelectedObjects selects specified objects in a form in a Form Design window as if you had selected them interactively. The array *objects* is an array of UIObjects (not the object names) to select. Use **attach** to assign a UIObject to an array.

The argument *yesNo* specifies whether to show selection handles: if *yesNo* is True, the selected objects have handles; otherwise, they do not.

setSelectedObjects example

The following example creates a form, then creates two boxes in it. Then it calls **setSelectedObjects** to select the boxes. Note that you must use **attach** assign a UIObject to an array.

```
method pushButton(var eventInfo Event)
   var
      foTemp       Form
      uiTemp       UIObject
      arObjects    Array[2] UIObject
   endVar

   const
      kOneInch = 1440 ; One inch = 1,440 twips.
      kShowHandles = Yes
   endConst

   foTemp.create()

   uiTemp.create(BoxTool, 300, 300, kOneInch, kOneInch, foTemp)
   uiTemp.Visible = Yes
   arObjects[1].attach(uiTemp)

   uiTemp.create(BoxTool, 300, 2200, kOneInch, kOneInch, foTemp)
   uiTemp.Visible = Yes
   arObjects[2].attach(uiTemp)

   foTemp.setSelectedObjects(arObjects, kShowHandles)
endMethod
```

▪

setStyleSheet method/procedure

See also Example Form Type

Specifies a form's style sheet.

Syntax

```
setStyleSheet ( const fileName String )
```

Description

setStyleSheet makes a form use the style sheet specified in *fileName*. If *fileName* does not specify a full path to the style sheet, this method searches for it in :WORK:. If called as a method, **setStyleSheet** operates on the specified form. If called as a procedure, it operates on the current form.

Any UIObjects created in the form while the style sheet is active will have the properties and methods of the corresponding prototype objects in the style sheet. **setStyleSheet** does not change the properties or methods of UIObjects that already exist. This method affects only the specified form; it does not affect the screen or printer style sheets. Use the System procedures **setDefaultScreenStyleSheet** and **setDefaultPrinterStyleSheet** for that.

setStyleSheet example

The following example opens a form, then calls **getStyleSheet** to see which style sheet the form is using. If the style sheet is not BORLAND.FT, the code calls **setStyleSheet** to set it, then calls **getStyleSheet** again to make sure it was set successfully. Note that **setStyleSheet** requires double backslashes in the path, but **getStyleSheet** returns single backslashes.

```
method pushButton(var eventInfo Event)
   var
      f Form
   endVar
   f.open("orders")
   ; Get and set the style sheet for this form.
   if f.getStyleSheet() <> "c:\\pdoxwin\\borland.ft" then
      f.setStyleSheet("c:\\pdoxwin\\borland.ft")
      if f.getStyleSheet() <> "c:\\pdoxwin\\borland.ft" then
         msgStop("Problem", "Could not set the style sheet.")
       endIf
   endIf
endMethod
```

setTitle method/procedure

See also Example Form Type

Sets the text in the window title bar.

Syntax

```
setTitle ( const text String )
```

Description

setTitle changes the text of the window title bar to the text specified in *text*. The maximum length of *text* is 78 characters. If you change a form's title, remember that you must use the new title when you want to attach to that form. (See the description of **attach** for more details.)

-

setTitle example

See the example for **openAsDialog**.

▪

show method/procedure

See also   Example   Form Type
Beginner

Displays a minimized window at its previous size; makes a hidden form visible.

Syntax
`show ( )`

Description
show makes a hidden form visible. show also restores a minimized window to the size before it was minimized. This method is similar to the Restore command on the Control menu.

show doesn't make a form the topmost window; use **bringToTop** to make a form the top layer and give it focus.

-

show example

See the example for **hide**.

▪

showToolbar procedure

See also Example Form Type

Makes the standard Toolbar visible.

Syntax

```
showToolbar ( )
```

Description

showToolbar displays the standard Toolbar.

▪

showToolbar example

See the example for **hideToolbar**.

▪

wait method

See also   Example   Form Type
Beginner

Suspends execution of a method.

Syntax

```
wait ( ) AnyType
```

Description

wait suspends execution of the current method until the form you're waiting for returns (see **formReturn**). This method is useful when you open a second form as a dialog box. Execution resumes in the first form when the second form (the one you're waiting for) calls **formReturn**, or when the second form closes. Once the called form returns, the calling form should close it with **close**. The called form does not automatically close, even if the user closes it; it stays open so that code on the calling form can examine it (for instance, to see settings on a dialog box).

▪

wait example

See the example for **formReturn**.

▪

windowClientHandle method/procedure

See also Example Form Type

Returns the handle of a window.

Syntax

`windowClientHandle ( ) LongInt`

Description

A window handle is a unique integer identifier assigned to a window by Windows. **windowClientHandle** returns an integer value representing the window handle of the client area of a form. When called as a procedure, it returns the window handle of the client area of the current form. This method should be used only by advanced programmers.

This information is useful only if you're using functions from a dynamic link library (DLL).

windowClientHandle example

In the following example, assume that a DLL called MYTEST.DLL exists and that it contains a function called *doSomething*. The *doSomething* function takes one argument, a window handle.

```
; someButton::pushButton
method pushButton(var eventInfo Event)
uses MYTEST
  doSomething(const wHandle CLONG)
endUses
doSomething(windowClientHandle())  ; call doSomething and supply the
                                   ; handle of the client portion
                                   ; of the current form
endMethod
```

▪

windowHandle method/procedure

See also Example Form Type

Returns the handle of a window.

Syntax

`windowHandle ( )` `LongInt`

Description

A window handle is a unique integer identifier assigned to a window by Windows. **windowHandle** returns an integer value representing the window handle of a form. When called as a procedure, it returns the window handle of the current form. This method should be used only by advanced programmers.

This information is useful only if you're using functions from a dynamic link library (DLL).

windowHandle example

In the following example, assume that a DLL called MYTEST.DLL exists and that it contains a function called *doSomething*. The *doSomething* function takes one argument, a window handle.

```
; someButton::pushButton
method pushButton(var eventInfo Event)
uses MYTEST
  doSomething(const wHandle CLONG)
endUses
doSomething(windowHandle())   ; call doSomething and supply the
                              ; window handle of the current form
endMethod
```

Graphic type

A Graphic variable provides a handle for manipulating a graphic object. That is, you can use Graphic variables in ObjectPAL code to manipulate graphic objects. Graphic objects contain and display graphics in bitmap format (BMP). However, Paradox can import the following graphic formats: bitmap (BMP), encapsulated Postscript (EPS), graphic interchange format (GIF), Paintbrush (PCX), and tagged information file format (TIF).

Using Graphic type methods **readFromClipboard**, **writeToClipboard**, **readFromFile,** and **writeToFile**, you can use Graphic variables to transfer bitmaps between forms (and reports), tables, the Clipboard, and disk files.

The Graphic type includes several derived methods from the AnyType type.

Methods for the Graphic type

AnyType	**Graphic**
blank	**readFromClipboard**
dataType	**readFromFile**
isAssigned	**writeToClipboard**
isBlank	**writeToFile**
isFixedType	

readFromClipboard method

See also Example Graphic Type

Reads a graphic from the Clipboard.

Syntax

```
readFromClipboard ( ) Logical
```

Description

readFromClipboard reads a graphic from the Clipboard to a variable of type Graphic. If the Clipboard contains a graphic that can be copied to the Graphic variable, **readFromClipboard** returns True. If the Clipboard is empty or does not contain a valid graphic, **readFromClipboard** returns False. **readFromClipboard** can read bitmap (BMP) and device independent bitmap (DIB) formats.

readFromClipboard example

In the following example, a form contains a multi-record object named *BIOLIFE* bound to the *Biolife* table, and a button named *getGraphic*. The **pushButton** method for *getGraphic* locates the record with a Common Name field value of "Firefish", then writes the contents of the Clipboard to that record's Graphic field. If the Clipboard is empty or does not contain a graphic, the **readFromClipboard** method returns False, and the value of the Graphic field is not changed.

```
; getGraphic::pushButton
method pushButton(var eventInfo Event)

var
  myGraphic Graphic
endVar

if BIOLIFE.locate("Common Name", "Firefish") then

  if myGraphic.readFromClipboard() then
    ; get the current clipboard contents to myGraphic
    BIOLIFE.edit()                 ; start Edit mode on the table
    BIOLIFE.Graphic = myGraphic    ; write the bitmap to the field
    BIOLIFE.endEdit()              ; end Edit mode
  endIf
endIf
endMethod
```

▪

readFromFile method

See also Example Graphic Type

Reads a graphic from a file.

Syntax

`readFromFile ( const fileName String ) Logical`

Description

readFromFile reads a graphic from a disk file specified in *fileName*. **readFromFile** returns True if the *fileName* name exists and contains a graphic format that can be imported; otherwise, it returns False. Paradox can import the following graphic formats:

{bullet.bmp} bitmap (BMP)

{bullet.bmp} encapsulated Postscript (EPS)

{bullet.bmp} graphic interchange format (GIF)

{bullet.bmp} Paintbrush (PCX)

{bullet.bmp} tagged information file format (TIF)

▪

readFromFile example

The following example assumes that a form contains a button named *getChess*, and an unbound graphic field named *bitmapField*. The **pushButton** method for *getChess* attempts to read the bitmap file CHESS.BMP from the C:\WINDOWS directory and stores CHESS.BMP in the *chessBmp* variable. If readFromFile is successful, *chessBmp* is written to the *bitmapField* object.

```
; getChess::pushButton
method pushButton(var eventInfo Event)
var
  chessBmp Graphic
endVar
; get the bitmap chess.bmp from the C:\Windows directory,
; and write it to the bitmapField graphic
if chessBmp.readFromFile("c:\\windows\\chess.bmp") then
  bitmapField = chessBmp
endIf
endMethod
```

writeToClipboard method

See also Example Graphic Type

Writes a bitmap to the Clipboard.

Syntax

`writeToClipboard ( ) Logical`

Description

writeToClipboard writes a bitmap to the Clipboard. **writeToClipboard** returns True if successful and False if it fails. Formats copied to Clipboard can be bitmap (BMP) or device independent bitmap (DIB).

writeToClipboard example

The following example assumes that a form contains a button named *getChessToClip*, and a bitmap field named *bitmapField*. The **pushButton** method for *getChessToClip* stores the value of *bitmapField* to *chessBmp*, then writes *chessBmp* to the Clipboard:

```
; getChessToClip::pushButton
method pushButton(var eventInfo Event)
var
   chessBmp Graphic
endVar
; get the bitmap from the bitmapField,
; and write it to the Clipboard
if NOT bitmapField.isblank() then
  chessBmp = bitmapField
  chessBmp.writeToClipboard()
endif
endMethod
```

▪

writeToFile method

See also Example Graphic Type

Writes a bitmap to a file.

Syntax

`**writeToFile** **(** const ***fileName*** String **)** Logical`

Description

writeToFile writes a bitmap to a disk file specified in *fileName*. If *fileName* does not specify a path, this method writes to :WORK:. **writeToFile** returns True if the file specified can be created; otherwise, it returns False.

▪

writeToFile example

The following example assumes that a form contains a button named *writeChessToFile*, and a bitmap named *bitmapField*. The **pushButton** method for *writeChessToFile* stores the value of *bitmapField* to *chessBmp*, then writes *chessBmp* to a file in the current directory named CHESS1.BMP:

```
; writeChessToFile::pushButton
method pushButton(var eventInfo Event)
var
   chessBmp Graphic
endVar
; get the bitmap from the bitmapField,
; and write it to the Clipboard
if NOT bitmapField.isblank() then
  chessBmp = bitmapField
  chessBmp.writeToFile("chess1.bmp")
endif
endMethod
```

KeyEvent type

A KeyEvent object gets and sets information about keystroke events.

The following built-in event methods are triggered by KeyEvents: **keyChar**, and **keyPhysical**.

The KeyEvent type includes several derived methods from the Event type.

Methods for the KeyEvent type

Event	KeyEvent
errorCode	**char**
getTarget	**charAnsiCode**
isFirstTime	**isAltKeyDown**
isPreFilter	**isControlKeyDown**
isTargetSelf	**isFromUI**
reason	**isShiftKeyDown**
setErrorCode	**setAltKeyDown**
setReason	**setChar**
	setControlKeyDown
	setShiftKeyDown
	setVChar
	setVCharCode
	vChar
	vCharCode

▪

char method

See also Example KeyEvent Type

Returns the character associated with a keypress.

Syntax

`char ( )` `String`

Description

char returns the character associated with a keypress. For example, if you type a, **char** returns "a". If you press Shift+A, **char** returns A. If a keypress results in an unprintable character, **char** returns an empty string ("").

char is the easiest way to check for an alphanumeric keypress when case matters. If case doesn't matter, use **vChar** to test against the string value of a virtual key code. For instance, if it matters whether the user presses a lowercase a or an uppercase A, use **char** to return the string value of the character pressed, and compare it to "a" or "A". If you want to find out if either a or A was pressed, use **vChar** and compare it to "A" (the virtual key code string for either a lowercase a or an uppercase A).

▪

char example

The following example displays the character typed into a field object as a message at the bottom of the screen. The code is attached to a field object's built-in **keyChar** method.

```
; thisField::keyChar
method keyChar(var eventInfo KeyEvent)
  doDefault                    ; put character in the field
  message(eventInfo.char())    ; then display character as a message
endMethod
```

▪

charAnsiCode method

See also Example KeyEvent Type

Returns the ANSI value associated with a keypress.

Syntax

```
charAnsiCode ( ) SmallInt
```

Description

charAnsiCode returns an integer representing the ANSI value associated with a keypress. For example, if you type a, **charAnsiCode** returns 97. If you press Shift+A, **charAnsiCode** returns 65. **charAnsiCode** works with unprintable characters as well. For example, if you press Enter, **charAnsiCode** returns 13.

charAnsiCode example

The following example beeps when a user presses Backspace or Ctrl+H. This code is attached to a field object's built-in **keyPhysical** method.

```
; thisField::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.charAnsiCode() = 8 then  ; if user presses Ctrl+H or Backspace
  beep()                             ; make a sound
endif
endMethod
```

▪

isAltKeyDown method

See also Example KeyEvent Type

Reports whether Alt was held down during a KeyEvent.

Syntax

`isAltKeyDown ( ) Logical`

Description

isAltKeyDown returns True if Alt was held down at the time a KeyEvent occurred; otherwise, it returns False.

isAltKeyDown example

The following example assumes a form has a box named *boxOne*. When the user presses Alt+C, the **keyPhysical** method for the form changes the color of *boxOne*. This code is attached to a form's **keyPhysical** method

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ;code here  executes for each object in form

    if eventInfo.isAltKeyDown() AND    ; if user presses Alt+C
       eventInfo.vChar() = "C" then
       disableDefault                  ; block normal processing
       ; alternate a boxOne's color between red and blue
       boxOne.color = iif(boxOne.color = Red, Blue, Red)
    endif

  else
    ;code here executes just for form itself
endif
endMethod
```

▪

isControlKeyDown method

See also Example KeyEvent Type

Reports whether Ctrl was held down during a KeyEvent.

Syntax

`isControlKeyDown ( ) Logical`

Description

isControlKeyDown returns True if Ctrl was held down at the time a KeyEvent occurred; otherwise, it returns False.

▪

isControlKeyDown example

See the example for **setControlKeyDown**.

▪

isFromUI method

See also Example KeyEvent Type

Reports whether an event was generated by the user interacting with Paradox.

Syntax

`isFromUI ( ) Logical`

Description

isFromUI reports whether a KeyEvent was generated by the user interacting with Paradox, or internally (for example, by an ObjectPAL statement). This method returns True only for the first KeyEvent generated by a keypress; for subsequent events and actions, it returns False.

▪

isFromUI example

The following example shows how to put one of two messages on the status bar depending on if a character is put in a field by a user or by ObjectPAL. This method returns True for user actions (including **sendKeys**, which mimics user input). It returns False with all other ObjectPAL methods, including **keyPhysical**.

The following code is attached to the **pushButton** method of a button named *btnAutoFill*. This method sends the character "a" to the field *fldPassword*.

```
; btnAutofill :: pushButton
method pushButton(var eventInfo Event)
   fldPassword.keyPhysical(97, 97, Shift)   ; send an "a"
endMethod
```

The following code is attached to the **keyPhysical** method of a field named *fldPassword*. This method sends one of two messages depending on whether the user typed in a character or used the *btnAutofill* button.

```
;fldPassword :: keyPhysical
method keyPhysical(var eventInfo KeyEvent)
   if eventInfo.isFromUI() then
      message("Try using the autofill button.")
   else
      message("Automatically typing value.")
   endIf
endMethod
```

▪

isShiftKeyDown method

See also Example KeyEvent Type

Reports whether Shift was held down during a KeyEvent.

Syntax

```
isShiftKeyDown ( ) Logical
```

Description

isShiftKeyDown returns True if Shift was held down at the time a KeyEvent occurred; otherwise, it returns False.

▪

isShiftKeyDown example

See the example for **setShiftKeyDown**.

▪

setAltKeyDown method

[See also](#) [Example](#) [KeyEvent Type](#)

Simulates pressing and holding Alt during a KeyEvent.

Syntax

```
setAltKeyDown ( const yesNo Logical )
```

Description

setAltKeyDown adds information about the state of Alt to a KeyEvent. You must specify Yes or No. Yes means Alt was pressed during a KeyEvent; No means Alt was not pressed.

setAltKeyDown example

The following example assumes a form has a box named *boxOne*. When the user presses Alt+C, the **keyPhysical** method for the form changes the color of *boxOne*. This code is attached to a form's **keyPhysical** method:

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    if eventInfo.isAltKeyDown() and    ; if user presses Alt+C
       eventInfo.vChar() = "C" then
       disableDefault                  ; block normal processing
       ; alternate a boxOne's color between red and blue
       boxOne.color = iif(boxOne.color = Red, Blue, Red)
    endif
  else
    ; code here executes just for form itself
endif
endMethod
```

To simulate pressing Alt+C, the code for this method creates a KeyEvent variable, then sets its virtual key character to "C" and sets the Alt key down.

```
; sendAltC::pushButton
method pushButton(var eventInfo Event)
var
  ke KeyEvent
endVar
ke.setVChar("C")            ; set the character to C
ke.setAltKeyDown(Yes)       ; set the Alt key state to pressed
thisForm.keyPhysical(ke)    ; send off the event
endMethod
```

▪

setChar method

See also Example KeyEvent Type

Specifies an ANSI character for a KeyEvent.

Syntax

setChar (const ***char*** String **)**

Description

setChar sets a KeyEvent to have an ANSI character based on the value of *char*, where *char* evaluates to single character string (example, a).

▪

setChar example

This code is attached to a field's built-in **keyChar** method. The **keyChar** method for *fieldOne* converts each space to an underscore as the user types characters into the field.

```
; thisField::keyChar
method keyChar(var eventInfo KeyEvent)
  if eventInfo.Char() = " " then  ; when user enters a space
    eventInfo.setChar("_")        ; convert it to underscore
  endif                           ; process other keystrokes normally
endMethod
```

▪

setControlKeyDown method

See also Example KeyEvent Type

Simulates pressing and holding Ctrl during a KeyEvent.

Syntax

setControlKeyDown (**const ***yesNo* Logical**)**

Description

setControlKeyDown adds information about the state of Ctrl to eventInfo for a KeyEvent. You must specify Yes or No. Yes means Ctrl was pressed during a KeyEvent; No means Ctrl was not pressed.

▪

setControlKeyDown example

The following example assumes a form has a box named *boxOne*. When the user presses Ctrl+C, the **keyPhysical** method for the form changes the color of *boxOne*. This code is attached to a form's **keyPhysical** method:

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter() then
  ; code here executes for each object in form
  if eventInfo.isControlKeyDown() and   ; if user presses Ctrl+C
    eventInfo.vChar() = "C" then
    disableDefault                      ; block normal processing
    ; alternate color of boxOne between red and blue
    boxOne.color = iif(boxOne.color = Red, Blue, Red)
  endif
else
  ; code here executes just for form itself
endif
endMethod
```

To simulate Ctrl+C, the code for this method creates a KeyEvent variable, then sets its virtual key character to "C" and sets the Ctrl key down.

```
; sendCtrlC::pushButton
method pushButton(var eventInfo Event)
var
  ke KeyEvent
endVar
ke.setChar("C")              ; set the character to C
ke.setControlKeyDown(Yes)    ; set the Ctrl key state to pressed
thisForm.keyPhysical(ke)     ; send off the event
endMethod
```

▪

setShiftKeyDown method

See also Example KeyEvent Type

Simulates pressing and holding Shift during a KeyEvent.

Syntax

setShiftKeyDown (const ***yesNo*** Logical **)**

Description

setShiftDown adds information about the state of Shift to a KeyEvent. You must specify Yes or No. Yes means Shift was pressed and held; No means Shift wasn't pressed.

▪

setShiftKeyDown example

The following example assumes a form has a box named *boxOne*. When the user presses Shift+C, the **keyPhysical** method for the form changes the color of *boxOne*. This code is attached to a form's **keyPhysical** method:

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter() then
  ; code here executes for each object in form
  if eventInfo.isShiftKeyDown() and    ; if user presses Ctrl+C
    eventInfo.vChar() = "C" then
    disableDefault                     ; block normal processing
    ; alternate color of boxOne between red and blue
    boxOne.color = iif(boxOne.color = Red, Blue, Red)
  endif
else
  ; code here executes just for form itself
endif
endMethod
```

To simulate pressing Shift+C, the code for this method creates a KeyEvent variable, then sets its virtual key character to "C" and sets the Shift key down.

```
; sendShiftC::pushButton
method pushButton(var eventInfo Event)
var
  ke KeyEvent
endVar
ke.setVChar("C")            ; set the character to C
ke.setShiftKeyDown(Yes)     ; set the Shift key state to pressed
thisForm.keyPhysical(ke)    ; send off the event
endMethod
```

▪

setVChar method

See also Example KeyEvent Type

Specifies a Windows virtual character for a KeyEvent.

Syntax

`setVChar` `( const` ***`char`*** `String )`

Description

setVChar specifies in *char* a one-character string for a KeyEvent. Use **setVChar** with an uppercase letter or a Keyboard constant to specify a code string for a single letter, but use the constant as a quoted string instead of an integer value. For example, the following statement specifies a tab character.

```
eventInfo.setVChar("VK_TAB")
```

The virtual character code string for any letter is the uppercase letter. For instance, the virtual character code string for the letter k is "K" (uppercase only).

-

setVChar example

See the example for **setAltKeyDown**.

String::chrToKeyName

▪

setVCharCode method

See also Example KeyEvent Type

Specifies a Windows virtual character for a KeyEvent.

Syntax

```
setVCharCode ( const VK_Constant SmallInt )
```

Description

setVCharCode uses a Keyboard constant in *VK_Constant* to specify a Windows virtual character for a KeyEvent.

setVCharCode example

This code is attached to a form's built-in **keyPhysical** method. When the user types ?, this code invokes the Paradox Help system.

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    if eventInfo.char() = "?" then   ; if user types ?
      eventInfo.setVCharCode(VK_HELP)  ; invoke built-in help system
    endif
  else
    ; code here executes just for form itself
endif
endMethod
```

▪

vChar method

See also   Example   KeyEvent Type

Returns a Windows virtual character.

Syntax

```
vChar ( ) String
```

Description

vChar returns a Windows virtual key name as a string. Use Keyboard constants to find out which Windows virtual character was returned, but use the constants as quoted strings instead of integer values. For example, the following statements are equivalent (they both beep when you press Return). The first statement uses **vCharCode** and the constant VK_RETURN to test for an integer value, the second statement uses **vChar** and "VK_RETURN" to test for a string value.

```
if vCharCode = VK_RETURN then beep() endIf
if vChar = "VK_RETURN" then beep() endIf
```

▪

vChar example

In the following example, assume a form contains a box named *boxOne*. When the user presses a movement key, this code moves *boxOne* in increments of 100 twips. If Shift is held down in combination with a movement key, *boxOne* moves 1000 twips. Since **vChar** returns the virtual key name as a string, this code must compare key names against string values such as "VK_LEFT". This code is attached to a form's built-in **keyPhysical** method.

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  kp      String        ; key name of the keystroke
  posPt   Point         ; x and y position of the box object
  boxStep SmallInt      ; number of Points to move the box
  x, y    LongInt       ; coordinates of the box object
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    disableDefault                   ; don't execute built-in code

    kp = eventInfo.vChar()           ; load kp with vChar string
    posPt = boxOne.position          ; posPt stores current position of box
    x = posPt.x()                    ; x stores the horizontal position
    y = posPt.y()                    ; y stores the vertical position

    ; if the Shift key was held down when the movement key was pressed,
    ; assign a large number to boxStep, else, a small number
    boxStep = iif(eventInfo.isShiftKeyDown(), 1000, 100)

    ; this block assigns x or y variables according to
    ; the key combination that the user presses
    switch
      case kp = "VK_LEFT"  : x = x - boxStep
      case kp = "VK_RIGHT" : x = x + boxStep
      case kp = "VK_UP"    : y = y - boxStep
      case kp = "VK_DOWN"  : y = y + boxStep
      otherwise            : enableDefault    ; let built-in code execute
    endswitch

    ; now move the box to location specified by x and y variables,
    ; and display the virtual key name associated with the keystroke
    boxOne.position = Point(x,y)
    message("Value of vChar() was " + kp)

  else
    ;code here executes just for form itself
endif
endMethod
```

vCharCode method

See also   Example   KeyEvent Type

Returns the integer value of a Windows virtual character.

Syntax

vCharCode ( ) SmallInt

Description

vCharCode returns the integer value of a Windows virtual character. Use Keyboard constants to find out which Windows virtual character the integer value represents.

vCharCode example

For the following example, assume a form has a field named *thisField*. When the user types a value in *thisField* and presses Return, the code creates and executes a query based on the value of the field. This code is attached to the built-in **keyPhysical** method for *thisField*.

```
; thisField::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  cName String      ; used as tilde var
  qVar Query          ; the query statement
  tv TableView      ; tableView handle
endVar

if eventInfo.vCharCode() = VK_RETURN then  ; if user presses Enter
  cName = self.value                           ; store value of field
  qVar = Query

         c:\pdoxwin\sample\biolife.db|Common Name |Species Name  |
                                    |check ~cName|check         |

       endQuery

  ; run query, write contents to myFish table
  qVar.executeQBE("myFish.db")
  tv.open("myFish")               ; view myFish view
endif
endMethod
```

Library type

Changes

A library is a Paradox object that stores custom methods, custom procedures, variables, constants, and user-defined data types. Libraries are useful for storing and maintaining frequently-used routines, and for sharing custom methods and variables among several forms.

In many ways, working with a library is like working with a form. For example, to create a form, choose File|New|Form; to create a library, choose File|New|Library. Like a form, a library has built-in event methods. You add code to a library just as you do to a form, using the Object Explorer and the ObjectPAL Editor. (However, you can't place design objects in the library.) As with a form, you can open Editor windows to declare custom ObjectPal methods, procedures, variables, constants, data types, and external routines.

The Library type includes several derived methods from the Form type.

Methods for the Library type

Form	**Library**
deliver	**close**
isCompileWithDebug	**create**
load	**enumSource**
methodDelete	**enumSourceToFile**
methodGet	**execMethod**
methodSet	**open**
save	
setCompileWithDebug	

Changes to Library type methods

The Form type has two new methods for version 7: **isCompileWithDebug** and **setCompileWithDebug**. Those two new Form methods now appear in the list of derived types for the Library type.

The following table lists new methods that were added for version 5.0.

New	Changed
create	(None)
deliver	
load	
methodDelete	
methodGet	
methodSet	
save	

close method

See also Example Library Type

Closes a library.

Syntax

```
close ( )
```

Description

close closes a library, and ends the association between a Library variable and the underlying library file.

▪

close example

The following example declares a Library variable named *lib*, and calls **open** to associate *lib* with the library TOOLS.LSL. The example executes a method from that library, then calls **close** to end the association between the variable and the library. Another call to **open** associates *lib* with the library KIT.LSL, making methods in that library available.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  lib Library           ; declare a Library variable
endVar

lib.open("TOOLS.LSL")   ; associate lib with the library TOOLS.LSL
lib.doThis()            ; execute a method from the library
lib.close()             ; end the association between lib and the library

lib.open("KIT.LSL")     ; associate lib with another library
lib.doThat()            ; execute a method from the library

endMethod
```

create method

See also Example Library Type

Creates a library.

Syntax

`create ( ) Logical`

Description

create creates a blank library and leaves it in a design window. You can use **methodSet** (derived from the Form type) to alter or add methods in the new library.

▪

create example

The following example uses **create** to create a new library, adds a custom method to it with **methodSet**, saves the library with **save**, then closes the library.

```
; btnCreateLibrary :: pushButton
method pushButton(var eventInfo Event)
   var
      lib   Library
   endVar

   ;Create library.
   lib.create()
   lib.methodSet("cmMessage", "method cmMessage()
       msgInfo(\"From new library\", \"Hello World!\") endMethod")
   lib.save("library")
   lib.close()

endmethod
```

▪

enumSource method

See also Example Library Type

Writes the code from a library to a Paradox table.

Syntax

`enumSource ( const tableName String [ , const recurse Logical ] )`

Description

enumSource lists, in the Paradox table specified in *tableName,* all the custom code (methods, procedures, variables, etc.) stored in a library. If the table does not exist, Paradox creates it in the current working directory; if the table does exist, information is appended to it.

The structure of the table is:

Field name	Type	Size
Object	A	128
MethodName	A	128
Source	M	64

The Object field stores the UIObject name of the library, the MethodName field stores the name of the method, procedure, or window (Var, Const, Proc, Type, or Uses), and the Source field stores the corresponding source code.

This method also applies to the Form type. For forms, the optional argument *recurse* specifies whether to include overridden methods for all objects contained by the form. Because a Library does not contain objects, the *recurse* argument is not meaningful in the context of a Library.

You must **open** or **load** the library before calling this method.

▪

enumSource example

The following example declares a Library variable named *lib*, and calls **open** to associate *lib* with the library TOOLS.LSL. Then the example calls **enumSource** to list the code from the library to a Paradox table named LIBSRC.DB:

```
; srcToTable::pushButton
method pushButton(var eventInfo Event)
var
  lib Library
endVar

if lib.open("TOOLS.LSL", PrivateToForm) then

  ; write contents of TOOLS.LSL to LIBSRC.DB--
  ; goes to :WORK: by default
  lib.enumSource("LIBSRC.DB")

else
  msgStop("TOOLS.LSL", "Could not open library.")
endIf

endMethod
```

▪

enumSourceToFile method

See also Example Library Type

Writes the code from a library to a text file.

Syntax

```
enumSourceToFile ( const fileName String [ , const recurse Logical ] )
```

Description

enumSourceToFile lists all the custom code (methods, procedures, variables, and so on) stored in a library to the text file specified in *fileName*. If the file does not exist, Paradox creates it. If the file does exist, Paradox overwrites it without asking for confirmation. If *fileName* contains no path or alias, the file is created in the working directory (:WORK:).

In the text file, comment lines are used to identify and mark the beginning and end of each method, procedure, and so on. The following example shows the code for a library's built-in **open** method:

```
;|BeginMethod|#Library1|open|
method open(var eventInfo Event)
   var
      myMsgTCursor   Tcursor
   endVar
   if not myMsgCursor.open("Msghelp.db") then
      msgStop("Error", "Couldn't open MsgHelp.db")
      fail()
   endIf
endMethod
;|EndMethod|#Library1|open|
```

This method also applies to the Form type. For forms, the optional argument *recurse* specifies whether to include overridden methods for all objects contained by the form. Because a Library does not contain objects, the *recurse* argument is not meaningful in the context of a Library.

You must call **open** or **load** the library before calling this method.

enumSourceToFile example

The following example declares a Library variable named *lib*, and calls **open** to associate *lib* with the library TOOLS.LSL. Then the example calls **enumSourceToFile** to list the code from the library to a text file named LIBSRC.TXT.

```
; getSource::pushButton
method pushButton(var eventInfo Event)
var
  lib Library
endVar

if lib.open("TOOLS.LSL", PrivateToForm) then

  ; write contents of TOOLS.LSL to LIBSRC.TXT--
  ; goes to :PRIV: by default
  lib.enumSourceToFile("LIBSRC.TXT")

else
  msgStop("TOOLS.LSL", "Could not open library.")
endIf

endMethod
```

■

execMethod method

See also Example Library Type

Calls a custom method that takes no arguments.

Syntax

execMethod (const ***methodName*** String **)**

Description

execMethod calls the custom method indicated by the string *methodName*. The method named in *methodName* takes no arguments. **execMethod** allows you to call a library method based on the contents of a variable, which means the compiler does not know the method to call until run time.

■

execMethod example

The following example creates an array of three items, where each item is the name of a custom method in a library. The code opens the library and calls **execMethod** for each item in the array:

```
var
   lib Library
   libMethods Array[3] String
   i SmallInt
endVar

libMethods[1] = "doThis"
libMethods[2] = "doThat"
libMethods[3] = "doOther"

if lib.open("tools.lsl", GlobalToDeskTop) then
   for i from 1 to libMethods.size()
      lib.execMethod(libMethods[i])
   endFor
else
   msgStop("TOOLS.LSL", "Could not open library.")
endIf
```

▪

open method

See also Example Library Type

Associates a Library variable with a library, and makes the library code available.

Syntax

```
open ( const libraryName String [ , const libScope SmallInt ] ) Logical
```

Description

open associates a Library variable with a library, and makes the library code, variables, constants, and type declarations available to the form. Variables declared in the library can be kept private to the form, or they can be shared with other forms and libraries that have opened this library, depending on the value of *libScope*. ObjectPAL defines LibraryScope constants for specifying the scope of variables declared in the library: PrivateToForm and GlobalToDesktop:

- PrivateToForm: Each form that opens the library has its own copy of the variables.
- GlobalToDesktop: every form in the Desktop (Paradox session) that opens the library shares the variables declared in the library.

To open a library and make its variables available to every form that opens the library in the current session of Paradox, use the constant GlobalToDesktop. For example, the following statement opens the library MYLIB.LSL:

```
lib.open("myLib.lsl", GlobalToDesktop)
```

For two or more forms to share the same library, each form must open the library global to the Desktop, and each form must have a Uses window that declares which library routines to use. This level of scope is useful in multiform applications, because it allows several forms access to the same custom methods and allows the forms to share the same global variables.

A library can be opened private to the form in one form and global to the Desktop in another form. Paradox will load a new instance of the library, if necessary.

By default, a library opens global to the Desktop. The following statements are equivalent:

```
lib.open("myLib.lsl") ; these statements are equivalent
lib.open("myLib.lsl", GlobalToDesktop)
```

open example

The following example shows how two forms can open a library global to the Desktop and share the library. In the following code, attached to a form's built-in **open** method, *libOne* is opened private to the form. *libOne* cannot be shared. *libTwo* is opened global to the Desktop and can be shared. *libOne* and *libTwo* are library variables that have been declared in the **var** block of the form.

```
; formOne::open
method open(var eventInfo Event)

if eventInfo.isPreFilter()
   then
   ; code here executes for each object in the form
else
   ; code here executes just for the form itself

   libOne.open("TOOLS.LSL", PrivateToForm)    ; no sharing variables
                                              ; with other forms
   libTwo.open("KIT.LSL", GlobalToDesktop)    ; can be shared
                                              ; with other forms
endIf
endMethod
```

The following code, attached to another form's built-in **open** method, calls **open** to open the library KIT.LSL global to the Desktop. This form and the previous form can now share KIT.LSL. *kitLib* is a library variable declared in the **var** block of the form.

```
; formTwo::open
method open(var eventInfo Event)

if eventInfo.isPreFilter()
   then
   ; code here executes for each object in the form
else
   ; code here executes just for the form itself
   kitLib.open("KIT.LSL", GlobalToDesktop) ; can be shared with other forms

endIf
endMethod
```

Logical type

Logical variables have two possible values: True or False. You can use the ObjectPAL constants Yes or On in place of True, and use No or Off in place of False.

A Logical variable occupies 1 byte of storage. In order of precedence, the logical operators are NOT, AND, and OR.

Logical variables often answer questions about other objects and operations, for example,

- Did that statement execute successfully?
- Is that table empty?
- Is that form displayed as an icon?

The Logical type includes several derived methods from the AnyType type.

Methods for the Logical type

AnyType	Logical
blank	**logical**
dataType	
isAssigned	
isBlank	
isFixedType	
view	

▪

logical procedure

See also   Example   Logical Type
Beginner

Casts a value as type Logical.

Syntax
`logical ( const value AnyType ) Logical`

Description
logical casts (converts) the data type of *value* to Logical. If *value* is a numeric data type, non-zero values evaluate to True and zero evaluates to False. If *value* is a string, it must evaluate to "True" or "False". (However, you can use True or False without the quotation marks.) ObjectPAL also provides Logical constants: On and Yes for True and Off and No for False.

■

logical example

In the following example, the **pushButton** method of a button named *showLogical* creates a string, casts it to a Logical type, then displays the result:

```
; showLogical::pushButton
method pushButton(var eventInfo Event)
var
  myVal      String
  theResult Logical
endVar
myVal = "True"               ; set a String of True
theResult = logical(myVal) ; and cast it to a Logical type
theResult.view()           ; show the result--Title displays Logical
endMethod
```

LongInt type

LongInt values are long integers; that is, they can be represented by a long series of digits. A LongInt variable occupies 4 bytes. ObjectPAL converts LongInt values to range from -2,147,483,648 to 2,147,483,647. An attempt to assign a value outside of this range to a LongInt variable causes an error. For example,

```
var
   x, y, z LongInt
endVar

x = 2147483647 ; The upper limit value for a LongInt variable.
y = 1
z = x + y      ; This statement causes an error.
```

When ObjectPAL performs an operation on LongInt values, it expects the result to be a LongInt, too. That's why the addition operation in the previous example causes an error: the result is too large to be a LongInt. To work with a boundary value (in either the positive or negative direction), convert it to a type that can accommodate it. In the following example, ObjectPAL converts one LongInt to a Number before doing the addition, and the statement succeeds. This example also assigns the result to a Number variable (which can handle the large value), instead of assigning it to a LongInt variable (which could not).

```
var
   x, y LongInt
   z    Number ; Declare z as a Number so it can hold the result.
endVar

x = 2147483647 ; The upper limit value for a LongInt variable.
y = 1
z = Number(x) + y ; This statement succeeds.
```

Note: Run-time library methods defined for the Number type also work with LongInt variables. The syntax is the same, and the returned value is a number.

The LongInt type includes several derived methods from the Number and AnyType types.

Methods for the LongInt type

AnyType	Number	**LongInt**
blank	abs	**bitAND**
dataType	acos	**bitIsSet**
isAssigned	asin	**bitOR**
isBlank	atan	**bitXOR**
isFixedType	atan2	**LongInt**
view	ceil	
	cos	
	cosh	
	exp	
	floor	
	fraction	
	fv	

■

bitAND method

See also Example LongInt Type

Performs a bitwise AND operation on two values.

Syntax

```
bitAND ( const value LongInt ) LongInt
```

Description

bitAND returns the result of a bitwise AND operation on value. **bitAND** operates on the binary representations of two integers, comparing them one bit at a time. The truth table for **bitAND** is:

a	b	a bitAND b
0	0	0
1	0	0
0	1	0
1	1	1

▪

bitAND example

In the following example, the **pushButton** method for a button named *andTwoNums* takes two integers and performs a bitwise AND calculation on them. The result of the calculation is displayed in a dialog box.

```
; andTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b LongInt
endVar
a = 33333   ; binary 00000000 00000000 10000010 00110101
b = -77777  ; binary 11111111 11111110 11010000 00101111
a.bitAND(b) ; binary 00000000 00000000 10000000 00100101
msgInfo("The result of a bitAND b is:", a.bitAND(b))
; displays 32805
endMethod
```

▪

bitIsSet method

See also Example1 Example2 LongInt Type

Reports whether a bit is 1 or 0.

Syntax

`bitIsSet ( const value LongInt ) Logical`

Description

bitIsSet examines the binary representation of an integer, reporting whether the **value** bit is 0 or 1. **bitIsSet** returns True if the bit specified is 1, and False if the bit is 0.

value is a number specified by 2*n*, where *n* is an integer between 0 and 30. The exponent *n* corresponds to one less than the position of the bit to test, counting from the right. For example, to specify the third bit from the right, use 4 (2(3-1), which is 22).

▪

bitIsSet example 1

In the following example, the **pushButton** method for a button named *isABitSet*, examines the values in two unbound field objects: *whichBit* and *whatNum*. *whichBit* contains the bit position (counting from the right) of the bit to test. *whatNum* contains the long integer to test.

The **pushButton** method uses *whichBit* to calculate the value of the position, then assigns the result to *bitNum*. The method then checks *Num* to see if the *bitNum* bit is set, and displays the Logical result with a **msgInfo** dialog box.

```
; isABitSet::pushButton
method pushButton(var eventInfo Event)
var
  bitNum,
  Num      LongInt
endVar
; get the bit position number from the whichBit
; field and convert to multiple of 2
bitNum = LongInt(pow(2, whichBit - 1))
; get the number to test from the whatNum field
Num = whatNum
; is the bit for value bitNum 1 in Num?
msgInfo("Is Bit Set?", Num.bitIsSet(bitNum))
endMethod
```

■

bitIsSet example 2

The next example illustrates how you can use **bitIsSet** to display a long integer as a binary number. The **pushButton** method for *showBinary* constructs a string of zeros and ones by testing each bit of a four-byte long integer. For readability, a blank is added to the string every 8 digits.

```
; showBinary::pushButton
method pushButton(var eventInfo Event)
var
  binString  String    ; to construct the binary string
  Num        LongInt
  i          SmallInt  ; for loop index
endVar
if NOT whatNum.isBlank() then
  Num = whatNum                ; get the number test from whatNum
  binString = ""               ; initialize the string
  for i from 0 to 30
    if Num.bitIsSet(LongInt(pow(2, i))) then
      binString = "1" + binString    ; add a 1 to the front of the string
    else
      binString = "0" + binString    ; add a 0 to the front of the string
    endif
    if i = 7 OR i = 15 OR i = 23 then
      binString = " " + binString    ; add a space every 8 digits
    endif
  endfor
  if Num < 0 then
    binString = "1" + binString      ; set the sign bit
  else
    binString = "0" + binString
  endif
  ; show the number
  message("The binary equivalent is ", binString)
endif
endMethod
```

▪

bitOR method

See also Example LongInt Type

Performs a bitwise OR operation on two values.

Syntax

```
bitOR ( const value LongInt ) LongInt
```

Description

bitOR returns the result of a bitwise OR operation on *value*. **bitOR** operates on the binary representations of two integers, comparing them one bit at a time. Here is the truth table for **bitOR**:

a	b	a bitOR b
0	0	0
1	0	1
0	1	1
1	1	1

bitOR example

For the following example, the **pushButton** method for a button named *orTwoNums* takes two integers and performs a bitwise OR calculation on them. The result of the calculation is displayed in a dialog box.

```
; orTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b LongInt
endVar
a = 33333  ; binary 00000000 00000000 10000010 00110101
b = -77777 ; binary 11111111 11111110 11010000 00101111
a.bitOR(b) ; binary 11111111 11111110 11010010 00111111
msgInfo("33333 OR -77777", a.bitOR(b)) ; displays -77249
endMethod
```

▪

bitXOR method

See also Example LongInt Type

Performs a bitwise XOR operation on two values.

Syntax

```
bitXOR ( const value LongInt ) LongInt
```

Description

bitXOR performs a bitwise XOR (exclusive OR) operation on *value*. **bitXOR** operates on the binary representations of two integers, comparing them one bit at a time. Here is the truth table for **bitXOR**:

a	b	a bitXOR(b)
0	0	0
1	0	1
0	1	1
1	1	0

bitXOR example

In the following example, the **pushButton** method for a button named *xorTwoNums* takes two integers and performs a bitwise XOR calculation on them. The result of the calculation is displayed in a dialog box.

```
; xorTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b LongInt
endVar
a = 33333   ; binary 00000000 00000000 10000010 00110101
b = -77777  ; binary 11111111 11111110 11010000 00101111
a.bitXOR(b) ; binary 11111111 11111110 01010010 00011010
msgInfo("33333 XOR -77777", a.bitXOR(b)) ; displays -110054
endMethod
```

■

LongInt procedure

See also Example LongInt Type
Beginner

Casts a value as a LongInt.

Syntax

```
LongInt ( const value AnyType ) LongInt
```

Description

LongInt casts (converts) the data type of value to a long integer. If you convert from a more precise type (for example, Number), precision may be lost.

LongInt example

The following example assigns a number to x, then casts x to **LongInt** and assigns the result to l. Notice that the decimal precision of x is lost when it is cast to a **LongInt** and assigned to l.

```
; convertToInt::pushButton
method pushButton(var eventInfo Event)
var
  x Number
  l LongInt
endVar
x = 12.34              ; give x a value
x.view()               ; view x, title of dialog will be "Number"
l = LongInt(x)         ; cast x as a LongInt and assign to l
l.view()               ; show l, note that decimal places are lost
                       ; displays 12
endMethod
```

Memo type

Changes

Memos contain text and formatting data up to 512MB in Paradox tables. Using Memo type methods **readFromFile** and **writeToFile**, you can transfer memos between forms (and reports), tables, and disk files.

You can also use the (=) operator to assign the value of a memo field to a Memo variable or a String variable.

Note: There are no arithmetic or comparison operators for Memo variables.

If you assign a memo field to a String variable, you get only the memo text without any formatting. If you assign a memo field to a Memo variable, you get the text and the formatting.

The Memo type includes several derived methods from the AnyType type.

Methods for the Memo type

AnyType	**Memo**
blank	**memo**
dataType	**readFromClipboard**
isAssigned	**readFromFile**
isBlank	**writeToClipboard**
isFixedType	**writeToFile**

Changes to Memo type methods

The following table lists new methods for version 7.

New
readFromClipboard
writeToClipboard

▪

memo procedure

See also   Example   Memo Type

Casts a value as a Memo.

Syntax

`memo ( const value AnyType [ , const value AnyType ] * ) Memo`

Description

memo casts (converts) the expression *value* to a Memo. If you specify multiple arguments, this method will cast all of them to Memos and concatenate them to one Memo.

memo example

The following example assumes that DOCFILES.DB exists and has an alpha field named Memo Name, a Date field named Memo Date, and a formatted memo field named Memo Data. For this example, a form has unbound fields named *stringObject* and *memoObject*, and a button named *getMemoData*. The code attached to *getMemoData's* **pushButton** method defines a TCursor to locate a particular record in *DocFiles*. Then, the code casts and concatenates the contents of the three *DocFiles* fields to a String value, then to a Memo value. The value cast as a String is displayed in the *stringObject* object and the value cast as a Memo is displayed in the *memoObject* object. Note that when cast as a String, formatting information is not displayed in *stringObject*. When cast as a Memo, *memoObject* displays all formatting information.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

if tc.open("DocFiles.db") then
  if tc.locate("Memo Name", "Project Notes") then

    ; this line casts data from three DOCFILES.DB fields as a String
    ; because this is cast as a String, the data that appears in stringObject
    ; displays WITHOUT formatting
    stringObject.value = string(tc."Memo Name", "\t",
                                tc."Memo Date", "\n", tc."Memo Data")

    ; this line casts data from three DOCFILES.DB fields as a memo

    ; because this is cast as a MEMO, the data that appears in memoObject
    ; displays with FORMATTED text
    memoObject.value = memo(tc."Memo Name", "\t",
                            tc."Memo Date", "\n", tc."Memo Data")

  else
    msgStop("Error", "Can't find Project Notes.")
  endif
else
  msgStop("Error", "Can't open DocFiles table.")
endif

endMethod
```

readFromClipboard method

See also Example Memo Type

Reads text from the Clipboard.

Syntax

`readFromClipboard ( )` `Logical`

Description

readFromClipboard reads text from the Clipboard. **readFromClipboard** will attempt to read in "Rich Text Format" if the format is available in the Clipboard. Otherwise, text (CF_TEXT) will be read in. readFromClipboard returns True if successful and False if unsuccessful.

readFromClipboard example

In the following example, a form has two buttons: readFromClipboard and writeToClipboard. The first button will read RTF formatted text from the Clipboard into a Memo variable which will then be stored in a table. The second button read a memo value from a table and writes it out to the Clipboard.

The following code is attached to the pushButton method for btnReadFromClipboard:

```
; btnReadFromClipboard::pushButton
method pushButton(var eventInfo Event)
var
   vrMemo   Memo
   tcMemo   TCursor
endVar

   ;// Open table to hold memos
   tcMemo.open(mymemos.db)
   if vrMemo.readFromClipboard() then
      ;// Add a record to the table and insert the value
      tcMemo.insertRecord()
      tcMemo.MemoField = vrMemo
      tcMemo.unlockRecord()
   endIf
   tcMemo.close()

endMethod
```

The following code is attached to the pushButton method for btnWriteToClipboard:

```
; btnWriteToClipboard::pushButton
method pushButton(var eventInfo Event)
var
   vrMemo   Memo
   tcMemo   TCursor
endVar

   ;// Open table to which contains memos
   tcMemo.open(mymemos.db)
   ;// Make sure there is data in the table
   if tcMemos.nRecords() <> 0 then
      ;// Copy a value to the Memo variable
      vrMemo = tcMemo.MemoField
      ;// Write it out to the Clipboard
      vrMemo.writeToClipboard()
   endIf
   tcMemo.close()

endMethod
```

▪

readFromFile method

See also Example Memo Type

Reads a memo from a file.

Syntax

`readFromFile ( const fileName String ) Logical`

Description

readFromFile reads a memo from a disk file specified in *fileName*. This method reads text only. It does not read the formatting of formatted memos.

▪

readFromFile example

The following example reads the contents of a text file to a memo field in a table. Assume that a table named *PJNotes* exists in the current directory and has the following fields: ProjDate, a Date field, and ProjNotes, a Memo field. The **pushButton** method for a button named *getFile* opens, edits, and inserts a new record in the *PJNotes* table, then fills the ProjDate field with the current date, and fills the ProjNotes field with text from a file named NOTES.TXT.

```
; getFile::pushButton
method pushButton(var eventInfo Event)
var
  MemoFile Memo
  pTC      TCursor
endVar

if pTC.open("pjNotes.db") then    ; open TCursor for PJNOTES.DB
  if MemoFile.readFromFile("notes.txt") then
    ; if memo file read was successful
    pTC.edit()                    ; edit PJNotes.DB table
    pTC.insertRecord()            ; insert a new blank record
    pTC.ProjDate = today()        ; fill the ProjDate field
    pTC.ProjNotes = MemoFile      ; write memo to ProjNotes field
    pTC.endEdit()                 ; end Edit mode
  endif
  pTC.close()                     ; close the TCursor
endIf
endMethod
```

▪

writeToClipboard method

See also Example Memo Type

Writes a memo to the Clipboard.

Syntax

`writeToClipboard ( ) Logical`

Description

writeToClipboard writes a memo to the Clipboard. The formats copied to the Clipboard are text (CF_TEXT) and "Rich Text Format". writeToClipboard returns True if successful and False if unsuccessful.

▪

writeToClipboard example

In the following example, a form has two buttons: readFromClipboard and writeToClipboard.   The first button will read RTF formatted text from the Clipboard into a Memo variable which will then be stored in a table.   The second button read a memo value from a table and writes it out to the Clipboard.

The following code is attached to the pushButton method for btnReadFromClipboard:

```
; btnReadFromClipboard::pushButton
method pushButton(var eventInfo Event)
var
   vrMemo   Memo
   tcMemo   TCursor
endVar

   ;// Open table to hold memos
   tcMemo.open(mymemos.db)
   if vrMemo.readFromClipboard() then
      ;// Add a record to the table and insert the value
      tcMemo.insertRecord()
      tcMemo.MemoField = vrMemo
      tcMemo.unlockRecord()
   endIf
   tcMemo.close()

endMethod
```

The following code is attached to the pushButton method for btnWriteToClipboard:

```
; btnWriteToClipboard::pushButton
method pushButton(var eventInfo Event)
var
   vrMemo   Memo
   tcMemo   TCursor
endVar

   ;// Open table to which contains memos
   tcMemo.open(mymemos.db)
   ;// Make sure there is data in the table
   if tcMemos.nRecords() <> 0 then
      ;// Copy a value to the Memo variable
      vrMemo = tcMemo.MemoField
      ;// Write it out to the Clipboard
      vrMemo.writeToClipboard()
   endIf
   tcMemo.close()

endMethod
```

▪

writeToFile method

See also Example Memo Type

Writes a memo to a file.

Syntax

`writeToFile ( const fileName String ) Logical`

Description

writeToFile writes a memo to a disk file specified in *fileName*. This method writes text only. It does not write the formatting of formatted memos.

▪

writeToFile example

The following example writes the contents of a memo to a text file. Assume that a table named *PJNotes* exists in the current directory and has the following fields: *ProjDate*, a Date field, and *ProjNotes*, a Memo field. The **pushButton** method for a button named *writeFile* opens the *PJNotes* table, locates a record with the current date, then writes the contents of the *ProjNotes* field for that record to a file named NOTETDAY.TXT.

```
; getFile::pushButton
method pushButton(var eventInfo Event)
var
  MemoFile Memo
  pTC      TCursor
endVar

if pTC.open("pjNotes.db") then                ; open PJNotes.DB table
  if pTC.locate("ProjDate", today()) then
    if NOT (pTC.ProjNotes = blank()) then  ; check if memo is blank
      MemoFile = pTC.ProjNotes             ; if not, write to MemoFile var
      MemoFile.writeToFile("notetday.txt") ; write MemoFile to text file
    endif
  endif
  pTC.close()                              ; close the TCursor
endIf
endMethod
```

Menu type

A Menu object is a list of items that appears in the application menu bar. When the user chooses an item from a menu, the text of that item is returned. Menus you build in ObjectPAL completely replace Paradox's built-in event menus (but you can get them back using **removeMenu**).

By default, menus do not persist across forms; each form has its own menu system associated with it. If you create a menu for a form, the menu appears only when that form is active. If you then open a second form, the second form uses the built-in event menus, not the menu you created for the first form. If you create a custom menu for each form, you can simulate context-sensitive menus in an application.

If you want two (or more) forms to display the same custom menu, set each form's StandardMenu property to Off. This instructs Paradox to retain the current menu when the user moves from one form to another. You can use the StandardMenu property to construct a single menu system for an entire application.

Note: A typical application uses both Menu objects and PopUpMenu objects. See the PopUpMenu type for more information.

Methods for the Menu type

Menu
addArray
addBreak
addPopUp
addStaticText
addText
contains
count
empty
getMenuChoiceAttribute
getMenuChoiceAttributeById
hasMenuChoiceAttribute
remove
removeMenu
setMenuChoiceAttribute
setMenuChoiceAttributeById
show

▪

addArray method

See also Example Menu Type

Appends elements of an array to a menu.

Syntax

```
addArray ( const items Array[ ] String )
```

Description

addArray appends *items* from an array to a menu. The array *items* are displayed from left to right across the menu bar. To create a drop-down menu or a cascading menu, use **addPopUp**.

addArray example

The following example constructs and displays an application menu bar when a form opens. This could be the application's main menu. Throughout the application, the menu displayed here can be changed by methods for other objects.

```
; thisForm::open
method open(var eventInfo Event)
var
  mMenu             Menu     ; main menu
  mmItems Array[3] String   ; main menu items
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    ;menu appears when the form first opens
    mmItems[1] = "File"        ; fill the array
    mmItems[2] = "Edit"
    mmItems[3] = "Window"
    mMenu.addArray(mmItems)    ; same as mMenu.addText(...) 3 times
    mMenu.show()               ; show the menu
endif
endMethod
```

▪

addBreak method

See also Example Menu Type

Starts a new row in a menu.

Syntax

```
addBreak ( )
```

Description

addBreak starts a new row in a menu. **addBreak** lets you explicitly "wrap" large menu constructs to two or more rows.

addBreak example

The following example constructs and displays an application menu bar when a form opens. It uses **addBreak** to add a second row on the menu bar.

```
; thisform::open
method open(var eventInfo Event)
var
  mMenu Menu
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    ;menu appears when the form first opens
    mMenu.addText("File")
    mMenu.addText("Edit")
    mMenu.addBreak()
    mMenu.addText("About...")  ; this appears on the second row
    mMenu.show()               ; show the menu
endif
endMethod
```

addPopUp method

See also Example Menu Type

Adds a pop-up menu to a menu bar item.

Syntax

`addPopUp ( const menuName String, const cascadedPopup PopUpMenu )`

Description

addPopUp adds the heading *menuName* and a pop-up menu *cascadedPopup* to a menu. This method is useful for creating drop-down menus and cascading menus.

Note: If you use **addPopUp** with a *menuName* of "&Window", Windows automatically appends a list of open windows to that pop-up menu.

■

addPopUp example

The code in the following example is attached to the built-in **arrive** method for each of two pages of a form. The **arrive** method for *pageOne* creates and displays a custom menu. The **arrive** method for *pageTwo* of the same form removes the custom menu. **addPopUp** is used to create a cascading pop-up menu and a drop-down menu.

Here is *pageOne*'s **arrive** method:

```
pageOne::arrive
method arrive(var eventInfo MoveEvent)
var
  p1, p2, p3  PopUpMenu
  m1          Menu
endVar

p1.addText("Passwords...")     ; add items to p1 popup
p1.addText("Attributes...")

p2.addText("Basic...")         ; add items to p2 popup
p2.addText("Scientific...")

p1.addPopUp("Calculator", p2) ; add another item to p1 popup,
                               ; and display p2 popup when the
                               ; item is selected

p3.addText("About...")         ; add an item to 3rd popup

m1.addPopUp("Utilities", p1)  ; add item to menu bar,
                               ; and drop-down p1 when selected
m1.addPopUp("Help", p3)        ; add item to menu bar,
                               ; and drop-down p3 when selected
m1.show()                      ; show the menu bar (not PopUpMenu)

endMethod
```

Here is *pageTwo*'s **arrive** method:

```
; pageTwo::arrive
method arrive(var eventInfo MoveEvent)
  removeMenu()     ; remove the custom menu■the default menu
                   ; will appear instead
endMethod
```

addStaticText method

[See also] [Example] [Menu Type]

Adds an unselectable text string to a menu.

Syntax

addStaticText (const *item* String)

Description

addStaticText appends *item* to a menu as unselectable text.

addStaticText example

In the following example, code attached to a form's **open** method creates a menu bar. This example uses **addStaticText** to add a static menu item to the menu bar.

```
thisForm::open
method open(var eventInfo Event)
var
  mMenu Menu
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    mMenu.addStaticText("Main menu")     ; first item is static
    mMenu.addText("File")                ; add two more items
    mMenu.addText("Edit")
    mMenu.show()                         ; show the menu
endif
endMethod
```

■

addText method

See also Example1 Example2 Menu Type

Adds a selectable text string to a menu.

Syntax

```
1. addText ( const menuName String )
2. addText ( const menuName String, const attrib SmallInt )
3. addText ( const menuName String, const attrib SmallInt, const id SmallInt
)
```

Description

addText adds a selectable text string to a menu.

Syntax 1 adds the item *menuName* to a menu. Menu items are displayed from left to right across the menu bar.

In syntax 2, you can use *attrib* to preset the display attribute of *menuName*. Use MenuChoiceAttributes constants to specify attributes.

In syntax 3, you can specify an *id* number (a SmallInt) to identify the menu by number instead of by *menuName*. Then, in the built-in event **menuAction** method, use the *id* number to determine which menu the user chooses. When you specify a menu *id*, you should use the built-in IdRanges constant **UserMenu** as a base constant, then add your own number to it or create a user-defined menu constant. For example, the following line adds "File" to the *myMenu* menu and specifies an *id* number for that menu item:

```
myMenu.addText("File", MenuEnabled, UserMenu + 1)
```

You can use an ampersand in an item to designate an accelerator key. For example, the item "&File" would display as File, and the user could choose it by pressing Alt + F. If you rely on *menuName* to test for the user's choice, you need to include the ampersand in the comparison string. In the following example, the return value is "&File", not "File".

To right-align menu items, you can precede *menuName* with the string value "\008". Once you include "\008" in *menuName*, all subsequent menu items appear right-aligned; you don't have to use "\008" again. For example, these lines display File on the left and Help and Utilities on the right:

```
myMenu.addText("File")
myMenu.addText("\008Help")
myMenu.addText("Utilities")
myMenu.show()
```

addText example 1

Examples 1 and 2 demonstrate how **addText** syntax influences the way you test for the user's menu choice.

The following example uses the first form of **addText** syntax to create a simple menu. It does not use *id* in the **addText** statements. The code attached to the built-in event **menuAction** method must evaluate the string specified in *menuName* to determine the user's menu choice. The code that follows is attached to the **open** method for *pageOne*.

```
; pageOne::open
method open(var eventInfo Event)
var
  mainMenu Menu
  utilPU PopUpMenu
endVar

 ; build a pop-up menu
utilPU.addText("&Time")
utilPU.addText("&Date")

 ; attach pop-up to the Utilities main menu item
mainMenu.addPopUp("&Utilities", utilPU)

 ; add "Help" to the menu and right-align "Help" with \008
mainMenu.addText("\008&Help")

 ; now display the menu
mainMenu.show()

endMethod
```

The following code is attached to the **menuAction** method for *pageOne*. This code uses the **menuChoice** method to obtain the string value defined by *menuName*.

```
; pageOne::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

choice = eventInfo.menuChoice()  ; assign string value to choice

  ; now use choice value to determine which menu was selected
switch
  case choice = "&Time"   :
    msgInfo("Current Time", time())
  case choice = "&Date"   :
    msgInfo("Today's date", today())
  case choice = "\008&Help" :
    ; open the built-in help system
    action(EditHelp)
endSwitch

endMethod
```

▪

addText example 2

The following example demonstrates how you can use the *id* clause with **addText** to refer to menu items by number instead of by name. This code establishes user-defined constants to make it easy to remember the menu *id* assignments. The following code goes in the Const window for *pageOne*:

```
; pageOne::Const
Const
  ; define constants for menu id's
  ; actual values (1, 2 and 3) are arbitrary
  TimeMenu = 1
  DateMenu = 2
  HelpMenu = 3
endConst
```

The following code is attached to the **open** method for *pageOne*. To control the menu display attributes, this code uses built-in constants such as **MenuEnabled**. To identify each menu item by number, the code uses the constants defined in the Const window for *pageOne* (TimeMenu, DateMenu, and HelpMenu).

```
; pageOne::open
method open(var eventInfo Event)
var
  mainMenu Menu
  utilPU PopUpMenu
endVar

 ; build a pop-up menu and use constants (ie: TimeMenu)
 ; defined in the Const window for thisPage
utilPU.addText("&Time", MenuEnabled, TimeMenu + UserMenu)
utilPU.addText("&Date", MenuEnabled, DateMenu + UserMenu)

 ; attach pop-up to the Utilities main menu item
mainMenu.addPopUp("&Utilities", utilPU)

 ; add "Help" to the menu bar and right-align "Help" with \008
mainMenu.addText("\008&Help", MenuEnabled, HelpMenu + UserMenu)

mainMenu.show()                  ; display the menu

endMethod
```

The code that follows is attached to the **menuAction** method for *pageOne*. This method evaluates menu selections by *id* number rather than by the name specified in *menuName*.

```
; pageOne::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice SmallInt
endVar

choice = eventInfo.id()      ; assign constant value (ie: 900) to choice

  ; now use constants to determine which menu was selected
switch
  case choice = TimeMenu + UserMenu:
    msgInfo("Current Time", time())
  case choice = DateMenu + UserMenu:
    msgInfo("Today's Date", today())
  case choice = HelpMenu + UserMenu:
    ; open the built-in help system
    action(EditHelp)
endSwitch

endMethod
```

▪

contains method

See also Example Menu Type

Reports whether an item is in a menu.

Syntax

`contains ( const item AnyType ) Logical`

Description

contains returns True if *item* is in the list of items in a menu; otherwise, it returns False.

contains example

The following example assumes that a multi-record object is on the form. When the user changes the value in a field contained in the multi-record object, an Undo menu item is added to the existing custom menu bar. When the user moves to another record, Undo is removed. The example uses **contains** to determine if Undo is present before adding or removing the item. The menu variable is defined in the form's Var window. The menu bar is created by the form's **open** method.

The following code goes in the form's Var window:

```
; thisForm::var
Var
  m1 Menu
endVar
```

The following code is for the form's **open** method:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    m1.addText("&Insert")
    m1.addText("&Delete")
    m1.show()               ; show two item menu
endif
endMethod
```

The following code is for the form's **action** method:

```
; thisForm::action
method action(var eventInfo ActionEvent)
if eventInfo.isPreFilter() then
  ;code here executes for each object in form

  switch
      ; when user locks a record (starts to change a field value)
    case eventInfo.id() = DataLockRecord :
            if not m1.contains("&Undo") then
         ; add Undo and redisplay the menu
         m1.addText("&Undo")
         m1.show()
            endIf

      ; when user posts the record (moves to another record)
    case eventInfo.id() = DataUnlockRecord :
            if m1.contains("&Undo") then
         ; remove Undo redisplay the menu
         m1.remove("&Undo")
         m1.show()
            endIf
  endswitch

endif
endMethod
```

The following code is for the form's **menuAction** method:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

if eventInfo.isPreFilter() then
  ;code here executes for each object in form

  choice = eventInfo.menuChoice()

  switch
    case choice = "&Insert" :
      active.action(DataInsertRecord) ; insert new record
    case choice = "&Delete" :
      active.action(DataDeleteRecord) ; delete current record
    case choice = "&Undo"   :
      active.action(DataCancelRecord) ; restore original state
      m1.remove("&Undo")              ; remove Undo menu item
      m1.show()                       ; redisplay menu without Undo
  endswitch

endif
endMethod
```

▪

count method

See also Example Menu Type

Returns the number of items in a menu.

Syntax

```
count ( ) SmallInt
```

Description

count returns the number of items in a menu, including separators, bars, and breaks.

count returns the number of items in a single menu. If you attach a pop-up menu to a menu bar item with **addPopUp**, **count** returns the number of items in the pop-up menu or the number of items in the menu bar, but not the total number of items in both menus.

count example

The following example constructs a menu and a pop-up menu, then displays the number of items in each menu. Note that **count** returns the number of items in a menu whether or not the menu is displayed.

```
; countMenus::pushButton
method pushButton(var eventInfo Event)
var
  m Menu
  p PopUpMenu
endVar

p.addText("&One")
p.addBar()
p.addText("T&wo")
p.addText("Th&ree")       ; 3 items + 1 bar = 4 elements

m.addText("&First")
m.addText("&Second")
m.addPopUp("&Third", p)    ; 3 items in menu bar

msgInfo("Menu bar items", m.count()) ; displays 3■counts menu bar only
msgInfo("Pop-up items", p.count())   ; displays 4
■counts pop-up only

endMethod
```

▪

empty method

See also Example Menu Type

Removes all items from a menu.

Syntax

```
empty ( )
```

Description

empty removes all items from a custom menu. Use **empty** when you need to clear an existing menu before rebuilding it.

▪

empty example

The following example uses two buttons to display alternate menus. Both methods affect the same menu, declared with the variable *mainMenu* in the form's Var window.

The following code goes in the form's Var window:

```
; thisForm::Var
Var
  mainMenu Menu  ; custom menu bar
endVar
```

Following is the code for *showMenuOne*'s **pushButton** method:

```
; showMenuOne::pushButton
method pushButton(var eventInfo Event)
  mainMenu.empty()          ; clear the menu
  mainMenu.addText("&One")  ; reconstruct it
  mainMenu.addText("&Two")
  mainMenu.show()           ; display the changed menu
endMethod
```

Following is the code for *showMenuTwo*'s **pushButton** method:

```
; showMenuTwo::pushButton
method pushButton(var eventInfo Event)
  mainMenu.empty()              ; clear the menu
  mainMenu.addText("File")      ; reconstruct it
  mainMenu.addText("Edit")
  mainMenu.show()               ; show it again
endMethod
```

■

getMenuChoiceAttribute procedure

See also Example Menu Type

Reports the display attributes of a menu item.

Syntax

```
getMenuChoiceAttribute ( const menuChoice String ) SmallInt
```

Description

getMenuChoiceAttribute returns an integer representing the display attributes of the menu item specified in *menuChoice*. The integer value represents the combination of attributes that apply. Use MenuChoiceAttributes constants to test attributes. Use **getMenuChoiceAttribute** with **hasMenuChoiceAttribute** to determine whether a specific display attribute applies for a menu item.

This procedure returns the attribute of the currently displayed menu; if you have not created a custom menu, **getMenuChoiceAttribute** operates on the built-in menu.

getMenuChoiceAttribute example

In the following example, the **open** method for *pageOne* constructs and displays a simple menu. The *getMenuState* button reports whether or not the Time menu item is enabled.

The following code is attached to the **open** method for *pageOne*:

```
; pageOne::open
method open(var eventInfo Event)
var
  mainMenu Menu
  utilPU PopUpMenu
endVar

 ; build a pop-up menu, disable Time option
utilPU.addText("&Time", MenuDisabled + MenuGrayed)
utilPU.addText("&Date")
 ; attach pop-up and show the menu bar
mainMenu.addPopUp("&Utilities", utilPU)
mainMenu.addText("&Help")
mainMenu.show()

endMethod
```

The following code is for *getMenuState's* **pushButton** method:

```
; getMenuState::pushButton
method pushButton(var eventInfo Event)
var
  attrib SmallInt
endVar

  ; store attributes of Time in attrib
attrib = getMenuChoiceAttribute("&Time")
  ; this displays False because Time is disabled
msgInfo("Time enabled?", HasMenuChoiceAttribute(attrib, MenuEnabled))
  ; this displays True because Time is grayed
msgInfo("Time grayed?", hasMenuChoiceAttribute(attrib, MenuGrayed))

endMethod
```

getMenuChoiceAttributeById procedure

See also Example Menu Type

Reports the display attribute of a menu item specified by its menu ID.

Syntax

`getMenuChoiceAttributeById ( const menuId SmallInt ) SmallInt`

Description

getMenuChoiceAttributeById returns an integer representing the display attributes of the menu item specified in *menuId*. The integer value represents the combination of attributes that apply. Use MenuChoiceAttributes constants to test attributes. Use **getMenuChoiceAttributeById** with **hasMenuChoiceAttribute** to determine whether a specific display attribute applies for a menu item.

This procedure returns the attribute of the currently displayed menu; if you have not created a custom menu, **getMenuChoiceAttributeById** operates on the built-in menu.

This procedure is similar to **getMenuChoiceAttribute** in that both report the display attributes for a specified menu item. The difference is that you specify the actual menu ID (a SmallInt value) for **getMenuChoiceAttributeById**, and the menu name (a String value) for **getMenuChoiceAttribute**. **getMenuChoiceAttributeById** is especially useful when you specify a menu ID as part of **addText** syntax.

getMenuChoiceAttributeById example

The following example demonstrates how you can use **getMenuChoiceAttributeById** with **hasMenuChoiceAttribute** to determine whether a menu item is disabled. In this example, the **open** method for *pageOne* constructs a small menu. The **pushButton** method for the *getMenuState* button reports on the state of the Undo menu item.

The following code goes in the form's Var window:

```
; thisForm::Var
Var
  m1      Menu
  p1, p2 PopUpMenu
endVar
```

The following code goes in the form's Const window:

```
; thisForm::Const
Const
  UndoMenu  = 1
  InsMenu   = 2
  DelMenu   = 3
  IndexMenu = 4
  AboutMenu = 5
endConst
```

The following code is for the page's **open** method:

```
; pageOne::open
method open(var eventInfo Event)

p1.addText("Undo",   MenuDisabled + MenuGrayed, UndoMenu + UserMenu)
p1.addText("Insert", MenuEnabled, InsMenu + UserMenu)
p1.addText("Delete", MenuEnabled, DelMenu + UserMenu)
p2.addText("Index",  MenuEnabled, IndexMenu + UserMenu)
p2.addText("About",  MenuEnabled, AboutMenu + UserMenu)

m1.addPopUp("&Record", p1)
m1.addPopUp("&Help", p2)
m1.show()

endMethod
```

The following code is attached to the *getMenuState's* **pushButton** method:

```
; getMenuState::pushButton
method pushButton(var eventInfo Event)

  ; store attributes of Undo menu in attrib
attrib = getMenuChoiceAttributeById(UndoMenu + UserMenu)

  ; this displays False because Undo is disabled
msgInfo("Undo enabled?", hasMenuChoiceAttribute(attrib, MenuEnabled))
  ; this displays True because Undo is grayed
msgInfo("Undo grayed?", hasMenuChoiceAttribute(attrib, MenuGrayed))
endMethod
```

hasMenuChoiceAttribute procedure

See also Example Menu Type

Reports whether a menu item contains a given display attribute.

Syntax

hasMenuChoiceAttribute (** const ***attrib* SmallInt **,** const ***attribSet*** SmallInt **)** Logical

Description

hasMenuChoiceAttribute returns True if *attribSet* contains the attribute specified in *attrib*; otherwise, it returns False. Use MenuChoiceAttributes constants to specify attributes.

Use **hasMenuChoiceAttribute** with **getMenuChoiceAttribute** or **getMenuChoiceAttributeById** to determine whether a particular display attribute for a menu item is represented in *attribSet*.

hasMenuChoiceAttribute example

The following code demonstrates how you can use **hasMenuChoiceAttribute** with **getMenuChoiceAttribute** to determine whether a particular attribute applies to the currently displayed menu. The following code is attached to the **open** method for *pageOne*.

```
; pageOne::open
method open(var eventInfo Event)
var
  m1 Menu
  p1 PopUpMenu
endVar

p1.addText("&Insert")    ; create a simple menu
p1.addText("&Delete")
p1.addText("&Undo")
m1.addPopUp("&Record", p1)
m1.show()

endMethod
```

The following code is attached to the **pushButton** method for the *toggleMenuState* button:

```
; toggleMenuState::pushButton
method pushButton(var eventInfo Event)
var
  attribSet SmallInt
endVar

 ; store composite menu attributes in attribSet
attribSet = getMenuChoiceAttribute("&Undo")

 ; this is True if Undo is enabled
if hasMenuChoiceAttribute(attribSet, MenuEnabled) then
   setMenuChoiceAttribute("&Undo", MenuDisabled + MenuGrayed)
else
  setMenuChoiceAttribute("&Undo", MenuEnabled)
endif

endMethod
```

remove method

See also Example Menu Type

Removes an item from a menu.

Syntax

```
remove ( const item AnyType )
```

Description

remove deletes the first occurrence of *item* from a menu. This method is useful for changing one item in a menu without having to rebuild the entire menu.

▪

remove example

The code shown in the following example changes a menu immediately by removing an item and adding another item in its place.

```
; changeMenu::pushButton
method pushButton(var eventInfo Event)
var
   mainMenu Menu
endVar

; First, assume the user is working with a form.
; You could display a menu like this:
mainMenu.addText("File")
mainMenu.addText("Edit")
mainMenu.addText("Form")
mainMenu.show()
msgInfo("Status", "About to change menus. Watch closely.")

; Then, suppose the user switches to work on a report.
; You could change the menu like this:
mainMenu.remove("Form")
mainMenu.addText("Report")
mainMenu.show()

msgInfo("Status", "About to remove the menus. Watch closely.")

; remove entire menu, reveal built-in menus
removeMenu()
endMethod
```

▪

removeMenu procedure

See also Example Menu Type

Removes a custom menu and displays the default menu.

Syntax

```
removeMenu ( )
```

Description

removeMenu replaces a menu built using ObjectPAL with Paradox's default menu.

removeMenu example

In the following example, the form's **open** method constructs a menu (but does not display it). The arrive method for *pageOne* displays the menu with **show**. The **arrive** method for *pageTwo* removes the menu and reveals the built-in Paradox menu.

The following code goes in the form's Var window:

```
; thisForm::var
Var
  m1 Menu
endVar
```

The following code is attached to the form's **open** method:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

     m1.addText("&File")   ; construct a menu
     m1.addText("&Edit")
     m1.addText("For&m")

endif

endMethod
```

The following code is attached to the **arrive** method for *pageOne*:

```
; pageOne::arrive
method arrive(var eventInfo MoveEvent)
m1.show()  ; display the application menu
endMethod
```

The following code is attached to the **arrive** method for *pageTwo*:

```
; pageTwo::arrive
method arrive(var eventInfo MoveEvent)
removeMenu()   ; remove application menu, reveal built-in menu
endMethod
```

▪

setMenuChoiceAttribute procedure

See also Example Menu Type

Sets the display attribute of a menu item.

Syntax

```
setMenuChoiceAttribute ( const menuChoice String, const menuAttribute
SmallInt )
```

Description

setMenuChoiceAttribute sets the display attribute of *menuChoice* to *menuAttribute*. Use MenuChoiceAttributes constants to specify attributes. This procedure affects the currently displayed menu; if you have not created a custom menu, **setMenuChoiceAttribute** affects the built-in menu.

Note: If a menu item's definition includes an accelerator key (for example, Print which is defined as "&Print"), remember to include the ampersand in the comparison string *menuChoice*.

▪

setMenuChoiceAttribute example

In the following example, you change the attribute of the Undo option, depending on whether there is anything to undo. As the user makes changes to the record, the Undo item becomes selectable. After posting the changes, Undo is unavailable.

The following code goes in the form's Var window:

```
; thisForm::var
Var
  m1 Menu
  p1 PopUpMenu
endVar
```

The following code is for the form's **open** method:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

    ; create a menu and show it
    p1.addText("&Undo", MenuDisabled + MenuGrayed)
    p1.addText("&Insert")
    p1.addText("&Delete")
    m1.addPopUp("&Record", p1)
    m1.show()

endif

endMethod
```

The following code is for the form's **action** method:

```
; thisForm::action
method action(var eventInfo ActionEvent)

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form

    switch
         ; when user locks a record (starts to change a field value)
      case eventInfo.id() = DataLockRecord :
         ; enable Undo menu item
        setMenuChoiceAttribute("&Undo", MenuEnabled)

        ; when user posts the record (moves to another record)
      case eventInfo.id() = DataUnlockRecord :
        ; disable and gray Undo menu item
        setMenuChoiceAttribute("&Undo", MenuDisabled + MenuGrayed)
    endswitch

  else
    ;code here executes just for form itself
  endif

endMethod
```

The following code is for the form's **menuAction** method:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form

    choice = eventInfo.menuChoice()
    switch
      case choice = "&Insert" :
        active.action(DataInsertRecord)  ; insert new record
      case choice = "&Delete" :
        active.action(DataDeleteRecord)  ; delete current record
      case choice = "&Undo"   :
        active.action(DataCancelRecord)  ; revert record to original state
        setMenuChoiceAttribute("&Undo", MenuDisabled + MenuGrayed)
    endswitch

  else
    ;code here executes just for form itself
endif

endMethod
```

setMenuChoiceAttributeById procedure

See also Example Menu Type

Sets the display attribute of a menu item.

Syntax

```
setMenuChoiceAttributeById ( const menuId String, const menuAttribute
SmallInt )
```

Description

setMenuChoiceAttributeById sets the display attribute of *menuId* to *menuAttribute*. Use MenuChoiceAttributes constants to specify attributes. This procedure affects the currently displayed menu; if you have not created a custom menu, **setMenuChoiceAttributeById** affects the built-in menu.

Note: If a menu item's definition includes an accelerator key (for example, Print which is defined as "&Print"), remember to include the ampersand in the comparison string *menuChoice*.

setMenuChoiceAttributeById example

In the following example, you change the attribute of the Undo option, depending on whether there is anything to undo. As the user makes changes to the record, the Undo item becomes selectable. After posting the changes, Undo is unavailable. This example uses the *menuId* clause in **addText** so that the code can refer to menu items by number rather than menu name.

The following code goes in the form's Var window:

```
; thisForm::var
Var
  m1 Menu
  p1 PopUpMenu
endVar
```

The following code goes in the form's Const Window:

```
; thisForm::const
Const
  InsMenu  = 1  ; use constants for menu id's
  DelMenu  = 2
  UndoMenu = 3
endConst
```

The following code is attached to the form's **open** method:

```
; thisForm::open
method open(var eventInfo Event)

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

     ; construct a menu and display it
     p1.addText("&Undo",   MenuDisabled + MenuGrayed, UndoMenu + UserMenu)
     p1.addText("&Delete", MenuEnabled, DelMenu + UserMenu)
     p1.addText("&Insert", MenuEnabled, InsMenu + UserMenu)
     m1.addPopUp("&Record", p1)
     m1.show()

  endif

endMethod
```

The following code is attached to the form's **action** method:

```
; thisForm::action
method action(var eventInfo ActionEvent)

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form

    switch
         ; when user locks a record (starts to change a field value)
      case eventInfo.id() = DataLockRecord :
         ; enable Undo menu item
        setMenuChoiceAttributeById(UndoMenu + UserMenu,
                                   MenuEnabled)

         ; when user posts the record (moves to another record)
      case eventInfo.id() = DataUnlockRecord :
         ; disable and dim Undo menu item
        setMenuChoiceAttributeById(UndoMenu + UserMenu,
                                   MenuGrayed + MenuDisabled)
    endswitch

  else
    ;code here executes just for form itself
endif

endMethod
```

The following code is attached to the form's **menuAction** method:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  menuItem SmallInt
endVar

if eventInfo.isPreFilter() then
  ;code here executes for each object in form

  menuItem = eventInfo.id()
  switch
    case menuItem = InsMenu :
      active.action(DataInsertRecord)    ; insert new record
    case menuItem = DelMenu :
      active.action(DataDeleteRecord)    ; delete current record
    case menuItem = UndoMenu :
      active.action(DataCancelRecord)    ; revert record to original state
      setMenuChoiceAttributeById(UndoMenu, MenuDisabled + MenuGrayed)
  endswitch

  endswitch

else
  ;code here executes just for form itself
endif

endMethod
```

▪

show method

See also Example Menu Type

Displays a menu.

Syntax

```
show ( )
```

Description

show displays a menu.

The user's choice is handled using the built-in event method **menuAction** and **menuChoice** from the MenuEvent type. Refer to the *Guide to ObjectPAL* for more information about working with menus.

▪

show example

In the following example, a form's **open** method constructs a simple menu, then displays it with **show**. The **menuAction** method for the form handles the user's menu choice. Following is the code attached to the **open** method for *thisForm*.

```
; thisForm::open
method open(var eventInfo Event)
var
  p1 PopUpMenu
  m1 Menu
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

    p1.addText("&Time")            ; construct a pop-up
    p1.addText("&Date")
    m1.addPopUp("&Utilities", p1) ; attach pop-up to menu item
    m1.show()                      ; display the m1 menu

endif

endMethod
```

The following code is attached to the form's **menuAction** method:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  menuName String
endVar

if eventInfo.isPreFilter() then
  ;code here executes for each object in form

  menuName = eventInfo.menuChoice()
  switch
    case menuName = "&Time" : msgInfo("Current Time", time())
    case menuName = "&Date" : msgInfo("Today's Date", date())
  endSwitch

else
  ;code here executes just for form itself
endif

endMethod
```

MenuEvent type

See also

MenuEvent variables contain data related to menu selections in the application menu bar. When the user chooses an item from a menu, it triggers the built-in **menuAction** method. By modifying an object's built-in **menuAction** method, you can define how the object responds.

The MenuEvent type includes several derived methods from the Event type.

Methods for the MenuEvent type

Event	MenuEvent
errorCode	**data**
getTarget	**id**
isFirstTime	**isFromUI**
isPreFilter	**menuChoice**
isTargetSelf	**reason**
setErrorCode	**setData**
	setId
	setReason

User-defined menu constants

See also MenuEvent Type

You can define your own menu constants, but you must keep them within a specific range. Because this range is subject to change in future versions of Paradox, ObjectPAL provides the IdRanges constants UserMenu and UserMenuMax to represent the minimum and maximum values allowed.

For example, suppose that you want to define two menu constants, ThisMenuItem and ThatMenuItem. In a Const window, define values for your custom constants as follows:

```
Const
   ThisMenuItem = 1
   ThatMenuItem = 2
EndConst
```

Then, to use one of these constants, add it to UserMenu. For example,

```
method menuAction(var eventInfo MenuEvent)
   if eventInfo.id() = UserMenu + ThisMenuItem then
      doSomething()
   endIf
endMethod
```

By adding UserMenu to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, use the value of UserMenuMax. One way to check the value is with a **message** statement:

```
message(UserMenuMax)
```

In this version of Paradox, the difference between UserMenu and UserMenuMax is 2047. That means the largest value you can use for a menu constant is UserMenu + 2047.

▪

data method

See also MenuEvent Type

Returns information about a MenuEvent.

Syntax

data ( ) LongInt

Description

data should be used by Windows programmers only. **data** returns the *lParam* argument (usually zero) of specific Windows messages, such as WM_SYSCOMMAND and WM_COMMAND. See your Windows programming documentation for more information.

▪

id method

See also Example1 Example2 MenuEvent Type
Beginner

Returns the ID of a MenuEvent.

Syntax

```
id ( ) SmallInt
```

Description

id returns the ID number of a MenuEvent. ObjectPAL provides MenuCommands constants (like MenuFileOpen) for many common menu choices, and you can also use user-defined menu constants to test the value returned by **id**.

▪

id example 1

This is attached to a form's built-in **menuAction** method. When the user selects Close from the System menu, attempts to toggle to a design window, or chooses File|Exit, the method asks the user to confirm whether or not to leave the form.

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    if eventInfo.id() = MenuControlClose OR
      eventInfo.id() = MenuFileExit OR
      eventInfo.id() = MenuFormDesign then
      disableDefault                      ; block departure
      ans = msgQuestion("Please confirm",
                        "Do you really want to leave?")
      if ans = "Yes" then
        dodefault
      endif
    endif
endif
endMethod
```

id example 2

The next example demonstrates how you can use the menu ID argument with **addText** to refer to menu items by number (ideally, user-defined constants) instead of by name. This code establishes user-defined constants to make it easy to remember the menu ID assignments. The following code defines constants global to *pageOne*.

```
; pageOne::Const
Const
  ; define constants for menu IDs
  ; actual values (1, 2 and 3) are arbitrary
  TimeMenu = 1
  DateMenu = 2
  HelpMenu = 3
endConst
```

The following code is attached to the **open** method for *pageOne*. To control the menu display attributes, this code uses built-in constants such as MenuEnabled. To identify each menu item by number, the code uses the constants defined in the Const window for *pageOne* (TimeMenu, DateMenu, and HelpMenu).

```
; pageOne::open
method open(var eventInfo Event)
var
  mainMenu Menu
  utilPU PopUpMenu
endVar

 ; build a pop-up menu and use constants (ie: TimeMenu)
 ; defined in the Const window for thisPage
utilPU.addText("&Time", MenuEnabled, TimeMenu + UserMenu)
utilPU.addText("&Date", MenuEnabled, DateMenu + UserMenu)
 ; UserMenu is an ObjectPAL constant
 ; attach pop-up to the Utilities main menu item
mainMenu.addPopUp("&Utilities", utilPU)

 ; add "Help" to the menu bar and right-justify "Help" with \008
mainMenu.addText("\008&Help", MenuEnabled, HelpMenu) + UserMenu

mainMenu.show()                  ; display the menu

endMethod
```

The following code is attached to the **menuAction** method for *pageOne*. This method evaluates menu selections by ID number rather than by the name specified in *menuName*.

```
; pageOne::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice SmallInt
endVar

choice = eventInfo.id()      ; assign constant value to choice

  ; now use constants to determine which menu was selected
switch
  case choice = TimeMenu + UserMenu:
    msgInfo("Current Time", time())
  case choice = DateMenu + UserMenu:
    msgInfo("Today's Date", today())
  case choice = HelpMenu + UserMenu:
    ; change menu ID to built-in constant (MenuHelpContents)▪
    ; this effectively opens the built-in help system.
    eventInfo.setId(MenuHelpContents)
    eventInfo.setReason(MenuDesktop)
endSwitch

endMethod
```

▪

isFromUI method

See also Example1 Example2 MenuEvent Type

Reports whether an event was generated by the user interacting with Paradox.

Syntax

```
isFromUI ( ) Logical
```

Description

isFromUI reports whether an event was generated by the user interacting with Paradox, or internally (for example, by an ObjectPAL statement). This method returns True if the event was generated by the user; otherwise, it returns False.

isFromUI example 1

The following example checks for a menu action to delete a record. If the action is from the UI (that is, if the user made the menu choice), a dialog box prompts for confirmation before deleting the record.

```
;frm :: menuAction
method menuAction(var eventInfo MenuEvent)

   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:

   else
      ;// This code executes only for the form:

      if eventInfo.id() = MenuRecordDelete and
      eventInfo.isFromUI() then
            if msgQuestion("Delete record?",
                     "Delete this record?") <> "Yes" then

               disableDefault
               return
            endIf
         endIf
   endif

endMethod
```

isFromUI example 2

This example shows how you can use **isFromUI** to indicate if the menu action was sent by **menuAction** or by **sendKeys**.

The following code is attached to the page's Const window. It declares constants to make it easy to remember the menu ID assignments.

```
; pageOne::Const
Const
  ; define constants for menu IDs
  ; actual values (1, 2 and 3) are arbitrary
  kTimeMenu = 1
  kDateMenu = 2
  kHelpMenu = 3
endConst
```

The following code is attached to the **open** method for *pageOne*. To control the menu display attributes, this code uses ObjectPAL constants such as MenuEnabled. To identify each menu item by number, the code uses the constants defined in the Const window for *pageOne* (kTimeMenu, kDateMenu, and kHelpMenu).

```
; pageOne::open
method open(var eventInfo Event)
var
  mainMenu Menu
  utilPU    PopUpMenu
endVar

 ; build a pop-up menu and use constants (ie: kTimeMenu)
 ; defined in the Const window for thisPage
utilPU.addText("&Time", MenuEnabled, kTimeMenu + UserMenu)
utilPU.addText("&Date", MenuEnabled, kDateMenu + UserMenu)
 ; UserMenu is an ObjectPAL constant
 ; attach pop-up to the Utilities main menu item
mainMenu.addPopUp("&Utilities", utilPU)

 ; add "Help" to the menu bar and right-justify "Help" with \008
mainMenu.addText("\008&Help", MenuEnabled, kHelpMenu + UserMenu)

mainMenu.show()                 ; display the menu

endMethod
```

The following code is attached to the **menuAction** method for *pageOne*. This method evaluates menu selections by ID number rather than by the name specified in *menuName*. In addition, it uses **isFromUI** to report whether the menu event was generated by **menuAction** or by **keyPhysical**.

```
; pageOne::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice          SmallInt
  youDoneIt   Logical
endVar

youDoneIt = eventInfo.isFromUI()
choice = eventInfo.id()      ; assign constant value to choice

  ; now use constants to determine which menu was selected
switch
  case choice = kTimeMenu + UserMenu:
    msgInfo("Did a user do this", youDoneIt)
    msgInfo("Current Time", time())
  case choice = kDateMenu + UserMenu:
    msgInfo("Did a user do this", youDoneIt)
    msgInfo("Today's Date", today())
  case choice = kHelpMenu + UserMenu:
    ; change menu ID to built-in constant (MenuHelpContents)▪
    ; this effectively opens the built-in help system.
    eventInfo.setId(MenuHelpContents)
    eventInfo.setReason(MenuDesktop)
endSwitch
endMethod
```

You can use the following two buttons to demonstrate the code above. The following code is attached to the **pushButton** method of a button named *btnObjectPAL*. It uses **menuAction** to send a menu event.

```
;btnObjectPAL :: pushButton
method pushButton(var eventInfo Event)
   menuAction(kDateMenu + UserMenu)
endMethod
```

The following code is attached to the **pushButton** method of a button named *btnSendKeys*. It uses **sendKeys** to send the keystrokes Alt+u+t . Use this button to simulate a user selecting a menu.

```
;btnSendKeys :: pushButton
method pushButton(var eventInfo Event)
   sendKeys("%ut")
endMethod
```

■

menuChoice method

See also		Example		MenuEvent Type
Beginner

Returns a string containing an item chosen from a menu.

Syntax

```
menuChoice ( ) String
```

Description

menuChoice returns a string containing an item chosen from a menu. Use **menuChoice** to modify an object's built-in **menuAction** method to specify how that object responds to menu choices.

Note: If a menu item's definition includes an accelerator key (for example "&Print"), remember to include the ampersand in the comparison string, for instance, the following code compares the return value of **menuChoice** with the string "&Print":

```
if eventInfo.menuChoice() = "&Print" then
   ; print the report
endif
```

menuChoice example

The following example assumes a form contains at least one memo field, named *thisMemoField*. When the user arrives on *thisMemoField*, the built-in **arrive** method displays a menu that lets the user perform basic cut and paste operations. The built-in **menuAction** method attached to *thisMemoField* uses **menuChoice** to evaluate the user's selection, and take appropriate action. Although this example mimics the behavior of the default menus, this technique is necessary when the default menus are replaced by custom menus.

This code is attached to the built-in **arrive** method for *thisMemoField*:

```
; thisMemoField::arrive
method arrive(var eventInfo MoveEvent)
Var
  EditPopUp PopUpMenu
  EditMenu  Menu
endVar

EditPopUp.addText("&Cut")              ; create a pop-up menu
EditPopUp.addText("&Copy")
EditPopUp.addText("&Paste")

EditMenu.addPopUp("&Edit", EditPopUp)  ; add pop-up menu bar item
EditMenu.show()                        ; display the menu
endMethod
```

This code is attached to the built-in **menuAction** method for *thisMemoField*. Note that comparisons in the **switch...endSwitch** statement must include the ampersand, such as "&Cut".

```
thisMemoField::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar
choice = eventInfo.menuChoice()       ; store the menu selection to choice

; now respond to the selection appropriately
switch
  case choice = "&Cut"   : self.action(EditCutSelection)
  case choice = "&Copy"  : self.action(EditCopySelection)
  case choice = "&Paste" : self.action(EditPaste)
endSwitch
endMethod
```

This code is attached to the built-in **depart** method for *thisMemoField*. When the user leaves *thisMemoField*, this code removes the menu. In this example, the default menus reappear when the user moves off the field. In a similar situation, you might want to display another custom menu structure.

```
; thisMemoField::depart
method depart(var eventInfo MoveEvent)
removeMenu()          ; remove the Edit menu
endMethod
```

reason method

See also Example MenuEvent Type

Reports the type of menu chosen.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report why a MenuEvent occurred. MenuEvent reasons occur when a built-in **menuAction** method is called. ObjectPAL provides MenuReasons constants for testing the value returned by **reason**.

▪

reason example

In the following example, the form's **menuAction** method examines every MenuEvent to determine the reason for the MenuEvent. The reason is then displayed in the *menuReasonField* field object.

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  reasonStr String
endVar
  if eventInfo.isPreFilter() then
    ; sort out the reason, and assign equivalent string to reasonStr
    reasonStr = iif(eventInfo.reason() = MenuNormal, "MenuNormal",
                iif(eventInfo.reason() = MenuControl, "MenuControl",
                 "MenuDesktop"))
    reasonId = eventInfo.reason()
    menuReasonField = String(reasonId) + " " + reasonStr
     ; Code here executes before each object
  else
     ; Code here executes afterwards (or for form)

  endif
endMethod
```

▪

setData method

See also MenuEvent Type

Specifies information about a MenuEvent.

Syntax

setData **(** const ***menuData*** LongInt **)**

Description

setData should be used by Windows programmers only. **setData** specifies the *lParam* argument (usually zero) of specific Windows messages, such as WM_SYSCOMMAND and WM_COMMAND. See your Windows programming documentation for more information.

▪

setId method

See also Example MenuEvent Type

Specifies the ID of a MenuEvent.

Syntax

```
setId ( const commandId SmallInt )
```

Description

setId specifies in *commandId* an action to take as the result of a menu choice, where *commandId* is a MenuCommands constant.

If you change the ID for a MenuEvent with **setId**, you may also need to change the reason for that MenuEvent with **setReason**.

Note: In many circumstances, you should use **menuAction** from the Form type or UIObject type to invoke a menu command. Although it is possible to change the reason and ID for an existing MenuEvent (*eventInfo*), and it is also possible to create a new MenuEvent and set the reason and ID for that event (only advanced users should try this), this technique is not always advisable.

▪

setId example

See the example for **id.**

▪

setReason method

See also Example MenuEvent Type

Specifies a reason for generating a MenuEvent.

Syntax

setReason (const ***reasonId*** SmallInt **)**

Description

setReason specifies in *reasonId* a reason for generating a MenuEvent, where *reasonId* is a MenuReasons constant.

Note: In many circumstances, you should use **menuAction** from the Form type or UIObject type to invoke a menu command. Although it is possible to change the reason and ID for an existing MenuEvent (*eventInfo*), and it is also possible to create a new MenuEvent and set the reason and ID for that event (only advanced users should try this), this technique is not always advisable.

▪

setReason example

See the example for **id.**

MouseEvent type

A MouseEvent object answers questions about the mouse, including

- Where is the mouse?
- Was a mouse button clicked?
- Which mouse button was clicked or held down during an operation?

The following built-in object variables can be useful when working with MouseEvents: lastMouseClicked and lastMouseRightClicked.

Many methods defined for the MouseEvent type use or return Point values. Methods defined for the Point type get and set information about screen coordinates and relative positions of points. For example, the size and position properties of a design object are specified in points.

Note: ObjectPAL calculates point values relative to the container of the design object in question. For example, if a box contains a button, ObjectPAL calculates the button's position relative to the box. If the button sits in an empty page, ObjectPAL calculates the button's position relative to the page. Methods that take or return Point values as arguments use this relative framework. The method **convertPointWithRespectTo** defined for the UIObject type is useful for converting values in different frameworks.

The following built-in event methods are triggered by MouseEvents: **mouseClick**, **mouseDown**, **mouseUp**, **mouseDouble**, **mouseRightUp**, **mouseRightDown**, **mouseRightDouble**, **mouseMove**, **mouseEnter**, and **mouseExit**.

The MouseEvent type includes several derived methods from the Event type.

Methods for the MouseEvent type

Event	**MouseEvent**
errorCode	**getMousePosition**
getTarget	**getObjectHit**
isFirstTime	**isControlKeyDown**
isPreFilter	**isFromUI**
isTargetSelf	**isInside**
reason	**isLeftDown**
setErrorCode	**isMiddleDown**
setReason	**isRightDown**
	isShiftKeyDown
	setControlKeyDown
	setInside
	setLeftDown
	setMiddleDown
	setMousePosition
	setRightDown
	setShiftKeyDown
	setX
	setY

$\underline{x}$

$\underline{y}$

▪

getMousePosition method

See also Example MouseEvent Type

Returns the mouse position as a Point.

Syntax

```
1. getMousePosition ( var p Point )
2. getMousePosition ( var xPosition LongInt, yPosition LongInt )
```

Description

getMousePosition returns the mouse position. This method gets the mouse position at the time the method was called. It doesn't track subsequent mouse movements.

Syntax 1 stores the value in a Point variable, *p*. When you use syntax 1, you can use Point type methods (for example, **isLeft** and **isRight**) to get more information.

Syntax 2 stores the value in *xPosition* and *yPosition*, two LongInt variables representing the x- and y-coordinates of the mouse pointer.

▪

getMousePosition example

The following example gets the position of the last **mouseUp** event and draws a small circle at that position. The method first checks if the source of the event was from the UI (in this case, from the user), and if the target of the event is the page itself (as opposed to whether it was bubbled up to the page from some other object). This method draws the circle only when the user clicks on the page.

```
; pageOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  crObj UIObject
  x, y  LongInt    ; point coordinates
endVar
if eventInfo.isFromUI() AND eventInfo.isTargetSelf() then
  ; create a small blue circle at the mouse position
  eventInfo.getMousePosition(x, y)
  crObj.create(ellipseTool, x, y, 1440, 1440)
  crObj.Color = DarkBlue
  crObj.Visible = True
endif
endMethod
```

▪

getObjectHit method

See also Example MouseEvent Type

Creates a handle to the UIObject that received the event.

Syntax

getObjectHit (var ***target*** UIObject **)** Logical

Description

getObjectHit returns in *target* a handle to the UIObject that was clicked. This method is useful for the internal MouseEvents that call the built-in event methods **mouseExit** and **mouseEnter**. **getObjectHit** can return a different object than **getTarget** during a **mouseExit** or **mouseEnter** method.

getObjectHit example

The following method is attached to the **mouseExit** method of a form. When the mouse exits an object, a message appears in the status window showing the name of the target object (**getTarget**) vs. the name of the object hit (**getObjectHit**).

```
; thisForm::mouseExit
method mouseExit(var eventInfo MouseEvent)
var
  targObj,
  hitObj   UIObject
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    eventInfo.getTarget(targObj)
    eventInfo.getObjectHit(hitObj)
    message(targObj.Name + " vs. " + hitObj.Name)
  else
    ;code here executes just for form itself

endif
endMethod
```

isControlKeyDown method

See also   Example   MouseEvent Type

Reports whether the user has held (or is holding) down Ctrl during a MouseEvent.

Syntax

```
isControlKeyDown ( ) Logical
```

Description

isControlKeyDown returns True if Ctrl is held down during a MouseEvent; otherwise, it returns False.

isControlKeyDown example

The following example examines the keyboard state during a mouse click to determine whether to automatically insert the highest value in the range, the lowest value in a range, or the default value. The following constants are declared in the Const window for *fieldOne*:

```
; fieldOne::Const
Const
  HighRangeVal = Number(10000)
  LowRangeVal = Number(100000)
  DefaultVal = Number(50000)
endConst
```

This is the method for **mouseUp** for *fieldOne*:

```
; fieldOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
; insert high, low, or default value depending on how mouse was clicked
switch
  case eventInfo.isControlKeyDown() : self.Value = LowRangeVal
                                      message("Ctrl-click")
  case eventInfo.isShiftKeyDown()   : self.Value = HighRangeVal
                                      message("Shift-click")
  otherwise                         : self.Value = LowRangeVal
                                      message("Click")
endswitch
endMethod
```

▪

isFromUI method

See also Example MouseEvent Type

Reports whether an event was generated by the user interacting with Paradox.

Syntax

`isFromUI ( ) Logical`

Description

isFromUI reports whether an event was generated by the user interacting with Paradox, or internally (for example, by an ObjectPAL statement). This method returns True if the event was generated by the user; otherwise, it returns False.

isFromUI example

Sometimes you need to know whether a mouseEvent was generated by the user interacting with the form or by ObjectPAL; for example, in a computer tutorial. In the following example, **isFromUI** is used to determine whether a button's built-in **mouseEnter** method was triggered by the user or by ObjectPAL.

```
;btnOpenCust :: mouseEnter
method mouseEnter(var eventInfo MouseEvent)
   if eventInfo.isFromUI() then
      message("This button opens the customer form.")
   else
      message("After you press this button, the customer form opens.")
   endIf
endMethod
```

▪

isInside method

See also Example MouseEvent Type

Reports whether the mouse is inside the border of the target object.

Syntax

`isInside ( ) Logical`

Description

isInside reports whether the mouse is inside the border of the target object at the time of the event.

isInside example

In the following example, the **mouseUp** method for *buttonOne* reports whether the last event is inside the borders of the target object. If you click *buttonOne*, the **mouseUp** MouseEvent is delivered to *buttonOne* and **isInside** returns True. If you drag from inside the button to outside the button, so that the **mouseUp** occurs outside of the borders of *buttonOne*, the MouseEvent occurs for *buttonOne*, and triggers the **mouseUp** method, but **isInside** returns False for that MouseEvent.

```
; buttonOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
msgInfo("Is the last event inside ?", eventInfo.isInside())
endMethod
```

▪

isLeftDown method

See also Example MouseEvent Type

Reports whether the left (or primary) mouse button is held down during a MouseEvent.

Syntax

```
isLeftDown ( ) Logical
```

Description

isLeftDown returns True if the left mouse button is held down during a MouseEvent, for instance, while dragging the mouse; otherwise, it returns False.

▪

isLeftDown example

In the following example, assume that the *Site Notes* field from the *Sites* table is placed on a form. This method, attached to the **mouseMove** method for *Site Notes*, checks whether the left or right button is down at the time of the move. If the left button is down, the field is selected from the point of the click to the beginning of the field. If the right button is down, the field is selected from the point of the click to the end of the field.

```
; Site Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isLeftDown() then
  self.action(SelectTop)            ; select from point to beginning
else
  if eventInfo.isRightDown() then
    self.action(SelectBottom)       ; select from point to end
  endif
endif
endMethod
```

-

isMiddleDown method

See also Example MouseEvent Type

Reports whether the middle mouse button is held down during a MouseEvent.

Syntax

`isMiddleDown ( ) Logical`

Description

isMiddleDown returns True if the middle mouse button is held down during a MouseEvent; otherwise (even if there is no middle mouse button), it returns False.

isMiddleDown example

The following example assumes that a form contains a button called *sendMove*, and a field from the *Sites* table called *Site Notes*. The **pushButton** method for *sendMove* constructs a MouseEvent with the middle button down, then sends the MouseEvent off to the *Site Notes* field.

```
; sendMove::pushButton
method pushButton(var eventInfo Event)
var
  mo  MouseEvent         ; declare a MouseEvent to send
  ui  UIObject
endVar
ui.attach("Site Notes")  ; attach to Site Notes
mo.setMiddleDown(Yes)    ; set middle button down on MouseEvent
ui.mouseMove(mo)         ; dispatch event to mouseMove for Site Notes
endMethod
```

This method is attached to the **mouseMove** method for *Site Notes*. If the middle button is down for the MouseEvent, the method moves to the beginning of the current word, then selects the entire word.

```
; Site_Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isMiddleDown() then
  self.action(MoveLeftWord)    ; go to the beginning of the word
  self.action(SelectRightWord) ; select the entire word
endif
endMethod
```

▪

isRightDown method

See also Example MouseEvent Type

Reports whether the right mouse button is pressed during a MouseEvent.

Syntax

`isRightDown ( ) Logical`

Description

isRightDown returns True if the right (or alternate) mouse button is held down during a MouseEvent, for instance, while right-dragging; otherwise, it returns False.

isRightDown example

In the following example, assume that the *Site Notes* field from the *Sites* table is placed on a form. The **mouseMove** method for *Site Notes* checks whether the left or right mouse button is down at the time of the move. If the left button is down, the field is selected from the point of the click to the beginning of the field; if the right button is down, the field is selected from the point of the click to the end of the field.

```
; Site Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isLeftDown() then
  self.action(SelectTop)            ; select from point to beginning
else
  if eventInfo.isRightDown() then
    self.action(SelectBottom)       ; select from point to end
  endif
endif
endMethod
```

isShiftKeyDown method

See also Example MouseEvent Type

Reports whether Shift is held down during a MouseEvent.

Syntax

`isShiftKeyDown ( ) Logical`

Description

isShiftKeyDown returns True if Shift is held down during a MouseEvent; otherwise, it returns False.

isShiftKeyDown example

The following example is attached to the **mouseUp** method for the *Site Notes* field. When the user presses Shift while clicking, the word to the right of the insertion point is selected.

```
; Site Notes::mouseUp
method mouseUp(var eventInfo MouseEvent)
;if Shift is down, select the word to the right
if eventInfo.isShiftKeyDown() then
   self.action(SelectRightWord)
endif
endMethod
```

▪

setControlKeyDown method

See also Example MouseEvent Type

Simulates pressing and holding Ctrl during a MouseEvent.

Syntax

setControlKeyDown (const *yesNo* Logical)

Description

setControlKeyDown adds information about the state of Ctrl for a MouseEvent. You must specify Yes or No. Yes means Ctrl was pressed and held during a MouseEvent; No means Ctrl was not pressed.

setControlKeyDown example

The following example creates a MouseEvent and sets Ctrl to Yes. The event is then sent to the **mouseUp** built-in event method for a field called *lcField*. This method is attached to the **pushButton** method for a button named *sendCtrl*.

```
; sendCtrl::pushButton
method pushButton(var eventInfo Event)
var
  ctrlMsEvent MouseEvent             ; declare the event
endVar

ctrlMsEvent.setControlKeyDown(Yes)   ; set the Control key
lcField.mouseUp(ctrlMsEvent)         ; send the event
endMethod
```

This code is attached to the **mouseUp** method for *lcField*. This method checks whether Ctrl is pressed when the mouse is clicked. If so, the value in the field is changed to all lowercase.

```
; lcField::mouseUp
method mouseUp(var eventInfo MouseEvent)
if eventInfo.isControlKeyDown() then   ; check for Control key
  self.Value = lower(self.Value)       ; change to lowercase
endif
endMethod
```

▪

setInside method

See also Example MouseEvent Type

Sets the mouse to be inside the current object.

Syntax

`setInside ( const TrueFalse Logical ) Logical`

Description

setInside sets the MouseEvent to be inside the current object.

setInside example

In the following example, the **mouseUp** method for *sendAnEvent* uses **setInside** to change the *eventInfo* variable, then sends the event to *buttonOne*.

```
; sendAnEvent::mouseUp
method mouseUp(var eventInfo MouseEvent)
eventInfo.setInside(Yes)
buttonOne.mouseUp(eventInfo)
endMethod
```

setLeftDown method

See also Example MouseEvent Type

Simulates pressing the left mouse button.

Syntax

`setLeftDown ( const yesNo Logical )`

Description

setLeftDown adds information about the state of the left mouse button for a MouseEvent. You must specify Yes or No. Yes means the left button was clicked; No means the left button was not clicked.

setLeftDown example

The following example constructs a MouseEvent with the left button set down. The MouseEvent is then sent to the **mouseMove** method for *Site_Notes*. This code is attached to the **pushButton** method for *sendLeftButton*:

```
; sendLeftButton::pushButton
method pushButton(var eventInfo Event)
var
  leftMoveMouse MouseEvent     ; create the mouse event
  ui            UIObject
endVar
leftMoveMouse.setLeftDown(Yes) ; set Left button to Yes
ui.attach("Site_Notes")
ui.mouseMove(leftMoveMouse)    ; send the event to Site_Notes
endMethod
```

This code is attached to the **mouseMove** method for *Site Notes*:

```
; Site_Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isLeftDown() then
  self.action(SelectTop)            ; select from point to beginning
else
  if eventInfo.isRightDown() then
    self.action(SelectBottom)       ; select from point to end
  endif
endif
endMethod
```

setMiddleDown method

See also Example MouseEvent Type

Simulates pressing the middle mouse button.

Syntax

```
setMiddleDown ( const yesNo Logical )
```

Description

setMiddleDown adds information about the state of the middle mouse button for a MouseEvent. You must specify Yes or No. Yes means the middle button was clicked; No means the middle button was not clicked.

setMiddleDown example

The following example assumes that a form contains a button called *sendMove* and a field object from the *Sites* table called *Site_Notes*. The **pushButton** method for *sendMove* constructs a MouseEvent with the middle button down, then sends MouseEvent to the *Site_Notes* field object.

```
; sendMove::pushButton
method pushButton(var eventInfo Event)
var
  mo  MouseEvent         ; declare a MouseEvent to send
  ui  UIObject
endVar
ui.attach("Site_Notes")  ; attach to Site_Notes
mo.setMiddleDown(Yes)    ; set middle button down on MouseEvent
ui.mouseMove(mo)         ; dispatch event to mouseMove for Site Notes
endMethod
```

This method is attached to the **mouseMove** method for *Site_Notes*. If the middle button is down for the MouseEvent, the method moves to the beginning of the current word, then selects the entire word.

```
; Site_Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isMiddleDown() then
  self.action(MoveLeftWord)    ; go to the beginning of the word
  self.action(SelectRightWord) ; select the entire word
endif
endMethod
```

▪

setMousePosition method

See also Example MouseEvent Type

Sets the position of the mouse for an event.

Syntax

```
1. setMousePosition ( const xPosition LongInt, const yPosition  LongInt )
2. setMousePosition ( const p Point )
```

Description

setMousePosition adds information about the position of the mouse for a MouseEvent. *xPosition* and *yPosition* specify the x- and y-coordinates in twips, relative to the upper left corner of the target object's container.

setMousePosition example

The following example creates a new event, sets the mouse position to 500 twips to the right and below the current mouse position, and sends the event to the **mouseRightUp** method for the same object. This code is attached to the **mouseUp** method for an object called *boxOne*:

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  rightEvent MouseEvent
endVar
; set the new position to current plus 500, 500
rightEvent.setMousePosition(eventInfo.x() + 500,
                            eventInfo.y() + 500)
mouseRightUp(rightEvent)            ; send off the new event
endMethod
```

▪

setRightDown method

See also Example MouseEvent Type

Simulates pressing the right mouse button.

Syntax

`setRightDown ( const yesNo Logical )`

Description

setRightDown adds information about the state of the right mouse button for a MouseEvent. You must specify Yes or No. Yes means the right button was clicked; No means the right button was not clicked.

▪

setRightDown example

The following example constructs a MouseEvent with the right button set down. The MouseEvent is then sent to the **mouseMove** method for *Site_Notes*. This code is attached to the **pushButton** method for *sendRightButton*:

```
; sendRightButton::pushButton
method pushButton(var eventInfo Event)
var
  rightMoveMouse MouseEvent      ; declare the event
  ui             UIObject
endVar
rightMoveMouse.setRightDown(Yes) ; set right button down
ui.attach("Site_Notes")
ui.mouseMove(rightMoveMouse)     ; send the event to Site Notes
endMethod
```

This code is attached to the **mouseMove** method for *Site_Notes*:

```
; Site_Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isLeftDown() then
  self.action(SelectTop)            ; select from point to beginning
else
  if eventInfo.isRightDown() then
    self.action(SelectBottom)       ; select from point to end
  endif
endif
endMethod
```

▪

setShiftKeyDown method

See also Example MouseEvent Type

Simulates pressing and holding Shift.

Syntax

setShiftKeyDown (`const` *yesNo* `Logical`)

Description

setShiftDown adds information about the state of Shift for a MouseEvent. You must specify Yes or No. Yes means Shift was pressed and held; No means Shift wasn't pressed.

▪

setShiftKeyDown example

The following example creates a MouseEvent and sets Shift to Yes. The event is then sent to the **mouseUp** built-in event method for a field called *ucField*. This method is attached to the **pushButton** method for a button named *sendShift*.

```
; sendShift::pushButton
method pushButton(var eventInfo Event)
var
  shiftMsEvent MouseEvent              ; declare the event
endVar

shiftMsEvent.setShiftKeyDown(Yes)      ; set the Shift key
ucField.mouseUp(shiftMsEvent)          ; send the event

endMethod
```

This code is attached to the **mouseUp** method for *ucField*. This method checks whether Shift is pressed when the mouse is clicked. If so, the value in the field is changed to all uppercase.

```
; ucField::mouseUp
method mouseUp(var eventInfo MouseEvent)
if eventInfo.isShiftKeyDown() then     ; check for Shift key
  self.Value = upper(self.Value)       ; change to uppercase
endif
endMethod
```

▪

setX method

See also Example MouseEvent Type

Specifies the horizontal coordinate of the mouse pointer position.

Syntax

`setX ( ``const `***`xPosition`***` LongInt `**`)`**

Description

setX sets the horizontal coordinate (in twips) of the mouse pointer position to *xPosition*. Coordinates must be specified relative to the upper-left corner of the current object.

▪

setX example

The following example involves two methods for the same object, *boxOne*. The **mouseUp** method creates a MouseEvent, setting the coordinates to 500 twips greater than the point of the click. The **mouseUp** method then sends the event to **mouseRightUp**. The **mouseRightUp** method gets the coordinates, converts them so they are placed properly on *boxOne*, and draws a box at the point indicated by the MouseEvent. If the MouseEvent is the result of a user interaction (**isFromUI** returns True), the new box is painted Red. If the MouseEvent is not the result of a user interaction, as when the event is passed from the **mouseUp** method, the new box is painted Green. The **mouseUp** method for *boxOne* is:

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  rightEvent MouseEvent
endVar
; set the new position to current plus 500, 500
rightEvent.setX(eventInfo.x() + 500)
rightEvent.setY(eventInfo.y() + 500)
mouseRightUp(rightEvent)            ; send off the new event
endMethod
```

This code is attached to the **mouseRightUp** method for *boxOne*:

```
; boxOne::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  ui     UIObject     ; to create object at point of click
  msPt   Point        ; the x, y point of click
endVar

; get the x and y coordinates of the click
msPt = Point(eventInfo.x(), eventInfo.y())

; convert the point from the page to the box
self.convertPointWithRespectTo(pageOne, msPt, msPt)

; create the box, color it, and set it to visible
ui.create(boxTool, msPt.x(), msPt.y(), 200, 200)
ui.Visible = True
if eventInfo.isFromUI() then
  ui.Color = Red          ; native event
else
  ui.Color = Green        ; mouse event passed from mouseUp
endif
endMethod
```

▪

setY method

See also Example MouseEvent Type

Specifies the vertical coordinate of the mouse pointer position.

Syntax

```
setY ( const yPosition LongInt )
```

Description

setY sets the vertical coordinate (in twips) of the mouse pointer position to *yPosition*. Coordinates must be specified relative to the upper left corner of the current object.

▪

setY example

See the example for **setX**.

▪

x method

See also Example MouseEvent Type

Returns the horizontal coordinate of the mouse pointer position.

Syntax

x ( ) LongInt

Description

x returns (in twips) the horizontal coordinate of the mouse pointer position.

-

x example

See the example for **setX**.

■

y method

See also Example MouseEvent Type

Returns the vertical coordinate of the mouse pointer position.

Syntax

y ( ) LongInt

Description

y returns (in twips) the vertical coordinate of the mouse pointer position.

▪

y example

See the example for **setX.**

MoveEvent type

Methods for the MoveEvent type enable you to get and set information about the events that occur as you navigate from one object to another in a form.

The following built-in event methods are triggered by MoveEvents: **arrive**, **canArrive**, **canDepart**, and **depart**.

The MoveEvent type includes several derived methods from the Event type.

Methods for the MoveEvent type

Event	**MoveEvent**
errorCode	**getDestination**
getTarget	**reason**
isFirstTime	**setReason**
isPreFilter	
isTargetSelf	
setErrorCode	

▪

getDestination method

See also Example MoveEvent Type

Reports which object is the destination of a move.

Syntax

`getDestination ( var dest UIObject )`

Description

getDestination returns in *dest* the object that Paradox is trying to move to in a form.

getDestination example

In the following example, assume that the form contains a multi-record object bound to the *Orders* table. The **canDepart** method for the form is called whenever the user attempts to move off a field or other object in the form. The **canDepart** method shown in this example uses **getDestination** to find the intended destination of the MoveEvent. This method uses **getTarget** to find the source of the move and compare it with the destination.

If the containers of the two objects are the same, such as when the user is moving from one field to the next in a multi-record object, the method displays a dialog box asking for confirmation. When the user responds, the move occurs and the field the user moved from is set to yellow. If the target's container and the destination's container are different, such as when the user is attempting to leave the form altogether, the method doesn't display the dialog box. The following code is attached to the **canDepart** method for a form:

```
; thisForm::canDepart
method canDepart(var eventInfo MoveEvent)
var
  destObj UIObject
  targObj UIObject
  doMove  String
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    eventInfo.getTarget(targObj)
    eventInfo.getDestination(destObj)
    if targObj.ContainerName = destObj.ContainerName then
      ; handle only field-to-field moves within the MRO
      doMove = msgQuestion("Move?", "Move to " + destObj.name + " ?")
      if doMove = "No" then
        eventInfo.setErrorCode(CanNotDepart)
      else
        targObj.Color = Yellow    ; leave a trail of yellow fields
      endIf
    endIf
  else
    ;code here executes just for form itself

endIf
endMethod
```

reason method

See also Example MoveEvent Type

Reports why a move occurred.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report why a MoveEvent occurred. MoveEvent reasons occur when a built-in **arrive**, **depart**, **canArrive**, or **canDepart** method is called. ObjectPAL provides MoveReasons constants for testing the value returned by **reason**.

reason example

In the following example, assume a form contains two field objects, *fieldOne* and *fieldTwo*, and a button named *moveToFieldOne*. A move away from *fieldOne* is treated as normal; however, to return to *fieldOne*, the user must press the *moveToFieldOne* button. The **canArrive** method for *fieldOne* checks the reason for the move, and blocks field arrival if the reason is not UserMove. The following code is attached to the **canArrive** method for *fieldOne*:

```
; fieldOne::canArrive
method canArrive(var eventInfo MoveEvent)
; don't allow user to move to field by tabbing or clicking
if eventInfo.reason() = UserMove then
  eventInfo.setErrorCode(CanNotArrive)
  beep()
  message("Press the Move to Field One button to move to Field One.")
endIf
endMethod
```

The following code is attached to the **pushButton** method for *moveToFieldOne*:

```
; moveToFieldOne::pushButton
method pushButton(var eventInfo Event)
; move to fieldOne if it does not currently have focus
if fieldOne.Focus = False then
  fieldOne.moveTo()
else
  fieldTwo.moveTo()
endIf
endMethod
```

▪

setReason method

See also Example MoveEvent Type

Specifies a reason for a Move Event.

Syntax

setReason (const ***reasonId*** SmallInt **)**

Description

setReason specifies a reason for generating a MoveEvent. This method takes a MoveReasons constant as an argument.

setReason example

In the following example, the **canArrive** method for *fieldOne* blocks field arrival if the reason for the move is UserMove. To temporarily circumvent this restriction, the form's **canArrive** method changes the reason for UserMove events to PalMove events. This code is attached to the **canArrive** method for *fieldOne*:

```
; fieldOne::canArrive
method canArrive(var eventInfo MoveEvent)
; don't allow user to move to field by tabbing or clicking
if eventInfo.reason() = UserMove then
  eventInfo.setErrorCode(CanNotArrive)
  beep()
  message("Press the Move to Field One button to move to Field One.")
endIf
endMethod
```

This code is attached to the **canArrive** method for the form:

```
; thisForm::canArrive
method canArrive(var eventInfo MoveEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    ; change events with a reason of UserMove to PalMove
    if eventInfo.reason() = UserMove then
      eventInfo.setReason(PalMove)
    endIf
  else
    ;code here executes just for form itself

endIf
endMethod
```

Number type

Number variables represent floating-point values consisting of a significand (fractional portion, for example, 3.224) multiplied by a power of 10. The significand can contain up to 18 significant digits, and the power of 10 can range from ± 3.4E-4930 to ± 1.1E4930. An attempt to assign a value outside of this range to a Number variable causes an error.

Note: ObjectPAL supports an alternate syntax:

methodName (*objVar*, *argument* [, *argument*])

methodName represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return the sine of a number:

```
theNum.sin()
```

This statement uses the alternate syntax:

```
sin(theNum)
```

It's best to use standard syntax for clarity and consistency, but you can use the alternate syntax wherever it's convenient.

Note: The display formats of numeric method may vary depending on the Windows number format of the user's system, but ObjectPAL's internal representation is always the same.

Note: Run-time library methods and procedures defined for the Number type also work with LongInt and SmallInt variables. The syntax is the same, and the returned value is a Number. For example, the following code works, even though **sin** does not appear in the list of methods for the LongInt type:

```
var
   abc LongInt
   xyz Number
endVar
abc = 43
xyz = abc.sin()
```

The Number type includes several derived methods from the AnyType type.

Methods for the Number type

AnyType	Number
blank	**abs**
dataType	**acos**
isAssigned	**asin**
isBlank	**atan**
isFixedType	**atan2**
view	**ceil**
	cos
	cosh
	exp
	floor
	fraction
	fv
	ln

▪

abs method

See also Example Number Type
Beginner

Returns the absolute value of a number.

Syntax
`abs ( ) Number`

Description
abs removes the sign from a numeric value

▪

abs example

For the following example, assume that a form contains three field objects: *forecastAmt*, *actualAmt*, and *diffPercent*. The **newValue** method for *actualAmt* finds the difference between *forecastAmt* and *actualAmt*, then calculates how far off the forecast was. Depending on the values in *forecastAmt* and *actualAmt*, the number by which they differ can be positive or negative. To find the percentage of error, **abs** is used to get the absolute value of the number, which is then multiplied by 100 to get a percentage. This code is attached to the **newValue** method for *actualAmt*:

```
; actualAmt::newValue
method newValue(var eventInfo Event)
var
  difference  Number
endVar
; don't execute if newValue is being called at startup, or
; if one of the fields involved is blank
if eventInfo.reason() <> StartupValue then
  if NOT self.isBlank() AND
     NOT forecastAmt.isBlank() then
    ; find out how much forecast differs from actual
    difference = (forecastAmt - Number(self.Value)) / forecastAmt
    diffPercent = difference.abs() * 100  ; get the variation as
                                          ; an absolute value
  else
    msgStop("Error", "The forecastAmt field can't be blank.")
  endIf
endIf
endMethod
```

▪

acos method

See also   Example   Number Type
Beginner

Returns the 2-quadrant arc cosine of a number.

Syntax
acos ( ) Number

Description
Given a number between -1 and 1, **acos** returns a numeric value between 0 and pi, expressed in radians. **acos** is called the 2-quadrant arc cosine because it returns values within quadrants 1 and 4 (that is, between -pi/2 and pi/2). **acos** is the inverse of **cos** (if **acos**(*x*) = *y*, then **cos**(*y*) = *x*).

▪

acos example

The **pushButton** method for the *findArcCos* button calculates and displays the arc cosine of a value entered by the user.

```
; findArcCos::pushButton
method pushButton(var eventInfo Event)
   var
      nuUserVal,
      nuArcCos   Number
      stPrompt   String
   endVar

   stPrompt = "Enter a number from -1 to 1"
   nuUserVal = 0

   nuUserVal.view(stPrompt)
   if (nuUserVal >= -1) and (nuUserVal <= 1) then
      nuArcCos = nuUserVal.acos()
      nuArcCos.view("Arc cosine of " + String(nuUserVal))
   else
      msgStop("You entered: " + String(nuUserVal), stPrompt)
   endIf
endMethod
```

▪

asin method

See also Example Number Type
Beginner

Returns the 2-quadrant arc sine of a number.

Syntax
`asin ( ) Number`

Description
Given a number between -1 and 1, **asin** returns a numeric value between -pi/2 and pi/2, expressed in radians. **asin** is the inverse of **sin** (if **asin**(*x*) = *y*, then **sin**(*y*) = *x*).

asin example

In the following example, the **pushButton** method for the *findASin* button displays the arc sine of a number.

```
; findASin::pushButton
var
  x Number
endvar
x = .5
msgInfo("arc sine of .5", x.asin())   ; displays .52
endMethod
```

▪

atan method

See also Example Number Type
Beginner

Returns the 2-quadrant arctangent of a number.

Syntax
`atan ( )` `Number`

Description
Given a tangent in radians, **atan** returns the angle in radians. **atan** is called the 2-quadrant arctangent because it returns values within quadrants 1 and 4 (that is, between -pi/2 and pi/2). **atan** is the inverse of **tan** (if atan(*x*) = *y*, then tan(*y*) = *x*).

atan example

In the following example, the **pushButton** method for *getAtan* calculates the 2-quadrant arctangent of *x* and *y*, then displays the result.

```
; getAtan::pushButton
method pushButton(var eventInfo Event)
var
  x   Number
  checkPi, fortyFiveDegrees Number
endvar
x = 1
fortyFiveDegrees = x.atan()
msgInfo("45 degrees in radians: ", fortyFiveDegrees)  ; 0.79
checkPi = fortyFiveDegrees * 4       ; pi radians = 180 degrees
msgInfo("pi: ", format("w12.10", checkPi))
endMethod
```

atan2 method

See also Example Number Type
Beginner

Returns the 4-quadrant arctangent of a number.

Syntax
`atan2 ( const x Number ) Number`

Description
Given a sine in radians, **atan2** returns an angle in radians whose cosine is x. **atan2** is called the 4-quadrant arctangent because it returns values in all four quadrants.

atan2 example

For the following example, assume that a form contains a button named *getAtan2*. The **pushButton** method for *getAtan2* calculates the 4-quadrant arctangent of *x* and *y*, then displays the results:

```
; getAtan2::pushButton
method pushButton(var eventInfo Event)
var
  x,
  y,
  checkpi,
  fortyFiveDegrees Number
endvar
x = 1                          ; The angle whose tangent is 1 / 1
y = 1                          ; is a 45 degree angle
fortyFiveDegrees = x.atan2(y)
msgInfo("45 degrees in radians: ", fortyFiveDegrees)  ; 0.79
checkpi = fortyFiveDegrees * 4.0      ; pi radians = 180 degrees
msgInfo("pi: ", format("w12.10", checkpi))
endMethod
```

▪

ceil method

[See also](#) [Example](#) [Number Type](#)
Beginner

Rounds a numeric expression up to the nearest whole number.

Syntax

`ceil ( )` `Number`

Description

ceil rounds a numeric expression up (toward positive infinity) to the nearest whole number.

ceil example

In the following example, the **pushButton** method for a button named *ceilVsRound* calculates the ceiling value of a number, then shows the rounded value of the same number:

```
; ceilVsRound::pushButton
method pushButton(var eventInfo Event)
var
  x  Number
endVar
x = 3.1
msgInfo("The ceil of " + String(x) + " is", ceil(x))     ; displays 4.0
msgInfo("The round of " + String(x) + " is", x.round(0)) ; displays 3
endMethod
```

▪

cos method

See also   Example   Number Type
Beginner

Returns the cosine of an angle.

Syntax
`cos ( )` Number

Description
cos returns a value between -1 and 1 for the cosine of a value or expression representing the size of the angle in radians.

cos example

In the following example, the **pushButton** method for the *findCosine* button calculates and displays the cosine of a 60-degree angle:

```
; findCosine::pushButton
method pushButton(var eventInfo Event)
var
  sixtyDegrees Number
endVar
sixtyDegrees = PI / 3.0
msgInfo("The cosine of 60 degrees", sixtyDegrees.cos()) ; displays 0.50
endMethod
```

▪

cosh method

See also Example Number Type
Beginner

Returns the hyperbolic cosine of an angle.

Syntax
`cosh ( )` `Number`

Description
cosh returns the hyperbolic cosine of a value or expression representing the size of the angle in radians. The formula used is

cosh(angle) = (exp(angle) + exp(-angle)) / 2

▪

cosh example

The **pushButton** method for the *findCosineH* button calculates and displays the *h* cosine of 60 degrees.

```
; findCosineH::pushButton
method pushButton(var eventInfo Event)
var
  sixtyDegrees Number
endVar
sixtyDegrees = PI / 3.0
msgInfo("The h cosine of " + format("W8.6", sixtyDegrees) + " radians",
         format("W14.12", sixtyDegrees.cosh()))
; displays 1.600286857702
endMethod
```

▪

exp method

See also Example Number Type
Beginner

Returns the exponential (base *e*) of a number.

Syntax
`exp ( ) Number`

Description
exp computes *e* to the *x* power, where *e* is the constant 2.7182845905, the so-called natural number. The return value is x, the exponent. The inverse method is the natural log, **ln.**

▪

exp example

In the following example, the **pushButton** method for a button named *getExponent* button calculates and displays the base *e* of 1:

```
; getExponent::pushButton
method pushButton(var eventInfo Event)
msgInfo("The exp of 1.0", format("W14.12", exp(1.0)))
; exp(1) formatted to display full precision
endMethod
```

▪

floor method

See also Example Number Type
Beginner

Rounds a numeric expression down to the nearest whole number.

Syntax
`floor ( )` `Number`

Description
floor rounds a numeric expression down (toward negative infinity) to the nearest whole number.

▪

floor example

In the following example, the **pushButton** method for a button named *floorVsRound* uses **floor** to round *x* down to the nearest integer. By comparison, for the same number, **round** results in a higher number.

```
; floorVsRound::pushButton
method pushButton(var eventInfo Event)
var
  x  Number
endVar
x = 3.9
msgInfo("The floor of " + String(x) + " is", floor(x))   ; displays 3.0
msgInfo("The round of " + String(x) + " is", x.round(0)) ; displays 4.0
endMethod
```

▪

fraction method

See also Example Number Type
Beginner

Returns the fractional part of a number.

Syntax
`fraction ( ) Number`

Description
fraction returns the fractional part of a number, the part to the right of the decimal.

▪

fraction example

In the following example, the **pushButton** method for *fractButton* displays the fraction portion of a numeric variable:

```
; fractButton::pushButton
method pushButton(var eventInfo Event)
var
  myNum Number
endVar
myNum = 12.23
msgInfo("Fractional part of " + String(myNum),
         myNum.fraction())                  ; displays .23
endMethod
```

▪

fv method

See also Example Number Type
Beginner

Returns the future value of a series of equal payments.

Syntax

```
fv ( const interestRate Number, periods Number ) Number
```

Description

fv returns the future value of a series of equal payment periods, invested at interest rate *interestRate*. *interestRate* is expressed as a decimal number (like .12), not as a percentage (12%). Make sure the rate period matches the deposit period; that is, if the deposits are monthly, the interest rate should be monthly too.

The formula used is

FV = payment(pow(1 + *rate*, *periods*) - 1) / *rate*

fv is sometimes called the future or compound value of an annuity because you can use it to calculate the amount accumulated in an annuity fund when making regular, equal payments over time.

▪

fv example

The following example calculates how much a 14.5% Individual Retirement Account would be worth if $166.67 were deposited every month for 30 years.

```
; findFutureVal::pushButton
method pushButton(var eventInfo Event)
var
  depositAmt,
  intRate,
  numPayments,
  iraValue       Number
endVar
intRate = .145 / 12      ; convert yearly interest to monthly interest
numPayments = 360        ; monthly payments for 30 years
depositAmt = 166.67      ; monthly deposit amount ($2000 a year)
iraValue = depositAmt.fv(intRate, numPayments)
msgInfo("IRA Value", "Depositing " + String(depositAmt) +
        " a month for " + String(numPayments/12) + " years at " +
        String(intRate * 12 * 100) + "% yields " + String(iraValue) +
        ". You'll be old but you'll be rich!")
; displays "Depositing 166.67 a month for 30 years
;           at 14.50% yields 1,027,394.23 ..."
endMethod
```

▪

ln method

See also Example Number Type
Beginner

Returns the natural logarithm of a numeric expression.

Syntax

ln ( ) Number

Description

ln calculates the natural logarithm to the base e of a positive value. The constant *e* is the natural number, approximated by the value 2.7182845905. If the value is 0 or negative, **ln** fails.

The inverse method is **exp.** Use **log** to compute base 10 logarithms.

▪

In example

In the following example, the **pushButton** method for the *findNatLog* button calculates and displays the natural logarithm of several numbers:

```
; findNatLog::pushButton
method pushButton(var eventInfo Event)
var
  x  Number
endVar
x = 2.71828
msgInfo("Natural log of " + Format("W10.6", x), ln(x)) ; displays 1.00
x = 7.3891
msgInfo("Natural log of " + Format("W10.6", x), ln(x)) ; displays 2.00
x = 20.0855
msgInfo("Natural log of " + Format("W10.6", x), ln(x)) ; displays 3.00
endMethod
```

▪

log method

See also Example Number Type
Beginner

Returns the base 10 logarithm of a numeric expression.

Syntax

`log ( )` Number

Description

log returns the base 10 logarithm of a value or numeric expression. If the value is 0 or negative, **log** fails.

Use **ln** to compute natural logarithms.

▪

log example

```
; findLog::pushButton
method pushButton(var eventInfo Event)
var
  x Number
endVar
x = 10
msgInfo("The logarithm of " + String(x), log(x)) ; displays 1.00
x = 100
msgInfo("The logarithm of " + String(x), log(x)) ; displays 2.00
x = 1000
msgInfo("The logarithm of " + String(x), log(x)) ; displays 3.00
endMethod
```

▪

max procedure

See also Example Number Type
Beginner

Returns the larger of two numbers.

Syntax
```
max ( const x1 AnyType, const x2 AnyType ) AnyType
```

Description
max returns the larger of the two values *x1* and *x2*.

▪

max example

In the following example, you want to find the medical deduction you're allowed for tax purposes. The **pushButton** method for *findMedDeduct* finds the maximum of 7.5% of *AGI* or *medExpense*, then deducts 7.5% of *AGI* from the result. Finding the maximum number first ensures that the calculation won't return a negative number.

```
; findMedDeduct
method pushButton(var eventInfo Event)
var
  medExpense,
  AGI          Number
endVar
AGI = 32000.45
medExpense = 4035.24
msgInfo("Allowed Medical Deduction",
        max(medExpense, AGI * .075) - (AGI * .075))  ; displays 1,635.21
; assumes that you can deduct only that part of your medical and dental
; expenses greater than 7.5% of Adjusted Gross Income
endMethod
```

▪

min procedure

See also Example Number Type
Beginner

Returns the smaller of two numbers.

Syntax

```
min ( const x1 AnyType, const x2 AnyType ) AnyType
```

Description

min returns the smaller of the two values, *x1* and *x2*.

▪

min example

In the following example, you want to calculate the maximum amount of tax-deductible charitable contributions, and no more than 30% of adjusted gross income can be deducted. The **pushButton** method for the *findCharityDeduct* button finds and displays the minimum of 30% of *AGI* and *charity*.

```
; findCharityDeduct::pushButton
method pushButton(var eventInfo Event)
var
  charity,
  AGI       Number
endVar
AGI = 32000.45     ; Adjusted Gross Income
charity = 12000    ; charitable contributions for the year
msgInfo("Allowed Charity Deduction", min(charity, AGI * .30))
; displays 9,600.13
; assumes charitable contributions up to 30% of AGI
; are allowed as deductions
endMethod
```

mod method

See also   Example   Number Type
Beginner

Returns the remainder when one number is divided by another.

Syntax

```
mod ( const modulo Number ) Number
```

Description

mod returns the remainder (or modulus) when a number is divided by the value of *modulo*. If the number is greater than the value of *modulo*, **mod** returns the remainder of the number divided by *modulo*. If the number is less than *modulo*, **mod** returns the number. If the number equals *modulo*, **mod** returns 0. The following table shows examples of each case.

Fraction	ObjectPAL code	Return value
5/2	`num = 5` `num.mod(2)`	1
2/5	`num = 2` `num.mod(5)`	2
2/2	`num = 2` `num.mod(2)`	0

▪

mod example

In the following example, the **pushButton** method for the *showRemainder* button calculates and displays the modulus for a series of division operations:

```
; showRemainder::pushButton
method pushButton(var eventInfo Event)
var
  x Number
endVar
x = 8
msgInfo("The remainder of " + String(x) + "/" + "3",
          x.mod(3))                    ; displays 2
msgInfo("The remainder of " + String(x) + "/" + "12",
         x.mod(12))                    ; displays 8
x = -2
msgInfo("The remainder of " + String(x) + "/" + "10",
          x.mod(10))                   ; displays -2
x = -10
msgInfo("The remainder of " + String(x) + "/" + "-100",
        x.mod(-100))                   ; displays -10
endMethod
```

▪

number procedure

See also Example Number Type
Beginner

Casts a value as a Number.

Syntax

`number ( const value AnyType ) Number`

Description

number casts (converts) *value* to a Number. *value* must be in the form of a valid number that can be entered in a field. number is used to cast a non-numeric type to a Number when a numeric operand is required in an expression, or a numeric argument is required in a procedure or method. **number** behaves the same as **numVal**.

▪

number example

In the following example, a variable *x* is declared as a String, then assigned a string of numbers. The **pushButton** method for the *showDouble* button casts *x* to a Number before doubling it, then displays the result:

```
; showDouble::pushButton
method pushButton(var eventInfo Event)
var
  x  String
endVar
x = "1123.54"
; cast x to a Number before multiplying by 2
msgInfo("Double " + x + " is", Number(x) * 2) ; displays 2,247.08
endMethod
```

▪

numVal procedure

See also Example Number Type

Casts a value as a Number.

Syntax

```
numVal ( const value AnyType ) Number
```

Description

numVal casts (converts) *value* to a Number. *value* must be in the form of a valid number that can be entered in a field. **numVal** is most often used to cast a non-numeric type to a Number when a numeric operand is required in an expression, or a numeric argument is required in a procedure or method. **numVal** behaves the same as **number.**

■

numVal example

In the following example, a variable *x* is declared as a String, then assigned a string of numbers. The **pushButton** method for the *showDouble* button casts *x* to a Number before doubling it, then displays the result:

```
; showDouble::pushButton
method pushButton(var eventInfo Event)
var
  x  String
endVar
x = "1123.54"
; cast x to a Number before multiplying by 2
msgInfo("Double " + x + " is", numVal(x) * 2) ; displays 2,247.08
endMethod
```

▪

pmt method

See also Example Number Type
Beginner

Returns the periodic payment required to pay off a loan.

Syntax

pmt (const ***interestRate*** Number, const ***periods*** Number **)** Number

Description

pmt returns the constant, regular payment required to amortize (pay off) a loan. The formula used is:

PMT = *p* * *i* / (1 - (1 + *i*) ^-t)

where *p* = principal amount, *i* = effective interest rate per period, and *t* = term of the loan (number of payment periods).

Payments are considered due at the end of each period.

pmt works for amortization-type loans (for example, conventional home mortgages), in which part of the payment consists of interest on the remaining principal, and the remainder pays off part of the principal of the loan. **pmt** does not work for consumer-type loans, such as repayments of credit accounts or automobile loans.

The interest rate used in **pmt** is expressed in *interestRate* as a decimal number (like .12), not as a percentage (12%). Make sure the rate period matches the payment periods; that is, if the payments are monthly, the interest rate should also be monthly. Since the interest rate usually quoted for amortization loans (mortgages) is annual, divide it by 12 for monthly payments, by 4 for quarterly payments, and so on.

Start with the nominal annual interest rate quoted, not the accompanying annual percentage rate (APR).

pmt example

In the following example, the **pushButton** method for the *findPayment* button calculates the monthly payment for a 24-month loan of $1,000 at 12%:

```
; findPayment::pushButton
method pushButton(var eventInfo Event)
var
  monthlyPayment,
  loanAmt,
  intRate,
  numPayments Number
endVar
loanAmt = 1000         ; borrow $1000
intRate = .12 / 12     ; 12 percent annual interest
numPayments = 24       ; 1 payment per month for 2 years
monthlyPayment = loanAmt.pmt(intRate, numPayments)
msgInfo("Monthly payment", "The monthly payment for a loan of " +
        String(loanAmt) + " at " + String(intRate * 12 * 100) +
        "% interest for " + String(SmallInt(numPayments)) +
        " months is " + String(monthlyPayment))    ; payment is $47.07
endMethod
```

■

pow method

See also Example Number Type
Beginner

Raises a number to a power.

Syntax

```
pow ( const exponent Number ) Number
```

Description

pow returns the value of a number raised to the power specified in *exponent*. If the return value is larger than 1E308 or smaller than 1E-308, **pow** returns an error.

▪

pow example

In the following example, the **pushButton** method for the *raiseTwo* button calculates *rootexpn* and displays the result:

```
; raiseTwo::pushButton
method pushButton(var eventInfo Event)
var
  root,
  expn    Number
endVar
root = 2
expn = 8
msgInfo(String(root) + " raised to the power of " +
        String(expn), root.pow(expn)) ; displays 256
endMethod
```

pow10 method

See also Example Number Type
Beginner

Calculates 10 to a specified power.

Syntax
pow10 ( ) Number

Description
pow10 returns the value of 10 raised to a specified power.

▪

pow10 example

In the following example, the **pushButton** method for the *raiseTen* button calculates *10expn* and displays the result:

```
; raiseTen::pushButton
method pushButton(var eventInfo Event)
var
  expn,
  result Number
endVar
expn = 9
result = expn.pow10()
msgInfo("Ten raised by a power of " + String(expn),
        format("EC", result))                  ; displays 1,000,000,000
endMethod
```

▪

pv method

See also Example Number Type
Beginner

Returns the present value of a series of equal payments.

Syntax

```
pv ( const interestRate Number, const periods Number ) Number
```

Description

pv calculates the present value of equal, regular payments on a loan (or withdrawals from an investment) at a rate specified in *interestRate* for a number of periods specified in *periods*. The payments reduce the principal, but the remaining balance continues to generate and compound interest.

The formula used is

$PV = payment * (1- (1+rate)^{-n} / rate)$

where *n* is the number of periods.

The interest rate used in **pv** is expressed as a decimal number (like .12), not as a percentage (12%). Make sure the rate period matches the payment period; that is, if the payments are monthly, the interest rate should also be monthly. You can use **pv** to calculate how large a mortgage you can afford. (Use **pmt** to work in reverse and find the monthly payment needed to amortize a given amount.) You can also use **pv** to calculate the amount you'll need to purchase an annuity that will make regular, equal payments to you over time. For this reason, **pv** is sometimes called the present value of an annuity.

■

pv example

Suppose you can afford to pay $1,200 per month and can get a 30-year mortgage at a fixed annual rate of 9% (0.75% monthly). The **pushButton** method for *findPV* calculates and displays the loan amount for which you can qualify:

```
; findPV::pushButton
method pushButton(var eventInfo Event)
var
  payAmt,
  intRate,
  term,
  mortgage    Number
endVar
payAmt   = 1200
intRate  = .09 / 12              ; monthly interest for 9% a year
term     = 360                   ; 30 years (expressed in months)
mortgage = payAmt.pv(intRate, term)
msgInfo("Maximum Mortgage", "If you can pay " + String(payAmt) +
        " a month for " + String(term /12) + " years at " +
        String(intRate * 12 * 100) + "% you can qualify for " +
        format("E$C", mortgage))     ; displays $149,138
endMethod
```

Suppose when you retire you would like to withdraw $2,500 each month for 30 years from an annuity account that accumulates 7.5% annual interest. This **pushButton** method for the *findAnnuity* button calculates how much you'll need in the account:

```
; findAnnuity::pushButton
method pushButton(var eventInfo Event)
var
  monthlyAmt,
  term,
  intRate,
  investment  Number
endVar

monthlyAmt = 2500.00  ; monthly amount you want annuity to pay
term = 360            ; 30 years, converted to 360 months
intRate = .075/12     ; 7.5% a year, converted to monthly rate
investment = monthlyAmt.pv(intRate, term)  ; what you need to start with
msgInfo("Annuity Required", "For an annuity to return $" +
        String(monthlyAmt) + " a month at " +
        format("W4.2", intRate * 12 * 100) + "% for " +
        String(SmallInt(term / 12)) +
        " years, the original amount must be " +
        String(investment))                ; displays 357,544.07
endMethod
```

▪

rand procedure

See also   Example   Number Type
Beginner

Generates a random value ranging from 0 to 1.

Syntax
`rand ( ) Number`

Description
rand generates a random value ranging from 0 to 1.

▪

rand example

In the following example, the **pushButton** method for the *getRand* button calculates and displays a random number *x* between 1 (*minNum*) and 10 (*maxNum*).

```
; getRand::pushButton
method pushButton(var eventInfo Event)
var
  x,
  minNum,
  maxNum  SmallInt
endVar
minNum = 1
maxNum = 10
; get a random integer between minNum and maxNum
x = SmallInt(rand() * (maxNum - minNum + 1) + minNum)
msgInfo("A number between " + String(minNum) + " and " +
        String(maxNum),  x)
endMethod
```

▪

round method

See also Example Number Type
Beginner

Rounds a number to a specified number of decimal places.

Syntax

```
round ( const places SmallInt ) Number
```

Description

round returns a number rounded to the number of decimal places specified in *places*.

▪

round example

In the following example, the **pushButton** method for the *showRound* button rounds a number to 4 decimal places and displays the result, then rounds and displays a number to the nearest 1000.

```
; showRound::pushButton
method pushButton(var eventInfo Event)
var
  roundMe Number
endVar
roundMe = 1.2356838
msgInfo(format("W9.7",roundMe) + " rounded to 4 decimal places",
          format("W6.4", roundMe.round(4))) ; displays 1.2357
roundMe = 678394
msgInfo(String(roundMe) + " rounded to -3 decimal places",
          roundMe.round(-3))                ; displays 678,000
endMethod
```

sin method

See also Example Number Type
Beginner

Returns the sine of an angle.

Syntax
`sin ( )` `Number`

Description
sin returns a numeric value between -1 and 1 for the sine of a value representing the size of the angle in radians.

sin example

The **pushButton** method for the *findSin* button finds the sine of a 45-degree angle:

```
; findSin::pushButton
method pushButton(var eventInfo Event)
var
  fortyFiveDegrees Number
endVar
fortyFiveDegrees  = PI / 4.0
msgInfo("The sine of 45 degrees",
         format("W14.12", fortyFiveDegrees.sin()))
; displays 0.707106781187
endMethod
```

sinh method

See also Example Number Type
Beginner

Returns the hyperbolic sine of an angle.

Syntax
`sinh ( )` `Number`

Description
sinh returns the hyperbolic sine of a value representing the size of the angle in radians. The formula used is

sinh (*angle*) = (*exp* (*angle*) - *exp* (-*angle*)) / *2*

▪

sinh example

In the following example, the **pushButton** method for the *getHSine* button finds the hyperbolic sine of a 45-degree angle:

```
; getHSine
method pushButton(var eventInfo Event)
var
  fortyFiveDegrees Number
endVar
fortyFiveDegrees = PI / 4.0
msgInfo("The hyperbolic sine of 45 degrees",
        format("w14.12", fortyFiveDegrees.sinh()))
; displays 0.868670961486
endMethod
```

▪

sqrt method

See also Example Number Type
Beginner

Returns the square root of a number.

Syntax
sqrt ( ) Number

Description
sqrt returns the square root of a positive value or numeric expression.

▪

sqrt example

In the following example, the **pushButton** method for the *getSqrt* button assigns the value from *fieldOne* (an unbound field object) to *x*, checks to see if *x* is negative, and, if not, calculates and displays the square root of *x*:

```
; getSqrt::pushButton
method pushButton(var eventInfo Event)
var
  x Number
endVar
x = fieldOne
if x < 0 then
  msgStop("Sorry",
          "Can't take the square root of a negative number.")
else
  msgInfo("The square root of " + String(x),
           format("w14.6", sqrt(x))) ; displays result
endIf
endMethod
```

tan method

See also Example Number Type
Beginner

Returns the tangent of an angle.

Syntax
`tan ( ) Number`

Description
tan returns the tangent of a value or numeric expression representing the size of the angle in radians.
tan diverges at -pi/2, pi/2, and every ± pi radians from those values.

▪

tan example

In the following example, the **pushButton** method for the *getTan* button calculates the tangent of a 45-degree angle and displays the result:

```
; getTan::pushButton
method pushButton(var eventInfo Event)
var
  fortyFiveDegrees Number
endVar
fortyFiveDegrees = PI / 4.0
msgInfo("Tangent of 45 degrees", fortyFiveDegrees.tan())   ; displays 1.00
endMethod
```

▪

tanh method

See also Example Number Type
Beginner

Returns the hyperbolic tangent of an angle.

Syntax

`tanh ( ) Number`

Description

tanh returns the hyperbolic tangent of a value or numeric expression representing the size of the angle in radians. The formula is

tanh (angle) = sinh (angle) / cosh (angle)

■

tanh example

In the following example, the **pushButton** method for a button named *getHTan* calculates the hyperbolic tangent of a 60-degree angle and displays the result:

```
; getHTan::pushButton
method pushButton(var eventInfo Event)
var
  sixtyDegrees Number
endVar
sixtyDegrees = PI / 3.0
msgInfo("The hyperbolic tangent of 60 degrees",
        format("W14.12", sixtyDegrees.tanh()))
; displays .780714435359
endMethod
```

▪

truncate method

See also Example Number Type
Beginner

Truncates a number to a specified number of decimal places.

Syntax

truncate (const ***places*** SmallInt **)** Number

Description

truncate returns a number truncated towards 0 to the number of decimal places in *places*. It does not round the value.

▪

truncate example

In the following example, the **pushButton** method for the *chopAValue* button assigns the value from *fieldOne* (an unbound field object) to *x*, truncates *x* to 3 decimal places, and displays the truncated result:

```
; chopAValue::pushButton
method pushButton(var eventInfo Event)
var
  x Number
endVar
x = fieldOne
msgInfo("x truncated to 3 places",
        format("W14.6", x.truncate(3))) ; displays truncated version of x
endMethod
```

■

OLE type

Changes ■

OLE stands for Object Linking and Embedding, a protocol that provides access to the functionality of another application without having to leave Paradox and open that application each time you want to make a change.

For example, suppose you have tables that contain bitmap graphics, and you want to create a Paradox application that enables users to edit those graphics. One approach is to create the graphics using a paint program that is an OLE server (defined below). Then, use ObjectPAL OLE type methods to make the functionality of the paint program available to your users (assuming, of course, that your users have the paint program installed on their systems).

Note: ObjectPAL and Paradox also support DDE (for Dynamic Data Exchange), another protocol for sharing data.

The following terms are used when discussing OLE operations:

- OLE *server* is an application that can provide access to its documents via the OLE mechanism. Paradox is an OLE server.
- OLE *container* is an application that can use the OLE mechanism to access documents created by an OLE server. Paradox is an OLE container.
- OLE *object* is a document created using an OLE server. It contains the data you want to use in your Paradox application.
- OLE *variable* is an ObjectPAL variable declared to be of type OLE. An OLE variable provides a handle for manipulating an OLE object. In other words, you can use OLE variables in ObjectPAL code to manipulate OLE objects.

OLE applications execute asynchronously■that is, code in each application executes independently; one application does not wait for the other. When you write a method that launches an OLE server for user input, declare the OLE variable in a Var window (or in a method window above the **method** keyword). This ensures that the OLE variable will be available (and in scope) even if the method finishes before the server application is closed.

The OLE type includes several derived methods from the AnyType type.

Methods for the OLE type

AnyType ■	**OLE**
blank	**canLinkFromClipboard**
dataType	**canReadFromClipboard**
isAssigned	**edit**
isBlank	**enumServerClassNames**
isFixedType	**enumVerbs**
unAssign	**getServerName**
	insertObject
	isLinked
	linkFromClipboard
	readFromClipboard
	updateLinkNow
	writeToClipboard

Changes to OLE type methods

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
canLinkFromClipboard	(None)
enumServerClassNames	
insertObject	
isLinked	
linkFromClipboard	
updateLinkNow	

▪

canLinkFromClipboard method

See also Example OLE Type

Reports whether an OLE object can be linked from the Clipboard to an OLE variable.

Syntax

```
canLinkFromClipboard ( ) Logical
```

Description

canLinkFromClipboard returns True if an OLE object can be linked from the Clipboard to an OLE variable; otherwise, it returns False. After an OLE object is linked (as opposed to read) from the Clipboard, changes made to the OLE object while in Paradox affect the underlying file.

This method is useful in a routine that finds out whether a **linkFromClipboard** operation is possible. For example, you can make a menu item dimmed and inactive when this method returns False.

canLinkFromClipboard example

This example tries to link an OLE object from the Clipboard to a field in a specified record in a table. If it can't, it prompts the user to embed (read) the OLE object instead.

```
; btnLinkOrRead::pushButton
method mouseClick(var eventInfo MouseEvent)
   var
      stReadOLE      String
      oleObj         OLE
      tcEmployee      TCursor
   endVar

   ; Move to specified record in table.
   tcEmployee.open("employee")
   tcEmployee.locate("EmpName", "Frank Borland")

   ; Link if you can, otherwise read (embed).
   switch
      case oleObj.canLinkFromClipboard() :
         oleObj.linkFromClipboard()

      case oleObj.canReadFromClipboard() :
         stReadOLE = msgQuestion("Can't link OLE object.",
                              "Do you want to embed it instead?")
         if stReadOLE = "Yes" then
            oleObj.readFromClipboard()
         else
            message("No update.")
            return
         endIf

      otherwise :
         msgInfo("Can't link or embed the OLE object.",
              "The Clipboard may be empty.")
         return
   endSwitch

   ; Update the table.
   tcEmployee.edit()
   tcEmployee.VoiceSample = oleObj
   tcEmployee.endEdit()
   message("Update complete")
endMethod
```

▪

canReadFromClipboard method

See also Example OLE Type

Reports whether an OLE object can be read (embedded) from the clipboard into an OLE variable.

Syntax

```
canReadFromClipboard ( ) Logical
```

Description

canReadFromClipboard returns True if an OLE object can be read (embedded) from the Clipboard into an OLE variable; otherwise, it returns False. After an OLE object is read (as opposed to linked) from the Clipboard, changes made to the OLE object while in Paradox do not affect the underlying file.

This method is useful in a routine that finds out whether a **readFromClipboard** operation is possible. For example, you can make a menu item dimmed and inactive when this method returns False.

▪

canReadFromClipboard example

See the example for **canLinkFromClipboard**.

▪

edit method

See also Example OLE Type

Launches the OLE server and lets the user edit the object or take some other action.

Syntax

```
edit ( const oleText String, const verb SmallInt ) Logical
```

Description

edit launches the OLE server application and gives control to the user. The argument *oleText* is a string that Paradox passes to the server application. Many server applications can display *oleText* in the title bar. **edit** passes *verb* to the application server to specify an action to take.

verb is an integer that corresponds to one of the OLE server's action constants. The meaning of *verb* varies from application to application, so a *verb* that is appropriate for one application may not be appropriate for another. Use **enumVerbs** to learn what verbs the server supports, then use one of them in the call to **edit**.

If you want to launch an OLE server without using **enumVerbs** first, use 0 (zero) for *verb*▪this value is the primary verb, and should be supported by all OLE servers.

edit example

Suppose the *Pics* table stores Paintbrush graphics in an OLE field. The table has two fields: PicName (A8) and PicData (O). When you click *editButton*, the code locates a record in the table then uses **edit** to invoke Paintbrush, thereby enabling the user to edit the graphic in the OLE field. When you click *updateButton*, the code updates the *Pics* table.

Code is attached to the page's Var window, *editButton*'s **pushButton** method, and *updateButton*'s **pushButton** method. Variables are declared in the page's Var window for two reasons: first, to make them available to both buttons; second, it makes sure the OLE variable is still available, even if this method finishes executing before the user closes Paintbrush.

The page's Var window contains the following code:

```
 var
   olePic  OLE
   picTC   TCursor
endVar
```

The *editButton's* **pushButton** method contains the following code:

```
 method pushButton(var eventInfo Event)
   if picTC.open ("pics.db") then
      if picTC.locate("PicName", "blueLine") then

                  ; The PicData field stores OLE objects
                  ; created using Paintbrush.
         olePic = picTC.PicData

                  ; Launch Paintbrush so user can edit the bitmap.
         olePic.edit("PDOXWIN", 0)
      else
         msgStop("Stop", "Couldn't find blueLine.")
      endIf
   else
      msgStop("Stop", "Couldn't open table.")
   endIf
endMethod
```

The *updateButton's* **pushButton** method contains the following code:

```
 method pushButton(var eventInfo Event)
   picTC.edit()
   picTC.PicData = olePic
   picTC.endEdit()
   picTC.close()
endMethod
```

enumServerClassNames method

See also   Example   OLE Type

Lists registered OLE servers.

Syntax

```
enumServerClassNames ( var serverClasses DynArray[ ] String ) Logical
```

Description

enumServerClassNames lists the OLE servers registered in the user's system. The information is assigned to *serverClasses*, a DynArray that you must declare and pass as an argument. This method returns True if it succeeds; otherwise, it returns False.

The indexes of the DynArray are the end-user server names (for example, "Paradox Table"), and the corresponding items are the names used internally by OLE (for example, "PdoxWin5Table").

Use **enumServerClassNames** to get a server name to pass to **insertObject.**

enumServerClassNames example

See the example for **insertObject**.

▪

enumVerbs method

See also Example OLE Type

Lists the actions supported by an OLE server.

Syntax

```
enumVerbs ( var verbs DynArray[ ] SmallInt ) Logical
```

Description

enumVerbs creates a DynArray listing the action commands (called *verbs*) supported by the OLE server associated with an OLE variable.

When you associate an OLE variable with an OLE object, Paradox knows which server application generated the object. Through OLE methods such as **enumVerbs** and **getServerName**, you can ask questions of the server.

enumVerbs asks the server for a list of supported action commands, then loads them into a dynamic array. Each DynArray index corresponds to the name that the server gives to a specific action; DynArray items correspond to the action constant used by the server. Because the meaning of *verb* varies from application to application, you need to know precisely what verb to pass to the server to instruct it to do what you want.

For example, Windows Paintbrush is an OLE server. Paintbrush has only one action command, named "Edit," with a value of 0. The following code reads from the Clipboard a graphic generated with Paintbrush, generates a dynamic array with **enumVerbs**, then displays the contents of the DynArray in a dialog box.

```
var
  oleVar OLE
  dy DynArray[] SmallInt
endVar

oleVar.readFromClipboard() ; read from the Clipboard into oleVar
oleVar.enumverbs(dy)       ; generate a DynArray of verbs
dy.view()                  ; display DynArray contents in a dialog
```

The preceding code assumes the Clipboard contains an OLE object (a graphic image in this case) that was generated in Paintbrush. The dynamic array contains one element whose index is "Edit" and whose value is 0. Some OLE servers use more than one verb, and would therefore generate a larger list. Other OLE servers use "Edit" but preface the name with an ampersand, such as "&Edit". The ampersand prefix is useful when you want to display action names in a menu. Paradox recognizes the ampersand as a special character and displays "&Edit" as Edit, and designates E as an accelerator key.

Refer to Menu methods for more information about menus and special characters.

enumVerbs example

For the following example, assume the *Sounds* table has an alpha field named SoundName and an OLE field named SoundData. Data in the OLE field were copied from the Windows Sound Recorder to the Clipboard. The following example uses **enumVerbs** to create a pop-up menu that lists the verbs (actions) for Sound Recorder when you click a button named *btnEditSounds*. Because Sound Recorder supports two actions (Edit and Play), this example lets the user choose to edit or play the sound contained in the OLE field.

The following code is attached to the button's Var window. It declares the OLE variable. By declaring the OLE variable in the Var window, you ensure that the variable is available, even if the method finishes before the server application is closed.

```
; btnEditSounds::Var
Var
   oleVar OLE
endVar
```

The following code is attached to the button's built-in **pushButton** method. It builds and displays a pop-up menu and launches the server application.

```
; btnEditSounds::pushButton
method pushButton(var eventInfo Event)
var
  oleVar  OLE
  p       PopUpMenu
  verbs   DynArray[] SmallInt
  tc      TCursor
  mChoice, tagName String
endvar
soundName = "tada.wav"
tblName = "Sounds.db"

if tc.open(tblName) then
  if tc.locate(1, soundName) then ; Search in first field for tada.wav
    oleVar = tc.SoundData    ; Assign field value to OLE var
    oleVar.enumVerbs(verbs)  ; Get list of Sound Recorder actions.
    forEach tagName in verbs ; Create a pop-up menu of verbs.
      p.addText(tagName) ; Sound Recorder's verbs are &Edit and &Play
    endForEach
    mChoice = p.show()  ; display "Edit" and "Play" in the pop-up menu

    ; If the user selects from the menu,
    ; pass the selected "verb" to the
    ; edit method. verbs[mChoice] evaluates to 0 or 1.
    ; "PdoxWin" appears in Sound Recorder's title bar
    ; when Edit is selected
    if not mChoice.isBlank() then
      oleVar.edit("PdoxWin", verbs[mChoice])
    endIf

  else
    errorShow("Can't find " + soundName + ".")
  endIf
else
  errorShow("Can't open " + tblName + ".")
endIf

endMethod
```

getServerName method

See also Example OLE Type

Reports the name of the OLE server for an OLE object.

Syntax

```
getServerName ( ) String
```

Description

getServerName returns as a string the name of the OLE server for an OLE object. This method is useful when you want to inform the user of the OLE server's name.

▪

getServerName example

For the following example, assume the *Media* table has an alpha field named MediaName, an alpha field named ServerName, and an OLE field named MediaData. The following code scans through *Media*'s records, filling the ServerName field with the name of the OLE server that generated data in the MediaData field.

```
; getServerName::pushButton
method pushButton(var eventInfo Event)
var
  oleVar  OLE
  tc      TCursor
endvar

if tc.open("Media") then
  tc.edit()
  scan tc for not isBlank(tc.SoundData) :
    oleVar = tc.SoundData
    tc.ServerName = oleVar.getServerName()
  endScan
  tc.close()
else
  msgStop("Error", "Can't open Media table.")
endIf

endMethod
```

insertObject method

See also Example1 Example2 Example3 OLE Type

Inserts a linked or embedded OLE object into an OLE variable.

Syntax

```
1. insertObject ( ) Logical
2. insertObject ( const fileName String , const link Logical ) Logical
3. insertObject ( const className String ) Logical
```

Description

insertObject assigns a linked or embedded OLE object to an OLE variable. This method returns True if the assignment succeeds; otherwise, it returns False.

Syntax 1 invokes the Insert Object dialog box, just as if you had chosen Edit|Insert Object interactively. It is up to the user to supply any necessary information and close the dialog box. For example, the user can choose Create New (to insert a new OLE object) or Create From File (to insert an existing OLE object from a file). Choosing Create New launches the server application for user input; choosing Create From File does not.

Syntax 2 inserts an object from the file specified in *fileName* without launching the server application for user input. The argument *link* specifies whether to link to the file. If *link* is True, changes made to the object in Paradox are reflected in the underlying file. If *link* is False, changes made in Paradox do not affect the file.

Syntax 3 launches the server application for user input and inserts an object from the class specified in *className*, where *className* is the name of a registered OLE server class. Use **enumServerClassNames** to get a list of OLE server class names.

insertObject example 1

In the following example, a form contains buttons named *btnInsertOLE* and *btnEditOLE* and a field object named *mugShot* which is bound to an OLE field named MugShot in a table in the form's data model. The variables *oleVar* and *loInserted* are declared in the page's Var window for two reasons: to make them available to both buttons, and to ensure that the OLE variable is available if a method finishes before the server application is closed.

The following code is attached to the page's Var window. It declares the OLE variable *oleVar* and a Logical flag variable *loInserted* that tracks whether an OLE object was inserted into the OLE variable.

```
; thePage::Var
Var
   oleVar         OLE
   loInserted   Logical
endVar
```

The following code is attached to the **pushButton** method of *btnInsertOLE*. It displays the Insert Object dialog box so the user can insert an OLE object.

```
; btnInsertOLE :: pushButton
method pushButton(var eventInfo Event)

   if not oleVar.insertObject() then ; Invoke Insert Object dialog box.
      errorShow()
      loInserted = No
      return
   else
      loInserted = Yes
   endIf

endMethod
```

The following code is attached to the **pushButton** method of *btnEditOLE*.

```
; btnEditOLE :: pushButton
method pushButton(var eventInfo Event)

   if not loInserted.isAssigned() then
      loInserted = No
   endIf

   if loInserted = Yes then
      edit()
      mugShot.Value = oleVar
      loInserted = No ; Reset the flag.
      endEdit()
   else
      msgInfo("No OLE object to insert.",
           "Click the Insert button.")
   endIf

endMethod
```

▪

insertObject example 2

In the following example, a form contains a button named *btnInsertOLE* and a field object named *fldOLE* which is bound to an OLE field in a table in the form's data model. The **pushButton** method uses an OLE variable *oleVar* and **insertObject** to read a wave file into the OLE variable *oleVar* and then assign it to the field *fldOLE*. This example does not launch the server application for user input.

```
;btnInsertOLEFile :: pushButton
const
   ; Changes made in Paradox will not
   ; affect the underlying file.
   kNoLink = False
endConst

var
   oleVar OLE
endVar

method pushButton(var eventInfo Event)
   var
      stFileName,
      stPrompt   String
   endVar

   stPrompt = "Type the file name here."
   stFileName = stPrompt
   stFileName.view("Enter a file name.")
   if stFileName = stPrompt then
      return ; User didn't type a file name and click OK.
   endIf

   if oleVar.insertObject(stFileName, kNoLink) then
      edit()
      fldOLE.Value = oleVar
      endEdit()
   else
      errorShow("Could not insert OLE object: " + stFileName)
   endIf
endMethod
```

■

insertObject example 3

Suppose you were using Paradox to maintain and publish a database for a school and each record was a course syllabus. Different instructors would likely prefer different word processor applications, so you could store syllabus data in an OLE field and let people edit it using the application of their choice (the application must be an OLE server).

In this example, suppose a form contains a table frame bound to the *Courses* table, and each record in the table frame contains a field object named *Syllabus*. The following code is attached to a button named *btnAddSyllabus* that lets the user add a new syllabus to the table. It gets a list of the OLE server applications installed in the user's system and displays the list in a pop-up menu. When the user chooses an application name from the pop-up menu, the call to **insertObject** launches the specified application.

```
; btnAddSyllabus :: pushButton
var
   oleVar   OLE
endVar

method pushButton(var eventInfo Event)
   var
      puServers      PopUpMenu
      stOLEServer,
      stUserServer      String
      dyOLEServers      DynArray[] String
   endVar

   ; Specify a title for the pop-up menu.
   puServers.addStaticText("Choose one:")
   puServers.addSeparator()

   ; enumServerClassNames returns a DynArray where the keys are
   ; the external names and the corresponding items are the
   ; names used internally by OLE.

   oleVar.enumServerClassNames(dyOLEServers)

   forEach stOLEServer in dyOLEServers
      puServers.addText(stOLEServer)
   endForEach

   stUserServer = puServers.show()
   if stUserServer <> "" then

     ; insertObject uses the internal name to specify an OLE server.
      if oleVar.insertObject(dyOLEServers[stUserServer]) then
         action(DataBeginEdit)
         Courses.Syllabus.Value = oleVar
         action(DataEndEdit)
      else
         errorShow("Could not insert " + stOLEServer)
      endIf
   else
      return ; User didn't choose a server.
   endIf
endMethod
```

▪

isLinked method

See also Example OLE Type

Reports whether an OLE object is a linked object.

Syntax

`isLinked ( ) Logical`

Description

isLinked returns True if an OLE object is a linked object; it returns False if it is an embedded object. When used with **updateLinkNow** this method provides a convenient way to update the linked OLE fields in a table.

▪

isLinked example

See the example for **updateLinkNow**.

▪

linkFromClipboard method

See also   Example   OLE Type

Pastes a link between an OLE object from the Clipboard and an OLE variable.

Syntax

```
linkFromClipboard ( ) Logical
```

Description

linkFromClipboard returns True if an OLE object is successfully read (pasted) from the Clipboard and linked to an OLE variable; otherwise, it returns False. Calling this method is equivalent to choosing Edit|Paste Link interactively.

After an OLE object is linked from the Clipboard, changes made to the OLE object while in Paradox affect the underlying file. Compare this method to **readFromClipboard**, where changes made in Paradox do not affect the underlying file.

-

linkFromClipboard example

See the example for **canReadFromClipboard**.

■

readFromClipboard method

See also Example OLE Type

Pastes an OLE object from the Clipboard into an OLE variable.

Syntax

```
readFromClipboard ( ) Logical
```

Description

readFromClipboard returns True if an OLE object is successfully read (pasted) from the Clipboard into an OLE variable; otherwise, it returns False.

After an OLE object is read from the Clipboard, changes made to the OLE object while in Paradox do not affect the underlying file. Compare this method to **linkFromClipboard**, where changes made in Paradox affect the underlying file.

▪

readFromClipboard example

See the example for **canReadFromClipboard**.

updateLinkNow method

See also Example OLE Type

Updates a linked OLE object.

Syntax

```
updateLinkNow ( ) Logical
```

Description

updateLinkNow updates a linked OLE object and returns True if successful. It returns False if the OLE object is an embedded object. When used with **isLinked** this method provides a convenient way to update the linked OLE fields in a table.

updateLinkNow example

This example scans the *Employee* table and updates any linked values in the OLE field named VoiceSample.

```
;btnUpdateLinks::pushButton
method pushButton(var eventInfo Event)
   var
      oleObj      OLE
      tcEmployee   TCursor
   endVar

   tcEmployee.open("employee")
   tcEmployee.edit()

   scan tcEmployee :
      oleObj = tcEmployee.VoiceSample    ; VoiceSample is an OLE field.
      if oleObj.isLinked() then
            oleObj.updateLinkNow()          ; Update the OLE variable.
            tcEmployee.VoiceSample = oleObj ; Assign the new value to the
field in the underlying table.
      endIf
   endScan

   tcEmployee.endEdit()
endMethod
```

▪

writeToClipboard method

See also Example OLE Type

Copies an OLE variable to the Clipboard.

Syntax

```
writeToClipboard ( ) Logical
```

Description

writeToClipboard copies the original OLE object to the Clipboard as if the server had done the copy. This method erases the Clipboard before doing the copy.

This method returns True if an OLE object is successfully written (copied) to the Clipboard; otherwise, it returns False.

writeToClipboard example

The following example reads an OLE field in a Paradox table and assigns its value to an OLE variable. Then it writes the variable to the Clipboard, where it can be used by Paradox, or by another application. The code assumes that EMPLOYEE.DB has an alpha field named Last Name and an OLE field named Picture.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  empTC TCursor
  oleImage OLE
endVar

empTC.open("Employee.db")      ; EMPLOYEE.DB has OLE images

if empTC.locate("Last Name", "Binkley") then

  oleImage = empTC.Picture    ; Picture is an OLE field
  oleImage.writeToClipboard() ; write contents of OLE field to variable

else
  msgStop("Error", "Can't find Binkley...")
endIf
endMethod
```

Point type

A Point variable holds information about a point on the screen. To ObjectPAL, the screen is a two-dimensional grid, with the origin at the upper left corner of the design object's container, positive x-values extending to the right, and positive y-values extending down. A Point has an x-value and a y-value, where *x* and *y* are measured in twips (a twip is 1/1440 of a logical inch; 1/20 of a printer's point.)

Methods defined for the Point type get and set information about screen coordinates and relative positions of points. For example, the size and position properties of a design object are specified in points.

Note: ObjectPAL calculates point values relative to the container of the design object in question. For example, if a box contains a button, ObjectPAL calculates the button's position relative to the box. If the button sits in an empty page, ObjectPAL calculates the button's position relative to the page. Methods that take or return Point values as arguments use this relative framework. The method **convertPointWithRespectTo** defined for the UIObject type is useful for converting values in different frameworks.

You can use Point operators (+, -, =, <, >, <=, and >=) to add, subtract, and compare Point variables. These operators operate on the x-coordinates of each point, then on the y-coordinates. For example,

```
var
   p1, p2, p3 Point
endVar

   p1 = Point(10, 30)
   p2 = Point(10, 30)
   p3 = Point(10, 33)

   message(p1 + p2)  ; Displays (20, 60), because 10 + 10 = 20, and 30 + 30 =
60.
   message(p1 = p2)  ; Displays True. Both x- and y-coordinates are equal.
   message(p1 = p3)  ; Displays False. Both coordinates must be equal.
   message(p3 > p1)  ; Displays False. Both coordinates must be greater.
   message(p3 >= p1) ; Displays True. Both coordinates are either greater or
equal.
```

The Point type includes several derived methods from the AnyType type.

Methods for the Point type

AnyType	Point
blank	**distance**
dataType	**isAbove**
isAssigned	**isBelow**
isBlank	**isLeft**
isFixedType	**isRight**
view	**point**
	setX
	setXY
	setY
	x

y

▪

distance method

See also Example Point Type

Returns the distance between two points.

Syntax

distance (**`const` ***pt* `Point` **)** `Number`

Description

distance returns the number of twips between a point and *pt*.

■

distance example

Suppose a form contains 2 boxes: *redBox* and *brownBox*. The **pushButton** method for a button named *getDistance* finds the distance between the upper left corners of the boxes:

```
; brownBox::pushButton
method pushButton(var eventInfo Event)
var
  p1, p2 Point
endVar
p1 = redBox.Position
p2 = brownBox.Position
msgInfo("Distance between boxes", p1.distance(p2))
; shows the distance between the top left corner of
; redBox and the top left corner of brownBox
endMethod
```

▪

isAbove method

See also Example Point Type

Reports whether a point is above another point.

Syntax

isAbove (const *pt* Point) Logical

Description

isAbove returns True if the y-coordinate of a point is less than the y-coordinate of *pt*; otherwise, it returns False.

▪

isAbove example

In the following example, the **pushButton** method for *convergeBoxes* moves *boxOne* closer to *boxTwo*, until the two boxes converge. Assume that *boxOne* starts to the left of and above *boxTwo*. Each time the button is clicked, *boxOne* will move down until it is on the same vertical plane, then move to the right until it is covered by *boxTwo*.

```
; convergeBoxes::pushButton
method pushButton(var eventInfo Event)
var
  p1, p2 Point
endVar
p1 = boxOne.position              ; get the position of boxOne
p2 = boxTwo.position              ; get the position of boxTwo
if p1.isAbove(p2) then            ; compare the two points
  ; if p1 is higher than p2, move boxOne down
  boxOne.position = Point(p1.x(), p1.y() + 100)
else
  if p1.isLeft(p2) then
    ; if p1 is to the left of p2, move boxOne to the right
    boxOne.position = Point(p1.x() + 100, p1.y())
  endIf
endIf
endMethod
```

▪

isBelow method

See also Example Point Type

Reports whether a point is below another point.

Syntax

isBelow (const *pt* Point) Logical

Description

isBelow returns True if the y-coordinate of a point is greater than the y-coordinate of *pt*; otherwise, it returns False.

▪

isBelow example

In the following example, the **pushButton** method for *convergeBoxes* moves *boxTwo* closer to *boxOne*, until the two boxes converge. Assume that *boxTwo* starts to the right of and below *boxOne*. Each time the button is clicked, *boxTwo* will move up until it is on the same vertical plane, then move to the left until it is covered by *boxOne*.

```
; convergeBoxes::pushButton
method pushButton(var eventInfo Event)
var
  p1, p2 Point
endVar
p1 = boxOne.position             ; get the position of boxOne
p2 = boxTwo.position             ; get the position of boxTwo
if p2.isBelow(p1) then           ; compare the two points
  ; if p2 is lower than p1, move boxTwo up
  boxTwo.position = Point(p2.x(), p2.y() - 100)
else
  if p2.isRight(p1) then
    ; if p2 is to the left of p1, move boxTwo to the left
    boxTwo.position = Point(p2.x() - 100, p2.y())
  endIf
endIf
endMethod
```

isLeft method

See also Example Point Type

Reports whether a point is to the left of another point.

Syntax

isLeft (const ***pt*** Point **)** Logical

Description

isLeft returns True if the x-coordinate of a point is less than the x-coordinate of *pt*; otherwise, it returns False.

▪

isLeft example

See the example for **isAbove.**

▪

isRight method

See also Example Point Type

Reports whether a point is to the right of another point.

Syntax

`isRight ( const pt Point ) Logical`

Description

isRight returns True if the x-coordinate of a point is greater than the x-coordinate of *pt*; otherwise, it returns False.

▪

isRight example

See the example for **isBelow**.

▪

point procedure

See also   Example   Point Type

Casts an expression as a Point.

Syntax

```
1. point ( const x LongInt, const y LongInt ) Point
2. point ( const newPoint Point ) Point
```

Description

point casts (converts) an expression as a Point.

▪

point example

In the following example, you want to vary the position of a box called *rateBox*. The values of an unbound field object named *rateField* range from 0 to 10. The position of *rateBox* is determined by the value in *rateField*. The following code is attached to the **changeValue** method for *rateField*:

```
; rateField::changeValue
method changeValue(var eventInfo ValueEvent)
Const
  baseXPosition = LongInt(3000)
  baseYPosition = LongInt(1000)
endConst
Var
  rateX   LongInt
endVar
try
  ; this if statement will fail if the field contents can't
  ; be compared to the integers 0 and 10 - for instance, if
  ; the user enters a string
  if eventInfo.newValue() >= 0 AND eventInfo.newValue() <= 10 then
    rateX = (eventInfo.newValue() * 400) + baseXPosition
    rateBox.Position = point(rateX, baseYPosition)
  else
    fail() ; if the value is a number but is out of range,
           ; call the fail block
  endIf
onFail
  disableDefault
  eventInfo.setErrorCode(CanNotDepart)
  msgStop("Stop", "Rating should be a number between 0 and 10.")
endTry

endMethod
```

setX method

See also Example Point Type

Specifies the x-coordinate of a point.

Syntax

setX (const *newXValue* LongInt)

Description

setX sets the x-coordinate of a point to *newXValue*. If *newXValue* is not a LongInt, it is converted to a LongInt, and precision may be lost.

▪

setX example

In the following example, a form contains an ellipse called *circleOne* and a button named *moveRight*. The **pushButton** method for *moveRight* uses **setX** to change the horizontal coordinate of a point, then sets the position of *circleOne* to the changed point:

```
; moveRight::pushButton
method pushButton(var eventInfo Event)
var
  p1 Point
endVar
p1 = circleOne.position     ; get the position of the circle
p1.setX(p1.x() + 100)       ; add 100 twips to the x-coordinate
circleOne.Position = p1     ; set the new position
message(p1)                 ; display coordinates
endMethod
```

▪

setXY method

See also Example Point Type

Specifies the x- and y-coordinates of a point.

Syntax

`setXY ( const newXValue LongInt, const newYValue LongInt )`

Description

setXY sets the x- and y-coordinates of a point to *newXValue* and *newYValue*. This method combines the functions of **setX** and **setY**. If *newXValue* and *newYValue* are not LongInts, they are converted to LongInts, and precision may be lost.

▪

setXY example

In the following example, a form contains an ellipse called *circleOne* and a button named *moveDiagonal*. The **pushButton** method for *moveDiagonal* uses **setXY** to change the horizontal and vertical coordinates of a point, then sets the position of *circleOne* to the changed point:

```
; moveDiagonal::pushButton
method pushButton(var eventInfo Event)
var
  p1 Point
endVar
p1 = circleOne.position                ; get the position of the circle
p1.setXY(p1.x() + 100, p1.y() + 100)  ; add 100 twips to each coordinate
circleOne.Position = p1                ; set the new position
message(p1)                            ; display coordinates
endMethod
```

setY method

See also Example Point Type

Specifies the y-coordinate of a point.

Syntax

setY (**const ***newYValue* LongInt **)**

Description

setY sets the y-coordinate of a point to *newYValue*. If *newYValue* is not a LongInt, it is converted to a LongInt, and precision may be lost.

setY example

In the following example, a form contains an ellipse called *circleOne* and a button named *moveDown*. The **pushButton** method for *moveDown* uses **setY** to change the vertical coordinate of a point, then sets the position of *circleOne* to the changed point:

```
; moveDown::pushButton
method pushButton(var eventInfo Event)
var
  p1 Point
endVar
p1 = circleOne.position  ; get the position of the circle
p1.setY(p1.y() + 100)    ; add 100 twips to y-coordinate
circleOne.Position = p1  ; set the new position
message(p1)              ; display coordinates
endMethod
```

▪

x method

See also Example Point Type

Returns the x-coordinate of a point.

Syntax

x ( ) LongInt

Description

x returns the x-coordinate of a point.

-

x example

See the example for **setX.**

▪

y method

See also Example Point Type

Returns the y-coordinate of a point.

Syntax

```
y ( ) LongInt
```

Description

y returns the y-coordinate of a point.

-

y example

See the example for **setY.**

PopUpMenu type

A PopUpMenu is a list of items that appears vertically in response to an Event (usually a mouse click). When the user chooses an item from a pop-up menu, the text of that item is returned to the method. A PopUpMenu is distinct from a Menu, a list of items that appears horizontally in the application menu bar.

Note: Choosing an item from a pop-up menu *does not* trigger the built-in **menuAction** method unless the pop-up menu is attached to a custom menu.

Using PopUpMenu methods, you can

- Build a pop-up menu
- Display the pop-up menu and return the selected item
- Inspect the items in a pop-up menu
- Provide keyboard access

The PopUpMenu type includes several derived methods from the Menu type.

Methods for the PopUpMenu type

Menu	**PopUpMenu**
contains	**addArray**
count	**addBar**
empty	**addBreak**
remove	**addPopUp**
removeMenu	**addSeparator**
	addStaticText
	addText
	show
	switchMenu

▪

addArray method

See also   Example   PopUpMenu Type

Appends elements of an array to a pop-up menu.

Syntax

`addArray ( const items Array[ ] String )`

Description

addArray appends *items* from an array to a pop-up menu.

addArray example

The following code is attached to a field object's built-in **mouseRightUp** method. When the user right-clicks the field, a list of available payment types appears in a pop-up menu. The following code is attached to the **mouseRightUp** method for *paymentField*.

```
; paymentType::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  items  Array[4] String
  p1     PopUpMenu          ;  addArray is called for this PopUpMenu
  choice String
endVar

disableDefault              ; don't show default Font menu

items[1] = "Visa"
items[2] = "MasterCharge"
items[3] = "Check"
items[4] = "Cash"

p1.addArray(items)          ; add items array to the PopUpMenu
choice = p1.show()          ; display menu, remember choice
if not choice.isBlank() then
  self.value = choice
endIf

endMethod
```

▪

addBar method

See also Example PopUpMenu Type

Adds a vertical bar to a pop-up menu.

Syntax

`addBar ( )`

Description

addBar adds a vertical bar to a pop-up menu. The **addBar** method marks the beginning of a new column of choices and inserts a vertical bar immediately before the new column. **addBar** is the vertical equivalent of **addSeparator**.

addBar example

The following code displays a pop-up menu with two columns of choices. The first two choices are displayed in the left column. The remaining choices are displayed in the right column. This code is attached to a field's **mouseRightUp** method.

```
; navField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  navPopUp   PopUpMenu    ; to show a navigate pop-up menu
  navChoice  String       ; store the menu choice
endVar

disableDefault                        ; don't show normal menu for field

navPopUp.addText("Previous record")   ; left menu
navPopUp.addText("First record")
navPopUp.addBar()                     ; add vertical bar
navPopUp.addText("Next record")       ; right menu
navPopUp.addText("Last record")

navChoice = navPopUp.show()           ; invoke menu
; ...
; process choice
; ...

endMethod
```

▪

addBreak method

See also Example PopUpMenu Type

Starts a new column in a pop-up menu.

Syntax

```
addBreak ( )
```

Description

addBreak starts a new column in a pop-up menu. The first item added after the call to **addBreak** appears at the top of a column to the right of the previous column, and subsequent items appear below it. The **addBreak** method behaves like **addBar** in that it marks the beginning of a new column of choices. However, **addBreak** doesn't create a vertical bar between columns. **addBreak** doesn't create a cascading menu; use **addPopUp** instead.

▪

addBreak example

The following code displays a pop-up menu with nine choices in three vertical columns. This code is attached to *whereToButton*'s **pushButton** method.

```
; whereToButton::pushButton
method pushButton(var eventInfo Event)
var
  navPopUp      PopUpMenu       ; a pop-up of navigation choices
  navChoice     String          ; navigation chosen
endVar

navPopUp.addText("Home")        ; left menu
navPopUp.addText("Left")
navPopUp.addText("End")

navPopUp.addBreak()             ; start second column
navPopUp.addText("Up")
navPopUp.addText("Center")
navPopUp.addText("Down")

navPopUp.addBreak()             ; start third column
navPopUp.addText("PgUp")        ; right menu
navPopUp.addText("Right")
navPopUp.addText("PgDn")

navChoice = navPopUp.show()     ; invoke menu

; ... process choice

endMethod
```

▪

addPopUp method

See also Example1 Example2 PopUpMenu Type

Adds a pop-up menu to the structure.

Syntax

addPopUp (const *menuName* String, const *cascadedPopup* PopUpMenu)

Description

addPopUp adds *menuName* and *cascadedPopup* to the current pop-up menu structure, creating a cascading menu. *menuName* appears as an item in the original pop-up menu, and the first item in *cascadedPopup* appears next to it. Subsequent items in *cascadedPopUp* appear in a column below the first item.

▪

addPopUp example 1

The following example uses **addPopUp** to attach a cascading menu to a menu bar item (a menu from the Menu type). In this example, the code attached to the built-in **open** method for *thisPage* creates and displays the menu structure. The code attached to *thisPage*'s **menuAction** handles the user's selection because the pop-up menus are attached to a menu bar item.

The following code is attached to the **open** method for *thisPage*:

```
; thisPage::open
method open(var eventInfo Event)
var
  mainMenu Menu
  subMenu1, subMenu2 PopUpMenu
endVar

  ; create 2nd level submenu
subMenu2.addText("&Time")
subMenu2.addText("&Date")

  ; add 2nd level to 1st level
subMenu1.addPopUp("&Utilities", subMenu2)

  ; add 1st level to menu bar
mainMenu.addPopUp("&File", subMenu1)

  ; display the menu bar
mainMenu.show()

endMethod
```

The following code is attached to *thisPage*'s **menuAction** method:

```
; thisPage::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

choice = eventInfo.menuChoice()
switch
  case choice = "&Time" : msgInfo("Current Time", time())
  case choice = "&Date" : msgInfo("Today's Date", date())
endSwitch

endMethod
```

addPopUp example 2

The following example uses **addPopUp** to create a cascading pop-up menu. This menu structure is not attached to a menu bar item. The code immediately following the call to **show** takes action based on the user's selection; the built-in **menuAction** method is not used.

The following code is attached to the **mouseRightUp** method for *pageTwo*:

```
; pageTwo::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  p1, p2, p3 PopUpMenu
  choice String
endVar

disableDefault                ; don't show normal pop-up menu

p2.addText("&Time")           ; build p2 and p3 submenus
p2.addText("&Date")
p3.addText("&Red")
p3.addText("&Green")
p3.addText("&Blue")

p1.addPopUp("&Utilities", p2) ; create Utilities item and attach p2 to it
p1.addPopUp("&Colors", p3)    ; create Colors item and attach p3 to it

choice = p1.show()            ; display menu and store selection to choice

switch                        ; now take action based on selection
  case choice = "&Red"   : self.color = Red
  case choice = "&Green" : self.color = Green
  case choice = "&Blue"  : self.color = Blue
  case choice = "&Time"  : msgInfo("Current Time", time())
  case choice = "&Date"  : msgInfo("Today's Date", date())
endSwitch

endMethod
```

▪

addSeparator method

See also Example PopUpMenu Type

Adds a horizontal bar to a pop-up menu.

Syntax

```
addSeparator ( )
```

Description

addSeparator appends a horizontal bar to separate item groups in a pop-up menu. **addSeparator** is used to group similar commands together within a menu.

addSeparator example

The following example uses **addSeparator** to group pop-up menu commands. The following code is attached to the built-in **open** method for *thisPage*:

```
; thisPage::open
method open(var eventInfo Event)
var
  mainMenu Menu
  subMenu1, clrMenu PopUpMenu
endVar

clrMenu.addText("&Red")
clrMenu.addText("&Blue")
clrMenu.addText("&White")

subMenu1.addText("&Time")
subMenu1.addText("&Date")
subMenu1.addSeparator()
subMenu1.addPopUp("&Page colors", clrMenu)
subMenu1.addSeparator()
subMenu1.addText("&About")

mainMenu.addPopUp("&Utilities", subMenu1)
mainMenu.show()
endMethod
```

The following code is attached to the built-in **menuAction** method for *thisPage*:

```
; thisPage::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar
choice = eventInfo.menuChoice()
switch
  case choice = "&Red"   : self.color = Red
  case choice = "&Blue"  : self.color = Blue
  case choice = "&White" : self.color = White
  case choice = "&Time"  : msgInfo("Current Time", time())
  case choice = "&Date"  : msgInfo("Today's Date", date())
  case choice = "&About" : eventInfo.setId(MenuHelpAbout)
endSwitch
endMethod
```

▪

addStaticText method

See also Example PopUpMenu Type

Adds an unselectable text string to a pop-up menu.

Syntax

```
addStaticText ( const item String )
```

Description

addStaticText appends an item to a pop-up menu as unselectable text. Static text is usually used as the title (first item) in a pop-up menu.

▪

addStaticText example

The following code is attached to a field object's built-in **mouseRightUp** method. When the user right-clicks the field, a list of available payment types appears in a pop-up menu. This example displays the first item as static text. The following code is attached to the **mouseRightUp** method for *paymentField*.

```
; paymentType::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  items  Array[4] String
  p1     PopUpMenu          ;  addArray is called for this PopUpMenu
  choice String
endVar

disableDefault              ; don't show default Font menu

items[1] = "Visa"
items[2] = "MasterCharge"
items[3] = "Check"
items[4] = "Cash"

                            ; display first item as static text
p1.addStaticText("Payment Method")
p1.addSeparator()           ; add a horizontal separator
p1.addArray(items)          ; add items array to the PopUpMenu
choice = p1.show()          ; display menu, remember choice
if not choice.isBlank() then
  self.value = choice
endIf

endMethod
```

▪

addText method

See also Example1 Example2 PopUpMenu Type

Adds a selectable text string to a pop-up menu.

Syntax

```
1. addText ( const menuName String )
2. addText ( const menuName String, const attrib SmallInt )
3. addText ( const menuName String, const attrib SmallInt, const id SmallInt
)
```

Description

addText appends a selectable item to a pop-up menu. The pop-up menu can be displayed alone, or as part of a menu in the menu bar.

Syntax 1 specifies the text of the item to append in *menuName*.

In syntax 2, you can use *attrib* to preset the display attribute of *menuName*. ObjectPAL provides MenuChoiceAttributes constants (like MenuDisabled) for display attributes, so you don't have to memorize numeric values.

Syntax 3 is used only when the pop-up menu is attached to a Menu object. You can specify an *id* number (of type SmallInt) to identify the menu by number instead of by *menuName*. Then, in the built-in **menuAction** method, you use the *id* number to determine which menu the user chooses.

You can use syntax 3 to create a menu that provides the same functions as a built-in Paradox menu. Use a MenuCommands constant to assign a value to the *id* argument. Then, when the user chooses that item from a menu, Paradox performs the default action. For example, the following line adds "Next" to the *puRecord* PopUpMenu and uses the MenuCommands constant MenuRecordNext to assign an ID value.

```
puRecord.addText("Next", MenuEnabled, MenuRecordNext)
```

It is up to you to display, enable, and disable menu items to ensure that the Paradox operation the user triggers is valid (for example, locking a record while the form is not in edit mode is not an allowed operation).

You can also specify custom menu IDs. To do this, use the IdRanges constant UserMenu as a base constant, then add a number or a user-defined menu constant to it. For example, the following line adds "File" to the *myPopup* PopUpMenu and specifies an *id* number for that menu item:

```
myPopup.addText("File", MenuEnabled, UserMenu + 1)
```

You can use an ampersand in an item so the user can select it using the keyboard. For example, the item "&File" would display as File, and the user could choose it by pressing F. When testing the user's choice, remember to include the ampersand. In this example, the returned value would be "&File", not "File".

You can also use "\t" to put a Tab between an item and its accelerator. For example, the item "&Edit Data\tF9" would display "Edit Data" left-aligned and "F9" right-aligned. The string value returned in this case would be "&Edit Data\tF9".

▪

addText example 1

The following example demonstrates a variation of **addText** syntax.

For this example, assume a form has an unbound field named *payField*. When the user right-clicks the field, a list of available payment methods appears in a pop-up menu. The user can choose from the list to insert that value into the field or press Esc to cancel. The following code goes in the Var window for *payField*:

```
; payField::var
var
  payPopUp PopUpMenu
  mChoice  String
endVar
```

The following code is attached to the **open** method for *payField*. When the field opens for the first time, this code adds four items to the *payPopUp* PopUpMenu. This code does not display the pop-up menu; it just prepares the menu for later.

```
; payField::open
method open(var eventInfo Event)

payPopUp.addText("Visa")
payPopUp.addText("MasterCard")
payPopUp.addText("Check")
payPopUp.addText("Cash")

endMethod
```

The following code is attached to *payField*'s built-in **mouseRightUp** method. When the user right-clicks the field, this method displays the menu with **show**, then inserts the user's choice in the unbound field.

```
; payField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)

disableDefault               ; don't show default pop-up menu

mChoice = payPopUp.show()    ; display menu, store selection to mChoice
if not isBlank(mChoice) then ; if user does not press Esc
  self.value = mChoice       ; insert mChoice in unbound field
endIf
endMethod
```

▪

addText example 2

The following example demonstrates a variation of **addText** syntax.

This example shows how you can use the *id* clause for pop-up menus attached to a Menu object. This code establishes user-defined constants to make it easy to remember the menu *id* assignments. The following code goes in the Const window for *thisPage*.

```
; thisPage::const
Const
  kMenuRed   = 1  ; define constant values for menu ids
  kMenuBlue  = 2
  kMenuWhite = 3
  kMenuTime  = 4
  kMenuDate  = 5
  kMenuAbout = 6
endConst
```

The following code is attached to the **open** method for *thisPage*. To control the menu display attributes, this code uses built-in constants such as MenuEnabled. To identify each menu item by number, the code uses the constants defined in the Const window for *thisPage* (*menuRed*, *menuBlue*, and so forth).

```
; thisPage::open
method open(var eventInfo Event)
var
  mainMenu Menu
  subMenu1, clrMenu, puRecord PopUpMenu
endVar

  ; add text to pop-up menus and use user-defined constants
clrMenu.addText("&Red", MenuEnabled, kMenuRed + UserMenu)
clrMenu.addText("&Blue", MenuEnabled, kMenuBlue + UserMenu)
clrMenu.addText("&White", MenuEnabled, kMenuWhite + UserMenu)

subMenu1.addText("&Time", MenuEnabled, kMenuTime + UserMenu)
subMenu1.addText("&Date", MenuEnabled, kMenuDate + UserMenu)
subMenu1.addSeparator()
subMenu1.addPopUp("&Page colors", clrMenu)
subMenu1.addSeparator()
subMenu1.addText("&About", MenuEnabled, kMenuAbout + UserMenu)
; Build a pop-up menu to attach to the Record menu.
; Use ObjectPAL MenuCommands constants to assign item IDs.
puRecord.addText("&First", MenuEnabled, MenuRecordFirst)
puRecord.addText("&Prev", MenuEnabled, MenuRecordPrevious)
puRecord.addText("&Next", MenuEnabled, MenuRecordNext)
puRecord.addText("&Last", MenuEnabled, MenuRecordLast)
  ; attach pop-up menus to mainMenu and display the menu bar
mainMenu.addPopUp("&Utilities", subMenu1)
mainMenu.addPopUp("&Record", puRecord)
mainMenu.show()
endMethod
```

The following code is attached to the **menuAction** method for *thisPage*. This example evaluates menu selections by ID number rather than by the name specified in *menuName*.

```
; thisPage::menuAction
method menuAction(var eventInfo MenuEvent)
var
  menuId SmallInt
endVar

menuId = eventInfo.id()   ; store menu id number in menuId

switch
  case menuId = kMenuRed + UserMenu   : self.color = Red
  case menuId = kMenuBlue + UserMenu  : self.color = Blue
  case menuId = kMenuWhite + UserMenu : self.color = White
  case menuId = kMenuTime  + UserMenu : msgInfo("Time", time())
  case menuId = kMenuDate + UserMenu  : msgInfo("Date", date())
  case menuId = kMenuAbout + UserMenu : eventInfo.setId(MenuHelpAbout)

   ; No extra code is needed to handle choices from the Record menu,
   ; because item IDs were assigned using MenuCommands constants.
   ; Paradox handles them automatically.
endSwitch

endMethod
```

▪

show method

See also Example PopUpMenu Type

Displays a pop-up menu and returns the item selected.

Syntax

```
show ( [ const xTwips SmallInt, const yTwips SmallInt ] ) String
```

Description

show displays a pop-up menu and returns the item selected. If the user presses Esc instead of making a selection, the returned value is a zero-length string. The optional arguments *xTwips* and *yTwips* specify the coordinates, in twips, of the upper left corner of the pop-up menu. If not specified, they are set to the x- and y-coordinates of the pointer.

▪

show example

For the following example, assume a form has an unbound field named *payField.* When the user right-clicks the field, a list of payment types appears in a pop-up menu. The user can choose from the list to insert that value into the field or press Esc to cancel. The following code goes in the Var window for *payField*:

```
; payField::var
var
  payPopUp PopUpMenu
  mChoice  String
endVar
```

The following code is attached to the **open** method for *payField*. When the field opens for the first time, this code adds four items to the *payPopUp* PopUpMenu. This code does not display the pop-up menu; it just prepares the menu for later.

```
; payField::open
method open(var eventInfo Event)

payPopUp.addText("Visa")
payPopUp.addText("MasterCard")
payPopUp.addText("Check")
payPopUp.addText("Cash")

endMethod
```

The following code is attached to *payField*'s built-in **mouseRightUp** method. When the user right-clicks the field, this method displays the menu with **show**, then inserts the user's choice into the unbound field.

```
; payField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)

disableDefault               ; don't show default pop-up menu

mChoice = payPopUp.show()    ; display menu, store selection to mChoice
if not isBlank(mChoice) then ; if user does not press Esc
  self.value = mChoice       ; insert mChoice into unbound field
endIf
endMethod
```

switchMenu procedure

See also Example PopUpMenu Type

Builds and displays a pop-up menu, and handles the menu choice.

Syntax

```
switchMenu
   CaseList
   [ otherwise : Statements ]
endSwitchMenu
```

CaseList is any number of statements in the following form:

```
CASE menuItem : Statements
```

Description

switchMenu uses the values of the *menuItem* argument in each *CaseList* to create and display a pop-up menu. The *Statements* following each *menuItem* specify how to handle each menu choice. The optional otherwise clause specifies what to do if the user closes the menu without making a choice (for example, by pressing Esc).

switchMenu example

The following example uses **switchMenu** to create, display, and process the choice from a pop-up menu. A string describing the selection is displayed in the message window of the status line.

```
; actionButton::pushButton
method pushButton(var eventInfo Event)
switchMenu
  case "Add"    : message("Add selected.")
  case "Edit"   : message("Edit selected.")
  case "Delete" : message("Delete selected.")
  otherwise     : message("No selection from menu.")
endSwitchMenu
endMethod
```

Query type

Changes

An ObjectPAL Query variable represents a QBE query. You can use ObjectPAL to create and execute queries from methods just as if you were using Paradox interactively. You can execute a query from a query file, a query statement, or a string. Some queries require Paradox to create temporary tables. Paradox creates these tables in the private directory.

Methods for the Query type

Query
appendRow
appendTable
checkField
checkRow
clearCheck
createAuxTables
createQBEString
enumFieldStruct
executeQBE
getAnswerFieldOrder
getAnswerName
getAnswerSortOrder
getCheck
getCriteria
getQueryRestartOptions
getRowID
getRowNo
getTableID
getTableNo
hasCriteria
insertRow
insertTable
isAssigned
isEmpty
isExecuteQBELocal
isCreateAuxTables
isQueryValid
query
readFromFile
readFromString
removeCriteria

removeRow
removeTable
setAnswerFieldOrder
setAnswerName
setAnswerSortOrder
setCriteria
setLanguageDriver
setQueryRestartOptions
setRowOp
wantInMemoryTCursor
writeQBE

Changes to Query type methods

The following table lists the new methods for version 7.

New
appendRow
appendTable
checkField
checkRow
clearCheck
createAuxTables
createQBEString
enumFieldStruct
getAnswerFieldOrder
getAnswerName
getAnswerSortOrder
getCheck
getCriteria
getRowID
getRowNo
getTableID
getTableNo
hasCriteria
insertRow
insertTable
isCreateAuxTables
isEmpty
isQueryValid
removeCriteria
removeRow
removeTable
setAnswerFieldOrder
setAnswerName
setAnswerSortOrder
setCriteria
setRowOp

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
readFromFile (replaces **Database::executeQBEFile**)	getQueryRestartOptions (previously in the Database type)
readFromString (replaces	isExecuteQBELocal (previously

Database::executeQBEString)	in the Database type)
wantInMemoryTCursor	setQueryRestartOptions (previously in the Database type)

▪

appendRow method

See also Example Query Type

Appends a row to a query table image.

Syntax

```
appendRow ( const tableID SmallInt ) SmallInt
appendRow ( const tableName String ) SmallInt
```

Description

appendRow adds a new row to the specified table image in a QBE. The table is specified with the numeric *tableID* or the *tableName*. **appendRow** returns the numeric *rowID* of the new row. The *rowID* is used to manipulate the contents of that row. After a row has been appended, its *rowID* does not change, even if rows are inserted or deleted ahead of it.

appendRow example

Appends a row to the query for the table CUSTOMER.DB.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   rowID SmallInt
endVar

   qVar.appendTable( "CUSTOMER.DB" )
   rowID = qVar.appendRow( "CUSTOMER.DB" )
   qVar.setCriteria( "CUSTOMER.DB", rowID, "State/Prov", "CA or HI" )
   qvar.checkRow("Customer.db", rowID, CheckCheck)
   qvar.writeQBE("MyQBE")
endMethod
```

▪

appendTable method

See also Example Query Type

Appends a table to a query image.

Syntax

```
appendTable ( const tableName String ) SmallInt
```

Description

appendTable adds the table specified with *tableName* to the query image in a QBE and returns a numeric ID which can then be used to manipulate the contents of that table image. After a table has been appended to the query image, its ID does not change even if table images are inserted or deleted ahead of it.

▪

appendTable example

Appends a table to a query image.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   rowID SmallInt
   tblID SmallInt
endVar

   tblID = qVar.appendTable( "CUSTOMER.DB" )
   rowID = qVar.appendRow( tblID )
   qVar.setCriteria( tblID , rowID, "State/Prov", "CA or HI" )
   qvar.checkRow("Customer.db", rowID, CheckCheck)
   qvar.writeQBE("MyQBE")
endMethod
```

▪

checkField method

See also Example Query Type

Puts a check in a field of a query table image.

Syntax

checkField (const *tableID* SmallInt, const *fieldID* SmallInt, const *checkType* SmallInt) Logical
checkField (const *tableID* SmallInt, const *fieldID* SmallInt, const *rowID* SmallInt, const *checkType* SmallInt) Logical
checkField (const *tableID* SmallInt, const *fieldName* String, const *checkType* SmallInt) Logical
checkField (const *tableID* SmallInt, const *fieldName* String, const *rowID* SmallInt, const *checkType* SmallInt) Logical
checkField (const *tableName* String, const *fieldID* SmallInt, const *checkType* SmallInt) Logical
checkField (const *tableName* String, const *fieldID* SmallInt, const *rowID* SmallInt, const *checkType* SmallInt) Logical
checkField (const *tableName* String, const *fieldName* String, const *checkType* SmallInt) Logical
checkField (const *tableName* String, const *fieldName* String, const *rowID* SmallInt, const *checkType* SmallInt) Logical

Description

checkField creates a check mark in the specified field. The table is specified with the numeric *tableID* or the *tableName*. The field is specified with the *fieldID* or the *fieldName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row.

The ***checkType*** is one of the following values:

CheckCheck	a check mark, unique keys only
CheckDesc	a descending order check
CheckGroup	a GroupBy check
CheckNone	invisible check
CheckPlus	a plus sign, include duplicate keys

▪

checkField example

Checks the "State/Prov" field in the CUSTOMER.DB table of the query image.

```
var
   qVar Query
   rowID SmallInt
   tblID SmallInt
   MyQBEValidateStr String
endVar

   tblID = qVar.appendTable( "CUSTOMER.DB" )
   rowID = qVar.appendRow( tblID )
   qVar.setCriteria( tblID , rowID, "State/Prov", "CA or HI" )
   qVar.checkField( tblID, rowID, "State/Prov", CheckPlus )
   MyQBEValidateStr = qVar.createQBEString()
   MyQBEValidateStr.view()
endMethod
```

▪

checkRow method

See also Example Query Type

Puts a check in each field of an entire row of a query table image.

Syntax

```
checkRow ( const tableName String, const rowID SmallInt, const checkType
      SmallInt ) Logical
checkRow ( const tableName String, const checkType SmallInt ) Logical
```

Description

checkRow puts a checkmark in each field of an entire row of a table image in a QBE query. The table is specified with the numeric *tableID* or the *tableName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row.

The ***checkType*** is one of the following values:

CheckCheck	a check mark, unique keys only
CheckDesc	a descending order check
CheckGroup	a GroupBy check
CheckNone	invisible check
CheckPlus	a plus sign, include duplicate keys

checkRow example

Puts the CheckPlus symbol in every field in first row of the CUSTOMER.DB table of the query image and saves the query to the query file ALLCust.QBE.

```
method pushButton(var eventInfo Event)
var
   qVar Query
endVar

   qVar.appendTable( "Customer.db" )
   qVar.checkRow( "Customer.db", CheckPlus )  ; row not specified,
                                              ; use first row
   qVar.writeQBE("ALLCust.QBE")
endMethod
```

▪

clearCheck method

See also   Example   Query Type

Deletes a check from a field or row of a query table image.

Syntax

```
clearCheck ( const tableID SmallInt, const fieldID SmallInt ) Logical
clearCheck ( const tableID SmallInt, const fieldName String ) Logical
clearCheck ( const tableID SmallInt, const rowID SmallInt, const fieldID
      SmallInt ) Logical
clearCheck ( const tableID SmallInt, const rowID SmallInt, const fieldName
      String ) Logical
clearCheck ( const tableName String, const fieldID SmallInt ) Logical
clearCheck ( const tableName String, const fieldName String ) Logical
clearCheck ( const tableName String, const rowID SmallInt, const fieldID
      SmallInt ) Logical
clearCheck ( const tableName String, const rowID SmallInt, const fieldName
      String ) Logical
```

Description

clearCheck clears a checkmark in the specified field in the QBE query. The table is specified with the numeric *tableID* or the *tableName*. The field is specified with the *fieldID* or the *fieldName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row.

clearCheck example

Clears the checkmark from the "State/Prov" field in the CUSTOMER.DB table in the query image and then runs the query.

```
method pushButton(var eventInfo Event)
var
   qVar Query
endVar

   qVar.readFromFile( "monthly.qbe" )
   qVar.clearCheck( "Customer.db" , "State/Prov" )
   qVar.executeQBE()
endMethod
```

createAuxTables method

See also Example Query Type

Sets the use of auxiliary tables on.

Syntax

```
createAuxTables ( const useAuxTables Logical ) Logical
```

Description

createAuxTables sets the use of auxiliary tables on if *useAuxTables* is set to True

createAuxTables example

This example contains a query that uses auxiliary tables.

```
method pushButton(var eventInfo Event)var
myQBE Query
endvar

myQBE = Query

      Customer.db  |  Name         |
      Delete       |  Johnson..    |

endQuery

myQBE.createAuxTables(True)
myQBE.executeQBE()
endMethod
```

▪

createQBEString method

See also Example Query Type

Returns the QBE string of a query.

Syntax

`createQBEString ( ) String`

Description

createQBEString returns the QBE string of a query variable. If the QBE is invalid, the returned string is blank and errorCode() is used to determine the cause of the failure. The QBE **must** be a valid query against existing tables in order for this function to return a query string, so it should not be used to generate partial (incomplete) query strings or queries which will generate syntax errors if compiled or executed.

▪

createQBEString example

Displays a QBE string from a modified version of the query MONTHLY.QBE.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   qStr String
endVar

   qVar.readFromFile( "Monthly.qbe" )
   qVar.clearCheck( "Customer.db" , "State/Prov" )
   qVar.checkField( "Customer.db" , "Name", CheckPlus )
   qStr = qVar.createQBEString()
   if isblank( qStr ) then
      errorShow()
   else
      qStr.view( "Query String" )
   endif

endMethod
```

enumFieldStruct method

See also Example Query Type

Lists the field structure of an answer table.

Syntax

```
1. enumFieldStruct ( const tableName String ) Logical
2. enumFieldStruct ( var inMemoryTC TCursor ) Logical
```

Description

enumFieldStruct lists the field structure of the answer table that would be generated from the QBE statement. Syntax 1 creates a Paradox table. Syntax 2 stores the information in a TCursor variable. **enumFieldStruct** returns True if successful and False if unsuccessful.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* exists, this method overwrites it without confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

In syntax 2, the structure information is stored in the TCursor variable *inMemoryTC* that you pass as an argument.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field Name	Field Type	Description
Field Name	A31	Name of field.
Type	A31	Data type of field.
Size	S	Size of field.
Dec	S	Number of decimal places, or 0 if field type doesn't support decimal places.
Key	A1	Is it a key field? * = key field, blank = not key field.
_Required Value	A1	Is the field required? T = required, N (or blank) = Not required.
_Min Value	A255	Field's minimum value, if specified; otherwise blank.
_Max Value	A255	Field's maximum value, if specified; otherwise blank.
_Default Value	A255	Field's default value, if specified; otherwise blank.
_Picture Value	A175	Field's picture, if specified; otherwise blank.
_Table Lookup	A255	Name of lookup table. Includes full path if the lookup table is not in :WORK:
_Table Lookup Type	A1	Type of lookup table. 0 (or blank) = no lookup table, 1 = Paradox
_Invariant Field ID	S	Field's ordinal position in table (first field = 1, second field = 2, etc.)

enumFieldStruct example

This example creates the Paradox table MYANSWER.DB containing the structure of the answer table that would be built by the query MYQUERY.QBE.

```
method pushButton(var eventInfo Event)
var
   qVar      Query
endVar
   qVar.readFromFile( "myquery.qbe" )
   qVar.enumFieldStruct("QSTRUCT.DB")
endMethod
```

▪

executeQBE method/procedure

See also Example1 Example2 Query Type

Executes a QBE query.

Syntax

```
1. (Method)
executeQBE ( [ { const ansTbl String |
                 var ansTbl Table |
               var ansTbl TCursor } ] ) Logical
2. (Procedure)
executeQBE ( var db Database, var qVar Query
                  [ , { const ansTbl String |
                          var ansTbl Table |
                        var ansTbl TCursor } ] ) Logical
```

Description

executeQBE executes the query assigned to a Query variable and writes the results to :PRIV:ANSWER.DB or to the table specified in *ansTbl*. You can assign a query to a Query variable using a **query** statement, by calling **readfromFile** or **readfromString**, or by building it with methods like **appendTable**, **appendRow**, and **setCriteria**.

Syntax 1 is for calling **executeQBE** as a method. You have the option to write the query result to *ansTbl* where *ansTbl* is a table name, a Table variable, or a TCursor. If *ansTbl* is not specified, **executeQBE** writes to ANSWER.DB in the user's private directory.

Syntax 2 (added in version 5.0) is for calling **executeQBE** as a procedure. You must specify a Database variable in *db* and a Query variable in *qVar*. You have the option to write the query result to *ansTbl* where *ansTbl* is a table name, a Table variable, or a TCursor. If *ansTbl* is not specified, **executeQBE** writes to ANSWER.DB in the user's private directory.

The following information applies to both syntaxes:

- If you specify the table name as a string and don't include a file extension, *ansTbl* is a Paradox table by default.
- If you specify *ansTbl* as a Table variable, *ansTbl* must be assigned and valid.
- If you specify *ansTbl* as a TCursor, the results are stored in memory only; a table is not created on disk.
- If **executeQBE** is successful

▪if *ansTbl* or ANSWER.DB is created

▪this method returns True (even if the resulting table is empty); otherwise it returns False.

▪

executeQBE example 1

This example calls **executeQBE** as a method. The **pushButton** method for the *getReceivables* button constructs a query statement, assigns it to a Query variable, then runs it with **executeQBE**. The query statement in this example is an insert query: it retrieves certain records from CUSTOMER.DB and ORDERS.DB and inserts them into the existing *MyCust* table. The selection criteria for this example uses a tilde variable *myState* that designates Oregon customers to be included in the results. Because "OR" is an ObjectPAL keyword, the *myState* variable must evaluate to a quoted string to distinguish it from the abbreviation for Oregon.

```
method pushButton(var eventInfo Event)
var
   qVar     Query
   myState  String
   tv       TableView
endVar

; add samp alias for the PdoxWin sample directory
addAlias("samp", "Standard", "c:\\pdoxwin\\sample")

; OR is the abbreviation for Oregon, but because it's
; also an ObjectPAL keyword, it must be enclosed in quotes.
myState = "\"OR\""

; now use myState as a tilde variable in this query statement
qVar = Query

      :samp:Customer.db|Customer No|Name |State/Prov|Phone  |
                       |_cust      |_name|  ~myState|_phone |

      :samp:Orders.db |Customer No|Balance Due|
                      |_cust      |0, _balDue |

            myCust.db |Customer No|Name |Balance Due|Phone |
              insert  |_cust      |_name|_balDue    |_phone|

      EndQuery

qVar.executeQBE("myCust.db")    ; put results into myCust.db
tv.open("myCust.db")            ; view the table

endMethod
```

■

executeQBE example 2

This example calls **executeQBE** as a procedure. The **pushButton** method for the *getReceivables* button constructs a query statement, assigns it to a Query variable, then runs it with **executeQBE**. The query statement in this example is an insert query: it retrieves certain records from CUSTOMER.DB and ORDERS.DB and inserts them into the existing *MyCust* table. The selection criteria for this example uses a tilde variable *myState* that designates Oregon customers to be included in the results. Because "OR" is an ObjectPAL keyword, the *myState* variable must evaluate to a quoted string to distinguish it from the abbreviation for Oregon.

```
method pushButton(var eventInfo Event)
var
   db      Database
   qVar    Query
   myState String
   tv      TableView
endVar

db.open() ; Get a handle to the default database.

; add samp alias for the PdoxWin sample directory
addAlias("samp", "Standard", "c:\\pdoxwin\\sample")

; OR is the abbreviation for Oregon, but because it is also
; a Paradox keyword it must be enclosed in quotes
myState = "\"OR\""

; now use myState as a tilde variable in this query statement
qVar = Query

       :samp:Customer.db|Customer No|Name |State/Prov|Phone  |
                        |_cust       |_name|  ~myState|_phone |

        :samp:Orders.db |Customer No|Balance Due|
                        |_cust       |0, _balDue |

              myCust.db |Customer No|Name |Balance Due|Phone |
                insert  |_cust       |_name|_balDue    |_phone|

        EndQuery

executeQBE(db, qVar, "myCust.db")    ; put results into myCust.db
tv.open("myCust.db")                 ; view the table

endMethod
```

getAnswerFieldOrder method

See also Example Query Type

Retrieves the field names of the custom field order of the answer table that would be generated by a query.

Syntax

```
getAnswerFieldOrder ( var fieldOrder Array[] String ) Logical
```

Description

getAnswerFieldOrder retrieves an array of the fields in the answer table for the current query, if a custom field order was specified. If no custom field order was specified for the query, this method returns an empty array. The query is compiled so that errors can be returned related to whether the query was valid and could be compiled or not. If the query compiles successfully, the array *fieldOrder* is filled with the names of the fields in the answer table. The names in the *fieldOrder* array can rearranged to reflect a different order and the array can be submitted to the **setAnswerFieldOrder** and **setAnswerSortOrder** methods.

The array must be resizable (not a fixed size array).

getAnswerFieldOrder example

This example retrieves the existing field order from MYQUERY.QBE, reorders the field order, and then puts the new ordering in place with **setAnswerFieldOrder**.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
   arFields  Array[] String
endVar
qVar.readFromFile( "myquery.qbe" )
qVar.getAnswerFieldOrder( arFields )
   if arFields.size() > 0 then   ; swap the first and third fields
                                 ; in the answer table.
   arFields.exchange(1,3)
   qVar.setAnswerFieldOrder( arFields )
   qVar.executeQBE()
   endif
endMethod
```

▪

getAnswerName method

See also Example Query Type

Retrieves the name of the answer table.

Syntax

```
getAnswerName () String
```

Description

getAnswerName retrieves the name of the answer table that would be produced by the query.

▪

getAnswerName example

This example enables the user to change the answer table name for the query file MYQUERY.QBE.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
   AnsTblName String
endVar
if msgQuestion("Query",
               "Would you like to change the "
             + "answer table name?") = "Yes" then
   qVar.readFromFile("MYQUERY.QBE")
   AnsTblName = qVar.getAnswerName()
   AnsTblName.view("Make changes below")
   qVar.setAnswerName(AnsTblName)
   qVar.writeQBE("MYQUERY.QBE")
endif
endMethod
```

▪

getAnswerSortOrder method

See also Example Query Type

Retrieves the custom sort order specified for the answer table.

Syntax

```
getAnswerSortOrder ( var sortFields Array[] String ) Logical
```

Description

getAnswerSortOrder specifies the order in which fields are sorted in the answer table, if a custom sort order was specified for the query. If no custom sort order was specified, this method returns an empty array. The array *sortFields* contains an ordered list of field names. After you retrieve an array of these field names with **getAnswerSortOrder**, you can change the field names as needed to create a different sort order.

If you retrieve the list of fields, then change the answer field list (for example, by unchecking a field), your array of fields will be out of date and you must also remove the field from your array before attempting to use that array as a basis for field sorting.

getAnswerSortOrder example

This example gets the field list from MYQUERY.QBE, changes the ordering, and saves the new sort order back into the query with **setAnswerSortOrder**.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
   arFields  Array[] String
endVar
qVar.readFromFile( "myquery.qbe" )
qVar.getAnswerSortOrder( arFields )
   if arFields.size() > 0 then   ; swap the first and third fields
                                 ; in the sort order.
      arFields.exchange(1,3)
      qVar.setAnswerSortOrder( arFields )
      qVar.executeQBE()
   endif
endMethod
```

getCheck method

See also Example Query Type

Returns the check type for the specified field in a query image.

Syntax

```
getCheck ( const tableID SmallInt, const fieldID SmallInt ) SmallInt
getCheck ( const tableID SmallInt, const fieldName String ) SmallInt
getCheck ( const tableID SmallInt, rowID SmallInt, const fieldID SmallInt )
      SmallInt
getCheck ( const tableID SmallInt, rowID SmallInt, const fieldName String )
      SmallInt
getCheck ( const tableName String, const fieldID SmallInt ) SmallInt
getCheck ( const tableName String, const fieldName String ) SmallInt
getCheck ( const tableName String, rowID SmallInt, const fieldID SmallInt )
      SmallInt
getCheck ( const tableName String, rowID SmallInt, const fieldName String )
      SmallInt
```

Description

getCheck returns the check type for the specified field in a query image. The table is specified with the numeric *tableID* or the *tableName*. The field is specified with the *fieldID* or the *fieldName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row.

The ***checkType*** is one of the following values:

CheckCheck	a check mark, unique keys only
CheckDesc	a descending order check
CheckGroup	a GroupBy check
CheckNone	invisible check
CheckPlus	a plus sign, include duplicate keys

getCheck example

Returns the type of checkmark used in the "State/Prov" field of the CUSTOMER.DB table in the query image and changes it to CheckDesc if it was CheckPlus.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   qStr String
endVar
qVar.readFromFile( "monthly.qbe" )
   if qVar.getCheck( "Customer.db" , "State/Prov" ) = CheckPlus then
      qVar.CheckField( "Customer.db" , "State/Prov" , CheckDesc )
      qVar.writeQBE("Monthly.QBE")
   endif
endMethod
```

▪

getCriteria method

See also Example Query Type

Returns the query expression used in a query image.

Syntax

```
getCriteria ( const tableID SmallInt, const fieldID SmallInt ) String
getCriteria ( const tableID SmallInt, const fieldName String ) String
getCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldID
      SmallInt ) String
getCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldName
      String ) String
getCriteria ( const tableName String, const fieldID SmallInt ) String
getCriteria ( const tableName String, const fieldName String ) String
getCriteria ( const tableName String, const rowID SmallInt, const fieldID
      SmallInt ) String
getCriteria ( const tableName String, const rowID SmallInt, const fieldName
      String ) String
```

Description

getCriteria returns the selection conditions and calculation statements in the specified field of a query image. The table is specified with the numeric *tableID* or the *tableName*. The field is specified with the *fieldID* or the *fieldName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row.

This expression does NOT include the Check mark, but does contain the remainder of the field contents.

■

getCriteria example

Changes the selection conditions for the "Name" field in the CUSTOMER.DB table in the query image.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   NameCriteria String
endVar
   qVar.readFromFile( "monthly.qbe" )
   NameCriteria = qVar.getCriteria ( "Customer.db" , "Name")
                                   ; default to the first row
   NameCriteria = NameCriteria + " or Unisco"
   qVar.setCriteria( "Customer.db" , "Name" , NameCriteria )

endMethod
```

getQueryRestartOptions method

See also Example Query Type

Returns a value representing the user's query restart option setting.

Syntax

```
getQueryRestartOptions ( ) SmallInt
```

Description

getQueryRestartOptions returns an integer value representing the user's query restart option setting. Use one of the following ObjectPAL QueryRestartOptions constants to test the value:

QueryDefault	Use the options specified interactively using the Query Restart Options dialog box.
QueryLock	Lock all other users out of the tables needed while the query is running. If Paradox cannot lock a table, it does not run the query. This is the least polite to other users, and you must wait until all the locks can be secured before the query will run.
QueryNoLock	Run the query even if someone changes the data while the query is running.
QueryRestart	Start the query over if a change is made to the data while the query is running. Specify QueryRestart when you want to make sure you get a snapshot of the data as it existed at some instant.

▪

getQueryRestartOptions example

See the example for **setQueryRestartOptions**.

▪

getRowID method

See also Example Query Type

Returns the row identifier for a specified sequence row number.

Syntax

```
getRowID ( const tableID SmallInt, const seqNo SmallInt ) SmallInt
```

Description

getRowID returns the *rowID* for the sequence specified. The *rowID* is any number, regardless of where the row resides in the table image on the query workspace. The table is specified with the numeric *tableID*.

If you want to find out what that *rowID* is, so you can manipulate its contents, you need to convert the sequence number of the row to the *rowID*. For example, the second row of the "Customer.db" table image might have a *rowID* of 32760.

getRowID example

Returns the row identifier of the second row (row number 2) in CUSTOMER.DB, assigns it the name "secondRow", changes the criteria of its "Country" field, and runs the query.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   secondRow SmallInt
endVar
   qVar.readFromFile( "monthly.qbe" )
   secondRow = qVar.getRowID( qvar.getTableID(1), 2 )
   qVar.setCriteria( "Customer.db", secondRow, "Country", "Fiji" )
   qVar.executeQBE()
endMethod
```

getRowNo method

See also Example Query Type

Returns the sequence number of a specific row.

Syntax

```
getRowNo ( const tableID SmallInt, const rowID SmallInt ) SmallInt
```

Description

getRowNo returns the sequence number of the row specified with *rowID*. This is the complement of **getRowID**. Given a unique numeric row identifier (*rowID*), **getRowNo** returns the current position (sequence) of the row in the query table image. For instance, a *rowID* of 32760 might be the third row in a table image.

getRowNo example

Appends a new row to the CUSTOMER.DB image in the query, then gets the sequence number of the new row and prints a message stating the new row's sequence number.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   seqNo, rowID  SmallInt
endVar
   qVar.readFromFile( "monthly.qbe" )
   rowID = qVar.appendRow( "Customer.db" )
   seqNo = qVar.getRowNo( rowID )
   message( "The newly appended row is row number" seqNo
            "in the customer.db query image" )
endMethod
```

getTableID method

See also Example Query Type

Returns the unique tableID for the nth table in the query image.

Syntax

```
getTableID ( const seqNo SmallInt ) SmallInt
```

Description

getTableID returns the unique tableID for the nth table in the query image. This ID is **not** the same as the sequential number of the table in the query image, and attempting to use the sequential number will result in errors. If you do not want to use the tableID to manipulate the query table image, use the table name in those methods which accept a table name in place of a tableID.

▪

getTableID example

This example retrieves the table ID for the third table and the row ID for the second row of the query MONTHLY.QBE. It then uses these IDs to determine the criteria set in the "Name" field.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   thirdTableID, secondrowID  SmallInt
   condition String
endVar

qVar.readFromFile("MONTHLY.QBE")
thirdTableID = qVar.getTableID(3)
secondRowID = qVar.getRowID(thirdTableID, 2)
condition = qVar.getCriteria(thirdTableID, secondRowID, "Name")
msgInfo("Condition", "The criteria for the Name field in the "
      + "second row of the third table is " + condition)
endMethod
```

▪

getTableNo method

See also Example Query Type

Returns the table number of the identified table.

Syntax

```
getTableNo ( const tableID SmallInt ) SmallInt
```

Description

getTableNo returns the table number of the table specified with *tableID*. Given a unique numeric *tableID*, this returns its current position in the query. For instance, a *tableID* of 32760 might correspond to the second table on the query workspace, so this function would return 2.

getTableNo example

This example displays a message with the specified table's position in the query.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   qStr String
   seqNo, rowID, newTableID   SmallInt
endVar
   qVar.readFromFile( "monthly.qbe" )
   newTableID = qVar.appendTable( "Vendors.db" )
   seqNo = qVar.getTableNo( newTableID )
   message( "The newly appended table is table number "
            "seqNo" in the query image" )
endMethod
```

hasCriteria method

See also Example Query Type

Indicates whether or not a specific field has query criteria in it.

Syntax

```
hasCriteria ( const tableID SmallInt, const fieldID SmallInt ) Logical
hasCriteria ( const tableID SmallInt, const fieldName String ) Logical
hasCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldID
      SmallInt ) Logical
hasCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldName
      String ) Logical
hasCriteria ( const tableName String, const fieldID SmallInt ) Logical
hasCriteria ( const tableName String, const fieldName String ) Logical
hasCriteria ( const tableName String, const rowID SmallInt, const fieldID
      SmallInt ) Logical
hasCriteria ( const tableName String, const rowID SmallInt, const fieldName
      String ) Logical
```

Description

hasCriteria returns a Logical value indicating whether or not the specified field has query criteria in it. The table is specified with the numeric *tableID* or the *tableName*. The field is specified with the *fieldID* or the *fieldName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row.

hasCriteria examines the field for a query expression; this does **not** include checkmarks. Use **getCheck** to determine whether or not a field is checked.

▪

hasCriteria example

This example examines the "Sale Date" field in the table ORDERS.DB in the query, then retrieves the criteria, and runs the query.

```
method pushButton(var eventInfo Event)
var
   qVar          Query
   newTableID    SmallInt
   DateCriteria  String
endVar
   qVar.readFromFile( "monthly.qbe" )
   if qVar.hasCriteria( "Orders.db" , "Sale Date" ) then
      DateCriteria = qVar.getCriteria( "Orders.db" , "Sale Date"
   else
      DateCriteria = ""
   endif
   DateCriteria.view( "Enter Date Criteria" )
   qVar.setCriteria( "Orders.db", "Sale Date", DateCriteria )
   qVar.executeQBE()
endMethod
```

▪

insertRow method

[See also](#) [Example](#) [Query Type](#)

Inserts a new row before an existing row in the query workspace.

Syntax

```
insertRow ( const tableID SmallInt, beforeRowID SmallInt ) Logical
insertRow ( const tableName String, beforeRowID SmallInt ) Logical
```

Description

insertRow inserts a new row before an existing row in the query workspace. The table is specified with the numeric *tableID* or the *tableName*. The parameter *beforeRowID* specifies the ID of the row which should be pushed down by the new row.

insertRow example

This example creates a query, based on the CUSTOMER.DB table, that retrieves customer records for two cities. After one row is appended and its query criteria set, another row is inserted and its criteria is set.

```
method pushButton(var eventInfo Event)
var
   qVar                 Query
   firstRow, secondRow  SmallInt
endVar

   qVar.appendTable( "CUSTOMER.DB" )
   secondRow = qVar.appendRow( "CUSTOMER.DB " )
   qVar.checkRow( "CUSTOMER.DB" , CheckCheck )
   qVarsetCriteria( "CUSTOMER.DB", "City", "Waterville")
   qVarsetCriteria( "CUSTOMER.DB", "Country", "USA")
   firstRow = qVar.insertRow( "CUSTOMER.DB", secondRow)
   qvar.checkRow( "CUSTOMER.DB", CheckCheck)
   qVarsetCriteria( "CUSTOMER.DB", "City", "Vancouver")
   qVarsetCriteria( "CUSTOMER.DB", "Country", "Canada")
   qVar.writeQBE( "TwoCity.QBE" )
endMethod
```

insertTable method

See also Example Query Type

Inserts a new table before an existing table in the query workspace.

Syntax

```
insertTable ( const tableName String, const beforeTableID SmallInt ) SmallInt
insertTable ( const tableName String, const beforeTableName String ) SmallInt
```

Description

insertTable inserts a new table before an existing table in the query workspace and returns the *tableID* for the inserted table. The parameter *tableName* specifies the name of the table to insert. The parameters *beforeTableID* and *beforeTableName* specify the ID and name (respectively) of the table which should follow the new table.

▪

insertTable example

This example creates a query that includes the Customer and Orders tables. The two tables are linked with an example element on their common field, "Customer No", and all fields are checked in the Customer table. The query is executed, producing an answer table that lists all customer records that have order records.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
endVar
qVar.appendTable("CUSTOMER.DB")
qVar.checkRow("CUSTOMER.DB", CheckCheck)
qVar.setCriteria("CUSTOMER.DB", "Customer No", "_Join1")
qVar.insertTable("ORDERS.DB". "CUSTOMER.DB")
qVar.setCriteria("CUSTOMER.DB", "Customer No", "_Join1")
qVar.executeQBE()
endMethod
```

▪

isAssigned method

[See also](#) [Example](#) [Query Type](#)

Reports whether a Query variable has an assigned value.

Syntax

`isAssigned ( )` `Logical`

Description

isAssigned returns True if a Query variable has been assigned a value; otherwise, it returns False. This method does not check the validity of the assigned query.

isAssigned example

In the following example, the call to **isAssigned** returns True, because the Query variable *qVar* has been assigned a value, even though the value is not a valid query.

```
method pushButton(var eventInfo Event)
var
   qVar Query
endVar

qVar = Query

      This is not a query

      endQuery

msgInfo("Assigned?", qVar.isAssigned())    ; displays True

endMethod
```

isCreateAuxTables method

See also Example Query Type

Reports whether the use of auxiliary tables is enabled or not.

Syntax

```
isCreateAuxTables ( ) Logical
```

Description

isCreateAuxTables reports whether the use of auxiliary tables is enabled or not. If isCreateAuxTables returns True, auxiliary tables will be used in the creation of a query's answer table.

isCreateAuxTables example

This example contains a query that uses auxiliary tables.

```
method pushButton(var eventInfo Event)
var
myQBE Query
endvar

myQBE = Query

      Customer.db  |  Name         |
      Delete       |  Johnson..    |

EndQuery

if myQBE.isCreateAuxTables = False then
   myQBE.createAuxTables(True)
else
endif
myQBE.executeQBE()
endMethod
```

■

isEmpty method

See also Example Query Type

Indicates whether or not the query is empty.

Syntax

```
isEmpty ( ) Logical
```

Description

isEmpty returns a Logical indicating whether or not the query is currently empty. This reflects whether you have added anything to your query, and not whether this query actually contains enough information to be run. For example, you might append a new row to a query, not put anything into it, and then ask whether the query is empty. **isEmpty** would return FALSE, because you have appended pieces of the query. But **isQueryValid** would return FALSE as well, because you did not complete the query.

▪

isEmpty example

This example reports if the query variable is empty, before and after a QBE file is read into the query variable. The query variable is always empty before it is assigned a value. If the readFromFile method is successful, the query variable will not be empty.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
endVar

msgInfo( "Before readFromFile", "Query is " +
         iif(qVar.isEmpty(), "empty", "not empty"))
qVar.readFromFile("MyQuery.QBE")
msgInfo( "After readFromFile", "Query is " +
         iif(qVar.isEmpty(), "empty", "not empty"))
endMethod
```

isExecuteQBELocal method

See also Example Query Type

Reports whether a QBE query was executed locally or on a server.

Syntax

`isExecuteQBELocal ( ) Logical`

Description

isExecuteQBELocal returns True if the query was executed locally; otherwise, it returns False. This method can be useful in situations where the server uses a different character set, sort order, or other feature that could make a query executed on the server yield a different result than the same query executed locally.

isExecuteQBELocal example

The following code calls **isExecuteQBELocal** to find out where a QBE query was executed. If the query was executed on the server, the code informs the user.

```
method pushButton (var eventInfo Event)
   var
      qbeVar        Query
      dlgTitleText,
      dlgBodyText   String
     endVar

    dlgTitleText = "Remote query"
    dlgBodyText  = "This query was not run on the server. \n" +
                   "Check the sort order"

    qbeVar = Query

             :WestData:orders.db |CustName|Qty        |
                                 |Check   |Check > 10 |

             endQuery

   if qbeVar.executeQBE() then
      if qbeVar.isExecuteQBELocal() then
         msgInfo(dlgTitleText, dlgBodyText)
      endIf
   else
         errorShow()
   endIf

endMethod
```

▪

isQueryValid method

See also Example Query Type

Compiles the current query.

Syntax

`isQueryValid ( ) Logical`

Description

isQueryValid compiles the current query and indicates whether or not the query contains errors which will prevent it from being run. This is the same procedure that occurs when you interactively save a query to disk or execute a query, or request a query string. This method returns False if the query contains an error. To get information on the error, use System::**errorCode**.

isQueryValid example

This example creates a query and reports an error if the result of isQueryValid is False.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   orderID SmallInt
endVar
orderID = qVar.appendTable( "Orders.db" )
qVar.setCriteria( orderID, "Sale Date", "> 1/1/95" )
if not qVar.isQueryValid() then
      errorShow()   ; note that no fields are checked
endif
endMethod
```

▪

query keyword

[See also](#) [Example](#) [Query Type](#)

Begins a query statement.

Syntax

```
query
   tableName|fieldName|[ fieldName|] *
            |criteria |[ criteria |] *
 [ tableName|fieldName|[ fieldName|] *
            |criteria |[ criteria |] * ] *
endQuery
```

Description

query marks the beginning of a QBE statement, which assigns a query to a Query variable. A QBE statement extracts data from one or more tables according to the fields specified in *fieldName* and the selection criteria, where *criteria* can be any valid QBE expression. Because this kind of query is not a string, it can contain tilde variables. (A query string cannot contain tilde variables; see **readFromString** for more information.)

A query statement begins with a Query variable, the = sign, and the keyword query followed by a blank line. Next comes the body of the query, and another blank line. The query ends with the keyword **endQuery**.

Note: You don't have to list all the fields in the table. Instead, you can list only those fields that affect the query, as in this example:

```
var myQBE Query endvar
myQBE = Query

             Customer|Customer No|Name   |
                     |Check      |A..    |

          endQuery
```

The previous query statement retrieves from the *Customer* table customer numbers whose name start with "A". (The *Customer* table has more than two fields, but only two fields are specified in the example.)

The blank lines above and below the body of the query are required. It is not necessary to align the vertical field separators (although alignment makes it more readable). ObjectPAL interprets the following code exactly as it interprets the code in the previous example.

```
var myQBE Query endvar
myQBE = Query

Customer|Customer No          |Name   |
|Check| A..    |

    endQuery
```

If you construct a query statement that includes two or more tables, you must separate each table with a blank line, as follows:

```
var myQBE Query endvar
myQBE = Query

            Customer|Customer No|Name |Phone |
                    |_x         |Check|Check |

            Orders  |Customer No|Balance Due|
                    |_x         |Check 0    |

         endQuery
```

You can use absolute paths or aliases to specify where to find tables in the query definition. Paradox searches for unqualified table names (that is, table names without paths or aliases) in the specified database, or in the default database (:WORK:) if a database is not specified.

query example

For the following example, the **pushButton** method for the *getReceivables* button constructs a query statement, assigns it to a Query variable, then runs it with **executeQBE**. The query statement in this example is an insert query; it retrieves certain records from CUSTOMER.DB and ORDERS.DB and inserts them into the existing *MyCust* table. The selection criteria for this example uses a tilde variable *myState* that designates Oregon customers to be included in the results. Since "OR" is the abbreviation for Oregon, the *myState* variable must evaluate to a quoted string to distinguish it from the **OR** query expression.

```
method pushButton(var eventInfo Event)
var
   qVar    Query
   myState String
   tv      TableView
endVar

; add samp alias for the PdoxWin sample directory
addAlias("samp", "Standard", "c:\\pdoxwin\\sample")

; OR is the abbreviation for Oregon, but because it's
; also an ObjectPAL keyword, it must be enclosed in quotes.
myState = "\"OR\""

; now use myState as a tilde variable in this query statement
qVar = Query

      :samp:Customer.db|Customer No|Name |State/Prov|Phone  |
                       |_cust      |_name|  ~myState|_phone |

       :samp:Orders.db |Customer No|Balance Due|
                       |_cust      |0, _balDue |

             myCust.db |Customer No|Name |Balance Due|Phone  |
               insert  |_cust      |_name|_balDue    |_phone |

        EndQuery

qVar.executeQBE("myCust.db")   ; put results into myCust.db
tv.open("myCust.db")            ; view the table

endMethod
```

▪

readFromFile method

See also   Example   Query Type

Assigns the contents of a QBE file to a Query variable.

Syntax

`readFromFile ( const qbeFileName String ) Logical`

Description

readFromFile opens *qbeFileName* and assigns the contents to a Query variable. There are several ways to create a query file; for example, in ObjectPAL using **writeQBE**, or interactively using the Query Editor. Use **executeQBE** to execute the query.

If the value of *qbeFileName* does not include a path or alias, this method looks for the file in the directory associated with the specified database (or the default database, if a database is not specified). If the value of *qbeFileName* does not include an extension, this method assumes an extension of .QBE. To specify a file name that does not have an extension, put a period at the end of the name. For example, the following table lists the resulting file names for various values of *qbeFileName*.

Value of *qbeFileName*	QBE file name
newcust	newcust.qbe
newcust.	newcust
newcust.q	newcust.q

readFromFile returns True if it succeeds; otherwise, it returns False.

▪

readFromFile example

This code reads a query from a file and executes the query.

```
method pushButton(var eventInfo Event)
var
   qVar   Query
endVar

 ; this writes results into :PRIV:ANSWER.DB
qVar.readFromFile("GetCust.qbe")
qVar.executeQBE()

endMethod
```

▪

readFromString method

See also Example Query Type

Assigns a query string to a Query variable.

Syntax

```
readFromString ( const QBEString String ) Logical
```

Description

readFromString assigns the query string specified in *QBEString* to a Query variable. Use **executeQBE** to execute the query.

readFromString is useful when you're building a QBE string from smaller strings▪a QBE string can be a combination of quoted strings and string variables. Double backslashes are required when specifying a path.

You can use absolute paths or aliases to specify where to find tables in the query definition. Paradox searches for unqualified table names (that is, table names without paths or aliases) in the specified database, or in the default database (:WORK:) if a database is not specified.

Because a QBE string is a quoted string, it cannot contain tilde variables (but you can use string variables to get the same effect). If you want to use tilde variables in a query, use a **query** statement or use **readFromFile** to assign the contents of a QBE file to a Query variable.

readFromString example

For the following example, the **pushButton** method for *btnFindName* defines a query as a string value, then uses **readFromString** to assign the string to a Query variable.

```
method pushButton(var eventInfo Event)
var
   db Database
   qs String
   tv TableView
   tc TCursor
   qVar   Query
endVar

; Add the sampData alias then open the database.
addAlias("sampData", "Standard", "c:\\pdoxwin\\sample")
db.open("sampData")

; Open a TCursor for the Stock table.
tc.open("Stock.db", db)

; If locate finds Krypton Flashlight in the Description field.
if tc.locate("Description", "Krypton Flashlight") then

   ; Now use the Stock No field value in Stock.db in a query string.
   qs = "Query\n\n" +
       ":sampData:Lineitem|Order No|Stock No |\n" +
                         "| _ordNo |" + tc."Stock No" + "|\n\n" +
         ":sampData:Orders|Order No|Customer No |\n" +
                         "| _ordNo|_cust |\n\n" +
       ":sampData:Customer|Customer No|Name|Phone |\n" +
                         "| _cust|Check|Check |\n\n" +
        "EndQuery"

   ; Note that the vertical lines (|) don't have to be aligned.

   qVar.readFromString(qs)

   if qVar.executeQBE() then
      tv.open(":priv:answer.db")        ; Display the answer table.
   else
      msgStop("Error", "Query failed") ; Otherwise, query failed.
   endIf

else
   msgStop("Error", "Can't find Krypton Flashlight")
endIf

endMethod
```

▪

removeCriteria method

See also   Example   Query Type

Clears the query expression in the specified field.

Syntax

```
removeCriteria ( const tableID SmallInt, const fieldID SmallInt ) Logical
removeCriteria ( const tableID SmallInt, const fieldName String ) Logical
removeCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldID
      SmallInt ) Logical
removeCriteria ( const tableID SmallInt, const rowID SmallInt, const
      fieldName String ) Logical
removeCriteria ( const tableName String, const fieldID SmallInt ) Logical
removeCriteria ( const tableName String, const fieldName String ) Logical
removeCriteria ( const tableName String, const rowID SmallInt, const fieldID
      SmallInt ) Logical
removeCriteria ( const tableName String, const rowID SmallInt, const
      fieldName String ) Logical
```

Description

removeCriteria clears the query expression in the specified field. It does **not** clear checkmarks (use **setCheck** and **clearCheck** to manage query checkmarks).

The affected table is specified with the numeric *tableID* or the *tableName*. The field is specified with the *fieldID* or the *fieldName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row.

removeCriteria example

This example removes the criteria on the "Name" field in the CUSTOMER.DB table in the query.

```
method pushButton(var eventInfo Event)
var
   qVar Query
endVar

   qVar.readFromFile( "Myquery.qbe" )
   qVar.removeCriteria( "Customer.db" , "Name" )
   qVar.executeQBE()       ; execute the saved query minus the
                           ; customer name criteria.
endMethod
```

▪

removeRow method

See also Example Query Type

Deletes a row and its contents from the query workspace.

Syntax

```
removeRow ( const tableID SmallInt, const rowID SmallInt ) Logical
removeRow ( const tableName String, const rowID SmallInt ) Logical
```

Description

removeRow deletes a row and its contents from the query workspace.

The table is specified with the numeric *tableID* or the *tableName*. The row must be specified with the row identifier *rowID*.

removeRow example

This example removes the second row from the ORDER.DB table in MYQUERY.QBE.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   rowID SmallInt
endVar
   qVar.readFromFile( "MyQuery.qbe" )
   rowID = qVar.getRowID( "ORDERS.DB", 2 )  ; get the 2nd row
   qVar.removeRow( "ORDERS.DB", rowID )
   qVar.executeQBE()
endMethod
```

removeTable method

See also Example Query Type

Removes a table from the query workspace.

Syntax

```
removeTable ( const tableID SmallInt ) Logical
removeTable ( const tableName String ) Logical
```

Description

removeTable removes a table from the query workspace. The table is specified with the numeric *tableID* or the *tableName*.

▪

removeTable example

This example removes the table ORDERS.DB from the query image MYQUERY.QBE.

```
method pushButton(var eventInfo Event)
var
   qVar Query
endVar

   qVar.readFromFile( "MyQuery.qbe" )
   qVar.removeTable( "Orders.db" )          ; remove Orders.db from
                                            ; the workspace
   qVar.removeCriteria("Customer.db", "Customer No" )  ; clear the
                                            ; example element link.
   qVar.executeQBE()
endMethod
```

▪

setAnswerFieldOrder method

[See also](#) [Example](#) [Query Type](#)

Sets the physical order of the field names of the answer table that would be generated by a query.

Syntax

```
setAnswerFieldOrder ( var fieldOrder Array[] String ) Logical
```

Description

setAnswerFieldOrder specifies the order in which fields are structured in the answer table. The parameter *fieldOrder* is an array of field names to use as the answer table structure. To retrieve an array of these field names use **getAnswerName**, then swap the elements as needed to create the correct field order.

If you retrieve the list of fields, then change the answer field list (for example, by unchecking a field), your array of fields will be out of date and you must also remove the field from your array before attempting to use that array as a basis for field ordering. A specified field order must contain the same number of elements as there are fields in the answer table. This method reorders existing elements, it doesn't extract a piece of the answer table and restructure that piece.

setAnswerFieldOrder example

This example retrieves the existing field order from MYQUERY.QBE, changes the field order, and then puts the new ordering in place with **setAnswerFieldOrder**.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
   arFields  Array[] String
endVar
   qVar.readFromFile( "myquery.qbe" )
   qVar.getAnswerFieldOrder( arFields )
      if arFields.size() > 0 then     ; swap the first and third fields
                                      ; in the answer table.
         arFields.exchange(1,3)
         qVar.setAnswerFieldOrder( arFields )
         qVar.executeQBE()
      endif
endMethod
```

setAnswerName method

See also   Example   Query Type

Sets the name of the answer table that would be generated by the query.

Syntax

`setAnswerName ( const tableName String ) Logical`

Description

setAnswerName specifies *tableName* as the name of the answer table to be created by the query.

setAnswerName example

This example enables the user to change the answer table name for the query file MYQUERY.QBE.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
   AnsTblName String
endVar
if msgQuestion("Query",
               "Would you like to change the "
             + "answer table name?") = "Yes" then
   qVar.readFromFile("MYQUERY.QBE")
   AnsTblName = qVar.getAnswerName()
   AnsTblName.view("Make changes below")
   qVar.setAnswerName(AnsTblName)
   qVar.writeQBE("MYQUERY.QBE")
endif
endMethod
```

▪

setAnswerSortOrder method

See also Example Query Type

Specifies the order in which fields are sorted in the answer table.

Syntax

```
setAnswerSortOrder ( var sortFields Array[] String ) Logical
```

Description

setAnswerSortOrder specifies the order in which fields are sorted in the answer table. The array *sortFields* contains an ordered list of field names. After you retrieve an array of these field names with **getAnswerSortOrder**, you can change the field names as needed to create a different sort order.

If you retrieve the list of fields, then change the answer field list (for example, by unchecking a field), your array of fields will be out of date and you must also remove the field from your array before attempting to use that array as a basis for field sorting.

▪

setAnswerSortOrder example

This example gets the field list from MYQUERY.QBE, changes the ordering, and saves the new sort order back into the query with **setAnswerSortOrder**.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
   arFields  Array[] String
endVar
   qVar.readFromFile( "myquery.qbe" )
   qVar.getAnswerSortOrder( arFields )
      if arFields.size() > 0 then     ; swap the first and third fields
                                      ; in the sort order.
         arFields.exchange(1,3)
         qVar.setAnswerSortOrder( arFields )
         qVar.executeQBE()
      endif
endMethod
```

setCriteria method

See also Example Query Type

Specifies the criteria for a specific field in a table.

Syntax

```
setCriteria ( const tableID SmallInt, const fieldID SmallInt, const
      newCriteria String ) Logical
setCriteria ( const tableID SmallInt, const fieldName String, const
      newCriteria String ) Logical
setCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldID
      SmallInt, const newCriteria String ) Logical
setCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldName
      String, const newCriteria String ) Logical
setCriteria ( const tableName String, const fieldID SmallInt, const
      newCriteria String ) Logical
setCriteria ( const tableName String, const fieldName String, const
      newCriteria String ) Logical
setCriteria ( const tableName String, const rowID SmallInt, const fieldID
      SmallInt, const newCriteria String ) Logical
setCriteria ( const tableName String, const rowID SmallInt, const fieldName
      String, const newCriteria String ) Logical
```

Description

setCriteria specifies a query expression string to be used as the criteria for a specific field in a table. The table is specified with the numeric *tableID* or the *tableName*. The field is specified with the *fieldID* or the *fieldName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row. The criteria is specified with *newCriteria*.

setCriteria does **not** handle checkmarks.

setCriteria example

This example set the criteria for the appended row ("State/Prov") to be either CA or HI.

```
method pushButton(var eventInfo Event)
var
   qVar Query
   rowID SmallInt
endVar

   qVar.appendTable( "CUSTOMER.DB" )
   rowID = qVar.appendRow( "CUSTOMER.DB" )
   qVar.setCriteria( "CUSTOMER.DB", rowID, "State/Prov", "CA or HI" )
   qvar.checkRow("Customer.db", rowID, CheckCheck)
   qvar.writeQBE("MyQBE")
endMethod
```

▪

setLanguageDriver method

See also   Example   System Type

Sets the name of the default language driver for the system.

Syntax

`setLanguageDriver ( const languageDriver String ) Logical`

Description

setLanguageDriver sets the default language driver to the driver specified with *languageDriver*.

The language driver is a part of the definition for a table. The language drivers for Paradox tables are listed as part of the description of the Table::**create** method.

When making a query of a table that uses a different language driver, get the language driver of the table by using System::**getLanguageDriver**, then set the language driver for the query so that the query's answer table is created using the same driver.

setLanguageDriver example

This example sets the language driver to Czech.

```
method pushButton(var eventInfo Event)
var
myQBE Query
endvar

myQBE = Query

   Customer|Customer No | Name   |
           |Check       | A..    |

endQuery
myQBE.setLanguageDriver ("ANCZECH")
myQBE.executeQBE()
endMethod
```

setQueryRestartOptions method

See also Example Query Type

Specifies what to do with the underlying tables while running a query.

Syntax

```
setQueryRestartOptions ( const qryRestartType SmallInt ) Logical
```

Description

setQueryRestartOptions tells Paradox what to do if data changes while you're running a query in a multiuser environment. The argument *qryRestartType* represents one of the following ObjectPAL QueryRestartOptions constants:

QueryDefault	Use the options specified interactively using the Query Restart Options dialog box.
QueryLock	Lock all other users out of the tables needed while the query is running. If Paradox cannot lock a table, it does not run the query. This is the least polite to other users, and you must wait until all the locks can be secured before the query will run.
QueryNoLock	Run the query even if someone changes the data while it's running.
QueryRestart	Start the query over if a change is made to the data while the query is running. Specify QueryRestart when you want to make sure you get a snapshot of the data as it existed at some instant.

setQueryRestartOptions example

The following example calls **getQueryRestartOptions** to get the user's current query restart options. If the setting is not QueryRestart, this code calls **setQueryRestartOptions** to set it. Then it executes a query.

```
method pushButton(var eventInfo Event)
   var
      qVar   Query
   endVar

   if getQueryRestartOptions() <> QueryRestart then
      setQueryRestartOptions(QueryRestart)
   endIf

   if qVar.readFromFile("newcust.qbe") then
      qVar.executeQBE()
   else
      errorShow()
   endIf

endMethod
```

▪

setRowOp method

See also Example Query Type

Sets the row operator for a specific row.

Syntax

```
setRowOp ( const tableID SmallInt, const rowID SmallInt, const rowOperator SmallInt) Logical
setRowOp ( const tableID SmallInt, const rowOperator SmallInt) Logical
setRowOp ( const tableName String, const rowID SmallInt, const rowOperator SmallInt) Logical
setRowOp ( const tableName String, const rowOperator SmallInt) Logical
```

Description

setRowOp sets one of the four row operators in the specified row. The table is specified with the numeric *tableID* or the *tableName*. The row must be specified with the row identifier *rowID*, or omitted to default to the first row.

The ***rowOperator*** is one of the following values:

qbeRowDelete - Delete operator

qbeRowInsert - Insert operator

qbeRowNone - No operator

qbeRowSet - Set operator

setRowOp example

This example deletes records where the "Customer No" field is blank.

```
method pushButton(var eventInfo Event)
var
   qVar  Query
endVar

   qVar.appendTable( "Customer.db" )
   qVar.setRowOp( "Customer.db" , qbeRowDelete)
   qVar.setCriteria( "Customer.db" , "Customer No" , "blank" )
                                  ; delete blank Customer No records.
   qVar.executeQBE()

endMethod
```

▪

wantInMemoryTCursor method

See also Example Query Type

Specifies how to create a TCursor resulting from a query.

Syntax

```
wantInMemoryTCursor ( [ const yesNo Logical ] )
```

Description

wantInMemoryTCursor specifies how to create a TCursor resulting from a query. When you call **wantInMemoryTCursor** with *yesNo* as Yes (or omitted), Paradox creates a "dead" TCursor in system memory, with no connection to underlying tables. When *yesNo* is No, Paradox creates a TCursor onto a live query view. By default, when you execute a query to a TCursor, that TCursor will point to a live query view▪changes made to the TCursor will affect the underlying tables. Set **wantInMemoryTCursor** to Yes when you *don't* want a live query view.

An in-memory TCursor can be useful for performing quick "what-if" analyses. For example, suppose you want to study the effect of giving each employee a 15 percent raise. You could query the employee data to increase everyone's salary by 15 percent. Of course, you wouldn't want to do this to the actual employee data (at least, not yet), so you would execute the query to an in-memory TCursor and work with the data there, without affecting the underlying data.

▪

wantInMemoryTCursor example

This example uses an in-memory TCursor to study the effects of giving every employee a 15 percent raise. It reads a pre-defined query from a file and executes it, then uses the results in a calculation.

```
method pushButton(var eventInfo Event)
   var
      qVar               Query
      tcRaise15      TCursor
      nuTotalPayroll   Number
   endVar

   qVar.wantInMemoryTCursor(Yes)
   qVar.readFromFile("raise15.qbe")
   qVar.executeQBE(tcRaise15)

   nuTotalPayroll = tcRaise15.cSum("Salary")
   nuTotalPayroll.view("Payroll after 15%   raise:")

endMethod
```

▪

writeQBE method/procedure

See also Example Query Type

Writes a query statement to a specified file.

Syntax

```
1. (Method) writeQBE ( const fileName String ) Logical
2. (Procedure) writeQBE ( const str String , const fileName String ) Logical
```

Description

writeQBE writes a previously defined query to the file specified in *fileName*. If *fileName* exists, it is overwritten without asking for confirmation. If *fileName* does not specify a path, Paradox writes to :WORK:. **writeQBE** returns True if the write succeeds; otherwise it returns False.

Syntax 1 is for calling **writeQBE** as a method. It writes the query represented by an assigned Query variable to the file specified in *fileName*.

Syntax 2 (added in version 5.0) is for calling **writeQBE** as a procedure. It writes the query string represented by *str* to the file specified in *fileName*.

▪

writeQBE example

In the following example, assume a form has a button named *getDest*. When the form opens, this example determines whether the GETDEST.QBE file exists in the current directory. If the file does not exist, the built-in **open** method for *pageOne* uses **writeQBE** to write a query string to GETDEST.QBE. The built-in **pushButton** for *getDest* runs the query, then opens the table. This code assumes the :MAST: alias has already been defined.

Following is the code attached to the **open** method for *pageOne*:

```
method pushButton(var eventInfo Event)
Var
   qVar Query
endVar

; if the GetDest.qbe query file doesn't exist
if not isFile("GetDest.qbe") then

   ; construct a query
   qVar = Query

            :mastApp:Dest|Destination Name|Avg Temp (F)|
                         |Check           |Check 70    |

          EndQuery

   ; write the query statement to the GetNames.qbe file
   qVar.writeQBE("GetDest.qbe")

endIf
endMethod
```

The following code is attached the built-in **pushButton** method for the *getDest* button. This code does not check whether GETDEST.QBE exists because the **open** method for the page ensures the file is available.

```
method pushButton(var eventInfo Event)
var
   qVar   Query
   tv    TableView
endVar

qVar.readFromFile("GetDest.qbe")
qVar.executeQBE("MyDest")
tv.open("MyDest")

endMethod
```

Another use for this method is to use ObjectPAL to create and save a query that the user can run interactively using the Query Editor.

Record type

ObjectPAL provides the Record type as a programmatic, user-defined collection of information, similar to a **record** in Pascal or a **struct** in C. Such records, defined in ObjectPAL code, are separate and distinct from records associated with a table.

Here's the syntax for declaring a Record data type:

```
TYPE
recordName = RECORD
                       fieldName fieldType
                     [ fieldName fieldType ] *
              ENDRECORD
ENDTYPE
```

One or more *fieldNames* identify fields (columns) of the record, and *fieldType* is one of the data types. Declare records in a design object's Type window.

Once you declare a Record data type, you can use the = and <> comparison operators to compare one record to another. You can also use the assignment (=) operator to copy the contents of one record to another.

The Record type includes several derived methods from the AnyType type.

Methods for the Record type

AnyType	Record
blank	**view**
dataType	
isAssigned	
isBlank	
isFixedType	

▪

view method

See also Example Record Type

Displays in a dialog box the value of a variable.

Syntax

```
view ( [ const title String ] )
```

Description

view displays in a modal dialog box the value or values assigned to a Record variable. ObjectPAL execution suspends until the user closes this dialog box. You can specify the dialog box's title in *title*, or you can omit *title* to display the variable's data type.

Note: Unlike many data types, values in a Record can't be changed when displayed in a **view** dialog box. Refer to AnyType for more information regarding **view** and other data types.

view example

The following example uses a type named MyRecord. The **pushButton** method for *getAndViewRec* declares a variable called *myRec* of type MyRecord. This method then opens a TCursor to the *Customer* table, fills *myRec* with the *Customer No* and *Name* field values from the first record, and uses **view** to display the record in a dialog box. This operation is then repeated for the second record in *Customer*.

The following code is attached to the Type window for *getAndViewRec*. This code creates a user-defined type named MyRecord.

```
; getAndViewRec::Type
Type
  MyRecord = RECORD          ; define a Record structure
               ID    String
               Name  String
             ENDRECORD
endType
```

This code is attached to the **pushButton** method for a button named *getAndViewRec*:

```
; getAndViewRec::pushButton
method pushButton(var eventInfo Event)
var
  recOne, recTwo MyRecord
  tc             TCursor
endVar

if tc.open("Customer.db") then
  recOne.ID = tc."Customer No"      ; put some values into the record
  recOne.Name = tc."Name"
  recOne.view("First record")       ; display the record in a dialog box

  tc.nextRecord()                   ; move to the next record

  recTwo.ID = tc."Customer No"      ; get new values
  recTwo.Name = tc."Name"
  recTwo.view("Second record")      ; display second record

  msgInfo("recOne = recTwo?", recOne = recTwo)  ; displays False

  recOne = recTwo                   ; now both records have the same values
  msgInfo("recOne = recTwo?", recOne = recTwo)  ; displays True

else
  msgStop("Stop", "Couldn't open the Customer table.")
endIf
endMethod
```

Report type

Changes

A Report variable is a handle to a report. You use Report variables in code to manipulate the report onscreen. Report methods control the window's size, position, and appearance, and to view and print the report.

Use **load** to load a report file in the Report Design window; use **open** to open the report in the Report window, and use **print** to open a report and print it. You cannot attach methods to objects in a report, but you can attach code to calculated fields.

The Report type includes several derived methods from the Form type.

Methods for the Report type

Form	**Report**
bringToTop	**attach**
create	**close**
deliver	**currentPage**
dmAddTable	**design**
dmBuildQueryString	**enumUIObjectNames**
dmEnumLinkFields	**enumUIObjectProperties**
dmGetProperty	**load**
dmHasTable	**moveToPage**
dmLinkToFields	**open**
dmLinkToIndex	**print**
dmRemoveTable	**run**
dmSetProperty	**setMenu**
dmUnlink	
enumDataModel	
enumSource	
enumTableLinks	
getFileName	
getPosition	
getProtoProperty	
getStyleSheet	
getTitle	
hide	
isDesign	
isMaximized	
isMinimized	
isVisible	
maximize	
menuAction	

minimize
moveTo
save
saveStyleSheet
selectCurrentTool
setIcon
setPosition
setProtoProperty
setSelectedObjects
setStyleSheet
setTitle
show
wait
windowClientHandle
windowHandle

Changes to Report type methods

Changes for version 7

The Form::setIcon method is new for version 7 and is a derived method of the Report type.

New
setIcon

Changes for version 5.0

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
deliver	attach
dmAddTable	load
dmBuildQueryString	print
dmEnumLinkFields	
dmGetProperty	
dmHasTable	
dmLinkToFields	
dmLinkToIndex	
dmRemoveTable	
dmSetProperty	
dmUnlink	
enumDataModel	
enumSource	
enumTableLinks	
getFileName	
getProtoProperty	
getStyleSheet	
isDesign	
menuAction	
moveTo	
saveStyleSheet	
selectCurrentTool	
setMenu	
setProtoProperty	
setStyleSheet	
wait	
windowClientHandle	

▪

attach method

See also Example Report Type

Associates a Report variable with an open report.

Syntax

`attach ( const reportTitle String ) Logical`

Description

attach associates a Report variable with an open report, where *reportTitle* specifies the title of an open report.

Note: The argument *reportTitle* refers to the text displayed in the title bar of the Report window, not to the file name. You can use **getTitle** to return this text, or you can use **setTitle** to specify a title yourself.

▪

attach example

In the following example, assume the form's **open** method opened the STOCK.RSL report and retitled the window to "Stock Report". The **pushButton** method for *printStock* attaches to the open report by way of its title, then prints it.

```
; printStock::pushButton
method pushButton(var eventInfo Event)
var
  stockRep  Report
endVar
; the Stock report was opened and retitled by the form's open method
stockRep.attach("Stock Report")  ; attach by report's title
stockRep.print()                 ; print the report
endMethod
```

This code is attached to the form's **open** method.

```
; thisForm::open
method open(var eventInfo Event)
var
  stockRep  Report
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    stockRep.open("stock.rsl")
    stockRep.setTitle("Stock Report")
    bringToTop()          ; bring this form back to the top
endIf
endMethod
```

▪

close method

See also Example Report Type

Closes a window.

Syntax

```
close ( )
```

Description

close closes a Report window. Closing a report with **close** is equivalent to choosing Close from the Control menu.

-

close example

In the following example, assume the form's **open** method opened the STOCK.RSL report and retitled the window to "Stock Report". The **close** method for the form attaches to the open report by way of its title, then closes it when the form closes.

```
; thisForm::close
method close(var eventInfo Event)
var
  stockRep  Report
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; the Stock report was opened and retitled by
    ; the form's open method
    stockRep.attach("Stock Report")
    stockRep.close()
endIf
endMethod
```

■

currentPage method

See also Example Report Type

Returns the current page number of a report.

Syntax

```
currentPage ( ) SmallInt
```

Description

currentPage returns the current page number of a report.

currentPage example

In the following example, the **pushButton** method for *plusTwoPages* attempts to attach to an open report, and, if this fails, opens the report. Once the *ordersRep* variable points to an open report, the method moves the report forward two pages.

```
; plusTwoPages::pushButton
method pushButton(var eventInfo Event)
var
  ordersRep Report
endVar
; report might be open already, so attempt an attach first
if NOT ordersRep.attach("Report : ORDERS.RSL") then
  if NOT ordersRep.open("Orders.rsl") then
    msgStop("FYI", "Could not open or attach to report.")
    return
  endIf
endIf
; move to two pages past the current page
ordersRep.moveToPage(ordersRep.currentPage() + 2)
bringToTop()   ; make this form the top layer again
endMethod
```

design method

See also Example Report Type

Switches a report from a Report window to a Report Design window.

Syntax

```
design ( ) Logical
```

Description

design switches a report from the Report window to the Report Design window. This method works only with saved reports (.RSL); it does not work with delivered reports (.RDL).

Use **run** to switch from a Report Design window to a Report window, or **load** to open a report in a Report Design window.

Note: Some report actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

▪

design example

In the following example, assume the form's **open** method opened the STOCK.RSL report and retitled the window to "Stock Report". The **pushButton** method for *stockDesign* attaches to the open report by way of its title, then switches the report to the Report Design window.

```
; stockDesign::pushButton
method pushButton(var eventInfo Event)
var
  stockRep  Report
endVar
; the form's open method opened and retitled the Stock report
stockRep.attach("Stock Report")
stockRep.design()            ; switch to Design mode
endMethod
```

▪

enumUIObjectNames method

See also Example Report Type

Creates a table listing the UIObjects contained in a report.

Syntax

```
enumUIObjectNames ( const tableName String ) Logical
```

Description

enumUIObjectNames creates a Paradox table listing the name and type of each object contained in a report. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of *tableName* is:

Field Name	Type	Size
ObjectName	A	128
ObjectClass	A	32

▪

enumUIObjectNames example

In the following example, the **pushButton** method for *describeReport* uses **enumUIObjectNames** and **enumUIObjectProperties** to document a report.

```
; describeReport::pushButton
method pushButton(var eventInfo Event)
var
  ordersRep Report
  tempTable TableView
endVar
ordersRep.load("Orders.rsl")                    ; load report in Report
Design window
ordersRep.enumUIObjectNames("ordnames.db")      ; write names to table
ordersRep.enumUIObjectProperties("ordprops.db") ; write properties to table
ordersRep.close()
tempTable.open("ordnames")                      ; observe your handiwork
tempTable.wait()
tempTable.open("ordprops")
tempTable.wait()
tempTable.close()
endMethod
```

▪

enumUIObjectProperties method

See also Example Report Type

Lists the properties of each UIObject contained in a report.

Syntax

`enumUIObjectProperties ( const tableName String ) Logical`

Description

enumUIObjectProperties creates a Paradox table listing the name, property name, and property value of each object contained in a report. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails.

The structure of *tableName* is:

Field Name	Type	Size
ObjectName	A	128
PropertyName	A	64
PropertyType	A	48
PropertyValue	A	255

▪

enumUIObjectProperties example

See the example for **enumUIObjectNames**.

▪

load method

See also Example Report Type

Opens a report in the Report Design window.

Syntax

`load ( const reportName String, [const windowStyle LongInt [ , const x LongInt, const y LongInt, const w LongInt, const h LongInt ] ] ) Logical`

Description

load opens *reportName* in the Report Design window. You have the option to specify in *windowStyle* a WindowStyles constant (or combination of constants). You also have the option to specify (in twips) the window's size and position: arguments *x* and *y* specify the position of the upper left corner, arguments *w* and *h* specify the width and height, respectively. Both of these options were added in version 5.0. This method works only with saved reports (.RSL); it does not work with delivered reports (.RDL).

Compare this method to **open**, which opens a report in the Report window.

Note: Some report actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

▪

load example

In the following example, the **pushButton** method for the *loadOrders* button loads the ORDERS.RSL report in the Report Design window, creates a text box in the page header, and writes a string to the text box.

```
; loadOrders::pushButton
method pushButton(var eventInfo Event)
var
  ordersRep Report
  pageTitle UIObject
endVar
if ordersRep.load("Orders.rsl") then
  ; assume report has room in the page header for a text box
  pageTitle.create(TextTool, 1440*3, 720, 1440*2, 360, ordersRep)
  pageTitle.Name = "NewTitleText"
  pageTitle.Text = "Orders Report " + String(time())
  pageTitle.Color = LightBlue
  pageTitle.Visible = True
  ordersRep.run()
endIf
endMethod
```

moveToPage method

See also Example Report Type

Displays a specified page of a report.

Syntax

```
moveToPage ( const pageNumber SmallInt ) Logical
```

Description

moveToPage displays the page of a report specified in *pageNumber*. This method doesn't make the report active. If you want to make the report active, follow **moveToPage** with **bringToTop** (see the Form type for more information on **bringToTop**).

▪

moveToPage example

See the example for **currentPage.**

▪

open method

See also   Example   Report Type

Opens a report.

Syntax

```
1. open ( const reportName String [, windowStyle LongInt ] ) Logical
2. open ( const reportName String, const windowStyle LongInt, const x
SmallInt, const y SmallInt, const w SmallInt, const h SmallInt ) Logical
3. open ( const openInfo ReportOpenInfo ) Logical
```

Description

open displays the report specified in *reportName*. Optional arguments specify the location of the upper left corner of the report (*x* and *y*), the width and height (*w* and *h*), and style (*windowStyle*).

The value of *windowStyle* must be one of the WindowStyles constants. You can specify more than one window style by adding the constants. For example, the following code opens a report window that has horizontal and vertical scroll bars:

```
salesReport.open("sales.rsl", WinStyleDefault + WinStyleHScroll +
WinStyleVScroll)
```

Syntax 3 lets you specify form settings from *openInfo*, a predefined record of type ReportOpenInfo. A ReportOpenInfo record, which is an instance of the Record Type, has the following structure:

```
x, y, w, h          LongInt  ;size and position of report
name                String   ;name of report to open (preView)
masterTable         String   ;master table name
queryString         String   ;run this query (actual query string)
restartOptions      SmallInt ;one of the ReportPrintRestart constants
windowStule         LongInt  ;one of the WindowStyle constants
```

Note: Some report actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** procedure in the System type for more information.

open example

In the following example, the **pushButton** method for *openSmall* opens the ORDERS.RSL report and minimizes it by supplying a window style constant of WinStyleMinimize.

```
; openSmall::pushButton
method pushButton(var eventInfo Event)
var
  ordersRep Report
endVar
ordersRep.open("Orders.rsl", WinStyleMinimize)   ; open Orders Report
minimized
endMethod
```

▪

print method

See also Example Report Type
Beginner

Prints a report.

Syntax

```
1. print ( ) Logical
2. print ( const reportName String, const reportPrintRestart SmallInt ) Logical
3. print (const ri ReportPrintInfo ) Logical
```

Description

print prints a report. With syntax 1, Paradox opens the Print dialog box for the current report, which allows the user to specify print settings. Syntax 2 lets you specify a report name in *reportName* and set restart options in *reportPrintRestart*. The value of *reportPrintRestart* must be one of the ReportPrintRestart constants. Syntax 3 lets you set print settings with a ReportPrintInfo record. The predefined ReportPrintInfo records, which are of the Record Type, have the following structure:

Field name	Type	Description
endPage	LongInt	The last page of a range. Default: last page of the report.
makeCopies	Logical	Specifies how copies are made: by Paradox or the printer. True = Paradox make copies; False = printer makes copies. Default: True. The value is ignored if the printer cannot print multiple copies.
masterTable	String	Name of the master table for the report.
name	String	Name of a report to run, if it's not already running.
nCopies	SmallInt	Number of copies. Default: 1
orient	SmallInt	Page orientation. Use one of the three ReportOrientation Constants: Landscape, Portrait, or the windows default.
pageIncrement	SmallInt	Page increment for multi-pass printing. Default: 1
panelOptions	SmallInt	Specifies how to handle overflow pages. Use one of the ReportPrintPanel constants. Default: PrintClipToWidth
printBackwards	Logical	Specifies whether to print forward (from first page to last page) or backward (from last page to first page). False = forward, True = backward. Default: False
queryString	String	Specifies a QBE string to execute.
restartOptions	SmallInt	Specifies what to do when data changes while printing report. Use one of the ReportPrintRestart constants. Default: PrintReturn
startPage	LongInt	The first page of a range. Default: 1
startPageNum	LongInt	(Added in version 5.0.) The page number to print on the first page of the report. Incremented for subsequent pages. Default: 1
xOffset	LongInt	Horizontal page offset. Default: 0
yOffset	LongInt	Vertical page offset. Default: 0

print example

For examples of printing using syntax 1, see the example for **attach.** The following example shows how to use syntax 3 to print using a ReportPrintInfo record.

```
; printWithRecord::pushButton
method pushButton(var eventInfo Event)
var
  stockRep  Report
  repInfo   ReportPrintInfo
endVar
; first, set up the repInfo record
repInfo.nCopies = 2
repInfo.makeCopies = True
repInfo.name = "Stock"
stockRep.print(repInfo)
endMethod
```

▪

run method

See also Example Report Type

Switches a report from the Report Design window to the Report window.

Syntax

```
run ( ) Logical
```

Description

run switches a report from the Design window to the View Data window. This method works only with saved reports (.RSL); it does not work with delivered reports (.RDL).

To switch from the View Data window to the Design window, use **design.**

Note: Some report actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** procedure in the System type for more information.

▪

run example

See the example for **load**.

▪

setMenu method

See also   Example   Report Type

Associates a menu with a report.

Syntax

```
setMenu ( const menuVar Menu )
```

Description

setMenu associates the menu specified in *menuVar* with a report. This method performs the same function as Menu::**show**, and adds the following features:

- When the report gets focus, Paradox displays the associated menu.
- Actions resulting from choices from that menu are sent to that report.

Note: When you build a custom menu for a report, use MenuCommands constants (like MenuFilePrint) to assign ID values to menu items. These are the *only* values a report can respond to, because (unlike a form) a report has no **menuAction** method you can modify to handle menu choices.

▪

setMenu example

The following example is a script. It opens a report, builds a simple menu, then uses **setMenu** to assign the menu to the report.

```
method run(var eventInfo Event)
   var
      reOrders    Report
      muOrderRpt    Menu
      puRptFile    PopUpMenu
   endVar

; Build a menu for the report.
   reOrders.open("orders")

; Setting the StandardMenu property to False
; (either in ObjectPAL code or interactively)
; can reduce flicker when changing menus.
   reOrders.StandardMenu = False

; IMPORTANT: When you build a custom menu for a report,
; use MenuCommands constants (like MenuFilePrint) to assign
; ID values to menu items. These are the only values a report
; can respond to, because (unlike a form) a report has no
; menuAction method you can modify to handle menu choices.

   puRptFile.addText("&Print Report", MenuEnabled, MenuFilePrint)
   puRptFile.addText("&Exit", MenuEnabled, MenuFileExit)
   muOrderRpt.addPopUp("&File", puRptFile)
   reOrders.setMenu(muOrderRpt)
endMethod
```

Script type

See also Changes

A Script type was added to ObjectPAL in version 5.0. It includes methods for manipulating scripts and the code they contain from within an ObjectPAL method or procedure.

The Script type includes several derived methods from the Form type.

Methods for the Script type

Form	Script
deliver	**attach**
enumSource	**create**
enumSourceToFile	**load**
formReturn	**run**
isCompileWithDebug	
methodDelete	
methodGet	
methodSet	
save	
setCompileWithDebug	

Changes to Script type methods

The Form type has two new methods for version 7: **isCompileWithDebug** and **setCompileWithDebug**. Those two new Form methods now appear in the list of derived types for the Script type.

The Script type is new for version 5.0, so all of the Script type methods are new. The Script type includes several derived methods from the Form type.

▪

attach method

See also Example Script Type

Associates a Script variable with the current script.

Syntax

`attach ( )` `Logical`

Description

attach associates a Script variable with the current script. This method only works if called in code attached to the script itself; therefore, the script must be running. In other words, this method lets a running script create a handle to itself. ObjectPAL can't return Script variables or pass them as arguments, so you can only use the handle within the method that created it. **attach** can be used with **enumSource** or **enumSourceToFile** to create a script that enumerates its own code.

This method returns True if it succeeds; otherwise, it returns False.

■

attach example

The following example shows how to use **attach** to create a script that enumerates its own source to a text file. The code is attached to the script's built-in **run** method, which executes when you run (play) the script.

```
method run(var eventInfo Event)
  var
    s Script
  endVar
  s.attach()
  s.enumSourceToFile("script.src", Yes)
endMethod
```

▪

create method

See also Example Script Type

Creates a script.

Syntax

`create ( ) Logical`

Description

create creates an empty script, but *does not* display an Editor window. Use **methodSet** to add code to the script.

▪

create example

In the following example, the **pushButton** method for a button named *editScript* creates a script named MSG. Then it calls **methodSet** to attach code to its built-in **run** method, calls **save** to save the script and name it NewMsg (Paradox appends the .SSL extension automatically), then calls **run** to run it.

```
; editScript::pushButton
method pushButton(var eventInfo Event)
  var
    theScript    Script
    stMsg       String
  endVar

stMsg =
"method newMsg()
  msgInfo("New message", "New message")
endMethod"

  theScript.create()
  theScript.methodSet("run", stMsg)
  theScript.save("NewMsg") ; Saves script as NEWMSG.SSL.
  theScript.run()          ; Calls the script's built-in run method.
endMethod
```

▪

load method

See also Example Script Type

Loads a script into system memory.

Syntax

```
load ( const scriptName String ) Logical
```

Description

load loads the script specified in *scriptName* into system memory. It does not display an Editor window. If you don't specify a path or an alias, Paradox looks for the script in :WORK:. This method returns True if it succeeds; otherwise, it returns False.

load example

In the following example, the **pushButton** method for a button named *editScript* loads the script named MSG (which must have been created and saved previously). Then it calls **methodSet** to add a custom method, calls **save** to save the script, then calls **run** to run it.

```
; editScript::pushButton
method pushButton(var eventInfo Event)
  var
    theScript    Script
    stMsg       String
  endVar

stMsg =
"method newMsg()
  msgInfo("New message", "New message")
endMethod"

  if theScript.load("msg") then
    theScript.methodSet("newMsg", stMsg)
    theScript.save()
    theScript.run() ; Executes the script's built-in run method.
  else
    errorShow("Couldn't load the script.")
  endIf
endMethod
```

run method

See also Example Script Type

Runs a script.

Syntax

```
run ( ) AnyType
```

Description

run runs a script by calling its built-in **run** method, the same as if you had called the System procedure **play**. To return a value from a script, you must call **formReturn** from within the script.

▪

run example

This example shows how to run a script and how to make a script return a value. The following code is attached to a button in a form. It runs a script which returns a value, then it displays the returned value in a dialog box.

```
method pushButton(var eventInfo Event)
   var
      scTest      Script
      atRetVal   AnyType
   endVar

   scTest.load("test")
   atRetVal = scTest.run()
   atRetVal.view()
endMethod
```

The following code is attached to a script's built-in **run** method. It assigns a value to a variable, then returns the value to the form.

```
method run(var eventInfo Event)
   var
      atNow   AnyType
   endVar
   atNow = time()
   formReturn(atNow)
endMethod
```

Session type

Changes

A Session object represents a channel to the database engine. Opening a Paradox application opens one session by default, and you can use ObjectPAL to open other sessions from within an application; it is not necessary to open other sessions to use procedures from the Session type. The number of other sessions you can open depends on the system environment. Each session uses one user count.

Only the default session can be managed using Paradox interactively. You must manage other sessions with ObjectPAL.

Locks set by ObjectPAL interact as peers with locks set interactively in the same session.

Methods for the Session type

Session
addAlias
addPassword
addProjectAlias
advancedWildcardsInLocate
blankAsZero
close
enumAliasLoginInfo
enumAliasNames
enumDatabaseTables
enumDriverCapabilities
enumDriverInfo
enumDriverNames
enumDriverTopics
enumEngineInfo
enumFolder
enumOpenDatabases
enumUsers
getAliasPath
getAliasProperty
getNetUserName
ignoreCaseInLocate
isAdvancedWildcardsInLocate
isAssigned
isBlankZero
isIgnoreCaseInLocate
loadProjectAliases
lock
open
removeAlias

removeAllPasswords
removePassword
removeProjectAlias
retryPeriod
saveCFG
saveProjectAliases
setAliasPassword
setAliasPath
setAliasProperty
setRetryPeriod
unLock

Changes to Session type methods

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
addProjectAlias	addAlias
loadProjectAliases	enumAliasNames
removeProjectAlias	enumDatabaseTables
saveProjectAliases	enumDriverCapabilities
	enumFolder
	enumOpenDatabases
	enumUsers

▪

addAlias method/procedure

See also Example1 Example2 Example3 Session Type

Adds a database alias to a session.

Syntax

`1. addAlias ( const aliasName String, const type String, const path String ) Logical`

`2. addAlias ( const aliasName String, const type String, const params DynArray[ ] String ) Logical`

`3. addAlias ( const aliasName String, const existingAlias String ) Logical`

Description

addAlias adds a public alias to a session. To add a project alias, use **addProjectAlias.** Although the concept of "public" and "project" aliases is new with version 5.0, the functionality of **addAlias** is unchanged.

In syntax 1, specify the alias name in *aliasName*, the alias type ("Standard") in *type*, and the full DOS path in *path*.

In syntax 2, added in version 5.0, specify the alias name in *aliasName*, the SQL alias type (Interbase, Oracle, Sybase, or Informix) in *type*, and the parameters in *params*.

Syntax 3, added in version 5.0, allows you to copy an alias from *existingAlias* to *aliasName*.

An alias added using **addAlias** is known only to the session for which it is defined, and exists only until the session is closed. Use **saveCFG** to save public aliases in a file. Public aliases are stored in IDAPI32.CFG by default (or in the .CFG file of your choice). They are available from any working directory and visible to any application that uses BDE.

addAlias example 1

The following example adds an alias to the current session, then supplies the new alias to the **open** method defined for the Database type. This code is attached to the built-in **open** method for the *pageOne* page.

```
; pageOne::open
method open(var eventInfo Event)
var
  custInfo Database
endVar

; add the CustomerInfo alias to the current session
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")

; now use the alias specify the database to open
custInfo.open("CustomerInfo") ; opens the CustomerInfo database

endMethod
```

▪

addAlias example 2

The following example adds an Oracle type alias to the current session, then supplies the new alias to the **open** method defined for the Database type. This code is attached to the built-in **open** method for the *pgeOne* page.

```
; pgeOne::open
method open(var eventInfo Event)
   var
      tv       TableView
      SQLdb         Database
      AliasInfo     DynArray[]  String
   endVar

   AliasInfo["SERVER NAME"]       = "Server1"
   AliasInfo["USER NAME"]            = "guest"
   AliasInfo["OPEN MODE"]            = "READ/WRITE"
   AliasInfo["SCHEMA CACHE SIZE"]    = "8"
   AliasInfo["NET PROTOCOL"]         = "SPX/IPX"
   AliasInfo["LANGDRIVER"]           = ""
   AliasInfo["SQLQRYMODE"]           = ""
   AliasInfo["PASSWORD"]             = "guest"

   addAlias("Guest_Account", "Oracle", AliasInfo)
   SQLdb.open("Guest_Account", AliasInfo)
   tv.open(":Guest_Account:mprestwood.customer")
endMethod
```

addAlias example 3

The following example adds an alias to the current session by copying the existing *work* alias to the new alias *NewAlias.*

```
; btnCopyWork::pushButton
method pushButton(var eventInfo Event)
   addAlias("NewAlias", "work")
endMethod
```

▪

addPassword method/procedure

See also Example Session Type

Presents a password allowing access to a protected table.

Syntax

addPassword (const *password* String)

Description

addPassword presents to a Paradox session the password specified in *password*. (Passwords apply to Paradox tables only.) The maximum length of a password is 31 characters. Subsequent attempts to access a table protected using that password are not challenged. The argument *password* can represent either an owner password or an auxiliary password. Auxiliary passwords generally confer less comprehensive rights than owner passwords. *password* is case-sensitive; a table protected with "Sesame" won't open for "SESAME".

Passwords added using this method are valid only for the session they are presented in, and are in effect only until the session is closed. Presenting a password does not affect the state of tables: for example, an open table remains open.

Access to tables opened before the password is presented is controlled by passwords previously presented. For instance, if a table was opened using an auxiliary password, the access rights to that table do not change upon presentation of the owner password. To confer owner rights to a previously-opened table, you should first close the table, then present the owner password, then reopen the table.

Use **removePassword** to restore protection.

▪

addPassword example

The following example acquires a password from the user, then presents it to the current session.

```
; getAddPass::pushButton
method pushButton(var eventInfo Event)
var
  newPass String
endVar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  newPass.view("Enter Password (up to 31 characters) to Add.")
  ses.addPassword(newPass)
else
  msgStop("Help!","Session variable is not Assigned!")
endIf
endMethod
```

addProjectAlias method/procedure

See also Example1 Example2 Example3 Session Type

Adds a project alias to a session.

Syntax

```
1. addProjectAlias ( const aliasName String, const type String, const path
String ) Logical
2. addProjectAlias ( const aliasName String, const type String, const params
DynArray[ ] String ) Logical
3. addProjectAlias ( const aliasName String, const existingAlias String )
Logical
```

Description

addProjectAlias adds a project alias to a session. Use **addAlias** to add a public alias.

In syntax 1, specify the alias name in *aliasName*, the alias type ("Standard") in *type*, and the full DOS path in *path*.

In syntax 2, specify the alias name in *aliasName*, the SQL alias type (Interbase, Oracle, Sybase, or Informix) in *type*, and the parameters in *params*.

Syntax 3 allows you to copy an alias from *existingAlias* to *aliasName*.

An alias added using **addProjectAlias** is known only to the project in which it is defined, and exists only until the working directory is changed. Use **saveProjectAliases** to save project aliases in a file.

When :WORK: is set (for example, at startup) or changed (either interactively or through ObjectPAL), Paradox discards all current project aliases and loads those project aliases that are specific to the new working directory. Public aliases remain active and available; if a project alias has the same name as a public alias, Paradox does not load the project alias. By default, Paradox reads project aliases from :WORK:PDOXWORK.CFG, but you can use **loadProjectAliases** to specify a different file.

■

addProjectAlias example 1

The following example adds an alias to the current project, then supplies the new alias to the **open** method defined for the Database type. This code is attached to the built-in **open** method for the *pageOne* page.

```
; pageOne::open
method open(var eventInfo Event)
var
  custInfo Database
endVar

; add the CustomerInfo alias to the project
addProjectAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")

; now use the alias specify the database to open
custInfo.open("CustomerInfo") ; opens the CustomerInfo database

endMethod
```

addProjectAlias example 2

The following example adds an Oracle type alias to the current project, then supplies the new alias to the **open** method defined for the Database type. This code is attached to the built-in **open** method for the *pgeOne* page.

```
; pgeOne::open
method open(var eventInfo Event)
   var
      tv       TableView
      SQLdb         Database
      AliasInfo     DynArray[]  String
   endVar

   AliasInfo["SERVER NAME"]        = "Server1"
   AliasInfo["USER NAME"]             = "guest"
   AliasInfo["OPEN MODE"]             = "READ/WRITE"
   AliasInfo["SCHEMA CACHE SIZE"]     = "8"
   AliasInfo["NET PROTOCOL"]          = "SPX/IPX"
   AliasInfo["LANGDRIVER"]            = ""
   AliasInfo["SQLQRYMODE"]            = ""
   AliasInfo["PASSWORD"]              = "guest"

   addProjectAlias("Guest_Account", "Oracle", AliasInfo)
   SQLdb.open("Guest_Account", AliasInfo)
   tv.open(":Guest_Account:mprestwood.customer")
endMethod
```

addProjectAlias example 3

The following example adds an alias to the current session by copying the existing *work* alias to the new alias *NewAlias.*

```
; btnCopyWork::pushButton
method pushButton(var eventInfo Event)
   addProjectAlias("NewAlias", "work")
endMethod
```

advancedWildcardsInLocate procedure

See also   Example   Session Type

Specifies whether this session can use advanced wildcards in locate operations.

Syntax

advancedWildcardsInLocate ([const ***yesNo*** Logical] **)**

Description

advancedWildcardsInLocate specifies whether the current session should use advanced wildcards found in pattern strings during locate operations. If *yesNo* is Yes, pattern strings used in locate operations can contain advanced wildcard characters; if set to No, pattern strings in locate operations cannot contain advanced wildcards. If omitted, *yesNo* is Yes by default.

advancedWildcardsInLocate example

The following example calls **advancedWildcardsInLocate**, if necessary, to specify that advanced wildcards can be used in a locate operation. Then the code continues with a call to **locatePattern** that uses an advanced wildcard pattern.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  thisSession Session
endVar

if tc.open("Orders.db") then

  ; if advanced wild cards can't be used in patterns
  if NOT isAdvancedWildcardsInLocate() then
    ; specify that this session can use advanced
    ; pattern characters in subsequent locate operations
    advancedWildcardsInLocate(Yes)
  endIf

  if tc.locatePattern("Ship VIA", "[^UPS]") then
    msgInfo("Order Number", tc."Order No")
  else
    msgStop("Error", "Can't find record")
  endIf
else
  msgStop("Error", "Can't open Orders table.")
endIf

endMethod
```

▪

blankAsZero method/procedure

See also Example Session Type

Specifies whether to treat blank values as zeros in calculations.

Syntax

```
blankAsZero ( const yesNo Logical )
```

Description

blankAsZero specifies whether to assign blank numeric fields a value of 0 in calculations. If *yesNo* is Yes, blanks are treated as zeros. If *yesNo* is No, they are not.

Calculations affected by **blankAsZero** include:

- Calculated fields in forms and reports
- Calculations in queries
- Column calculations that involve either the number of fields or the number of non-blank fields, for example, those performed with cCount, cAverage, and others

You may want to perform these calculations differently depending on the state of **blankAsZero**. You can use **isBlankZero** to test the state, and **blankAsZero** to set it.

blankAsZero example

The following example sets **blankAsZero**, if necessary, to True so that a call to the **cAverage** method treats blank field values as zeros.

```
; getAvgPmt::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

if tc.open("Orders.db") then
  if not isBlankZero() then
    blankAsZero(True)
  endIf

  msgInfo("Average Amount Paid", tc.cAverage("Amount Paid"))

else
  msgStop("Error", "Can't open Orders table.")
endIf

endMethod
```

close method

See also Example Session Type

Closes a session.

Syntax

```
close ( ) Logical
```

Description

close ends a session by closing the channel to the database engine. **close** frees one user count, and makes the Session variable unassigned.

▪

close example

For the following example, assume that the variable *ses* is assigned to an open session. This example closes the session *ses*.

```
; closeSession::pushButton
method pushButton(var eventInfo Event)
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  if ses.close() then
    msgInfo("We have TouchDown","Session close Successful.")
  else
    msgStop("Crash and Burn","Session close Unsuccessful.")
  endIf
else
  msgStop("Help!","Session variable is not Assigned! Who am I?")
endIf
endMethod
```

▪

enumAliasLoginInfo method

See also Example Session Type

Writes data about a specified server alias to a table.

Syntax

`enumAliasLoginInfo ( const tableName String, const aliasName String ) Logical`

Description

enumAliasLoginInfo writes information about the server alias specified in *aliasName* to the Paradox table specified in *tableName*. Returns True if successful; otherwise returns False. **enumAliasLoginInfo** operates on aliases stored in IDAPI32.CFG and on new aliases opened and stored in system memory. This method fails if the table specified in *tableName* is already open.

This method is only applicable to remote databases, and not to "standard" (Paradox or dBASE) databases.

The structure of the resulting *tableName* table is:

Field name	Type	Description
DBName	A32*	Name of the database.
Property	A32*	Name of the property. Examples of properties (which vary depending on the database type) are: OPEN MODE, NET PROTOCOL, SERVER NAME, and USER NAME.
PropertyValue	A82	Value of the property.

enumAliasLoginInfo example

The following example calls **enumAliasLoginInfo** to write data about an alias to a Paradox table. Then it searches the table to test whether the OPEN MODE property for the alias is set to READ/WRITE. If it is, the code calls a custom procedure named doSomething (which is assumed to be defined elsewhere) to continue processing. Otherwise, the code displays information about properties and property values in a modal dialog box to inform the user of the problem.

```
method pushButton(var eventInfo Event)

   var
      db                 Database
      aliasInfoTC        TCursor
      aliasName,
      infoTableName,
      fieldName1,
      fieldName2,
      propName,
      propVal            String
      propValDA          DynArray[] AnyType
   endVar

   ; initialize variables
   aliasName = "itchy"
   infoTableName = "dbAlias.db"
   fieldName1 = "Property"
   fieldName2 = "PropertyValue"
   propName  = "OPEN MODE"
   propVal  = "READ/WRITE"


   ; open database, get alias info
   if db.open(aliasName) then
      if enumAliasLoginInfo(infoTableName, aliasName) then
         aliasInfoTC.open(infoTableName)

         ; search for info of interest
         if aliasInfoTC.locate(fieldName1, propName) then

            ; compare expected and actual values
            if aliasInfoTC.(fieldName2) <> propVal then

               ; inform user if values don't match
               propValDA["Property:"] = aliasInfoTC.(fieldName1)
               propValDA["Expected value:"] = propVal
               propValDA["Actual value:"] = aliasInfoTC.(fieldName2)
               propValDA.view("Property mismatch")
               return
            endIf

         else
            errorShow("Property not found.")
            return
         endIf
      else
         errorShow("Can't write to table: " + infoTableName)
         return
      endIf
   else
      errorShow("Couldn't open " + aliasName)
      return
   endIf

   doSomething() ; if property values are OK, continue processing
```

```
endMethod
```

▪

enumAliasNames method/procedure

See also Example Session Type

Lists the names of database aliases available to a session.

Syntax

```
1. enumAliasNames ( const tableName String [ , const LoginInfoTableName
String ] ) Logical
2. enumAliasNames ( var aliasNames Array[ ] String ) Logical
```

Description

enumAliasNames lists the names of database aliases available to a session.

Syntax 1 creates a Paradox table *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates the table in the working directory.

The structure of *tableName* is

Field Name	Type	Description
DBName	A32*	The database alias name
DBType	A32	The driver type
DBPath	A82	The alias path

The optional argument *LoginInfoTableName* was added in version 5.0. If you include this argument, Paradox writes login data to the table, just as if you had called **enumAliasLoginInfo.**

The structure of the resulting table is

Field name	Type	Description
DBName	A32*	Name of the database.
Property	A32*	Name of the property. Examples of properties (which vary depending on the database type) are OPEN MODE, NET PROTOCOL, SERVER NAME, and USER NAME.
PropertyValue	A82	Value of the property.

Syntax 2 (added in version 5.0) assigns the database names to items in an array *aliasNames* that you declare and pass as an argument.

▪

enumAliasNames example

In the following example, the **pushButton** method for *getAliasButton* writes the alias names active for the current session to an array. If the array does not contain the name of a specified alias, a call to **addAlias** adds it to the session.

```
; getAliasButton::pushButton
method pushButton(var eventInfo Event)
   var
      stAliasName,
      stAliasPath   String
      arAliasNames   Array[] String
   endVar

   stAliasName = "NewCust"
   stAliasPath = "g:\\netdata\\newcust"

   enumAliasNames(arAliasNames)   ; List names to an array.
   if arAliasNames.contains(stAliasName) then
         return
   else
         addAlias(stAliasName, "STANDARD", stAliasPath)
   endIf
endMethod
```

▪

enumDatabaseTables method/procedure

See also Example Session Type

Lists the tables and other files in a database.

Syntax

```
1. enumDatabaseTables ( const tableName String, const databaseName String,
const fileSpec String )
2. enumDatabaseTables ( var tableNames Array[ ] String, const databaseName
String, const fileSpec String )
```

Description

enumDatabaseTables lists the tables and other files in the database specified by *databaseName*, where *databaseName* is an alias known to the session. *fileSpec* specifies a DOS file specification (it can include the wildcards * and ?).

Syntax 1 creates the Paradox table *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of the table is

Field name	Type	Description
DBName	A32*	The database alias
TableName	A32*	The name of the table (or other file, depending on the file specification)

Syntax 2 (added in version 5.0) assigns the table names to items in an array *tableNames* that you pass as an argument.

▪

enumDatabaseTables example

The following example lists the Paradox and dBASE tables (and any other file whose extension is DB followed by 0 or 1 characters) in the user's private directory. It uses **enumDatabaseTables** as a procedure and works in the current session.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dbName,
  fileSpec,
  tbName    String
  tv1       TableView
endVar

; Init variables.
dbName   = ":PRIV:"
fileSpec = "*.db?" ; Lists <filename>.db and <filename>.dbf
tbName   = "TabList"

enumDatabaseTables(tbName, dbName, fileSpec)
tv1.open(tbName)  ; Open the created table.
endMethod
```

▪

enumDriverCapabilities procedure

See also   Example   Session Type

Lists the capabilities of the current driver.

Syntax

```
enumDriverCapabilities ( const drvCapName String, const tblCapName String,
const fldCapName String [ , const inxCapName String ] ) Logical
```

Description

enumDriverCapabilities creates three Paradox tables that list the capabilities of the current driver. Paradox overwrites the tables (if they exist) without asking for confirmation. You can include an alias or path in the specified table names; if no alias or path is specified, Paradox creates the tables in the working directory.

Driver capabilities are written to the table *drvCapName* (each supported table type is described by a record), which has the following structure:

Field name	Type	Description
DriverType	A32*	Name of driver; for example, DBASE.
Description	A32	Text describing the driver; for example, dBASE driver.
Category	A32	Driver category.
DB	A4	If True or Yes, the driver supports a true database concept; otherwise, the driver uses a file system to simulate a database.
DBType	A32	Database type to be used; for example, STANDARD.
MultiUser	A4	If True or Yes, the driver supports multiuser access; otherwise, it doesn't.
ReadWrite	A4	If True or Yes, the driver is read-write; otherwise, it's read only.
Transactions	A4	If True or Yes, the driver supports transactions; otherwise, it doesn't.
PassThruSQL	A4	If True or Yes, the driver supports pass-through SQL; otherwise, it doesn't.
Login	A4	If True or Yes, the driver requires an explicit login (for example, to access a SQL server); otherwise, it doesn't.
CreateDb	A4	If True or Yes, the driver can create a database; otherwise, it can't.
DeleteDb	A4	If True or Yes, the driver can delete a database; otherwise, it can't.
CreateTable	A4	If True or Yes, the driver can create a table; otherwise, it can't.
DeleteTable	A4	If True or Yes, the driver can delete a table; otherwise, it can't.
MultiPasswords	A4	If True or Yes, the driver supports multiple passwords; otherwise, it doesn't.

Table capabilities are written to the table *tblCapName* (each supported table type is described by a record), which has the following structure:

Field name	Type	Description
DriverType	A32*	Type of table; for example, dBASE.
TableType	A32*	Text describing the table type; for example, PDOX 5.0.
Format	A32*	Table format; for example, CLUSTERED.
ReadWrite	A4	If True or Yes, the user can read from and write to the table; otherwise, the user can read only.
Create	A4	If True or Yes, the user can create a table of this type; otherwise, the

		user cannot.
Restructure	A4	If True or Yes, the user can restructure a table of this type; otherwise, the user cannot.
ValChecks	A4	If True or Yes, the user can specify validity checks for a table of this type; otherwise, the user cannot.
Security	A4	If True or Yes, the table can be password-protected; otherwise, it cannot.
RefInt	A4	If True or Yes, the table can participate in a referential integrity relationship; otherwise, it cannot.
PrimaryKey	A4	If True or Yes, the table supports primary keys; otherwise, it does not.
Indexing	A4	If True or Yes, the table can have other (secondary) indexes; otherwise, it cannot.
NoFieldType	A6	Number of physical field types supported.
MaxRecSize	A6	Maximum record size (in bytes).
MaxFlds	A6	Maximum number of fields per record.

Field capabilities are written to the table *fldCapName*, which has the following structure:

Field name	**Type**	**Description**
DriverType	A32*	Type of table; for example, dBASE.
TableType	A32*	Text describing the table type; for example, PDOX 5.0.
Format	A32*	Table format; for example, CLUSTERED.
FieldType	A32*	Field type.
Description	A32	Description of field type; for example, Long integer.
NativeType	A6	Numeric value of native field type; for example, 266.
XType	A6	Numeric value of translated field type; for example, 3.
XSubType	A6	Numeric value of translated field subtype; for example, 3.
MaxUnits1	A6	Maximum places to the left of the decimal point (or number of characters); for example, 240.
MaxUnits2	A6	Maximum places to the right of the decimal point; for example, 19.
Size	A6	Field size; for example, 8.
Required	A4	If True or Yes, the field is a required field; otherwise, it isn't.
Default	A4	If True or Yes, the field has a specified default value; otherwise, it doesn't.
Min	A4	If True or Yes, the field has a specified minimum value; otherwise, it doesn't.
Max	A4	If True or Yes, the field has a specified maximum value; otherwise, it doesn't.
RefInt	A4	If True or Yes, the field is part of a referential integrity relationship; otherwise, it isn't.
Other	A4	Reserved.
Key	A4	If True or Yes, the field can be part of an index (keyed).
Multi	A4	If True or Yes, the driver supports more than one of these fields per

		record.
MinUnits1	A6	Minimum places to the left of the decimal point (or number of characters); for example, 240. Added in version 5.0.
MinUnits2	A6	Minimum places to the right of the decimal point; for example, 19. Added in version 5.0.
Createable	A4	If True or Yes, the driver can create a table using this field type. Added in version 5.0.

If the optional argument *inxCapName* (added in version 5.0) is included, index capabilities are written to the table specified in *inxCapName*, which has the following structure:

Field name	**Type**	**Description**
DriverType	A32*	Type of table; for example, dBASE.
TableType	A32*	Text describing the table type; for example, PDOX 5.0.
Format	A32*	Table format; for example, CLUSTERED.
Name	A32*	Internal name describing the type of index; for example, SECONDARY. A corresponding description is provided in the Description field.
Format1	A32*	Index format; for example, BTREE.
Description	A32	Description of the index; for example, Nonmaintained Secondary index.
Composite	A4	Yes if the index supports composite keys; otherwise No.
Primary	A4	Yes if the index is a primary index; otherwise No.
Unique	A4	Yes if the index is a unique index; otherwise No.
keyDescending	A4	Yes if the whole key can be descending; otherwise No.
fldDescending	A4	Yes if the index is field level descending; otherwise No.
Maintained	A4	Yes if the index is a maintained index; otherwise No.
Subset	A4	Yes if the index is a subset index; otherwise No.
KeyExp	A4	Yes if the index is an expression index; otherwise No.
CaseInsensitive	A4	Yes if the index is insensitive to case; otherwise No.

Categories for Session::enumDriverCapabilities

Value	Description
File	File-based (Paradox, dBASE).
SQL Server	SQL-based server.
Other Server	Other kind of server (not file, not SQL).

Field types for Session::enumDriverCapabilities

The following tables (updated for version 5.0) list the field types for Paradox and dBASE tables:

Paradox field type	Return value
Alpha	ALPHA
Autoincrement	AUTOINCREMENT
BCD	BCD
Binary	BINARY
Bytes	BYTES
Date	DATE
FmtMemo	FMTMEMO
Graphic	GRAPHIC
Logical	LOGICAL
LongInt	LONG
Memo	MEMO
Money	MONEY
Number	NUMBER
OLE	OLE
Short	SHORT
Time	TIME
TimeStamp	TIMESTAMP

dBASE field type	Return value
BINARY	BINARY
CHAR	CHARACTER
DATE	DATE
FLOAT	FLOAT
LOGICAL	LOGICAL
MEMO	MEMO
NUMBER	NUMERIC
OLE	OLE

enumDriverCapabilities example

In the following example, the *describeDriver* button creates and views three tables that describe the engine driver.

```
; describeDriver::pushButton
method pushButton(var eventInfo Event)
var
  tv1, tv2, tv3  TableView
endVar
enumDriverCapabilities("dbcap", "tblcap", "fldcap")
tv1.open("dbcap")
tv2.open("tblcap")
tv3.open("fldcap")
endMethod
```

▪

enumDriverInfo procedure

See also Example Session Type

Lists information about available drivers.

Syntax

`enumDriverInfo ( const tableName String )`

Description

enumDriverInfo lists information about driver types currently available. Information is written to the table *tableName*. If *tableName* exists, it is overwritten without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is

Field name	Type	Description
DriverType	A32*	Type (or name) of driver; for example, PARADOX. See **enumDriverNames** for more information.
Topic	A32*	Driver function; for example, TABLE CREATE. See **enumDriverTopics** for more information.
Property	A32*	Property of corresponding driver function; for example, BLOCK SIZE.
PropertyValue	A68	Value of corresponding property; for example, 2048.

enumDriverInfo example

The following example enumerates driver information to a table named *DriveInf*, then views the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tv1 TableView
endVar
; create and view the DriveInf table
enumDriverInfo("Driveinf")
tv1.open("DriveInf")
endMethod
```

enumDriverNames method/procedure

See also Example Session Type

Creates a Paradox table listing the names of the drivers available in the current session.

Syntax

```
enumDriverNames ( const tableName String )
```

Description

enumDriverNames writes the driver names currently available to the table *tableName*. If *tableName* already exists, it is overwritten without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of the table is DriverType, A32*.

▪

enumDriverNames example

The following example enumerates available driver names to a table named *DrivName*, then views the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tv1  TableView
endVar
; create and view the DrivName table
enumDriverNames("DrivName")
tv1.open("DrivName")
endMethod
```

▪

enumDriverTopics procedure

See also Example Session Type

Lists the topics currently available for each driver type.

Syntax

`enumDriverTopics ( const tableName String )`

Description

enumDriverTopics writes the driver topics available for each driver type to the table *tableName*. If *tableName* already exists, it is overwritten without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is:

Field name	Type	Description
DriverType	A32*	Type (or name) of driver; for example, PARADOX. See **enumDriverNames** for more information.
Topic	A32*	Driver function. For Paradox and dBASE tables, the topics are INIT and TABLE CREATE.

▪

enumDriverTopics example

The following example enumerates available driver topics to a table named *DrivTop*, then views the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tv1  TableView
endVar
; create and view the DrivTop table
enumDriverTopics("drivtop")
tv1.open("drivtop")
endMethod
```

■

enumEngineInfo procedure

See also Example Session Type

Creates a Paradox table listing the current BDE engine properties.

Syntax

```
enumEngineInfo ( const tableName String )
```

Description

enumEngineInfo creates a Paradox table that describes the contents of the BDE System Information dialog box. Each setting name and value is written to a record in the table *tableName*. If *tableName* already exists, it is overwritten without asking confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is:

Field name	Type	Description
Property	A32*	Engine property
PropertyValue	A68	Value of corresponding property.

▪

enumEngineInfo example

The following example enumerates engine information to a table named *EngInf*, then views the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tv1  TableView
endVar
enumEngineInfo("EngInf")
tv1.open("EngInf")
endMethod
```

Properties for Session::enumEngineInfo

Engine property	Description
LANGDRIVER	Name of language driver; for example, ascii.
LANGDRVDIR	Language driver directory.
LOCAL SHARE	TRUE if Local Share is active, FALSE if it is not active.
MAXBUFSIZE	Maximum buffer size (in bytes).
MAXFILEHANDLES	Maximum number of file handles.
MINBUFSIZE	Minimum buffer size (in bytes).
NET DIR	Path to NET directory.
NET TYPE	Network type.
SYSFLAGS	Number of system flags.
VERSION	BDE version number.

▪

enumFolder procedure

See also Example Session Type

Lists the names of files in a folder (project).

Syntax

```
1. enumFolder ( const tableName String [ , const fileSpec String ] ) Logical
2. enumFolder ( var result Array[ ] String [ , const fileSpec String ] ) Logical
```

Description

enumFolder lists the names of files in a folder (project). In version 5.0, the Project Viewer replaced the Folder. By default, a project includes all the objects in :WORK: and :PRIV:. You can also add references to objects in other directories.

Syntax 1 creates the Paradox table *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

Syntax 2 lists the files in the array *result* which you must declare and pass as an argument. For each file, the array lists the file name (and extension, if one exists), and includes the path if the file is not in :WORK:.

Both syntaxes include an option to specify a *fileSpec* to list files with a particular extension. For instance, to list all forms in a file, include a *fileSpec* of ".FSL".

The structure of the table created by syntax 1 is

Field name	Type	Description
Name	A128	File name (and extension, if one exists). Includes the path if the file is not in :WORK:.
LocalName	A68	File name without extension. Includes the path if the file is not in :WORK:.
IsReference	A4	If True or Yes, the file name refers to a file in a directory other than :WORK:; if No, the file is in :WORK:.
IsPrivate	A4	If True or Yes, the file name refers to a file in :PRIV:; otherwise, the file resides elsewhere.
IsTemp	A4	Reserved.
Position	A10	Reserved.

▪

enumFolder example

In the following example, the method prompts the user to enter a file specification (such as "*.FSL"). The file specification entered is then used by **enumFolder** to create a table listing the files that match the specification.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  filespec String
  tv1      TableView
endVar
filespec.view("Enter file name specification")
enumFolder("PartCat", filespec)
message("Table lists files that match your specification.")
tv1.open("PartCat")
endMethod
```

▪

enumOpenDatabases method/procedure

See also Example Session Type

Lists the open databases.

Syntax

```
1. enumOpenDatabases ( const tableName String ) Logical
2. enumOpenDatabases ( var tableNames Array[ ] String ) Logical
```

Description

enumOpenDatabases lists the databases open in the current session.

Syntax 1 creates the Paradox table *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of the table is:

Field name	Type	Description
DBName	A32*	Database alias name.
DBType	A32	Database driver type.
ShareMode	A32	Database share mode.
OpenMode	A32	Database open mode.

Syntax 2 (added in version 5.0) writes the data to an array *tableNames* that you declare and pass as an argument.

▪

enumOpenDatabases example

In the following example, **enumOpenDatabases** creates a table *OPENDB.DB,* and then a tableview is opened to display the table.

```
; btnOpenDB :: pushButton
method pushButton(var eventInfo Event)
   var
      tv   TableView
   endVar

   enumOpenDatabases("OPENDB.DB")
   tv.open("OPENDB.DB")
endMethod
```

▪

enumUsers procedure

See also Example Session Type

Creates a Paradox table listing all known users with an open channel to the BDE engine.

Syntax

```
1. enumUsers ( const tableName String ) LongInt
2. enumUsers ( var userNames Array[ ] String ) LongInt
```

Description

enumUsers creates a list of all users with an open path to the BDE database engine.

Syntax 1 creates the table *tableName* that lists all users with an open path to BDE. If *tableName* exists, it is overwritten without asking for confirmation. You can include an alias or path in *tableName;* if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of the table is

Field Name	Type	Description
UserName	A15	Network user name
NetSession	N	Network session number
ProductClass	N	User's product class ID number
SerialNumber	A22	Serial number (version 1.0 only)

Syntax 2 (added in version 5.0) fills the array *userNames* with the network user names of users who currently have an open path to BDE. You must declare the array before calling this procedure.

enumUsers example

The following example writes information about current users to the table *Users*, then displays the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tv1 TableView
endVar
  enumUsers("users")
  tv1.open("users")
endMethod
```

getAliasPath method/procedure

See also Example Session Type

Returns the path for a specified alias.

Syntax

```
getAliasPath ( const aliasName String ) String
```

Description

getAliasPath returns the path for the alias *aliasName*.

getAliasPath example

The following example prompts the user for an alias name, then shows the path currently associated with that alias.

```
; getShowPath::pushButton
method pushButton(var eventInfo Event)
   var
      stPrompt,
      stAliasName,
      stCurrentPath,
      stMyPath      String
   endVar

   stPrompt    = "Enter an Alias Name."
   stAliasName = stPrompt
   stMyPath    = "d:\\pdoxwin\\data"

   stAliasName.view(stPrompt)    ; prompt for an alias name
   if stAliasName = stPrompt then
         return ; User didn't click the OK button.
   else
         stCurrentPath = getAliasPath(stAliasName)  ; get the path
   endIf

   if stCurrentPath = stMyPath then
         return
   else
         setAliasPath(stAliasName, stMyPath)
   endIf
endMethod
```

▪

getAliasProperty method

See also Example Session Type

Returns the value of a specified property for a specified server alias.

Syntax

```
getAliasProperty ( const aliasName String, const property String ) String
```

Description

getAliasProperty returns a string representing the value of the property specified in *property* for the server alias specified in *aliasName*. If the property is not valid for the alias, this method returns an error. **getAliasProperty** operates on aliases stored in IDAPI32.CFG and on new aliases opened and stored in system memory.

This method is only applicable to remote databases, and not to "standard" (Paradox or dBASE) databases.

▪

getAliasProperty example

The following example uses **getAliasProperty** to get the value of the OPEN MODE property. It compares the returned (actual) value with the expected value. If they match, the code calls a custom procedure named doSomething (assumed to be defined elsewhere) to continue processing. Otherwise, the code informs the user of a property mismatch and calls **setAliasProperty** to set the property to the expected value.

```
method pushButton(var eventInfo Event)

   var
      db                Database
      aliasName,
      propName,
      expectedPropVal,
      actualPropVal     String
      propValDA         DynArray[] AnyType
   endVar

   ; initialize variables
   aliasName = "itchy"
   propName = "OPEN MODE"
   expectedPropVal = "READ/WRITE"

   if db.open(aliasName) then

      ; get property value and compare with expected value
      actualPropVal = getAliasProperty(aliasName, propName)
      if actualPropVal = expectedPropVal then
         doSomething() ; continue processing
         return
      else

         ; inform the user if there's a mismatch
         propValDA["Property name"] = propName
         propValDA["Expected value"] = expectedPropVal
         propValDA["Actual value"] = actualPropVal
         propValDA.view("Property mismatch:")

         ; let user decide what to do
         if msgQuestion("Set property value?",
            "Set "+propName+" to " + expectedPropVal + "?") = "Yes" then

            ; set property to expected value and continue processing
          if setAliasProperty(aliasName, propName, expectedPropVal) then
               doSomething() ; Continue processing
               return
          else
               errorShow("Couldn't set property value.",
                         "Operation canceled.")
               return
          endIf

         else
            msgInfo("Operation canceled.", "Property not set.")
            return
         endIf

      endIf

   else
      msgStop(aliasName, "Couldn't open database.")
      return
   endIf
```

```
endMethod
```

getNetUserName method/procedure

See also Example Session Type

Returns the network user name for a session.

Syntax

```
getNetUserName ( ) String
```

Description

getNetUserName returns the name of the current network user.

-

getNetUserName example

The following example displays the current user's network name in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
msgInfo("Who Am I?", getNetUserName())
endMethod
```

▪

ignoreCaseInLocate procedure

See also Example Session Type

Specifies whether to ignore case sensitivity in locates.

Syntax

ignoreCaseInLocate ([const ***yesNo*** Logical] **)**

Description

ignoreCaseInLocate specifies whether the current session should ignore case-sensitivity during locate operations. If optional argument *yesNo* is Yes (or omitted), this subsequent locate operations will ignore case in string comparisons; if *yesNo* is No, locate operations will be case-sensitive.

ignoreCaseInLocate example

The following example calls **ignoreCaseInLocate**, if necessary, to set up for a call to the **locate** method.

```
; findName::pushButton
method pushButton(var eventInfo Event)
var
  tc                TCursor
   loIgnoreCase   Logical
endVar

if tc.open("Customer.db") then

   loIgnoreCase = isIgnoreCaseInLocate() ; Get user's setting.

  if loIgnoreCase then

    ; locate values based on value as entered
    ; (do not ignore case in string compares)
    ignoreCaseInLocate(No)
  endIf

  ; search for case-sensitive MacAnaly in Name field
  if tc.locate("Name", "MacAnaly") then
    tc.edit()
    tc.Name = "Macanaly"
    tc.endEdit()
  else
    message("Couldn't find MacAnaly...")
  endIf

   ignoreCaseinLocate(loIgnoreCase) ; Restore user's setting.

else
  msgStop("Error", "Can't open Customer table.")
endIf

endMethod
```

isAdvancedWildcardsInLocate procedure

See also Example Session Type

Reports whether this session is using advanced wildcards in locate operations.

Syntax

`**isAdvancedWildcardsInLocate ( )** Logical`

Description

isAdvancedWildcardsInLocate reports whether the current session is using advanced wildcards during locate operations that include pattern strings.

▪

isAdvancedWildcardsInLocate example

The following example calls **advancedWildcardsInLocate**, if necessary, to specify that advanced wild cards can be used in a locate operation. Then the code continues with a call to **locatePattern** that uses an advanced wildcard pattern.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  thisSession Session
endVar

if tc.open("Orders.db") then

  ; if advanced wild cards can't be used in patterns
  if NOT isAdvancedWildcardsInLocate() then
    ; specify that this session can use advanced
    ; pattern characters in subsequent locate operations
    advancedWildcardsInLocate(Yes)
  endIf

  if tc.locatePattern("Ship VIA", "[^UPS]") then
    msgInfo("Order Number", tc."Order No")
  else
    msgStop("Error", "Can't find record")
  endIf
else
  msgStop("Error", "Can't open Orders table.")
endIf

endMethod
```

▪

isAssigned method

See also Example Session Type

Reports whether a session variable is assigned.

Syntax

`isAssigned ( )` `Logical`

Description

isAssigned reports whether a Session variable is assigned.

▪

isAssigned example

See the example for **close**.

▪

isBlankZero method/procedure

See also Example Session Type

Reports whether blank values are being treated as zero in calculations.

Syntax

```
isBlankZero ( ) Logical
```

Description

isBlankZero returns True if blank fields are treated as fields with a value of zero in calculations**,** or are counted as filled fields in counting calculation (for example**, cCount**). If blank fields are treated as blanks or are being ignored in calculations and counts, **isBlankZero** returns False. Use **blankAsZero** to change this setting.

-

isBlankZero example

See the example for **blankAsZero**.

▪

isIgnoreCaseInLocate procedure

See also Example Session Type

Reports whether the current session is ignoring case sensitivity in locate operations.

Syntax

`isIgnoreCaseInLocate ( ) Logical`

Description

isIgnoreCaseInLocate reports whether the current session is ignoring case-sensitivity during locate operations.

isIgnoreCaseInLocate example

See the example for **ignoreCaseInLocate**.

▪

loadProjectAliases procedure

See also Example Session Type

Loads project alias specifications from a file.

Syntax

```
loadProjectAliases ( const cfgFileName String ) Logical
```

Description

loadProjectAliases loads project alias specifications from the file specified in *cfgFileName*. If *cfgFileName* does not specify a path, Paradox searches for it in the working directory. By default, Paradox automatically reads project aliases from :WORK:PDOXWORK.CFG when the working directory is set or changed. This method lets you specify a different file.

When :WORK: is set (for example, at startup) or changed (either interactively or through ObjectPAL), Paradox discards all current project aliases and loads those project aliases that are specific to the new working directory. Public aliases remain active and available; if a project alias has the same name as a public alias, Paradox does not load the project alias. This method returns True if it succeeds; otherwise, it returns False.

loadProjectAliases example

In this example, **loadProjectAliases** loads the project aliases in the **open** method of the first page in the form. It reads a list of custom aliases from C:\OFFICE\PDOXWIN\CUSTOM.CFG instead of from :WORK:PDOXWORK.CFG (the Paradox default configuration file).

```
;pge1 :: open
method open(var eventInfo Event)
   loadProjectAliases("C:\\OFFICE\\PDOXWIN\\CUSTOM.CFG")
endMethod
```

▪

lock procedure

[See also] [Example] [Session Type]

Locks one or more tables.

Syntax

`lock ( const` ***`table`*** `{ Table|TCursor|String },  const` ***`lockType`*** `String [ , const` ***`table`*** `{ Table|TCursor| String },  const` ***`lockType`*** `String ] * ) Logical`

Description

lock locks one or more tables specified in comma-separated pairs of tables and lock types. You can use a TCursor or a Table to specify a table, and you can mix TCursor and Table variables in the list.

lockType must be a string expression that evaluates to one of the following values, listed in order of decreasing strength and increasing concurrency.

String value	Description
Full	The current session has exclusive access to the table. No other session can open the table. Cannot be used with dBASE tables.
Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

If this method locks all the tables in the list, it returns True; otherwise, it returns False. If it can't lock all the tables, it doesn't lock any.

lock example

The following example attempts to place a write lock on the *Orders* table and a read lock on the *Customer* table. If **lock** is able to lock both tables, the code displays data from both of the tables in a dialog box. Then, the code calls **unlock** to remove the explicit locks placed on *Customer* and *Orders*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ordTB    Table
  custTC   TCursor
  sampDB   Database
  otherSes Session
endVar

otherSes.open("other") ; Open another session
otherSes.addAlias("samples", "Standard", "c:\\pdoxwin\\sample")
sampDB.open("samples", otherSes)

custTC.open("Customer.db", sampDB)
ordTB.attach("Orders.db", sampDB)

if lock(custTC, "Read", ordTB, "Write") then
  if custTC.locate("Name", "Unisco") then
    custNo = custTC."Customer No"
    ordTB.setIndex("Customer No")
    ordTB.setFilter(custNo, custNo)
    msgInfo(String("Total for order ", custNo),
            ordTB.cSum("Total Invoice"))
    unlock(custTC, "Read", ordTB, "Write")
  else
    msgStop("Error", "Can't find Unisco.")
  endIf
else
  errorShow()
endIf

endMethod
```

open method

See also Example Session Type

Opens a session (a channel to the database engine).

Syntax

```
1. open ( ) Logical
2. open ( const sessionName String ) Logical
```

Description

Calling **open** with no arguments (syntax 1) gives you a handle to the current session; it does not exhaust a user count. When you use *sessionName* to specify a session name (syntax 2), you open another channel to the database engine and exhaust one user count. The actual value of *sessionName* doesn't matter, as long as it is a valid string.

You can open more than one session from the same workstation, and Paradox will view each session as a separate user; for example, locks set in one session block access from the other.

open example

The following code calls **open** twice: once to get a handle to the current session, and once to open a new session. Next it calls **blankAsZero** to specify how each session handles blank values in calculations. Then it passes the Session variables to a custom procedure named doSomething. The results of doSomething will be different for each session because of the different **blankAsZero** settings.

```
; openSession::pushButton
method pushButton(var eventInfo Event)
var
  currentSes,
  otherSes    Session
endVar

; Open sessions.
currentSes.open()
otherSes.open("other")

; Set session properties.
currentSes.blankAsZero(Yes)
otherSes.blankAsZero(No)

; Pass session handles to a custom procedure.
; Results will differ depending on settings for each session.
doSomething(currentSes)
doSomething(otherSes)

endMethod
```

▪

removeAlias method/procedure

See also Example Session Type

Removes an alias from a session.

Syntax

```
removeAlias ( const aliasName String ) Logical
```

Description

removeAlias removes the alias *alias* from a session. You cannot remove :WORK: or :PRIV: or an alias that is open.

removeAlias example

The following example adds an alias to the current session, then supplies the new alias to the **open** method defined for the Database type. When the alias is no longer needed, this code calls **removeAlias** to remove the alias name from the current session.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  custInfo Database
endVar

; Add the CustomerInfo alias to the current session.
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")

; Now use the alias specify the database to open.
custInfo.open("CustomerInfo") ; Opens the CustomerInfo database.

; Do something with the opened database,
; then when the alias is no longer needed, close the
; database and remove the alias from the current session.

custInfo.close()
removeAlias("CustomerInfo")

endMethod
```

▪

removeAllPasswords method/procedure

See also Example Session Type

Removes all passwords presented to a session.

Syntax

```
removeAllPasswords ( )
```

Description

removeAllPasswords reverses the effects of all **addPassword** statements issued for a session. It does not remove security from tables; it withdraws the passwords required to access protected tables.

removeAllPasswords example

The following example removes all the passwords from the session *ses*.

```
; removePasses::pushButton
method pushButton(var eventInfo Event)
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  ses.removeAllPasswords()
else
  msgStop("Help!","Session variable is not Assigned!")
endIf
endMethod
```

▪

removePassword method/procedure

See also Example Session Type

Removes a password presented to a session.

Syntax

removePassword (const ***password*** String **)**

Description

removePassword reverses the effect of an **addPassword** statement issued for a session. It does not unprotect the table; it merely withdraws the password specified in the argument *password* that was presented to access the table. Note that *password* is case-sensitive.

removePassword example

In the following example, the *getRemovePass* button acquires a password to remove from the user, then removes the password from the current session. Subsequent attempts to open tables protected by that password will fail.

```
; getRemovePass::pushButton
method pushButton(var eventInfo Event)
var
  newPass string
endVar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  newPass.view("Enter Password to Remove")
  ses.removePassword(newPass)
else
  msgStop("Help!", "Session variable is not Assigned!")
endIf
endMethod
```

▪

removeProjectAlias procedure

See also Example Session Type

Removes a project alias. For information on aliases, see Public and Project Aliases in the User's Guide help.

Syntax

```
removeProjectAlias ( const alias String ) Logical
```

Description

removeProjectAlias removes the project alias specified in *alias*.

When :WORK: is set (for example, at startup) or changed (either interactively or through ObjectPAL), Paradox discards all current project aliases and loads those project aliases that are specific to the new working directory. Public aliases remain active and available; if a project alias has the same name as a public alias, Paradox does not load the project alias. By default, Paradox reads project aliases from :WORK:PDOXWORK.CFG, but you can use **loadProjectAliases** to specify a different file.

removeProjectAlias example

This example uses **addProjectAlias** in the page's built-in **arrive** method to add an alias to the current project, then uses **removeProjectAlias** in the page's built-in **depart** method to remove it.

The following code is attached to the page's built-in **arrive** method.

```
;pge1 :: arrive
method arrive(var eventInfo MoveEvent)
   ;Add the CustomerInfo alias to the project.
   addProjectAlias("CustomerInfo", "Standard", "D:\\OFFICE\\PDOXWIN\\SAMPLE")
endMethod
```

The following code is attached to the page's built-in **depart** method.

```
;pge1 :: depart
method depart(var eventInfo MoveEvent)
   ;Remove the CustomerInfo alias from the project.
   if not removeProjectAlias("CustomerInfo") then
      errorShow("Could not remove project alias CustomerInfo.")
   endIf
endMethod
```

retryPeriod method/procedure

See also Example Session Type

Returns the number of seconds to retry an operation on a locked record or table.

Syntax

```
retryPeriod ( ) SmallInt
```

Description

retryPeriod returns the number of seconds to retry an operation on a locked record or table. The default value is 0, which means that operations are not retried.

retryPeriod example

The following example displays the current retry period to the user.

```
; getShowRetry::pushButton
method pushButton(var eventInfo Event)
var
  rp smallint
endVar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  rp = ses.RetryPeriod()            ; get the current retry period
  rp.view("The Retry Period is...") ; display the value
else
  msgStop("Help!","Session variable is not assigned!")
endIf
endMethod
```

saveCFG method/procedure

See also Example Session Type

Saves the current session's alias information to a file.

Syntax

```
saveCFG ( const fileName String ) Logical
```

Description

saveCFG saves the BDE configuration for the current session to *fileName*. The configuration file specified by *fileName* can be loaded (with the **-o** command-line option) in place of IDAPI32.CFG to set session information when you start Paradox.

saveCFG example

This example saves the current BDE settings to MyConfig.cfg.

```
;// saveconfiguration::pushButton
method pushButton(var eventInfo Event)
;// saves the BDE setting to file MyConfig.cfg
saveCfg("MyConfig.cfg")
endMethod
```

saveProjectAliases procedure

See also Example Session Type

Saves project alias specifications to a file. For information on aliases, see Public and project aliases in the User's Guide help.

Syntax

```
saveProjectAliases ( [ const fileName String ] ) Logical
```

Description

saveProjectAliases saves project alias specifications to a file. You can use the optional argument *fileName* to specify a file name. If you omit *fileName*, Paradox saves the alias to :WORK:PDOXWORK.CFG.

When :WORK: is set (for example, at startup) or changed (either interactively or through ObjectPAL), Paradox discards all current project aliases and loads those project aliases that are specific to the new working directory. Public aliases remain active and available; if a project alias has the same name as a public alias, Paradox does not load the project alias. By default, Paradox reads project aliases from :WORK:PDOXWORK.CFG, but you can use **loadProjectAliases** to specify a different file.

saveProjectAliases example

In the following example, **saveProjectAliases** is used to save the project aliases to MYPROJ.CFG after a project alias is added using **addProjectAlias**.

```
;pge1 :: open
method open(var eventInfo Event)
   ; Add project alias.
   addProjectAlias("MYPROJ", "Standard", "D:\\OFFICE\\PDOXWIN\\MYPROJ")

   ; Save project aliases.
   saveProjectAliases("MYPROJ.CFG")
endMethod
```

■

setAliasPassword method

See also Example Session Type

Sets the in-memory password for a specified alias.

Syntax

`setAliasPassword ( const aliasName, const password String ) Logical`

Description

setAliasPassword sets the in-memory password for the alias specified in *aliasName* to the value specified in *password*. The maximum length of a password is 31 characters. Then, the next time you open that alias, you can do so without supplying the password. In other words, calling **setAliasPassword** has the same effect as presenting a password interactively using the Alias Manager dialog box. It has no effect on the password stored and maintained on the server. This method returns True if successful; otherwise, it returns False.

setAliasPassword example

The following example calls **setAliasPassword** to present the password for a specified alias. Then, when the call to **open** executes, it opens the database without prompting the user for a password.

```
method pushButton(var eventInfo Event)
   var
      aliasName,
      aliasPassword   String
   endVar

   ; initialize variables
   aliasName = "bedrock"
   aliasPassword = "fred" ; Max length: 31 characters

   ; set alias password and open database
   if setAliasPassword(aliasName, aliasPassword) then
   db.open(aliasName) ; opens without prompting for password
   else
      errorShow("Couldn't set alias password.")
      return
   endIf

endMethod
```

▪

setAliasPath method/procedure

See also Example Session Type

Sets the path for an alias.

Syntax

```
setAliasPath ( const aliasName String, const aliasPath String ) Logical
```

Description

setAliasPath sets the path *aliasPath* for the alias *aliasName*.

▪

setAliasPath example

See the example for **getAliasPath**.

▪

setAliasProperty method

See also Example Session Type

Sets the value of a specified property for a specified alias.

Syntax

`setAliasProperty ( const aliasName String, const property String, const propertyValue String ) Logical`

Description

setAliasProperty sets the value of the property specified in *property* to the value specified in *propertyValue* for the alias specified in *aliasName*. Returns True if successful; otherwise, returns False.

Properties you set using this method show up in the Alias Manager dialog box just as if you had set them interactively. However, property settings are *not* automatically saved to IDAPI32.CFG. Use the Session procedure **saveCFG** to save alias properties to a file.

This method is only applicable to remote databases, and not to "standard" (Paradox or dBASE) databases.

▪

setAliasProperty example

See the example for **getAliasProperty.**

setRetryPeriod method/procedure

See also	Example	Session Type

Sets the number of seconds to retry an action on a locked table or record.

Syntax

`setRetryPeriod ( ``const `***`period`***` SmallInt `**`)`**` Logical`

Description

setRetryPeriod specifies in *period* the number of seconds to retry an action on a locked table or record. A value of 0 means that actions are not retried.

setRetryPeriod example

The following example prompts the user for a retry period, then sets the retry for the session to that value.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  rp Smallint
endVar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  rp = ses.retryPeriod()
  rp.view("Enter retry period")  ; get a retry period from user
  ses.setRetryPeriod(rp)         ; set the session's retry period
else
  msgStop("Help!","Session variable is not assigned!")
endIf
endMethod
```

unlock procedure

See also Example Session Type

Unlocks one or more tables.

Syntax

```
unlock ( const table { Table|TCursor|String } ,
         const lockType String [ , const table { Table|TCursor|String } ,
         const lockType String ] * ) Logical
```

Description

unlock unlocks one or more tables specified in a comma-separated list of tables and lock types.

unlock removes locks explicitly placed by a particular user or application using **lock**; it has no effect on locks placed automatically by Paradox. *lockType* must be a string expression that evaluates to one of the following values: Exclusive, Write, Read, or Full. "Read" and "Full" apply only to Paradox tables.)

If one **unlock** in the list fails, previous locks are not restored; the tables remain unlocked. You don't have to specify a session to use this method, because session data is set when you **open** a TCursor or **attach** to a Table.

Each time you lock a table explicitly, be sure to unlock it as soon as you no longer need the explicit lock. This ensures maximum concurrent availability of tables. Also, when you lock a table twice, you must unlock it twice. You can use the **lockStatus** method (defined for the TCursor and UIObject types) to determine how many explicit locks you have placed on a table. **unlock** returns False if you try to unlock a table that isn't locked or cannot be unlocked.

▪

unlock example

See the example for **lock.**

SmallInt type

SmallInt values are small integers; that is, they can be represented by a small (short) series of digits. A SmallInt variable occupies 2 bytes of storage.

ObjectPAL converts SmallInt values to range from -32,768 to 32,767. An attempt to assign a value outside of this range to a SmallInt variable causes an error. For example,

```
var
   x, y, z SmallInt
endVar

x = 32767 ; The upper limit value for a SmallInt variable.
y = 1
z = x + y ; This statement causes an error.
```

When ObjectPAL performs an operation on SmallInt values, it expects the result to be a SmallInt, too. That's why the addition operation in the previous example causes an error: the result is too large to be a SmallInt. To work with a boundary value (in either the positive or negative direction), convert it to a type that can accommodate it. In the following example, ObjectPAL converts one SmallInt to a LongInt before doing the addition, and the statement succeeds. This example also assigns the result to a LongInt variable (which can handle the large value), instead of assigning it to a SmallInt variable (which could not).

```
var
   x, y SmallInt
   z    LongInt   ; Declare z as a LongInt so it can hold the result.
endVar

x = 32767 ; the upper limit value for a SmallInt variable
y = 1
z = LongInt(x) + y
```

Note: The SmallInt value -32,768 cannot be stored in a Paradox table because, to Paradox, -32,768 = Blank. However, you can use this value in calculations, and you can store it in a dBASE table. Store such large numbers as LongInt or Number data types.

Note: Run-time library methods and procedures defined for the Number type also work with LongInt and SmallInt variables. The syntax is the same, and the returned value is a Number. For example, the following code will work, even though **sin** does not appear in the list of methods for the SmallInt type:

```
var
   abc LongInt
   xyz Number
endVar
abc = 43
xyz = abc.sin()
```

Note: ObjectPAL supports an alternate syntax:

methodName **(** ***objVar*** , *argument* [, *argument* **])**

methodName represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return the sine of a number:

```
theNum.sin()
```

The following statement uses the alternate syntax:

```
sin(theNum)
```

We recommend using standard syntax for clarity and consistency, but you can use the alternate syntax wherever it's convenient.

The SmallInt type includes several derived methods from the Number and AnyType types.

Methods for the SmallInt type

AnyType	Number	**LongInt**
blank	abs	**bitAND**
dataType	acos	**bitIsSet**
isAssigned	asin	**bitOR**
isBlank	atan	**bitXOR**
isFixedType	atan2	**int**
view	ceil	**smallInt**
	cos	
	cosh	
	exp	
	floor	
	fraction	
	fv	
	ln	
	log	
	max	
	min	
	mod	
	number	
	numVal	
	pmt	
	pow	
	pow10	
	pv	
	rand	
	round	
	sin	
	sinh	
	sqrt	
	tan	
	tanh	
	truncate	

bitAND method

See also Example SmallInt Type

Performs a bitwise AND operation on two values.

Syntax

`bitAND` `(` `const` ***`value`*** `SmallInt` `)` `SmallInt`

Description

bitAND returns the result of a bitwise AND operation on *value*. **bitAND** operates on the binary representations of two integers, comparing them one bit at a time. The truth table for **bitAND** is:

a	b	a bitAND b
0	0	0
1	0	0
0	1	0
1	1	1

▪

bitAND example

In the following example, the **pushButton** method for a button named *andTwoNums* takes two integers and performs a bitwise AND calculation on them. The result of the calculation is displayed in a dialog box.

```
; andTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b SmallInt
endVar
a = 30233   ; binary 01110110 00011001
b = 1233    ; binary 00000100 11010001
a.bitAND(b) ; binary 00000100 00010001
msgInfo("The result of 30233 bitAND 1233 is:", a.bitAND(b))
; displays 1041
endMethod
```

bitIsSet method

See also Example1 Example2 SmallInt Type

Reports whether a bit is 1 or 0.

Syntax

```
bitIsSet ( const value SmallInt ) Logical
```

Description

bitIsSet examines the binary representation of an integer, reporting whether the **value** bit is 0 or 1. It returns True if the bit specified is 1, and False if the bit is 0.

value is a number specified by 2*n*, where *n* is an integer between 0 and 14. The exponent *n* corresponds to one less than the position of the bit to test, counting from the right. For example, to specify the third bit from the right, use 4 (2(3-1), which is 22).

▪

bitIsSet example 1

In the following example, the **pushButton** method for a button named *isABitSet*, examines the values in two unbound field objects: *whichBit*, and *whatNum*. *whichBit* contains the bit position (counting from the right) of the bit test. *whatNum* contains the integer to test. The **pushButton** method uses *whichBit* to calculate the value of the position, then assigns the result to *bitNum*. The method then checks *Num* to see if the *bitNum* bit is set, and displays the Logical result with a **msgInfo** dialog box.

```
; isABitSet::pushButton
method pushButton(var eventInfo Event)
var
  bitNum,
  Num      SmallInt
endVar
; get the bit position number from the whichBit
; field and convert to multiple of 2
bitNum = SmallInt(pow(2, whichBit - 1))
; get the number to test from the whatNum field
Num = whatNum
; is the bit for value bitNum 1 in Num?
msgInfo("Is Bit Set?", Num.bitIsSet(bitNum))
endMethod
```

▪

bitIsSet example 2

The following example illustrates how you can use **bitIsSet** to display an integer as a binary number. The **pushButton** method for *showBinary* constructs a string of zeros and ones by testing each bit of a four-byte long integer. For readability, a blank is added to the string after 8 digits.

```
; showBinary::pushButton
method pushButton(var eventInfo Event)
var
  binString  String    ; to construct the binary string
  Num        SmallInt  ; number to test
  i          SmallInt  ; for loop index
endVar
if NOT whatNum.isBlank() then
  Num = whatNum                ; get the number test from whatNum
  binString = ""               ; initialize the string
  for i from 0 to 14
    if Num.bitIsSet(SmallInt(pow(2, i))) then
      binString = "1" + binString    ; add a 1 to the front of the string
    else
      binString = "0" + binString    ; add a 0 to the front of the string
    endIf
    if i = 7 then
      binSTring = " " + binString    ; add a space every 8 digits
    endIf
  endfor
  if Num < 0 then
    binString = "1" + binString      ; set the sign bit
  else
    binString = "0" + binString
  endIf
  ; show the number
  message("The binary equivalent is ", binString)
endIf
endMethod
```

■

bitOR method

See also Example SmallInt Type

Performs a bitwise OR operation on two values.

Syntax

```
bitOR ( const value SmallInt ) SmallInt
```

Description

bitOR returns the result of a bitwise OR operation on *value*. **bitOR** operates on the binary representations of two integers, comparing them one bit at a time. The truth table for **bitOR** is:

a	b	a bitOR b
0	0	0
1	0	1
0	1	1
1	1	1

bitOR example

In the following example, the **pushButton** method for a button named *orTwoNums* takes two integers and performs a bitwise OR calculation on them. The result of the calculation is displayed in a dialog box.

```
; orTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b SmallInt
endVar
a = 30233  ; binary 01110110 00011001
b = 1233   ; binary 00000100 11010001
a.bitOR(b) ; binary 01110110 11011001
msgInfo("30233 OR 1233", a.bitOR(b)) ; displays 30425
endMethod
```

▪

bitXOR method

See also Example SmallInt Type

Performs a bitwise XOR operation on two values.

Syntax

bitXOR (const ***value*** SmallInt **)** SmallInt

Description

bitXOR performs a bitwise XOR (exclusive OR) operation on *value*. **bitXOR** operates on the binary representations of two integers, comparing them one bit at a time. The truth table for **bitXOR** is:

a	b	a bitXOR(b)
0	0	0
1	0	1
0	1	1
1	1	0

■

bitXOR example

In the following example, the **pushButton** method for a button named *xorTwoNums* takes two integers and performs a bitwise XOR calculation on them. The result of the calculation is displayed in a dialog box.

```
; xorTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b SmallInt
endVar
a = 30233   ; binary 01110110 00011001
b = 1233    ; binary 00000100 11010001
a.bitXOR(b) ; binary 01110010 11001000
msgInfo("30233 XOR 1233", a.bitXOR(b)) ; displays 29384
endMethod
```

▪

int procedure

See also Example SmallInt Type

Casts a value as an integer.

Syntax

```
int ( const value AnyType ) SmallInt
```

Description

int casts (converts) the numeric expression *value* to an integer. If *value* is of a more precise type (for example, Number), precision is lost.

■

int example

The following example assigns a number to *nn*, views the value of *nn* in a dialog box, then displays *nn* as an integer. This code is attached to the **pushButton** method for the *showInt* button.

```
; showInt::pushButton
method pushButton(var eventInfo Event)
var
   nn Number
endVar
nn = 123.12
view(nn)                              ; displays 123.12
msgInfo("nn as Integer", int(nn))    ; displays 123
endMethod
```

smallInt procedure

See also   Example   SmallInt Type
Beginner

Casts a value as a small integer.

Syntax

```
smallInt ( const value AnyType ) SmallInt
```

Description

smallInt casts (converts) the numeric expression *value* to a SmallInt. If *value* is of a more precise type (for example, Number), precision is lost.

▪

smallInt example

The following example assigns a number to *x*, then casts *x* to SmallInt and assigns the result to *s*. The decimal precision of *x* is lost when it is cast to a SmallInt.

```
; convertToInt::pushButton
method pushButton(var eventInfo Event)
var
  x Number
  s SmallInt
endVar
x = 12.34              ; give x a value
x.view()               ; view x, title of dialog will be "Number"
s = SmallInt(x)        ; cast x as a LongInt and assign to s
s.view()               ; show s, note that decimal places are lost
                       ; displays 12
endMethod
```

■

SQL type

Changes ■

An ObjectPAL SQL variable represents an SQL statement. You can use ObjectPAL to create and execute SQL commands from methods just as if you were using Paradox interactively. You can execute SQL commands from an SQL file, an SQL statement, or a string. Some queries require Paradox to create temporary tables. Paradox creates these tables in the private directory.

Methods for the SQL type

SQL

executeSQL

getQueryRestartOptions

isAssigned

readFromFile

readFromString

setQueryRestartOptions

wantInMemoryTCursor

writeSQL

Changes to SQL type methods

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
getQueryRestartOptions	readFromFile (replaces **executeSQLFile**)
isAssigned	readFromString (replaces **executeSQLString**)
setQueryRestartOptions	
wantInMemoryTCursor	

■

executeSQL method/procedure

See also Example SQL Type

Executes an SQL statement.

Syntax

```
Method:
1. executeSQL ( const db Database) Logical
2. executeSQL ( const db Database, ansTbl String ) Logical
3. executeSQL ( const db Database, ansTbl Table ) Logical
4. executeSQL ( const db Database, ansTbl TCursor ) Logical
Procedure:
1. executeSQL ( const db Database, const qbeVar SQL ) Logical
2. executeSQL ( const db Database, const qbeVar SQL, ansTbl String ) Logical
3. executeSQL ( const db Database, const qbeVar SQL, ansTbl Table ) Logical
4. executeSQL ( const db Database, const qbeVar SQL, ansTbl TCursor ) Logical
```

Description

executeSQL executes a pass through SQL query created in an ObjectPAL method or procedure.

In syntax 1, where the answer table is not specified, **executeSQL** writes to ANSWER.DB in the user's private directory.

In syntax 2, specify the answer table as a string; if you do not include a file extension, the answer table is a Paradox table by default.

In syntax 3, where *ansTbl* is a Table variable, *ansTbl* must be assigned and valid.

In syntax 4, a TCursor is opened onto the answer set, which may be an in-memory table or a cursor onto the answer set.

executeSQL returns True if the query is executed on the server (even if the resulting table is empty); otherwise, it returns False.

An SQL query in ObjectPAL code begins with an SQL variable, the = sign, and the keyword SQL followed by a blank line. Next come the SQL statements that make up the body of the query, and another blank line. The query ends with the keyword **endSQL**. Because this kind of query is not a quoted string, it can contain tilde variables. Compare this method with **readFromString.**

Note: **executeSQL** is a pass through function: the SQL statements are sent directly to the server as if by another user. They do not execute within the context of a database handle or active transaction.

▪

executeSQL example

The following example prompts the user to enter an item name and stores the user's input in a variable. Then it uses the variable as a tilde variable in an SQL query and calls **executeSQL** to execute the query and store the results in a TCursor in system memory. If the query executes successfully, the results are passed to a custom procedure for more processing.

```
method pushButton(var eventInfo Event)
   var
      itemNameSQL      SQL
      aliasNamTC,
      itemNameTC       TCursor
      db               Database
      myAlias,
      promptString,
      aliasTableName,
      userItemName     String
   endVar

   ; initialize variables
   myAlias = "itchy"
   aliasTableName = ":PRIV:aliasNam.db"
   promptString = "Enter an item name here."
   userItemName = promptString

   enumAliasNames(aliasTableName) ; create a table of alias names

   ; use alias to open database
   aliasNamTC.open(aliasTableName)
   if aliasNamTC.locate("DBName", myAlias) then
      db.open(myAlias)   ; use alias to get database handle to server
   else
      msgStop("Stop", "The alias " + myAlias + " has not been defined.")
      return
   endIf

   userItemName.view("Item name:")
   if userItemName = promptString then
      return ; exit the method

   else
      ; set query (use a tilde variable to store user input)
      itemNameSQL = SQL

                   SELECT CustomerName, Order_no, ItemName
                   FROM   Customer, Sales
                   WHERE  Sales.ItemName = ~userItemName AND
                          Customer.CustNo = Sales.CustNo

                   endSQL

   endIf

   ; execute the query and process the results
   if itemNameSQL.executeSQL(db, itemNameTC) then
      doSomething(itemNameTC) ; call custom proc to process data
   else
      errorShow("executeSQL failed")
   endIf

endMethod
```

getQueryRestartOptions method

See also Example SQL Type

Returns a value representing the user's query restart option setting.

Syntax

```
getQueryRestartOptions ( ) SmallInt
```

Description

getQueryRestartOptions returns an integer value representing the user's query restart option setting. Use one of the following ObjectPAL QueryRestartOptions constants to test the value:

QueryDefault	Use the options specified interactively using the Query Restart Options dialog box.
QueryLock	Lock all other users out of the tables needed while the query is running. If Paradox cannot lock a table, it does not run the query. This is the least polite to other users. And you must wait until all the locks can be secured before the query will run.
QueryNoLock	Run the query even if someone changes the data while it's running.
QueryRestart	Start the query over. Specify QueryRestart when you want to make sure you get a snapshot of the data as it existed at some instant. Another user might change the data after the query is completed but before the Answer table is displayed, but at least you got a snapshot. This is just the nature of multi-user work.

▪

getQueryRestartOptions example

See the example for **setQueryRestartOptions**.

▪

isAssigned method

See also Example SQL Type

Reports whether an SQL variable has an assigned value.

Syntax

`isAssigned ( ) Logical`

Description

isAssigned returns True if a SQL variable has been assigned a value; otherwise, it returns False. This method does not check the validity of the assigned SQL statement.

isAssigned example

In the following example, the call to **isAssigned** returns True, because the SQL variable *sqlVar* has been assigned a value, even though the value is not a valid SQL variable.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  sqlVar SQL
endVar
sqlVar = SQL
     This is not a valid SQL statement
     endSQL
msgInfo("Assigned?", sqlVar.isAssigned())    ; displays True
endMethod
```

▪

readFromFile method

See also Example SQL Type

Assigns the contents of an SQL file to an SQL variable.

Syntax

```
readFromFile ( const sqlFileName SQL ) Logical
```

Description

readFromFile opens *sqlFileName* (created with the **writeSQL** method or interactively using the SQL Editor) and assigns the contents to an SQL variable. Do not use the **SQL** and **endSQL** keywords. Use **executeSQL** to execute the query.

If the value of *fileName* does not include a path or alias, this method looks for the file in the directory associated with the specified database (or the default database, if a database is not specified). If the value of *fileName* does not include an extension, this method assumes an extension of .SQL. To specify a file name that does not have an extension, put a period at the end of the name. For example, the following table lists the resulting file names for various values of *fileName*.

Value of *fileName*	SQL file name
newcust	newcust.sql
newcust.	newcust
newcust.s	newcust.s

readFromFile returns True if it succeeds; otherwise, it returns False.

Note: **readFromFile** is a pass through function: the SQL statements are sent directly to the server as if by another user. They do not execute within the context of a database handle or active transaction.

readFromFile replaces executeSQLFile, which existed in versions of ObjectPAL prior to 5.0.

▪

readFromFile example

The following example creates a pop-up menu listing the SQL files in the user's private directory. When the user chooses a file from the menu, this code calls **readFromFile** to read the query, assign it to an SQL variable, execute the query, and store the results in a TCursor. Then it passes the TCursor to a custom procedure (assumed to be defined elsewhere) for more processing.

```
method pushButton(var eventInfo Event)
   var
      myAlias,
      aliasTableName,
      sqlFileName,
      sqlFileSpec      String
      aliasNamTC,
      answerTC         TCursor
      sqlPop           PopUpMenu
      db               Database
      sqlFS            FileSystem
      sqlFileAr        Array[] String
            sqlVar      SQL
   endVar

   ; initialize variables
   myAlias = "itchy"
   aliasTableName = ":PRIV:aliasNam.db"
   sqlFileSpec = ":PRIV:*.SQL"

   enumAliasNames(aliasTableName) ; create a table of aliases

   aliasNamTC.open(aliasTableName)
   if aliasNamTC.locate("DBName", myAlias) then
      db.open(myAlias)  ; use alias to get database handle to server
   else
      msgStop("Stop",
              "The alias " + myAlias +
              " has not been defined.")
      return ; exit the method
   endIf

   ; build a pop-up menu listing SQL files in the target directory
   if sqlFS.findFirst(sqlFileSpec) then
      sqlFS.enumFileList(sqlFileSpec, sqlFileAr)
      sqlPop.addArray(sqlFileAr)
      sqlFileName = sqlPop.show() ; variable stores user's menu choice

      ; read and execute the SQL file chosen by the user
      sqlVar.readFromFile(sqlFileName)
      if sqlVar.executeSQL(answerTC) then
         doSomething(answerTC) ; call custom proc to process data
      else
         errorShow("readFromFile failed")
      endIf

   else
      msgStop("File not found:", sqlFileSpec)
   endIf

endMethod
```

▪

readFromString method

See also Example SQL Type

Assigns a query string to an SQL variable.

Syntax

```
readFromString ( const sqlString SQL ) Logical
```

Description

readFromString assigns the SQL query string specified in *sqlString* to an SQL variable. Do not enclose the string between the **SQL** and **endSQL** keywords. Use **executeSQL** to execute the query.

Note: **readFromString** is a pass through function: the SQL statements are sent directly to the server as if by another user. They do not execute within the context of a database handle or active transaction.

readFromString replaces **executeSQLString**, which existed in versions of ObjectPAL prior to 5.0.

▪

readFromString example

The following example prompts the user to enter an SQL keyword, then uses that keyword as part of an SQL string. If the user enters a valid SQL keyword and the query executes successfully, the results are stored in a TCursor and passed to a custom procedure (assumed to be defined elsewhere) for more processing.

```
method pushButton(var eventInfo Event)
   var
      sqlKeyword,
      promptString,
      bigOrderString   String
      aliasNamTC,
      bigOrderTC       TCursor
      db               Database
      myAlias,
      aliasTableName   String
      sqlVar           SQL
   endVar

   ; Initialize variables.
   myAlias = "itchy"
   aliasTableName = ":PRIV:aliasNam.db"
   promptString = "Enter an SQL keyword (e.g. SELECT):"

   enumAliasNames(aliasTableName)

   ; Prompt user to enter an SQL keyword.
   sqlKeyword.view("SQL Keyword")
   if sqlKeyword = promptString then
      return ; Exit method if user doesn't enter a keyword.
   endIf

   ; Use alias to open database.
   aliasNamTC.open(aliasTableName)
   if aliasNamTC.locate("DBName", myAlias) then
      db.open(myAlias)  ; Use alias to get database handle to server
   else
      msgStop("Stop", "The alias " + myAlias +
                      " has not been defined.")
      return
   endIf


   ; Combine SQL statements and String variable sqlKeyword
   ; to create an SQL string.
   bigOrderString = sqlKeyword +
                    "CustName, Order_no, Sale_date, Qty
                    FROM     Customer
                    WHERE    Qty > 1000

   ; Read and execute the query and process the results.
   sqlVar.readFromString(bigOrderString)
   if sqlVar.executeSQL(bigOrderTC) then
      doSomething(bigOrderTC) ; call custom proc to process data
   else
      errorShow()
   endIf

endMethod
```

▪

setQueryRestartOptions method

See also Example SQL Type

Specifies what to do with the underlying tables while running a query.

Syntax

`setQueryRestartOptions ( const qryRestartType SmallInt ) Logical`

Description

setQueryRestartOptions tells Paradox what to do if data changes while you're running a query in a multiuser environment. The argument *qryRestartType* represents one of the following ObjectPAL QueryRestartOptions constants:

QueryDefault	Use the options specified interactively using the Query Restart Options dialog box.
QueryLock	Lock all other users out of the tables needed while the query is running. If Paradox cannot lock a table, it does not run the query. This is the least polite to other users. And you must wait until all the locks can be secured before the query will run.
QueryNoLock	Run the query even if someone changes the data while it's running.
QueryRestart	Start the query over. Specify QueryRestart when you want to make sure you get a snapshot of the data as it existed at some instant. Another user might change the data after the query is completed but before the Answer table is displayed, but at least you got a snapshot. This is just the nature of multi-user work.

▪

setQueryRestartOptions example

The following example calls **getQueryRestartOptions** to get the user's current query restart options. If the setting is not QueryRestart, this code calls **setQueryRestartOptions** to set it. Then it executes a query.

```
method pushButton(var eventInfo Event)
   var
      qVar   SQL
   endVar

  if getQueryRestartOptions <> QueryRestart then
     setQueryRestartOptions(QueryRestart)
  endIf

   if qVar.readFromFile("newcust.sql") then
      qVar.executeSQL()
   else
      errorShow()
   endIf
endMethod
```

wantInMemoryTCursor method

See also Example SQL Type

Specifies how to create a TCursor resulting from a query.

Syntax

```
wantInMemoryTCursor ( [ const yesNo Logical ] )
```

Description

wantInMemoryTCursor specifies how to create a TCursor resulting from a query. By default, when you execute a query to a TCursor, that TCursor will point to a live query view changes made to the TCursor will affect the underlying tables. When you call **wantInMemoryTCursor** with *yesNo* as Yes (or omitted), Paradox creates the TCursor in system memory, with no connection to underlying tables.

An in-memory TCursor can be useful for performing quick "what-if" analyses. For example, suppose you want to study the effect of giving each employee a 15 percent raise. You could query the employee data to increase everyone's salary by 15 percent. Of course, you wouldn't want to do this to the actual employee data (at least, not yet), so you would execute the query to an in-memory TCursor and work with the data there, without affecting the underlying data.

wantInMemoryTCursor example

This example uses an in-memory TCursor to study the effects of giving every employee a 15 percent raise. It reads a pre-defined query from a file and executes it, then uses the results in a calculation.

```
method pushButton(var eventInfo Event)
   var
      qVar               SQL
      tcRaise15      TCursor
      nuTotalPayroll   Number
   endVar

   qVar.wantInMemoryTCursor(Yes)
   qVar.readFromFile("raise15.qbe")
   qVar.executeQBE(tcRaise15)

   nuTotalPayroll = tcRaise15.cSum("Salary")
   nuTotalPayroll.view("Payroll after 15%   raise:")
endMethod
```

▪

writeSQL method/procedure

See also Example SQL Type

Writes an SQL statement or an SQL string to a file.

Syntax

```
Method:
1. writeSQL ( const fileName String ) Logical
Procedure:
2. writeSQL ( const sqlString String, const fileName String ) Logical
```

Description

writeSQL writes a previously defined SQL statement or SQL string to the file specified in *fileName*. If *fileName* exists, Paradox overwrites it without asking for confirmation. **writeSQL** returns True if successful; otherwise, it returns False. This method does not evaluate the SQL commands.

Syntax 1 is a method; use dot notation to specify an SQL variable▪for example, sqlVar.writeSQL("bigOrder.sql").

Syntax 2 is a procedure; instead of using dot notation, use a String variable as the first argument▪for example, writeSQL(sqlString, "bigOrder.sql").

▪

writeSQL example

The following example prompts the user to enter a table name and stores the name in a String variable. Then it uses the String variable as a tilde variable in an SQL statement. The call to **writeSQL** writes the SQL statement (including the expanded tilde variable) to a file. In other words, if the user enters ORDERS as a table name, the resulting SQL file would contain the following statements:

```
SELECT *
FROM ORDERS
```

This method does not verify that the file contains valid SQL statements.

```
method pushButton(var eventInfo Event)
   var
      sqlString,
      userTableName,
      sqlFileName,
      promptString    String
   endVar

   ; Initialize variables.
   sqlFileName = "user001.sql"
   promptString = "Enter table name here."
   userTableName = promptString

   ; Display a view() dialog box and prompt user for input.
   userTableName.view("Select * from table:")

   ; If user enters a string, use it in a tilde variable
   ; in the following SQL query.
   if userTableName <> promptString then

      sqlString = SELECT *
                  FROM ~userTableName

      writeSQL(sqlString, sqlFileName) ; Write user's query to a file.

   endIf

endMethod
```

StatusEvent type

StatusEvent type methods control messages that appear in the Desktop status bar. Using StatusEvent type methods, you can attach code to the built-in event method to find out where and why messages will be displayed. You can block messages or display them somewhere else, in a different status area, or in another object (for example, a field object or a text file). You can also specify the text to be displayed in the message.

You can use the StatusReasons constants to refer to the areas of the status bar.

The StatusEvent type includes several derived methods from the Event type.

Methods for the StatusEvent type

Event	StatusEvent
errorCode	**reason**
getTarget	**setReason**
isFirstTime	**setStatusValue**
isPreFilter	**statusValue**
isTargetSelf	
reason	
setErrorCode	
setReason	

reason method

See also Example StatusEvent Type

Reports why a StatusEvent occurred.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report why a StatusEvent occurred. StatusEvent reasons occur when a built-in **status** method is called. ObjectPAL provides StatusReasons constants for testing the value returned by **reason**.

reason example

The following example copies all the messages sent to the status bar to a field. Assume a form contains a field called *fldStatus*. The form's built-in **status** examines the event packet for the reason. If the reason is StatusWindow, then it sends the status value to the field *fldStatus*.

```
;frm1 :: status
method status(var eventInfo StatusEvent)
if eventInfo.isPreFilter()
   then
      ; This code executes for each object on the form.
   else
      ; This code executes only for the form.
      if eventinfo.reason() = StatusWindow then
         fldStatus.Value = eventinfo.statusValue()
      endIf
endIf
endMethod
```

setReason method

See also Example StatusEvent Type

Specifies a reason for a StatusEvent.

Syntax

`setReason ( const reasonId SmallInt )`

Description

setReason specifies a reason for generating a StatusEvent. The StatusEvent reasons tell you which window on the status bar the message was sent to. ObjectPAL provides StatusReasons constants for setting the reason for a StatusEvent.

▪

setReason example

In the following example, for StatusEvent bubbled up to the form from a field, the form's **status** method changes the reason and the content of the message. The method changes the reason to ModeWindow1, and sets the value of the message to the name of the object that started the original event (the target).

```
; thisForm::status
method status(var eventInfo StatusEvent)
var
  targObj  UIObject
  nameStr  String
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; after regular message has displayed, also show
    ; field name in ModeWindow1
    eventInfo.getTarget(targObj)
    if targObj.Class = "Field" then      ; if this is a field
      nameStr = targObj.Name       ; get the field name
      eventInfo.setReason(ModeWindow1)    ; set the window
      eventInfo.setStatusValue(nameStr)   ; send the string
    endIf
endIf
endMethod
```

setStatusValue method

See also Example StatusEvent Type

Specifies the text of a status message.

Syntax

setStatusValue (const ***statusValue*** AnyType **)**

Description

setStatusValue specifies the text of a status message in *messageText*.

▪

setStatusValue example

See the example for **setReason**.

▪

statusValue method

See also Example StatusEvent Type

Returns the text of a status message.

Syntax

`statusValue ( )` AnyType

Description

statusValue returns the text of a status message.

▪

statusValue example

The following example makes the default status messages more prominent to the user by copying each message to a field on the form. This feature is controlled by the *magnifyMessage* button, also on the same form. The following code is attached to the **pushButton** method of the *magnifyMessage* button:

```
; magnifyMessage::pushButton
method pushButton(var eventInfo Event)
; toggle statusMessageField to visible or invisible and
; toggle label between "Magnified Messages" and "Normal Messages"
if self.LabelText = "Magnified Messages" then
  statusMessageField.Visible = True
  self.LabelText = "Normal Messages"
else
  statusMessageField.Visible = False
  self.LabelText = "Magnified Messages"
endIf
endMethod
```

This code is attached to the form's **status** method:

```
; thisForm::status
method status(var eventInfo StatusEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    ; write every status event to a field on the form
    if statusMessageField.Visible = True then
      if eventInfo.reason() = StatusWindow then
        statusMessageField = eventInfo.statusValue()
      endIf
    endIf
  else
    ; code here executes just for form itself

endIf
endMethod
```

String type

Changes

A String variable can contain up to 32,000 characters (use Memo objects for longer text). A quoted string can contain up to 255 characters. Use double quotes ("") to represent an empty string. Strings occupy 1 byte of storage per character.

Note: ObjectPAL supports an alternate syntax:

methodName **(** *objVar* **,** *argument* [**,** *argument*] **)**

methodName represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return a lowercase version of a string:

```
theString.lower()
```

The following statement uses the alternate syntax:

```
lower(theString)
```

It's best to use standard syntax for clarity and consistency, but you can use the alternate syntax wherever it's convenient.

The String type includes several derived methods from the AnyType type.

Methods for the String type

AnyType	**String**
blank	**advMatch**
dataType	**ansiCode**
isAssigned	**breakApart**
isBlank	**chr**
isFixedType	**chrOEM**
view	**chrToKeyName**
	fill
	format
	ignoreCaseInStringCompares
	isIgnoreCaseInStringCompares
	isSpace
	keyNameToChr
	keyNameToVKCode
	lower
	lTrim
	match
	oemCode
	readFromClipboard
	rTrim
	search
	size
	space
	string

[strVal]
[substr]
[toANSI]
[toOEM]
[upper]
[vkCodeToKeyName]
[writeToClipboard]

Changes to String type methods

The following table lists new methods and methods that were changed for version 7.

New	Changed
readFromClipboard	(None)
writeToClipboard	

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
(None)	format

▪

advMatch method

[See also](#) [Example](#) [String Type](#)
Beginner

Searches text for a specified string.

Syntax

advMatch (const *pattern* String [, var *matchVar* String] *) Logical

Description

advMatch returns True if *pattern* is found within the string; otherwise, it returns False. To specify *pattern*, use a string and the optional symbols listed in the table. This method is case sensitive by default, but you can use the String procedure **ignoreCaseInStringCompares** to change the case behavior.

matchVar is a variable to which the matching portion will be assigned. **advMatch** assigns matched patterns to one or more *matchVar* variables as the patterns are found. The portions of the string matching wildcard elements are assigned to the variables from left to right. Since there may be multiple matches, the first matching substring is assigned to the first variable, the second matching substring to the second variable, and so on. If no match is found, variables are not assigned values.

If you supply *pattern* from within a method, you need to use two backslashes when you want to tell **advMatch** to treat a special character as a literal; for example, \\(tells **advMatch** to treat the parenthesis as a literal character. Two backslashes are required in this situation because of the ambiguity between the compiler's interpretation of a backslash (used in escape sequences such as \t for a tab) and **advMatch's** understanding of a backslash. When the compiler sees a string with an embedded escape sequence, such as a "\tstart", it interprets the "\t" as a tab. The backslash character has a special meaning to the compiler, but it also has a special meaning to **advMatch**.

For example, if you're trying to search for a question mark embedded in a string, you might call **advMatch** like so:

```
s = "a string?"
advMatch(s, "\?")        ; this won't work!
```

You might think that you're telling **advMatch** to search for the literal question mark. However, the compiler sees the string first and returns a syntax error because "\?" is not a valid escape sequence. To prevent the compiler from interpreting the backslash as the beginning of an escape sequence, precede the backslash by another backslash. This will work:

```
s = "a string?"
advMatch(s, "\\?")        ; this does work!
```

If you supply *pattern* from a field in a table or a TextStream, special **advMatch** symbols are recognized without a preceding backslash, and one backslash and plus symbol (\+) yields a literal character.

Symbol	Matches
\	Use backslash to include special characters (for example, \t for Tab) as regular characters. (Remember to use two backslashes in quoted strings.)
[ ]	Match the enclosed set. For instance, [aeiou0-9] match a, e, i, o, u, and 0 through 9.
[^]	Do not match the enclosed set. For instance, [^aeiou0-9] match anything except a, e, i, o, u, and 0 through 9.
( )	Grouping.
^	Beginning of string (do not confuse this with [^]), where the ^ acts as a logical NOT).
$	End of string.
..	Match anything.
@	Match any single character.
*	Zero or more of the preceding character or expression.

+	One or more of the preceding character or expression.
?	None or one of the preceding character or expression.
|	OR operation.

For examples, see Sample search strings with wildcards in the User's Guide help.

▪

advMatch example

These statements demonstrate **advMatch** functionality:

```
method pushButton(var eventInfo Event)
var
  w, x, y, z      String
  l               Logical
endVar

l = advMatch("this is", "s")
l.view()
  ; returns True (different from match)

l = advMatch("this is", "^s")
l.view()
  ; returns False, because it requires s to be at the beginning of the line

l = advMatch("this is", "S")
l.view()
  ; returns False, it is case sensitive.

l = advMatch("this is", "[sS]")
l.view()
  ; returns True, because [sS] specifies any in this set

l = advMatch("this is", "[a-z]")
l.view()
  ; returns True, because [a-z] specifies any in this set of a through z

l = advMatch("this is", "[a-c]")
l.view()
  ; returns False, because [a-c] specifies any in this set of a through c
  ; and "this is" does not contain a, b, or c

l = advMatch("this is", "[a-cs]")
l.view()
  ; returns True, because [a-cc] specifies any in this set of a through c
  ; or s and "this is" does contain s
  ; note that [a-c, s] would specify any in the set of a through c,
  ; a comma, a space, or an s

l = advMatch("this is", "(@)s", x)
l.view()
x.view()
  ; returns True, x = "i" because the "()" operators specify a group,
  ; unlike match, advMatch places only those things that you group
  ; in the variables

l = advMatch("this is a test", "((t@@s)|(t@s))|(@s)", w, x, y, z)
l.view()   ; returns True, and
w.view()   ; "this", the result of the first set of parentheses,
           ; that is, for the entire expression ((t@@s)|(t@s))
           ; also, "this" was matched before "test"
x.view()   ; also "this", for the result of the second set of
           ; parentheses, (t@@s)
y.view()   ; the result of (t@s), blank, because the t@@s
           ; satisfied the expression ((t@@s)|(t@s))
z.view()   ; also blank, because the expression ((t@@s)|(t@s)) satisfied
           ; the entire pattern ((t@@s)|(t@s))|(@s)
; NOTE: Match variables are matched to groups in the order of occurrence,
```

```
 ;       not in the order of precedence: The first group▪starting from
 ;       the left
▪is assigned to the first variable.

l = advMatch("this is so", "(..)is(..)", x, y)
l.view()
x.view()
y.view()
  ; returns True, x = "this", y = " so"

l = advMatch("this is so", "[a-c]|[f-l]s" )
l.view()
  ; returns True, because an s is preceded by either a through
  ; c or f through l

l = advMatch("this as so", "[a-c]|[t-z]s" )
l.view()
  ; returns True, because an s is preceded by either a through
  ; c or t through z

endMethod
```

ansiCode procedure

See also Example String Type

Returns the ANSI code of a one-character string.

Syntax

ansiCode (const *char* String) SmallInt

Description

ansiCode returns the ANSI code of *char*, where *char* is a one-character string. The ANSI code returned is an integer between 1 and 255.

ansiCode example

In the following example, assume a form contains four field objects: *showAllChars*, *ANSIField*, *OEMField*, and *KeyNameField*. The **keyPhysical** method for *showAllChars* examines every character, then translates it to its ANSI code, OEM code, and key-name equivalent. The various character codes are written to *ANSIField*, *OEMField*, and *KeyNameField*.

```
; showAllChars::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  anyChar   String
  anyANSI   SmallInt
  anyKeyN   String
  anyOEM    SmallInt
endVar
anyChar = eventInfo.char()            ; get the character typed
anyANSI = ansiCode(anyChar)           ; convert to ANSI code
ANSIField = anyANSI                   ; write ANSI code to ANSIField

anyCode = eventInfo.vCharCode()       ; get the VK_Code of character

anyKeyN = VKCodeToKeyName(anyCode)    ; convert VK_Code to key name
KeyNameField = anyKeyN                ; write key name to KeyNameField

anyOEM = oemCode(anyChar)             ; convert char to OEM code
OEMField = anyOEM                     ; write OEM code to OEMField
beep()
endMethod
```

▪

breakAPart method

See also Example String Type
Beginner

Splits a string into substrings.

Syntax

`breakApart ( var tokenArray Array[ ] String [ , const separators String ] )`

Description

breakApart splits a string into an array of substrings; each substring is written to an element of the array *tokenArray*. You can specify one or more delimiting characters in *separators*. If you omit *separators*, substrings are delimited by a space. In either case, delimiting characters are not included in *tokenArray*. This method is useful for importing data from a text file into a table.

Note: Two delimiters with nothing between them parse as a token and result in an empty array element.

▪

breakApart example

In the following example, the **pushButton** method for a button named *breakToArray* creates three arrays from the same string. The first time, the call to the **breakApart** method does not specify any delimiters; by default, the method treats spaces as delimiters. The second time, the call to **breakApart** specifies the asterisk as a delimiter. Empty array elements are created each time an asterisk immediately follows another asterisk. The third time, the question mark, comma, and semicolon are listed as delimiters. The space is not used as a delimiter.

```
; breakToArray::pushButton
method pushButton(var eventInfo Event)
var
  ar Array[] String ; Must be resizable
  s String
endvar

s = "this is, a : delimited ? string"

s.breakApart(ar) ; breaks on spaces by default
ar.view()
{
ar = this
     is,
     a
     :
     delimited
     ?
     string
}

s = "this*is*a*delimited**string"
s.breakApart(ar, "*") ; breaks on specified characters
ar.view()
{
ar = this
     is
     a
     delimited

     string
}

s = "this is, a : delimited ? string"
s.breakApart(ar, ",:?") ; breaks on specified characters
                        ; this time, no space in list of delimiters
ar.view()
{
ar = this is
      a
      delimited
      string
}

endMethod
```

▪

chr procedure

See also Example String Type
Beginner

Returns the one-character string represented by an ANSI code.

Syntax

`chr ( const ansiCode SmallInt ) String`

Description

chr returns a one-character string containing the ANSI character corresponding to *ansiCode*. If *ansiCode* is not an integer between 1 and 255, an error results.

You can use **chr** to generate characters that are not easily accessible through the keyboard.

▪

chr example

In the following example, the **pushButton** method for a button named *showChar* assigns the ANSI character 167 to the *sectionChar* variable, converts character 167 to its key name, and assigns it to *sectionKeyName*. The method then displays both versions of the character in a dialog box.

```
; showChar::pushButton
method pushButton(var eventInfo Event)
var
  sectionChar    String
  sectionKeyName String
endVar
sectionChar = chr(167)                           ; get the character
sectionKeyName = chrToKeyName(chr(167))          ; get the key name
msgInfo("The section character", sectionChar +   ; show the character and
        " has a key name of " + sectionKeyName)  ; the key name
endMethod
```

chrOEM procedure

See also Example String Type

Returns the one-character string of an OEM code.

Syntax

```
chrOEM ( const oemCode SmallInt ) String
```

Description

chrOEM returns a one-character string containing the OEM character corresponding to *oemCode*. If *oemCode* is not an integer between 1 and 255, an error results.

You can use **chrOEM** to generate characters that are not easily accessible through the keyboard. See the *Guide to ObjectPAL* for more information.

▪

chrOEM example

In the following example, a form has a button named *showOEM* and a field named *fieldOne*. The **pushButton** method for *showOEM* displays the OEM character specified by the number in *fieldOne*.

```
; showOEM::pushButton
method pushButton(var eventInfo Event)
msgInfo("OEM char described by fieldOne", chrOEM(fieldOne))
endMethod
```

■

chrToKeyName procedure

See also Example String Type

Returns the virtual key-code string of a one-character string.

Syntax

`chrToKeyName ( const char String ) String`

Description

chrToKeyName returns the virtual key code of *char* as a string. A key name is one of the virtual key codes (such as VK_BACK for Backspace), but this method returns the Keyboard constant name as a string (such as "VK_BACK"). Alphanumeric characters and most symbols have a key name that consists simply of the character, for instance, "J" for the letter *J*.

▪

chrToKeyName example

See the example for **chr.**

▪

fill procedure

See also Example String Type
Beginner

Returns a string containing repeated instances of a character.

Syntax

```
fill ( const fillCharacter String, const fillNumber LongInt ) String
```

Description

fill returns a string containing the first character in *fillCharacter* (usually a one-character string), where *fillCharacter* is repeated the number of times specified in *fillNumber*. *fillNumber* must be a non-negative integer; if *fillNumber* is 0, **fill** returns an empty string.

▪

fill example

In the following example, the **pushButton** method for the *fillAndView* button creates two strings with the **fill** procedure. The method creates the first string by filling a variable with the same letter five times. The second string is created by repeating the string "Shakespeare" four times.

```
; fillAndView::pushButton
method pushButton(var eventInfo Event)
var
  str String
endVar
str = fill("X", 5)
str.View()                      ; displays the string
str = fill("Shakespeare", 4)    ; add a line break
                                ; after every occurrence
str.View()
; displays:   Shakespeare
;             Shakespeare
;             Shakespeare
;             Shakespeare
endMethod
```

▪

format procedure

See also Example String Type
Beginner

Returns a formatted string for display or printing.

Syntax

```
format ( const FormatSpec String, const value AnyType ) String
```

Description

format lets you control the way values are displayed or printed. *formatSpec* is a string expression containing one or more format specifications to be applied to String.

The following table lists the default format specification for each format category. This table also lists the valid data types for each format category. In addition to the data types listed, you can use AnyType values, as long as the value can be interpreted consistently with the format category.

Format category	meaning	Data types allowed	Default
Width	Set allowable field width and decimal precision	All	Entire data value
Alignment	Alignment within width	All	AR (right-aligned) for all numeric types, AL (left-aligned) for all others (including point)
Case	Uppercase or lowercase strings	All string types	No default
Edit	Specify characters and spacing	All numeric types	See following defaults
	Include a specified symbol		No default
	Decimal point character		ED. (period as decimal point)
	Whole number separator		No separator
	Number of leading zeros		None
	Symbol spacing		None
	Scientific notation		No
	Hide trailing spaces		No (show spaces)
	Use zeros as fill pattern		No
	Scale numbers up		No
	Precede with dollar sign		No
	U.S. or Int'l separators	U.S.	
Sign	Format of positive and negative numbers	All numeric	See following
	Positive		No leading positive sign 999
	Negative		Leading minus sign -999
Date	Specify date formats	Date & DateTime	mm/dd/yy(yy) for Date or hh:mm:ss am(pm), mm/dd/yy(yy) for DateTime
Time	Specify time formats	Time & DateTime	hh:mm:ss am(pm) for Date or hh:mm:ss am(pm), mm/dd/yy(yy)

			for DateTime
Logical	Logical value representation	Logical	True/False

You can combine two or more format specifications in *formatSpec* by separating them with commas.

Type	Spec	Meaning
Width	W*n*	*n* specifies total format width, including all special characters, leading symbols or spaces, decimal point, and whole number separators
	W.*n*	*n* specifies number of decimal places, so W12.2 specifies a field of 12 characters, two of which are after the decimal character
	W.W	Use decimal places from Windows numbers
	W.$	Use decimal places from Windows currency
Alignment	AL	Left align in field
	AR	Right align in field
	AC	Center in field
Case	CU	Convert to uppercase
	CL	Convert to lowercase
	CC	Convert to initial capitals
Edit	E(*s*)	*s* specifies symbol to precede number
	E$W	Include currency symbol from Windows
	ED*d*	*d* specifies decimal point character
	EDW	Use Windows decimal-point character
	EN*c*	*c* specifies whole-number separator
	ENW	Use Windows whole-number separator
	EL*n*	*n* specifies the number of leading zeros
	ELW	Use Windows leading zero setting
	EP0	No symbol spacing
	EP-	Make symbol spacing for negatives
	EP+	Make symbol spacing for positives
	EPB	Make symbol spacing for all numbers
	EPW	Use Windows symbol spacing setting
	ES	Use scientific notation
	ET	Hide trailing spaces
	EZ	Use zeros as fill pattern
	EB	Use blanks as fill pattern
	E*	Use '*' as fill pattern
	E+*n*	Scale the number up
	E-*n*	Scale the number down
	E$	The same as E($)
	EC	The same as EN (or EN.D)
	EI	The same as ED (or ED,N. if EC is set)
Sign	S+0	Format positives as $999
	S+1	Format positives as +$999

	S+2	Format positives as $+999
	S+3	Format positives as $999+
	S+4	Format positives as 999$
	S+5	Format positives as +999$
	S+6	Format positives as 999+$
	S+7	Format positives as 999$+
	S+8	Format positives as $999DB
	S+W	Format positives as Windows currency
	S-0	Format negatives as ($999)
	S-1	Format negatives as -$999
	S-2	Format negatives as $-999
	S-3	Format negatives as $999-
	S-4	Format negatives as (999$)
	S-5	Format negatives as -999$
	S-6	Format negatives as 999-$
	S-7	Format negatives as 999$-
	S-8	Format negatives as $999CR
	S-W	Format negatives as Windows currency
	SP	(The same as S-0)
	S-	(The same as S-1)
	S+	(The same as S-1+1)
	SC	(The same as S-8)
	SD	(The same as S-8+8)
Date	DW1	Day of week as Mon
	DW2	Day of week as 'Monday'
	DWL	Day of week from Windows Long Date
	DM1	Month as 1
	DM2	Month as 01
	DM3	Month as Jan
	DM4	Month as January
	DML	Month from Windows Long Date
	DMS	Month from Windows Short Date
	DD1	Day as 1
	DD2	Day as 01
	DDL	Day from Windows Long Date
	DDS	Day from Windows Short Date
	DY1	Year as 1
	DY2	Year as 01
	DY3	Year as 1901
	DYL	Year from Windows Long Date

	DYS	Year from Windows Short Date
	DO(*s*)	*s* specifies order and separators, use %W for weekday,%D for numeric day, %M for month, and %Y for year, separators are literal, so 12/28/92 as DO(%W %M-%D-%Y) is Mon 12-28-92
	DOL	Order and separators as Windows Long Date
	DOS	Order and separators as Windows ShortDate
	D1	This is the default date format
	D2	As DM4Y3O(%M %D,%Y)
	D3	As DO(%M/%D)
	D4	As DO(%M/%Y)
	D5	As DM3O(%D-%M-%Y)
	D6	As DM3O(%M %Y)
	D7	As DM3Y3O(%D-%M-%Y)
	D8	As DY3O(%M/%D/%Y)
	D9	As DO(%D.%M.%Y)
	D10	As DO(%D/%M/%Y)
	D11	As DO(%Y-%M-%D)
	DEYEA(*s*)	*s* specifies A.D. dates
	DEYEB(*s*)	*s* specifies B.C. dates
Time	TH1	Hours as 1T
	TH2	Hours as 01
	THW	Hours from Windows
	TM1	Minutes as 1
	TM2	Minutes as 01
	TMW	Minutes from Windows
	TS1	Seconds as 1
	TS2	Seconds as 01
	TSW	Seconds from Windows
	TNA(*s*)	*s* is string to show after times before noon
	TNP(*s*)	*s* is string to show after times after noon
	TNW	Noon settings from Windows
	TO(*s*)	*s* specifies order and separators, use %H for hours, %M for minutes, %S for seconds, %N for am/pm
	TOW	Order and separators from Windows
Logical	LT(*s*)	*s* specifies representation of logical True value
	LF(*s*)	*s* specifies representation of logical False value
	LY	Logical values as Yes and No
	LO	Logical values as On and Off

-

format example

In the following examples, assume a form contains a field called *formatField* and a button named *demoFormat*. The **pushButton** method for *demoFormat* demonstrates a number of different format specifications. For each example, the method fills the *formatField* with the formatted string, then shows a copy of the format specification in a dialog box (with view). The method won't proceed to the next example until the View dialog box is closed; this gives you a way to examine both the format specification and the formatted output before moving to the next example.

```
; demoFormat::pushButton
method pushButton(var eventInfo Event)
var
  x  AnyType
  fs String
endVar
fs = "\"w6\",\"This is a test\""
formatField = format("w6","This is a test")
; displays This i
fs.view("Format Spec")

fs = "\"w6\",1234567"
formatField = format("w7",1234567)
; displays 1.e+6
fs.view("Format Spec")

fs = "\"w1\",( =5)"
formatField = format("w1",( =5))
; returns True, displays T
fs.View()

fs = "\"w9.2\",1234.567"
formatField = format("w9.2",1234.567)
; displays   1234.57
fs.View()

; Here are some examples of alignment specifications:
fs = "\"w20,ac\",\"This is\""
formatField = format("w20,ac","This is")
; displays              This is
fs.view()

fs = "\"w20,ac\",\"The Title\""
formatField = format("w20,ac","The Title")
; displays            The Title
fs.view()

fs = "\"w20,ac\",\"Of the Book\""
formatField = format("w20,ac","Of the Book")
; displays          Of the Book
fs.view()

fs = "\"w20,al\",123456"
formatField = format("w20,al",123456)
; displays 123456
fs.view()

fs = "\"w20,ar\",123456"
formatField = format("w20,ar",123456)
; displays               123456
fs.view()

; Here are some examples of case specifications:
fs = "\"cu\",\"the quick brown fox\""
formatField = format("cu","the quick brown fox")
; displays THE QUICK BROWN FOX
fs.view()
```

```
fs = "\"cl\",\"JUMPS OVER THE LAZY\""
formatField = format("cl","JUMPS OVER THE LAZY")
; displays jumps over the lazy
fs.view()

fs = "\"cc\",\"dOG.\""
formatField = format("cc","dOG.")
; displays Dog.
fs.view()

fs = "\"cc\",\"widgets'r us \" + \"too\""
formatField = format("cc","widgets'r us " + "too")
; displays Widgets'R Us Too
fs.view()

; Here are some examples of edit specifications:
x = 34567.89
fs = "\"w10.2, e$c\", x"
formatField = format("w10.2, e$c", x)      ; displays $34,567.89
fs.view()

fs = "\"w10.2, e$ci\", x"
formatField = format("w10.2, e$ci", x)     ; displays $34.567,89
fs.view()

fs = "\"w13.2, e$c\", x"
formatField = format("w13.2, e$c", x)      ; displays    $34,567.89
fs.view()

fs = "\"w14.2, e$cb, al\", x"
formatField = format("w14.2, e$cb, al", x) ; displays $   34,567.89
fs.view()

fs = "\"w15.2, e$cz, al\", x"
formatField = format("w15.2, e$cz, al", x) ; displays $000034,567.89
fs.view()

fs = "\"w15.2, e$c*, al\", x"
formatField = format("w15.2, e$c*, al", x) ; displays $****34,567.89
fs.view()

; Here are some examples of sign specifications:
x = -3456.12
fs = "\"w8.2, s+\", x"
formatField = format("w8.2, s+", x)          ; displays <196>3456.12
fs.view()

fs = "\"w11.2, e$c, sc\", x"
formatField = format("w11.2, e$c, sc", x)    ; displays $3,456.12CR
fs.view()

fs = "\"w14.2, e$c*, sp\", x"
formatField = format("w14.2, e$c*, sp", x)   ; displays ($***3,456.12)
fs.view()

fs = "\"w13.2, e$c*, s+\", x"
```

```
formatField = format("w13.2, e$c*, s+", x)   ; displays -$***3,456.12
fs.view()

fs = "\"w14.2, e$c*, sd\", x"
formatField = format("w14.2, e$c*, sd", x)   ; displays $***3,456.12CR
fs.view()

; Here are some miscellaneous examples:
fs = "\"D2\", Date(\"3/7/1948\""
formatField = format("D2", Date("3/7/1948"))   ; displays March 7, 1948
fs.view()

fs = "\"W9.2, AL\", 1234.123"
formatField = format("W9.2, AL", 1234.123)
; displays 1234.12 in field of 9 digits with 2 decimal places
fs.view()

fs = "\"W9.2, AR\", 1234.123"
formatField = format("W9.2, AR", 1234.123)
; displays 1234.12 right aligned in same field
fs.view()

; to display date and time in 24-hour format

fs = "\"TNA(), TNP(), TO(%H:%M:%S %D), DO(%W %M/%D/%Y)\"," +

     " dateTime(\"2:30:00 pm 11/24/92\")"

formatField = format("TNA(), TNP(), TO(%H:%M:%S %D), DO(%W %M/%D/%Y)",

                     dateTime("2:30:00 pm 11/24/92"))

; displays   14:30:00 Tue 11/24/92

fs.view("Format Spec")

; To display a date including the era (B.C. or A.D.):

fs = "\"DEYEA(A.D.)EB(B.C.)O(%M/%D/%Y %E)\",
       date(\"11/15/81\")"

formatField = format("DEYEA(A.D.)EB(B.C.)O(%M/%D/%Y %E)",
                     date("11/15/81"))
; displays 11/15/81 A.D.
fs.view()
endMethod
```

ignoreCaseInStringCompares procedure

See also Example String Type

Specifies whether to consider case when comparing strings.

Syntax

```
ignoreCaseInStringCompares ( const yesNo Logical )
```

Description

ignoreCaseInStringCompares specifies whether to consider case when comparing strings. Normally, upper- and lower-case letters don't match. For example, "Q" and "q" are not the same. But when you use **ignoreCaseInStringCompares(Yes)**, case doesn't matter, so "Q" equals "q." Once you call **ignoreCaseInStringCompares(Yes)**, it stays in effect until you call **ignoreCaseInStringCompares(No)**.

To find out if case is being considered, use **isIgnoreCaseInStringCompares**.

▪

ignoreCaseInStringCompares example

In the following example, the **pushButton** method for the *tryCompare* button checks whether Paradox is set to ignore case in string comparisons. If **isIgnoreCaseInStringCompares** returns Yes, the method uses **ignoreCaseInStringCompares** to set it to No▪which means that case is considered

▪then compares an uppercase and lowercase string. A message window informs the user that the strings are not equivalent. Next, the method turns on case ignore, and attempts the same comparison, which returns True.

```
; tryCompare::pushButton
method pushButton(var eventInfo Event)
var
  s1,
  s2  String
endVar
s1 = "cat"
s2 = "CAT"
if isIgnoreCaseInStringCompares() then
  ignoreCaseInStringCompares(No)
endIf
x = (s1 = s2)              ; the first  "=" assigns, all others compare
msgInfo(s1 + " = " + s2 + "?", x)     ; displays False
ignoreCaseInStringCompares(Yes)
x = (s1 = s2)
msgInfo(s1 + " = " + s2 + "?", x)     ; displays True
endMethod
```

isIgnoreCaseInStringCompares procedure

See also Example String Type

Reports whether case is considered when comparing strings.

Syntax

```
isIgnoreCaseInStringCompares ( ) Logical
```

Description

isIgnoreCaseInStringCompares returns True if case is considered when comparing strings; otherwise, it returns False.

To specify whether to consider case, use **ignoreCaseInStringCompares**.

▪

isIgnoreCaseInStringCompares example

See the example for **ignoreCaseInStringCompares**.

▪

isSpace method

See also Example String Type
Beginner

Reports whether a string contains only spaces (or is empty).

Syntax

isSpace (**const ***string* String **)** Logical

Description

isSpace returns True if *string* contains only whitespace, or if *string* is the empty string (""); otherwise, it returns False. Whitespace characters include spaces, tabs, carriage returns, linefeeds, and formfeeds.

isSpace example

The following example creates and checks several strings to see if the strings either contain only spaces, or contain nothing at all. This is the code for the **pushButton** method for the *valString* button:

```
; valString::pushButton
method pushButton(var eventInfo Event)
var
  s String
endVar
s = space(3)                         ; 3 spaces
msgInfo("3 Spaces", s.isSpace())     ; True
s = ""                               ; empty String
msgInfo("Empty String", s.isSpace()) ; True
s = "Z" + space(2)                   ; Z and 2 spaces
msgInfo("Z and 2 Spaces", s.isSpace()) ; False
endMethod
```

▪

keyNameToChr procedure

See also Example String Type

Returns the one-character string represented by a virtual key-code string.

Syntax

`keyNameToChr ( const keyName String ) String`

Description

keyNameToChr returns the one-character string represented by the virtual key code *keyName*.

keyName must be one of the Keyboard constants (such as VK_BACK for Backspace), but must be supplied as a string (such as "VK_BACK"), not a constant. Alphanumeric characters and most symbols have a key name that consists simply of the character, for instance, "J" for the letter *J*.

▪

keyNameToChr example

See the example for **keyNameToVKCode.**

▪

keyNameToVKCode procedure

See also Example String Type

Returns the VK_Code of a virtual key-code string.

Syntax

keyNameToVKCode (** const ***keyName* String **)** SmallInt

Description

keyNameToVKCode returns the virtual key code (VK_Code) of the character represented by the virtual key code *keyName*, given as a string.

keyName must be one of the Keyboard constants (such as VK_BACK for Backspace), but must be supplied as a string (such as "VK_BACK"), not a constant. Alphanumeric characters and most symbols have a key name that consists simply of the character, for instance, "J" for the letter *J*.

▪

keyNameToVKCode example

In the following example, the **pushButton** method for *showCode* sets the string variable *keyStr* to an open bracket ([), then displays the ANSI code and the key name of *keyStr* in a dialog box.

```
; showCode::pushButton
method pushButton(var eventInfo Event)
var
  keyStr  String
endVar
keyStr = "["              ; set the key name for open bracket
msgInfo("VK_Code/Char", "VK_Code: " +           ; VK_Code 91
        String(keyNameToVKCode(keyStr)) +
        "\nCharacter: " + keyNameToChr(keyStr)) ; char "["
endMethod
```

▪

lower method

See also Example String Type
Beginner

Converts a string to lowercase.

Syntax
`lower ( ) String`

Description
lower converts a string to lowercase letters. Use **upper** to convert a string to uppercase letters.

lower example

In the following example, the **pushButton** method for *makeLower* creates an uppercase string, then uses **lower** to display it in lowercase.

```
; makeLower::pushButton
method pushButton(var eventInfo Event)
var
  myText String
endVar
myText = "HEY, EVERYBODY! IT'S QUITTIN' TIME"
msgInfo("Official Notice", myText.lower())
; displays "hey everybody! it's quittin' time"
endMethod
```

▪

lTrim method

See also Example String Type
Beginner

Removes leading blanks from a string.

Syntax

lTrim ( ) String

Description

lTrim removes spaces and Tab characters from the left end of a string.

lTrim example

In the following example, the **pushButton** method for *trimLeft* creates a string with leading spaces and a leading tab (the escape sequence \t). The method displays the original string, uses **lTrim** to remove the leading nonprinting characters, then displays the trimmed version.

```
; trimLeft::pushButton
method pushButton(var eventInfo Event)
var
  trimMe, trimmed String
endVar
trimMe = "  \t   First word"  ; string with spaces and a tab
msgInfo("Original string", trimMe)

trimmed = trimMe.lTrim()      ; trim off spaces and tab
msgInfo("A slightly shorter version", trimmed)
; displays "First word"
endMethod
```

▪

match method

See also Example String Type
Beginner

Compares a string with a pattern.

Syntax

```
match ( const pattern String [ , var matchVar String ] * ) Logical
```

Description

match tests whether a string matches a pattern, and if so, extracts the components of the string that match the wildcard elements. The value of *pattern*, like patterns in queries, consists of characters interlaced with the wildcard operators .. and @. The .. matches any number of characters (including none), while the @ matches any single character. Also as in queries, **match** ignores or considers case depending on system settings (default: ignore case). Use **isIgnoreCaseInStringCompares** to find out what the system setting is, and use **ignoreCaseInStringCompares** to turn case-sensitivity on or off.

matchVar is a variable to which the matching portion will be assigned. **match** assigns matched patterns to one or more *matchVar* variables as the patterns are found. The portions of the string matching the wildcard elements are assigned to the variables from left to right. Since there may be multiple matches, the first matching substring is assigned to the first variable, the second matching substring to the second variable, and so on. If no match is found, variables are not assigned values.

Quotes in *pattern* require special handling, periods do not. To embed a quote in *pattern*, precede it with a backslash, as follows: \". **match** in ObjectPAL treats periods as alphanumeric characters, unlike earlier versions of PAL which required backslashes to delimit periods.

match example

These statements demonstrate match functionality.

```
var
  s, x, y, z String
endVar

s = "this and that"

msgInfo("match?", s.match("t.."))          ; displays True
msgInfo("match?", s.match("@his.."))       ; displays True
msgInfo("match?", s.match("@ and that"))   ; displays False
msgInfo("match?", s.match("..and.."))      ; displays True

msgInfo("match?", s.match("..and..", x, y))
                      ; displays True (x = this, y = that)

msgInfo("match?", s.match("T..", z))
  ; If isIgnoreCaseInString() is False, this statement displays
  ; False, and z is not assigned. Use
  ; ignoreCaseInStringCompares(Yes) to get this to display
  ; True, and set z to "his and that"
```

▪

oemCode procedure

See also Example String Type

Returns the OEM code of a one-character string.

Syntax

`oemCode ( const char String ) SmallInt`

Description

oemCode returns the OEM code of *char*, where *char* is a one-character string. The OEM code returned is an integer between 1 and 255.

oemCode example

See the example for **ansiCode**.

▪

readFromClipboard method

See also Example String Type

Reads text from the Clipboard.

Syntax

`readFromClipboard ( ) Logical`

Description

readFromClipboard reads a string from the Clipboard. The format read from the Clipboard is text (CF_TEXT). readFromClipboard returns True if successful and False if unsuccessful.

readFromClipboard example

In the following example, a form has two buttons: readFromClipboard and writeToClipboard. The first button will read text from the Clipboard into a String variable which will then be stored in a table. The second button read a String value from a table and writes it out to the Clipboard.

The following code is attached to the pushButton method for btnReadFromClipboard:

```
; btnReadFromClipboard::pushButton
method pushButton(var eventInfo Event)
var
   vrString  String
   tcString  TCursor
endVar

   ;// Open table to hold Strings
  tcString.open(mystrings.db)
   if vrString.readFromClipboard() then
      ;// Add a record to the table and insert the value
      tcString.insertRecord()
      tcString.stringField = vrString
      tcString.unlockRecord()
   endIf
   tcString.close()
endMethod
```

The following code is attached to the pushButton method for btnWriteToClipboard:

```
; btnWriteToClipboard::pushButton
method pushButton(var eventInfo Event)
var
   vrString  String
   tcString  TCursor
endVar

   ;// Open table to which contains strings
   tcString.open(mystrings.db)
   ;// Make sure there is data in the table
   if tcStrings.nRecords() <> 0 then
      ;// Copy a value to the String variable
      vrString = tcString.stringField
      ;// Write it out to the Clipboard
      vrString.writeToClipboard()
   endIf
   tcString.close()
endMethod
```

▪

rTrim method

See also Example String Type
Beginner

Removes trailing blanks from a string.

Syntax
`rTrim ( ) String`

Description
rTrim removes spaces, tabs, carriage returns, and linefeed characters from the right end of a string.

rTrim example

In the following example, the **pushButton** method for *trimRight* creates a string with trailing spaces. The method displays the original string, uses **rTrim** to remove the trailing nonprinting characters, then displays the trimmed version.

```
; trimRight::pushButton
method pushButton(var eventInfo Event)
var
  trimMe, trimmed String
endVar
trimMe = "Last word     "     ; string with trailing spaces
msgInfo("Original string", trimMe + "The end")
; displays "Last word     The end"

trimmed = trimMe.rTrim()      ; trim off spaces
msgInfo("A slightly shorter version", trimmed + "The end")
; displays "Last wordThe end"
endMethod
```

▪

search method

See also Example String Type
Beginner

Returns the position of one string inside another.

Syntax

`search ( const str String ) SmallInt`

Description

search tests for an occurrence of *str* within a target string. If *str* is found, **search** returns the starting character position of *str* within the target string; otherwise, it returns 0. The search always begins at the first character of the target string.

By default, **search** is case-sensitive, but you can use **ignoreCaseInStringCompares** to make it case-insensitive.

search example

The following example searches for parts of the string "Goliath" and "Golgolithic". The following code is attached to the **pushButton** method for the *searchStr* button:

```
; searchStr::pushButton
method pushButton(var eventInfo Event)
var
  s String
endVar
s = "Goliath"
msgInfo("Where is lia in Goliath?", s.search("lia")) ; displays 3
msgInfo("Where is lai in Goliath?", s.search("lai")) ; displays 0
ignoreCaseInStringCompares(No)
s = "Golgolithic"
msgInfo("Where is gol in Golgolithic?", s.search("gol"))
; displays 4
; Note: If ignoreCaseInStringCompares is on, the last
; search yields a 1 instead.
endMethod
```

▪

size method

See also   Example   String Type
Beginner

Returns the length of a string.

Syntax
```
size ( ) LongInt
```

Description
size returns the number of characters (including spaces) in a string.

▪

size example

In the following example, the **pushButton** method for *getSize* assigns a string to the variable *sourceText*, then displays the sentence and its size in a dialog box. The method then uses **size** to get the first half of *sourceText*, and assign it back to *sourceText*. The size of the *sourceText* and the smaller *sourceText* are displayed in a dialog box.

```
; getSize::pushButton
method pushButton(var eventInfo Event)
var
  sourceText String
endVar
sourceText = "This is a short sentence."
msgInfo("Size", "Length: " + String(sourceText.size()) +
                "\n" + sourceText)
; displays   Length: 25
;            This is a short sentence.

; now chop the sentence in half
sourceText = subStr(sourceText, 1, SmallInt(sourceText.size()/2))
msgInfo("Half-Size", "Length: " + strVal(sourceText.size())
                     + "\n" + sourceText)
; displays   Length: 12
;            This is a sh
endMethod
```

▪

space method

See also Example String Type
Beginner

Creates a string of a specified number of spaces.

Syntax

```
space ( const numberOfSpaces LongInt ) String
```

Description

space returns a string of *numberOfSpaces* spaces.

▪

space example

See the example for **isSpace**.

▪

string procedure

See also Example String Type
Beginner

Casts a value as a String.

Syntax
string (const ***value*** AnyType [**,** const ***value*** AnyType] * **)** String

Description
string casts (converts) an expression *value* to a String. If you specify multiple arguments, **string** will cast them all to strings and concatenate them to one string.

▪

string example

In the following example, the **pushButton** method for *getNumToString* requests a number from the user, then casts it as a string and concatenates it with another string for display in a **msgInfo** dialog box.

```
; getNumToString::pushButton
method pushButton(var eventInfo Event)
var
   nn Number
endVar
nn = 0.0                     ; initialize the number
nn.View("Enter a number")    ; display it, and ask for input

; Note: Because you can enter only one argument for the text of
; the msgInfo dialog box, if you have any non-string elements, they
; must be cast as strings, then concatenated. Here, nn is cast
; to a String type before being concatenated with "You entered "
msgInfo("Status", "You entered " + string(nn))
msgInfo("Status", string("You entered ", nn))  ; also works
endMethod
```

▪

strVal procedure

See also Example String Type

Converts a value to a string.

Syntax

```
strVal ( const value AnyType ) String
```

Description

strVal converts *value* to a string. The data type of *value* can be any of the types represented by AnyType.

▪

strVal example

See the example for **size**.

subStr method

See also Example String Type
Beginner

Returns a portion of a string.

Syntax

`**substr** **(** const ***startIndex*** LongInt [ , const ***numberOfChars*** LongInt ] **)** String`

Description

substr returns a portion of a string that starts at *startIndex* and continues for *numberOfChars* characters. The value of *startIndex* must be greater than 0 and less than or equal to the size of the string. If *numberOfChars* is 0, **substr** returns a null string. If *numberOfChars* is omitted, **substr** returns the character at position *startIndex*.

▪

subStr example

In the following example, assume a form contains a button named *getPhone* and four fields named *wholePhone*, *phAreaCode*, *phExchange*, and *phNumber*. The method in this example uses **substr** to extract the three groups of digits from a U.S. phone number. The following code is attached to the **pushButton** method for *getPhone*.

```
; getPhone::pushButton
method pushButton(var eventInfo Event)
var
  phoneNum  String
endVar
phoneNum = wholePhone.Value
; assume phone number has been entered as ###-###-####
; start from first position, take three characters
phAreaCode.Value = phoneNum.substr(1, 3)  ; get the area code
phExchange.Value = phoneNum.substr(5, 3)  ; get the exchange
phNumber.Value   = phoneNum.substr(9, 4)  ; get the number
beep()
endMethod
```

toANSI method

See also Example String Type

Converts a string of OEM characters to ANSI characters.

Syntax

```
toANSI ( ) String
```

Description

toANSI converts a string of OEM characters to ANSI characters.

toANSI example

In the following example, the **pushButton** method for a button named *showANSI* displays a string in two ways: in the title of the dialog box the string is displayed as is; in the window of the dialog box, the string is first converted to ANSI. The last character in the string is the copyright symbol (©). This symbol prints in the title of the dialog box; however, in the window of the dialog box, the symbol is replaced by an underbar (_).

```
; showANSI::pushButton
method pushButton(var eventInfo Event)
var
  ss String
endVar
; string plus copyright symbol
ss = "A string of characters " + chr(169)
msgInfo(ss, ss.toANSI())
; displays string plus "_" in window of dialog box - system-dependent
endMethod
```

toOEM method

See also Example String Type

Converts a string of ANSI characters to OEM characters.

Syntax

`toOEM ( ) String`

Description

toOEM converts a string of ANSI characters to OEM characters.

toOEM example

In the following example, the **pushButton** method for a button named *showOEM* displays a string in two ways: in the title of the dialog box the string is displayed as is; in the window of the dialog box, the string is first converted to OEM. The last character in the string is the copyright symbol (©). This symbol prints in the title of the dialog box; however, in the window of the dialog box, the symbol is replaced by the letter *c*.

```
; showOEM::pushButton
method pushButton(var eventInfo Event)
var
  ss String
endVar
; string plus copyright symbol
ss = "A string of characters " + chr(169)
msgInfo(ss, ss.toOEM())
; displays string plus "c" in window of dialog box
endMethod
```

upper method

See also Example String Type
Beginner

Converts a string to uppercase.

Syntax

```
upper ( ) String
```

Description

upper converts a string to uppercase letters. Use **lower** to convert a string to lowercase letters.

-

upper example

In the following example, the **pushButton** method for *makeUpper* gets a string from the user, then converts it to uppercase. The converted string is then compared to an uppercase string constant.

```
;makeUpper:pushButton
method pushButton(var eventInfo Event)
const
  ORDERTYPE = "BIDORDER"   ; concatenate two valid types
endConst
var
  myText String
  x      SmallInt
endVar
myText = ""                           ; initialize the string
myText.view("Enter 'Bid' or 'Order'") ; get a response
myText = myText.upper()               ; convert to uppercase
if search(ORDERTYPE, myText) > 0 then
  ; search for a matching string -- returns location
  ; of match, or zero if no match
  msgInfo("Status", "You entered a valid type.")
else
  msgStop("Stop", "You must enter either Bid or Order.")
endIf
endMethod
```

▪

vkCodeToKeyName procedure

See also Example String Type

Converts a virtual keycode constant to a virtual keycode string.

Syntax

vkCodeToKeyName (const ***vkCode*** SmallInt **)** String

Description

vkCodeToKeyName returns the virtual key-code name, as a String, of the character represented by the integer value *vkCode*.

This method returns the name of a Keyboard constant (such as VK_BACK for Backspace) as a string (such as "VK_BACK"), not a constant. Alphanumeric characters and most symbols have a key name that consists simply of the character, for instance, "J" for the letter *J*.

▪

vkCodeToKeyName example

See the example for **ansiCode**.

▪

writeToClipboard method

See also　　Example　　String Type

Writes a string to the Clipboard.

Syntax

`writeToClipboard ( ) Logical`

Description

writeToClipboard writes a string to the Clipboard. The format copied to the Clipboard is text (CF_TEXT). **writeToClipboard** returns True if successful and False if unsuccessful. The text copied to the Clipboard is ANSI.

writeToClipboard example

In the following example, a form has two buttons: readFromClipboard and writeToClipboard. The first button will read text from the Clipboard into a String variable which will then be stored in a table. The second button read a String value from a table and writes it out to the Clipboard.

The following code is attached to the pushButton method for btnReadFromClipboard:

```
; btnReadFromClipboard::pushButton
method pushButton(var eventInfo Event)
var
   vrString  String
   tcString  TCursor
endVar

   ;// Open table to hold Strings
  tcString.open(mystrings.db)
   if vrString.readFromClipboard() then
      ;// Add a record to the table and insert the value
      tcString.insertRecord()
      tcString.stringField = vrString
      tcString.unlockRecord()
   endIf
   tcString.close()
endMethod
```

The following code is attached to the pushButton method for btnWriteToClipboard:

```
; btnWriteToClipboard::pushButton
method pushButton(var eventInfo Event)
var
   vrString  String
   tcString  TCursor
endVar

   ;// Open table to which contains strings
   tcString.open(mystrings.db)
   ;// Make sure there is data in the table
   if tcStrings.nRecords() <> 0 then
      ;// Copy a value to the String variable
      vrString = tcString.stringField
      ;// Write it out to the Clipboard
      vrString.writeToClipboard()
   endIf
   tcString.close()
endMethod
```

System type

Changes

The System type contains methods and procedures for displaying messages, finding out about your system, setting up a printer, manipulating the File Browser, working with the Help system, and more.

Methods and procedures for the System type

System

beep

close

compileInformation

constantNameToValue

constantValueToName

cpuClockTime

debug

deleteRegistryKey

desktopMenu

dlgAdd

dlgCopy

dlgCreate

dlgDelete

dlgEmpty

dlgNetDrivers

dlgNetLocks

dlgNetRefresh

dlgNetRetry

dlgNetSetLocks

dlgNetSystem

dlgNetUserName

dlgNetWho

dlgRename

dlgRestructure

dlgSort

dlgSubtract

dlgTableInfo

enableExtendedCharacters

enumDesktopWindowHandles

enumDesktopWindowNames

enumEnvironmentStrings

enumExperts

enumFonts

formatStringToDateTime
formatStringToNumber
formatStringToTime
getDefaultPrinterStyleSheet
getDefaultScreenStyleSheet
getDesktopPreference
getLanguageDriver
getMouseScreenPosition
getRegistryValue
getUserLevel
helpOnHelp
helpQuit
helpSetIndex
helpShowContext
helpShowIndex
helpShowTopic
helpShowTopicInKeywordTable
isErrorTrapOnWarnings
isMousePersistent
message
msgAbortRetryIgnore
msgInfo
msgQuestion
msgRetryCancel
msgStop
msgYesNoCancel
pixelsToTwips
play
projectViewerClose
projectViewerIsOpen
projectViewerOpen
printerGetInfo
printerGetOptions
printerSetCurrent
printerSetOptions
readEnvironmentString
readProfileString
resourceInfo
runExpert
searchRegistry

sendKeys
sendKeysActionID
setDefaultPrinterStyleSheet
setDefaultScreenStyleSheet
setDesktopPreference
setMouseScreenPosition
setMouseShape
setMouseShapeFromFile
setRegistryValue
setUserLevel
sleep
sound
sysInfo
tracerClear
tracerHide
tracerOff
tracerOn
tracerSave
tracerShow
tracerToTop
tracerWrite
twipsToPixels
version
winGetMessageID
winPostMessage
winSendMessage
writeEnvironmentString
writeProfileString

Changes to System type methods

The following table lists new methods and methods that were changed for version 7.

New	Changed
deleteRegistryKey	sysInfo
enumDesktopWindowHandles	dlgExport (moved to Data Transfer type for 7)
enumExperts	dlgExport (moved to Data Transfer type for 7)
enumRegistryKeys	dlgImportAsciiFix (moved to Data Transfer type for 7)
enumRegistryValueNames	dlgImportAsciiVar (moved to Data Transfer type for 7)
enumWindowHandles	dlgImportSpreadSheet (moved to Data Transfer type for 7)
formatStringToDateTime	exportASCIIFix (moved to Data Transfer type for 7)
formatStringToTime	exportASCIIVar (moved to Data Transfer type for 7)
getDesktopPreference	exportParadoxDOS (moved to Data Transfer type for 7)
getRegistryValue	exportSpreadsheet (moved to Data Transfer type for 7)
isMousePersistent	importASCIIFix (moved to Data Transfer type for 7)
runExpert	importASCIIVar (moved to Data Transfer type for 7)
searchRegistry	importSpreadsheet (moved to Data Transfer type for 7)
setMouseShapeFromFile	enumDesktopWindowNames
setDesktopPreference	enumWindowNames
setRegistryValue	resourceInfo
	sendKeys
	setMouseShape
	winGetMessageID
	winPostMessage
	winSendMessage
	disablePreviousError(moved to Form type for 7)

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
compileInformation	enumDesktopWindowNames
desktopMenu	enumWindowNames
disablePreviousError	sleep

enableExtendedCharacters
enumEnvironmentStrings
enumFormats
enumPrinters
enumRTLErrors
exportParadoxDOS
formatGetSpec
formatStringToDate
formatStringToNumber
getDefaultPrinterStyleSheet
getDefaultScreenStyleSheet
getLanguageDriver
getUserLevel
isErrorTrapOnWarnings
projectViewerClose
projectViewerIsOpen
projectViewerOpen
printerGetInfo
printerGetOptions
printerSetCurrent
printerSetOptions
resourceInfo
setDefaultPrinterStyleSheet
setDefaultScreenStyleSheet
setUserLevel

beep procedure

See also Example System Type
Beginner

Sounds the Windows default beep.

Syntax

```
beep ( )
```

Description

The beep is audible only if a sound device is installed on the system and set to active.

To send a sound of specified pitch and duration to the system speaker, use **sound.**

beep example

Prompts you to enter a number and beeps if the number is out of range. The following code is attached to a button's **pushButton** method.

```
; getANumber::pushButton
method pushButton(var eventInfo Event)
var
  someNumber  SmallInt
endVar
someNumber = 1
someNumber.view("Pick a number between 1 and 10")
while someNumber < 1 OR someNumber > 10
  beep()          ; beep
  sleep(100)      ; slight pause, otherwise beeps run together as one
  beep()
  msgStop("Oops", "That number is too large or too small. Try again.")
  someNumber.view("Pick a number between 1 and 10")
endwhile
endMethod
```

▪

close procedure

See also Example System Type
Beginner

Closes the current form.

Syntax

`close ( [ const returnValue AnyType ] )`

Description

close returns a value to the calling form when *returnValue* (optional) is specified. (Does not generate an error if *returnValue* is specified and there is no calling form.) Starts the process of closing the form, which includes removing the focus and departing.

close example

Closes the current form after asking you for confirmation.

```
; closeButton::pushButton
method pushButton(var eventInfo Event)
var
  qAnswer String
endVar
qAnswer = msgYesNoCancel("Closing Application",
          "Do you want to close this form?")
if qAnswer = "Yes" then
  close()                 ; close the current form
else
  message("Application not closed.")
endIf
endMethod
```

▪

compileInformation procedure

See also Example System Type

Lists information about the form most recently compiled.

Syntax

```
compileInformation ( var info DynArray[ ] AnyType )
```

Description

compileInformation is useful for analyzing large forms, libraries, scripts, and reports. It writes the data to the DynArray *info* that you declare and pass as an argument; its structure is:

Index	Definition
CodeSize	The compiled size of the code segment (in bytes).
CompileTime	Compile time (in milliseconds).
DataSize	The compiled size of the data segment (in bytes).
MethodCount	The number of methods that have code and/or comments.
SourceSize	The size of the uncompiled source code (in bytes).
SymbolTableSize	The compiled size of the symbol table (in bytes).

compileInformation example

Writes compiler information to a dynamic array *dynCompileInfo*, and then displays it in a **view** dialog box.

```
;analyzeObject::pushButton
method pushButton(var eventInfo Event)
  var
    dynCompileInfo   Dynarray[]   AnyType
  endVar

  compileInformation(dynCompileInfo)
  dynCompileInfo.view()
endmethod
```

▪

constantNameToValue procedure

See also Example System Type

Returns the numeric value of the constant *constantName*.

Syntax

`constantNameToValue ( const constantName String ) AnyType`

Description

Returns values only for predefined ObjectPAL constants; not for user-defined constants.

Note: For readability, ease of maintenance, and portability, use constants rather than numeric values.

constantNameToValue example

Returns the numeric value for the action constant DataBeginEdit.

```
; showValOfConst::pushButton
method pushButton(var eventInfo Event)
var
  constValue   AnyType
  constString  String
  tf           Logical
endvar
constValue = constantNameToValue("DataBeginEdit")  ; constant is passed as a
                                                   ; String
msgInfo("The value of DataBeginEdit is", constValue)
tf = constantValueToName("ActionDataCommands", constValue, constString)
if tf then   ; if the conversion worked properly, display the string
  msgInfo("The name of " + String(constValue) + " is", constString)
else
  msgInfo("Status", "Something went wrong with that conversion.")
endIf
endMethod
```

▪

constantValueToName procedure

See also Example System Type

Reports on the name of a constant.

Syntax

```
constantValueToName ( const groupName String, const value AnyType, var
constName String ) Logical
```

Description

Writes to *constName* the name of a constant whose value equals *value* that belongs to the group *groupName*, where *groupName* is one of the Types of Constants. Returns True if successful; otherwise, returns False.

Works for names of predefined ObjectPAL constants only; not for user-defined constants.

■

constantValueToName example

See the example for **constantNameToValue**.

cpuClockTime procedure

See also Example System Type
Beginner

Returns the number of milliseconds since the computer was booted.

Syntax

```
cpuClockTime ( ) LongInt
```

Description

The minimum clock increment is 55 milliseconds. This procedure is useful for measuring the interval between two events.

cpuClockTime example

Compares execution times for two for loops: one with an undeclared variable, the other with a declared variable. The code executes significantly faster when the variable is declared, although execution times vary by system.

```
; clockVars::pushButton
method pushButton(var eventInfo Event)
var
  fastVar    SmallInt
  delta      String
  startTime,
  stopTime   LongInt
endvar
startTime = cpuClockTime()                  ; clock's time before starting
for slowVar from 1 to 10000                 ; slowVar is undeclared
  slowVar = slowVar + 1
endFor
stopTime = cpuClockTime()                   ; clock's time after 10000 loops
delta = String(stopTime - startTime)        ; find the elapsed time using
delta.view("Time for undeclared variable") ; an undeclared variable --
                                            ; times vary by system
startTime = cpuClockTime()
for fastVar from 1 to 10000                 ; fastVar is declared
  fastVar = fastVar + 1
endFor
stopTime = cpuClockTime()
delta = String(stopTime - startTime)        ; find the elapsed time using
delta.view("Time for declared variable")   ; a declared variable
msgInfo("And the moral is:", "For the best performance, " +
        "declare variables!")
endMethod
```

▪

debug procedure

See also Example System Type

Halts execution of a method and invokes the Debugger.

Syntax

```
debug ( )
```

Description

A **debug** statement has the same effect as setting a breakpoint: The method stops executing, and the Debugger window opens with the pointer on the line containing **debug**. Unlike breakpoints, **debug** statements are saved with the method's source code. Useful for setting persistent breakpoints in methods while you are developing an application.

debug statements only take effect when you choose Program|Compile With Debug; otherwise, they are ignored. This makes it easy to toggle **debug** statements without having to remove them from your code. You test the application with Program|Compile With Debug turned on, and deliver the application with it turned off.

Note: **debug** works only in methods and procedures that you write, not for methods and procedures in the ObjectPAL run-time library.

▪

debug example

Executes a **for** loop. Halfway through the loop, the call to **debug** suspends execution and opens an Editor window containing the code. Choose Program|Run to resume execution, or use the other Debugger features. Assume the command Program|Compile With Debug on the ObjectPAL Editor menu is selected.

```
; startDebugAt50::pushButton
method pushButton(var eventInfo Event)
var
  i SmallInt
endVar
for i from 1 to 100
  message(i)
  if i = 50 then
      debug()     ; will work only if Program|Compile With Debug
                  ; ObjectPAL Editor menu command is checked
  endIf
endFor
endMethod
```

▪

deleteRegistryKey method

See also Example System Type

Deletes a registry key and/or value.

Syntax

`deleteRegistryKey ( const key String, const value String, const rootKey LongInt ) Logical`

Description

deleteRegistryKey deletes the registry key specified by *key*. **deleteRegistryKey** returns True, if successful, and False, if unsuccessful. If the paramater *value* is not empty, then the value name of the specified *key* is deleted, but not *key*. If *value* is empty, then only *key* will be deleted. If *key* has subkeys, *key* is not deleted. In this case, a warning is generated.

▪

deleteRegistryKey example

This example adds and then deletes a registry key. If the value parameter is left blank, the entire key is deleted, otherwise only the value and corresponding data are deleted.

```
var
   ar Array[] String
endvar

   setRegistryValue( "Software\\Borland\\Paradox\\Pdoxwin\\IDE\\MyKey",
"MyKeyValue", "MyKeyData", RegKeyCurrentUser )

   enumRegistryKeys( "Software\\Borland\\Paradox\\Pdoxwin\\IDE",
RegKeyCurrentUser, ar )
   ar.view()

   deleteRegistryKey( "Software\\Borland\\Paradox\\Pdoxwin\\IDE\\MyKey", "",
RegKeyCurrentUser )

   enumRegistryKeys( "Software\\Borland\\Paradox\\Pdoxwin\\IDE",
RegKeyCurrentUser, ar )
   ar.view()
```

▪

desktopMenu procedure

Example System Type

Displays the Paradox Desktop menu.

Syntax

```
desktopMenu ( )
```

Description

desktopMenu is useful when you use a form as a dialog box that doesn't have an associated menu.

After you call **desktopMenu**, the Paradox Desktop menu persists until it is replaced by one of the following actions:

- The current form or report loses focus.
- A call to **removeMenu** restores the default menu for the form or report.
- A call to **show** displays a custom menu.

▪

desktopMenu example

Calls **desktopMenu** in the **setFocus** method on the page of a dialog box to display the Paradox default menu.

```
;pge1 :: setFocus
method setFocus(var eventInfo Event)
   desktopMenu()
endMethod
```

▪

dlgAdd procedure

See also Example System Type

Displays the Add Records In <table> To dialog box.

Syntax

```
dlgAdd ( const tableName String )
```

Description

tableName specifies the source table.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgAdd example

Displays the Add Records In <table> To dialog box and fills in the *Customer* table name as the source table. You type the destination table name and close the dialog box.

```
; showAddDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Add Records In <table> To dialog box with Customer as the source
dlgAdd("customer.db")
endMethod
```

▪

dlgCopy procedure

See also Example System Type

Displays the Copy <table> To dialog box.

Syntax

```
dlgCopy ( const tableName String )
```

Description

dlgCopy displays the Copy <table> To dialog box. The argument *tableName* specifies the source table.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgCopy example

Displays the Copy <table> To dialog box and fills in the *Customer* table name as the source table. You provide the destination table name and close the dialog box.

```
; showCopyDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Copy <table> To dialog box with the Customer table as the source
DlgCopy("customer.db")
endMethod
```

▪

dlgCreate procedure

See also   Example   System Type

Displays dialog boxes to create a table.

Syntax

```
dlgCreate ( const tableName String )
```

Description

Displays the Create Table dialog box. The argument *tableName* specifies the name of table to create. When you choose a table type and close the dialog box, opens a Table Type dialog box for the specified table type.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgCreate example

Displays the Table Type dialog box. You choose the table type, fill out the field roster, and save the created table.

```
; showCreateDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Type dialog box -- table name is not used
dlgCreate("sometbl.db")
endMethod
```

■

dlgDelete procedure

See also Example System Type

Displays a warning dialog box prompting the user to confirm deletion of the table.

Syntax

`dlgDelete ( const tableName String )`

Description

dlgDelete displays a warning dialog box prompting the user to confirm deletion of the table. The argument *tableName* specifies the name of table to delete.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgDelete example

Displays a warning dialog box and fills in the *Customer* table name as the table to delete. You close the dialog box and confirm the deletion.

```
; showDeleteDlg::pushButton
method pushButton(var eventInfo Event)
; invoke warning dialog box for the Customer table
dlgDelete("Customer.db")  ; same as Tools|Utilities|Delete
endMethod
```

■

dlgEmpty procedure

See also Example System Type

Displays a warning dialog box prompting the user to confirm the emptying of the table.

Syntax

```
dlgEmpty ( const tableName String )
```

Description

dlgEmpty displays a warning dialog box prompting the user to confirm the emptying of the table. The argument *tableName* specifies the name of table to empty.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgEmpty example

Displays the warning dialog box and fills in the *Customer* table name as the table to empty. You close the dialog box and confirm the data deletion.

```
method pushButton(var eventInfo Event)
; Displays the warning dialog box for Customer table
dlgEmpty("Customer.db")
endMethod
```

▪

dlgNetDrivers procedure

See also Example System Type

Opens the BDE page of the Preferences dialog box.

Syntax

```
dlgNetDrivers ( )
```

Description

dlgNetDrivers opens the BDE page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

For information on drivers, see About language drivers in the User's Guide help.

▪

dlgNetDrivers example

Opens the BDE page of the Preferences dialog box.

```
; showNetDrivers::pushButton
method pushButton(var eventInfo Event)
; invoke the BDE page of the Preferences dialog box
dlgNetDrivers()
endMethod
```

▪

dlgNetLocks procedure

See also Example System Type

Creates and displays a table of lock information.

Syntax

```
dlgNetLocks ( )
```

Description

dlgNetLocks displays the Select File dialog box and prompts you to choose a table. Click Open to create a Paradox table named LOCKS.DB in your private directory. If the table already exists, Paradox overwrites it without asking for confirmation. This method fails if the table is already open.

Here is the structure of LOCKS.DB:

Field name	Type & size		Description
Type	S	25	Lock type value.
Username	A	14	User name of lock owner.
Net Session	S		Net level session number.
Our Session	S		BDE session number, if the lock is a BDE lock.
Record Number	A	33	Record number of locked record (if Type = Record Lock (Write)).

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

Paradox creates the *Locks* table and displays it in a Table window.

Lock type values for System::dlgNetLocks

0 = Record lock

1 = Special record lock

2 = Group lock

3 = Image lock

4 = Table open (no lock)

5 = Table read lock

6 = Table write lock

7 = Table exclusive lock

9 = Unknown lock

dlgNetLocks example

Opens the Select File dialog box. Creates and displays a *Locks* table after you choose a file.

```
; showNetLocks::pushButton
method pushButton(var eventInfo Event)
; creates a table of lock info :PRIV:LOCKS.DB, then displays it
dlgNetLocks()
endMethod
```

▪

dlgNetRefresh procedure

See also Example System Type

Displays the Database page of the Preferences dialog box.

Syntax

```
dlgNetRefresh ( )
```

Description

dlgNetRefresh displays the Database page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

For more information, see Database page (Preferences dialog box) in the User's Guide help.

dlgNetRefresh example

Opens the Database page of the Preferences dialog box.

```
; showNetRefresh::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog
dlgNetRefresh()
endMethod
```

▪

dlgNetRetry procedure

See also Example System Type

Displays the Database page of the Preferences dialog box.

Syntax

```
dlgNetRetry ( )
```

Description

dlgNetRetry displays the Database page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

For more information, see Database page (Preferences dialog box) in the User's Guide help.

dlgNetRetry example

Opens the Database page of the Preferences dialog box.

```
; showNetRetryDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog box
dlgNetRetry()
endMethod
```

dlgNetSetLocks procedure

See also Example System Type

Displays the Table Locks dialog box, where you place a lock on a table.

Syntax

```
dlgNetSetLocks ( )
```

Description

dlgNetSetLocks displays the Table Locks dialog box, where you place a lock on a table. ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

▪

dlgNetSetLocks example

Opens the Table Locks dialog box.

```
; showSetLocks::pushButton
method pushButton(var eventInfo Event)
dlgNetSetLocks()  ; invoke the Table Locks dialog box
endMethod
```

▪

dlgNetSystem procedure

See also Example System Type

Displays the BDE page of the Preferences dialog box.

Syntax

```
dlgNetSystem ( )
```

Description

dlgNetSystem displays the BDE page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgNetSystem example

Opens the BDE page of the Preferences dialog box.

```
; showNetSystem::pushButton
method pushButton(var eventInfo Event)
; invoke the BDE page of the Preferences dialog box
dlgNetSystem()
endMethod
```

▪

dlgNetUserName procedure

See also Example System Type

Displays the Database page of the Preferences dialog box, which shows the current user's network name.

Syntax

```
dlgNetUserName ( )
```

Description

dlgNetUserName displays the Database page of the Preferences dialog box, which shows the current user's network name. ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

For more information, see Database page (Preferences dialog box) in the User's Guide help.

dlgNetUserName example

Opens the Database page of the Preferences dialog box, which shows the current network user's name.

```
; showUserName::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog box
dlgNetUserName()
endMethod
```

▪

dlgNetWho procedure

See also   Example   System Type

Displays the Database page of the Preferences dialog box.

Syntax

```
dlgNetWho ( )
```

Description

dlgNetWho displays the Database page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

For more information, see Database page (Preferences dialog box) in the User's Guide help.

dlgNetWho example

Opens the Database page of the Preferences dialog box.

```
; showUserList::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog box
dlgNetWho()
endMethod
```

▪

dlgRename procedure

See also Example System Type

Displays the Rename <table> To dialog box.

Syntax

```
dlgRename ( const tableName String )
```

Description

dlgRename displays the Rename <table> To dialog box. The argument *tableName* specifies the table to rename.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgRename example

Displays the Rename <table> To dialog box and fills in *Customer* as the table to rename. You enter a new name and close the dialog box.

```
; showRenameDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Rename <table> To dialog box
dlgRename("customer.db")
endMethod
```

dlgRestructure procedure

See also Example System Type

Displays the Restructure Table dialog box.

Syntax

```
dlgRestructure ( const tableName String )
```

Description

dlgRestructure displays the Restructure Table dialog box. The argument *tableName* specifies the table to restructure, including the filename's extension. If *tableName* does not specify a path, **dlgRestructure** searches for the table in the working directory.

If *tableName* does not specify an extension, or specifies an extension of .DB, **dlgRestructure** displays the Restructure Paradox Table dialog box.

If *tableName* specifies an extension of .DBF, **dlgRestructure** displays the Restructure dBASE Table dialog box.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgRestructure example

Displays the Restructure Table dialog box and fills in *Customer* as the table to restructure. You modify the structure and close the dialog box.

```
; showRestructureDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Restructure Table dialog box for Customer table
dlgRestructure("customer.db")
endMethod
```

▪

dlgSort procedure

See also Example System Type

Displays the **Sort Table** dialog box.

Syntax

```
dlgSort ( const tableName String )
```

Description

tableName specifies the name of table to sort.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgSort example

Displays the Sort Table dialog box and chooses *Customer* as the table to sort. You create a sort specification and close the dialog box.

```
; showSortDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Sort Table dialog box
dlgSort("customer.db")
endMethod
```

dlgSubtract procedure

See also Example System Type

Displays the Subtract Records In <table> From dialog box.

Syntax

```
dlgSubtract ( const tableName String )
```

Description

dlgSubtract displays the Subtract Records In <table> From dialog box. The argument *tableName* specifies the table from which to subtract records.

The dialog box opens with the argument *tableName* filled in, prompting the user to choose the table to subtract from *tableName*. ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgSubtract example

Displays the Subtract Records In <table> From dialog box and fills in *Customer* as the source table from which to subtract records. You close the dialog box.

```
; showSubtractDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Subtract Records In <table> From dialog box
dlgSubtract("customer.db")  ;
endMethod
```

▪

dlgTableInfo procedure

See also Example System Type

Displays the Structure Information dialog box.

Syntax

```
dlgTableInfo ( const tableName String )
```

Description

Displays the Structure Information dialog box. The argument *tableName* specifies the table from which to obtain the structure information.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

dlgTableInfo example

Displays the Structure Information dialog box for the *Customer* table.

```
; showTableInfo::pushButton
method pushButton(var eventInfo Event)
; invoke the Structure Information dialog box for the Customer table
dlgTableInfo("customer.db")
endMethod
```

▪

enableExtendedCharacters procedure

See also Example System Type

Specifies whether you can enter extended character codes from the numeric keypad without turning NumLock on.

Syntax

```
enableExtendedCharacters ( const yesNo Logical ) Logical
```

Description

If *yesNo* is True, enables extended characters without NumLock. If *yesNo* is False, NumLock must be on to enter extended character codes; when NumLock is off, keypad keys function as navigation keys. Affects all forms, and remains active as long as Paradox is running. This setting is not saved when you exit.

Useful in international applications or other environments where keyboards do not have NumLock keys. Returns True if successful, otherwise False.

enableExtendedCharacters example

Enables extended characters when the form opens.

```
method open(var eventInfo Event)

   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:

   else
      ;// This code executes only for the form:
      doDefault
      enableExtendedCharacters(Yes)
   endIf

endMethod
```

▪

enumDesktopWindowHandles procedure

See also Example System Type

Lists the window handles of open windows owned by the Paradox desktop.

Syntax

enumDesktopWindowHandles (`var` ***windowHandles*** `DynArray [ ] AnyType [, const` ***className*** `String ]` **)**

Description

enumDesktopWindowHandles lists the handles of open windows owned by the Paradox Desktop. This procedure writes the list to the DynArray *windowHandles*. The index of *windowHandles* contains the hande and the value is the name of the window. The optional *className* argument specifies that the list generated will contain only windows whose *className* equals the name of the window class.

enumDesktopWindowHandles example

This example builds a DynArray of all the window titles open on the Paradox desktop and displays the DynArray.

```
method pushButton(var eventInfo Event)
var
   winHandles DynArray[] String
endvar

enumDesktopWindowHandles(winHandles)    ;// enumerate desktop window
                                        ;// handles to a dynarray
winHandles.view()                       ;// lists all windows open
                                        ;// in the Paradox Desktop

endMethod
```

▪

enumDesktopWindowNames procedure

See also   Example   System Type

Lists the names of open windows owned by the Paradox Desktop.

Syntax

```
1. enumDesktopWindowNames ( const tableName String ) Logical
2. enumDesktopWindowNames ( const windowNames Array [ ] String [, const
className String] )
```

Description

enumDesktopWindowNames lists the names of open windows owned by the Paradox Desktop. Syntax 1 creates the Paradox table *tableName* listing the name, class, position, and size of each window owned by Paradox. If *tableName* does not specify a path, **enumDesktopWindowNames** creates the table in the working directory. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

Here is the structure of the table:

Field name	Type & size		Description
WindowName	A	64	Name of window, or blank if the window has no name. Field size changed in version 5.0.
ClassName	A	63	Window type. Field size changed in version 5.0.
Position	A	12	Coordinates of upper left corner, for example (456, 553)
Size	A	12	Coordinates of lower right corner, for example (889, 221).
Handle	I		Window handle.
ChildId	I		ID number of child window (0 = no child window).
ParentHandle	I		Handle of parent window.
InstanceHandle	I		Handle of window instance.

Syntax 2 fills the array named in *winArray* with the names of the windows; you declare *winArray* before calling this method. Applications are listed in Windows z-order; that is, the application displayed "on top" is listed first in the array, the window in the second layer is listed second, and so on. The optional argument *className* specifies that only the names of windows whose class is equal to *className* will appear in *winArray*.

Compare this method to **enumWindowNames**, which lists all Windows applications running on your system.

enumDesktopWindowNames example

Writes the open desktop window titles to an array and shows the array. Next, creates and displays a table that lists the open desktop window names.

```
; getDesktopWinNames::pushButton
method pushButton(var eventInfo Event)
var
  winNames Array[] String
  tempTV           TableView
endvar
tempTV.open("Customer")               ; open a table view
enumDesktopWindowNames(winNames)      ; enum desktop window names to an array
winNames.view() ; lists all windows open in the Paradox Desktop, if
                ; method editor window is open, lists first 32 chars
enumDesktopWindowNames("wNameTbl.db") ; enum to a table
tempTV.open("wNameTbl")               ; show the table
endMethod
```

▪

enumEnvironmentStrings procedure

See also Example System Type

Lists all the items from the DOS environment.

Syntax

`enumEnvironmentStrings ( var values DynArray[ ] String ) Logical`

Description

enumEnvironmentStrings lists all the items from the DOS environment. This method writes the items to the dynamic array *values*, which you declare and pass as an argument.

enumEnvironmentStrings example

Creates a dynamic array *dyn* that lists items from the DOS environment and shows the dynamic array.

```
;thisButton::pushButton
method pushButton(var eventInfo Event)
  var
    dyn   DynArray[] String
  endVar

  enumEnvironmentStrings(dyn)
  dyn.view()
endmethod
```

▪

enumExperts procedure

See also Example System Type

Lists all experts available to Paradox.

Syntax

```
1. enumExperts ( const expertType String, var expertNames DynArray [ ]
AnyType )
2. enumExperts ( const expertType String, const expertName String )
```

Description

enumExperts lists the all the experts available to Paradox. The *expertType* parameter specifies the type of experts to list. ObjectPAL provides ExpertTypes constants for this purpose. Syntax 1 fills the DynArray *expertNames* with the names of the experts. Syntax 2 lists the experts out to a table. The format of the table is as follows :

Field name	Type & size		Description
Expert	Alpha	25	The registered name of the expert
Name	Alpha	25	The visible name of the expert
Description	Alpha	255	The help description text
File Name	Alpha	255	The expert file name, including the path
Icon	Graphic		The experts icon graphic

enumExperts example

This example enumerates the available experts, determines if expertForm is in the list, and runs it if it is available.

```
method pushButton(var eventInfo Event)
var
   da dynarray[] anytype
   expert string
endvar

expertForm = "Form"
enumExperts(  expTypeDocument, da )

if da.contains( expertForm ) then
   runExpert( expTypeDocument, expertForm )
else
   msgStop( "Error", "Unable to run the expert :" + expertForm )
endif
endmethod
```

▪

enumFonts procedure

See also   Example   System Type

Creates a table listing the fonts in your system.

Syntax

```
1. enumFonts ( const tableName String )
2. enumFonts ( const deviceType SmallInt, var fontList Array[] String )
```

Description

enumFonts creates a table listing the fonts in your system. The argument *tableName* specifies the table. By default, creates *tableName* in your working directory. If *tableName* already exists, overwrites it without asking for confirmation. Fails if *tableName* is open.

The structure of *tableName*:

Field name	Type & size		Description
FaceName	A	64	Font name. Example: Arial
FontSize	A	8	Font size in printer's points. Example: 12
Attribute	A	64	Display/print attribute. Example: Normal

Syntax 2 builds an array of fonts in *fontList*. The argument *deviceType* has two possible values: 1 (indicating screen display fonts), and 2 (indicating printer fonts).

▪

enumFonts example

Creates and lists system fonts in the table FONTS.DB. Then searches a TCursor for a font named Modern. If Modern is in the table, sets the Font.TypeFace property of a field object named *balanceField* to Modern.

```
; getFonts::pushButton
method pushButton(var eventInfo Event)
var
  fontsTC TCursor
  tempTV  TableView
endVar
enumFonts("fonts.db")         ; write font names to a table
tempTV.open("fonts.db")       ; show the table
dlgTableInfo("fonts.db")      ; show the table structure
fontsTC.open("fonts.db")
if fontsTC.locate("FaceName", "Modern") then
  balanceField.Font.TypeFace = "Modern"
endIf
fontsTC.close()
endMethod
```

▪

enumFormats procedure

See also Example System Type

Lists the current formats.

Syntax

enumFormats (const ***formatType*** String, var ***formats*** DynArray[ ] String **)** Logical

Description

enumFormats lists the current formats. The argument *formatType* is one of the following: Date, Number, Time, or Logical. This method writes the list to *formats*, a dynamic array that you declare and pass as an argument.

Returns True if successful, otherwise False.

▪

enumFormats example

Creates a dynamic array *dyn* that lists the formats for Date; then displays *dyn*.

```
; btnInspectFormat :: pushButton
method pushButton(var eventInfo Event)
  var
    s   String
    dyn   DynArray[] String
  endVar

  s = "Date"
  s.view("Enter format to inspect")
  enumFormats(s, dyn)
  dyn.view()
endmethod
```

▪

enumFormNames procedure

See also Example System Type

Creates an array listing open forms.

Syntax

```
enumFormNames ( var formNames Array[ ] String )
```

Description

enumFormNames creates an array listing open forms. It fills the array *formNames* with the file names of the open forms in your desktop. You declare *formNames* as a resizeable array before calling this method. Forms are listed in Windows z-order; that is, the form displayed "on top" is listed first in the array, the form in the second layer is listed second, and so on.

enumFormNames example

Writes the file names of open forms to the array *openForms*, then displays *openForms*.

```
; getFormNames::pushButton
method pushButton(var eventInfo Event)
var
  openForms Array[] String
endVar
enumFormNames(openForms)
openForms.view()         ; Lists file names of open forms.
endMethod
```

▪

enumPrinters procedure

See also Example System Type

Lists printers installed in your system.

Syntax

```
enumPrinters ( var printers Array[ ] String ) Logical
```

Description

enumPrinters lists printers installed in your system. It fills the array *printers* with elements that each contain the name, driver name, and port (separated by commas) for every printer installed in your system. You declare *printers* as a resizeable array before calling this method.

For example, if the printer name is Postscript Printer, the driver is PSCRIPT.DRV, and the port is LPT1:.

```
PostScript Printer,pscript,LPT1:
```

You pass an array item to **printerSetCurrent** to specify the active printer. Use the String method **breakApart** to separate the components (for example, to display a list of printer names).

enumPrinters example

Gets a list of printers installed in your system. If the list includes a PostScript printer, **printerSetCurrent** makes it the current (active) printer. Assumes that PostScript printers use the driver PSCRIPT.DRV.

```
method pushButton(var eventInfo Event)
   var
      arPrinters,
      arPrnNames  Array[] String
      stDrvName,
      stPrnName,
      stPrnInfo   String
      i           SmallInt
   endVar

   stDrvName = "pscript"

   enumPrinters(arPrinters) ; Get a list of installed printers.

   ; See if the list includes a PostScript printer that
  ; uses the "pscript" driver.
   for i from 1 to arPrinters.size()
      stPrnInfo = arPrinters[i]

      ; Info is separated by commas.
      stPrnInfo.breakApart(arPrnNames, ",")

      ; After breakApart, array item 1 is the printer name,
      ; array item 2 is the driver name.
      if arPrnNames[2] = stDrvName then
         ; If a PostScript printer is found, make it current.
         if printerSetCurrent(stPrnInfo) then
            msgInfo("Current printer:", arPrnNames[1])
         else
            errorShow()
         endIf
         return
      endIf
   endFor

   msgStop("Printer setup", "A PostScript printer must be installed.")
endMethod
```

enumRegistryKeys method

[See also](#) [Example](#) [System Type](#)

Fills an array with keys from the registry

Syntax

enumRegistryKeys (const ***key*** String, const ***rootKey*** LongInt , var ***keyinfo*** Array[] String **)** Logical

Description

enumRegistryKeys fills an array with keys from the registry. **enumRegistryKeys** returns True, if successful, and False, if unsuccessful. The array *keyinfo* is the array that is filled with the full key path from the specified *key* and *rootKey*. The subkeys of *key* are also placed in the array. If *key* is blank then all the subkeys from the *rootKey* will be enumerated.

▪

enumRegistryKeys example

```
enumRegistryKeys( "software\\Borland", regKeyLocalMachine, ar )
;\\ values in array
ar[1] = "software\\Borland"
ar[2] = "software\\Borland\\Database Engine "
ar[3] = "software\\Borland\\BLW32"
```

The following example will list and display all registry Keys residing under the "Software\Borland\Paradox\Pdoxwin" key.

```
var
   ar Array[] String
endvar

   enumRegistryKeys( "Software\\Borland\\Paradox\\Pdoxwin",
RegKeyCurrentUser, ar )
   ar.view()
```

■

enumRegistryValueNames method

See also Example System Type

Fills a dynamic array with values and data from the registry.

Syntax

enumRegistryValueNames (const ***key*** String, const ***rootKey*** LongInt, var ***keyInfo*** Array[] String **)** Logical

Description

enumRegistryValueNames fills the array *keyInfo* with the value names of the specified registry *key*. **enumRegistryValueNames** returns True, if successful, and False, if unsuccessful. *key* is entered as a path similar to a file path. However, unlike a file path, wildcards are not expanded. *key* cannot contain a single backslash and cannot be empty. The size of *key* is limited to 65,534 bytes. The array *keyInfo* is filled with the value names for the specified *key*. *rootKey* is analogous to a directory drive. The *rootKey* should be set with the predefined constants:

regKeyCurrentUser

regKeyClassesRoot

regKeyLocalMachine

regKeyUser

enumRegistryValueNames example

The following example lists all the Value Names under the Software\Borland\Paradox\Pdoxwin\IDE key, assigns them and their corresponding values to a dynarray, and then displays it to the user.

```
var
   ar      Array[]       String
   dyn   DynArray[]    AnyType
   i      SmallInt
endvar

   enumRegistryValueNames( "Software\\Borland\\Paradox\\Pdoxwin\\IDE",
RegKeyCurrentUser, ar )

   if ar.size() > 0 then
      for i from 1 to ar.size()
         dyn[ ar[ i ] ] = getRegistryValue( "Software\\Borland\\Paradox\
\Pdoxwin\\IDE", ar[ i ], RegKeyCurrentUser )
      endfor
   endif

   dyn.view()
```

▪

enumReportNames procedure

See also Example System Type

Creates an array listing open reports.

Syntax

```
enumReportNames ( var reportNames Array[ ] String )
```

Description

enumReportNames fills the array *reportNames* with the names of open reports in your desktop. You declare *reportNames* as a resizeable array before calling this method. Reports are listed in Windows z-order; that is, the report displayed "on top" is listed first in the array, the report in the second layer is listed second, and so on.

enumReportNames example

Writes the open report names to the array *openReports*; then displays *openReports*.

```
; getReportNames::pushButton
method pushButton(var eventInfo Event)
var
  openReports Array[] String
endVar
enumReportNames("openReports")
openReports.view()           ; lists forms, reports
endMethod
```

▪

enumRTLClassNames procedure

See also Example System Type

Creates a table listing the object types (also called classes) known to ObjectPAL.

Syntax

```
enumRTLClassNames ( const tableName String ) Logical
```

Description

enumRTLClassNames creates, in the working directory by default, the table *tableName* listing the names of all object types (also called classes) in the ObjectPAL run-time library. Overwrites *tableName* without asking for confirmation if it already exists. Fails if *tableName* is open. Returns True if successful, otherwise False.

The structure of the table:

Field name	Type & size		Description
ClassName	A	32	ObjectPAL type name. Example: UIObject

▪

enumRTLClassNames example

Writes the run-time library class names to the table *Rtlclass;* then displays *Rtlclass*.

```
; getRTLClasses::pushButton
method pushButton(var eventInfo Event)
var
  tempTV TableView
endVar
enumRTLCLassNames("rtlclass.db")       ; write class names to table
tempTV.open("rtlclass")                ; show the table
endMethod
```

▪

enumRTLConstants procedure

See also Example System Type

Creates a table listing the constants defined by ObjectPAL.

Syntax

`enumRTLConstants ( const tableName String ) Logical`

Description

enumRTLConstants creates, in the working directory by default, the table *tableName* listing all the constants defined in the ObjectPAL run-time library. Overwrites *tableName* without asking for confirmation if it already exists. Fails if *tableName* is open.

Here is the structure of the table:

Field name	Type & size		Description
GroupName*	A	32	One of the types of constants. Example: ActionDataCommands
ConstantName*	A	48	Symbolic name of the constant. Example: DataArriveRecord
Type	A	48	Data type of the constant. Example: SmallInt
Value	A	64	Value of the constant. Example: 3111

(* = key field)

Note: Although Paradox provides the values of constants, you should not use the values in code; rather, refer to constants by name. Use the **constantValueToName** and **constantNameToValue** methods to convert values and constants.

enumRTLConstants example

Writes the run-time library constant descriptions to the table *Rtlconst*; displays *Rtlconst*.

```
; getRTLConsts::pushButton
method pushButton(var eventInfo Event)
var
  tempTV TableView
endVar
enumRTLConstants("rtlconst.db")      ; write constants names to table
tempTV.open("rtlconst")              ; show the table
endMethod
```

▪

enumRTLErrors procedure

See also Example System Type

Lists the error codes and messages used by ObjectPAL.

Syntax

`enumRTLErrors ( const tableName String ) Logical`

Description

enumRTLErrors creates, in the working directory by default, the table *tableName* listing the error codes and messages used by ObjectPAL . Overwrites *tableName* without asking for confirmation if it already exists.

Here is the structure of the table:

Field name	Type & size		Description
ErrorNo*	N		Error number (decimal)
ErrorNoX	A	8	Error number (hex)
Name	A	48	Error constant name, if it exists (for example, peNoMemory); otherwise, it's the string <<Unmapped Error>>
Value	M	230	Error message. Example: Insufficient memory for this operation.

(* = key field)

Returns True if successful, otherwise False. If, for example, you pass **enumRTLErrors** an invalid table name, it fails and returns False.

enumRTLErrors example

Writes the run-time library error codes and descriptions to the table *Rtlerror;* displays *Rtlerror*.

```
;getRTLErrors::pushButton
method pushButton(var eventInfo Event)
  var
    tv  TableView
  endVar

  enumRTLErrors("RTLerror.db")
  tv.open("RTLerror.db")
endMethod
```

▪

enumRTLMethods procedure

See also Example System Type

Creates a table listing the RTL methods and RTL procedures in ObjectPAL.

Syntax

`enumRTLMethods ( const tableName String ) Logical`

Description

enumRTLMethods creates a table *tableName*, in the working directory by default, listing all the methods defined in the ObjectPAL run-time library. Overwrites *tableName* if it already exists, without asking for confirmation. Fails if *tableName* is open.

The table structure:

Field name	Type & size		Description
ClassName*	A	32	ObjectPAL type name. Example: FileSystem
MethodType*	A	8	Method (for methods) or Proc (for procedures)
MethodName*	A	64	Name of method (or procedure). Example: isDir
MethodArgs*	A	255	Arguments to the method (or procedure). Example: (const *dirName* String)
ReturnType*	A	32	Data type of returned value, or blank if no return value. Example: Logical

(* = key field)

enumRTLMethods example

Writes the run-time library method descriptions to the table *Rtlmeth*; displays *Rtlmeth*.

```
; getRTLMethods::pushButton
method pushButton(var eventInfo Event)
var
  tempTV TableView
endVar
enumRTLMethods("rtlmeth.db")    ; write method names to table
tempTV.open("rtlmeth")          ; show the table
endMethod
```

enumWindowHandles procedure

See also Example System Type

Lists the open window handles.

Syntax

`enumWindowHandles (` `var` ***`windowHandles`*** `DynArray [ ] AnyType [, const` ***`className`*** `String ]` **`)`**

Description

enumWindowHandles lists the handles of the open windows running under Windows. This procedure writes the list to the DynArray *windowHandles*. The index of *windowHandles* contains the hande and the value is the name of the window. The optional *className* argument specifies that the generated list will contain only windows whose *className* equals the name of the window class.

enumWindowHandles example

This example builds a DynArray of all the window handles and displays it.

```
method pushButton(var eventInfo Event)
var
   winHandles DynArray[] String
endvar

enumWindowHandles(winHandles)    ;// enumerate desktop window
                                 ;// handles to a dynarray
winHandles.view()                ;// lists all open windows

endMethod
```

▪

enumWindowNames procedure

Example1 Example2 System Type

Creates a list of applications currently running under Windows.

Syntax

```
1. enumWindowNames ( const tableName String ) Logical
2. enumWindowNames ( var windowNames Array [ ] String [, const className
String] )
```

Description

enumWindowNames creates a list of applications currently running under Windows. Syntax 1 creates *tableName* (in the working directory if no path is specified), which lists the name, class, position, size, and handles to each open application in your system (not just Paradox). Overwrites an existing *tableName* without asking for confirmation. Fails if *tableName* is open.

The table structure:

Field name	Type & size		Description
WindowName	A	64	Name of window, or blank if no name. Field size changed in version 5.0.
ClassName	A	64	Window type. Field size changed in version 5.0.
Position	A	12	Coordinates of upper left corner, for example (456, 553).
Size	A	12	Coordinates of lower right corner, for example (889, 221).
Handle	I		Window handle.
ChildId	I		ID number of child window (0 = no child window).
ParentHandle	I		Handle of parent window.
InstanceHandle	I		Handle of window instance.

Syntax 2 fills the array *winArray*, which you declare before calling **enumWindowNames**, with the names of all current applications, in Windows z-order. That is, the application displayed "on top" is listed first in the array, the application in the second layer is listed second, and so on. The optional argument *className* specifies that only the names of windows whose class is equal to *className* will appear in *winArray*.

Compare this method to **enumDesktopWindowNames,** which lists only open windows owned by Paradox.

▪

enumWindowNames example 1

The **pushButton** method for the button *getWindowNames* writes and displays open window information in two ways. First, it fills an array with the titles of the open windows and displays the array. Next, it fills a table with descriptions of the open windows, and displays the table and its structure.

```
; getWindowNames::pushButton
method pushButton(var eventInfo Event)
var
  winNames Array[] String
  tempTV           TableView
endvar
enumWindowNames(winNames)       ; write names to an array
winNames.view()                 ; lists all open windows
                                ; if a method editor window is open,
                                ; lists first 32 chars
enumWindowNames("wNameTbl.db")  ; write window descriptions to a table
tempTV.open("wNameTbl")         ; show the table
dlgTableInfo("wNameTbl.db")     ; show the table structure
endMethod
```

enumWindowNames example 2

The **pushButton** method for the button *btnCalc* writes the open window information to the table *:PRIV:APPS.DB.* Then it searches the table for *Calculator;* if found, it uses the Windows API call BringToTop, registered in the Uses window of the button, to switch to it. Otherwise, the **pushButton** method executes *CALC.EXE*.

```
;btnCalc :: Uses
uses USER32
   BringWindowToTop(WinHandle CWORD)
endUses

;btnCalc :: pushButton
method pushButton(var eventInfo Event)
   var
      stApps       String
      tc           TCursor
      siWinHandle  SmallInt
   endVar

   stApps = ":PRIV:APPS.DB"
   enumWindowNames(stApps)

   tc.open(stApps)

   if tc.locate("WindowName", "Calculator") then
      siWinHandle = tc.handle
      BringWindowToTop(siWinHandle)
   else
      execute("CALC.EXE")
   endIf
endmethod
```

▪

errorClear procedure

See also Example System Type

Clears the error stack.

Syntax

```
errorClear ( )
```

Description

errorClear clears (empties) the error stack of all error codes and error messages. For more information about the error stack, refer to the *Guide to ObjectPAL*.

▪

errorClear example

Clears the error stack.

```
; clearError::pushButton
method pushButton(var eventInfo Event)
errorClear()           ; clear the error stack
endMethod
```

errorCode procedure

See also Example System Type
Beginner

Returns a number that signifies the most recent run-time error or error condition.

Syntax

```
errorCode ( ) SmallInt
```

Description

errorCode returns a number that signifies the most recent run-time error or error condition. ObjectPAL provides error constants for these integers (for example, peObjectNotFound), so you don't have to remember numeric values. Use **enumRTLErrors** to create a list of error codes and error messages.

Calling **errorCode** is not the same as calling **eventInfo.setErrorCode**, which adds error information to the event packet, but not to the error stack.

For more information about the error stack and the event packet, refer to the *Guide to ObjectPAL* .

▪

errorCode example

Uses a **try** clause to attempt to attach to an object *boxOne* on the current form. If the object doesn't exist, a critical error occurs, and control moves to the **onFail** clause. The **onFail** clause uses **errorCode** to discover the error, then takes appropriate action.

```
; handleErrorcode::pushButton
method pushButton(var eventInfo Event)
var
  obj UIObject
endVar
try
  obj.attach("boxOne")
  obj.color = Red
onFail
  if errorCode() = peObjectNotFound then
     obj.create(BoxTool, 180, 180, 360, 360)
     obj.name = "boxOne"
     obj.visible = Yes
     reTry
  else
     fail()
  endIf
endTry
endMethod
```

▪

errorHasErrorCode method

See also Example System Type

Checks the error stack for a specific error code.

Syntax

```
errorHasErrorCode ( const errCode SmallInt ) Logical
```

Description

errorHasErrorCode searches the error stack for the error specified by *errCode*, which is an Errors constant or a user-defined error constant. This method returns True if the error is found; otherwise, returns False.

Use **enumRTLErrors** to create a list of error codes and error messages.

▪

errorHasErrorCode example

Searches the error stack for a reported key violation:

```
if errorHasErrorCode(peKeyViol) then

     ; error handling code goes here

endIf
```

▪

errorHasNativeErrorCode method

See also Example System Type

Checks the error stack for a SQL error code.

Syntax

```
errorHasNativeErrorCode ( const errCode LongInt ) Logical
```

Description

errorHasNativeErrorCode checks the error stack for an SQL error code. The argument *errCode* is the SQL error to search for. Error codes vary depending on the server and may overlap with some Paradox error codes. This method returns True if the error is found; otherwise, returns False.

errorHasNativeErrorCode example

Searches the error stack to check whether the server error associated with the constant peServerFailure, which is set to an error code listed in the server's documentation, exists in the error stack:

```
if errorHasNativeErrorCode(peServerFailure) then

     ; error handling code goes here

endIf
```

■

errorLog procedure

See also Example System Type

Adds error information to the error stack.

Syntax

```
errorLog ( const errorCode SmallInt, const errorMessage String )
```

Description

errorLog adds error information to the error stack. Use Errors constants or user-defined error constants to specify *errorCode*. Use **enumRTLErrors** to create a list of error codes and error messages.

Calling **errorLog** is not the same as calling **eventInfo.setErrorCode**, which adds error information to the event packet, but not to the error stack.

For more information about the error stack and the event packet, refer to the *Guide to ObjectPAL* .

▪

errorLog example

Uses a **try** clause to attempt to attach to an object *boxOne* to the current form. If the object doesn't exist, a critical error occurs, and control moves to the **onFail** clause. If the error code isn't peObjectNotFound, the method creates and logs a custom error.

```
; pushMessage::pushButton
method pushButton(var eventInfo Event)
var
  obj    UIObject
  eCode  LongInt
  eMsg   String
endVar
try
  obj.attach("boxOne")
  obj.color = "RedBlue"   ; invalid color constant--will cause an error
                          ; other than peObjectNotFound
onFail
  if errorCode() = peObjectNotFound then
    msgInfo("And the error was", errorMessage())
    obj.create(BoxTool, 180, 180, 360, 360)
    obj.name = "boxOne"
    obj.visible = Yes
    reTry
 else
   ; pop off the original error
   eCode = errorCode()
   eMsg = errorMessage()
   errorPop()
   ; push the original error back onto the stack, but
   ; modify the error message
   errorLog(eCode, self.Name + "::pushButton failed at " +
            String(time()) + ". " + eMsg)
   msgInfo("And the new error is", errorMessage())
   fail()
 endIf
endTry
endMethod
```

▪

errorMessage procedure

See also   Example   System Type
Beginner

Returns a string containing the error message displayed by the most recent run-time error or error condition from the error stack.

Syntax

`errorMessage ( )` `String`

Description

errorMessages returns a string containing the error message displayed by the most recent run-time error or error condition from the error stack. This method returns the empty string ("") if no error occurred. It is useful for logging error messages during a session.

-

errorMessage example

See the example for **errorLog**.

▪

errorNativeCode method

See also Example System Type

Returns the SQL server's error code.

Syntax

```
errorNativeCode ( ) LongInt
```

Description

errorNativeCode returns the SQL server's error code. The SQL server's error code varies by server and might overlap some Paradox error codes. Usually returns zero. If **errorCode** returns the constant peGeneralSQL, returns the server's error code; any error message would be from the server.

▪

errorNativeCode example

Checks whether a server error occurred, then displays the server error code (if any):

```
if errorCode() = peGeneralSQL then
   message("SQL server error number " + string(errorNativeCode()))
endIf
```

▪

errorPop procedure

See also Example System Type

Removes the top layer of information (that is, the most recently added error code and error message) from the error stack.

Syntax

```
errorPop ( ) Logical
```

Description

errorPop removes the top layer of information (that is, the most recently added error code and error message) from the error stack. It gives access to the stack layer below the current layer.

For more information about the error stack, refer to the *Guide to ObjectPAL*.

▪

errorPop example

See the example for **errorLog**.

▪

errorShow procedure

See also   Example   System Type

Opens the Error dialog box and displays the current error information.

Syntax

```
errorShow ( [ const topHelp String [ , const bottomHelp String ] ] ) Logical
```

Description

errorShow opens the Error dialog box and displays the current error information. The argument *topHelp* labels the top portion, and *bottomHelp* the bottom portion, of the dialog box.

For more information about the error stack, refer to the *Guide to ObjectPAL*.

errorShow example

The button *tryAnError* logs several errors onto the error stack; then **errorShow** displays them.

```
; tryAnError::pushButton
method pushButton(var eventInfo Event)
; add two errors to the error stack
errorLog(1, "First error")
errorLog(2, "Second error")
; show the error dialog box (error 2 shows first)
errorShow("Title for top", "Title for bottom")
endMethod
```

▪

errorTrapOnWarnings procedure

See also Example System Type
Beginner

Specifies whether to handle warning errors as critical errors.

Syntax

```
errorTrapOnWarnings ( const yesNo Logical )
```

Description

errorTrapOnWarnings specifies whether to handle warning errors as critical errors. By default, warning errors are not trapped in a **try...onFail** block. Setting the argument *yesNo* to Yes traps warning errors as critical errors.

errorTrapOnWarnings example

Attempts to open an invalid form. Does not generate an error If **errorTrapOnWarnings** is set to No (the default). Generates an error message when **errorTrapOnWarnings** is set to Yes.

```
; warningToError::pushButton
method pushButton(var eventInfo Event)
var
  someForm Form
endVar
someForm.open("someFile.fsl") ; attempt to attach to a nonexistent form
                              ; normally, this doesn't cause an error
errorTrapOnWarnings(Yes)      ; set the trap
someForm.open("someFile.fsl") ; this time, you get an error message
errorTrapOnWarnings(No)       ; restore to normal
endMethod
```

▪

execute procedure

See also Example System Type
Beginner

Executes a program or DOS command.

Syntax

```
execute ( const programName String [ , const wait Logical [ , const
displayMode SmallInt ] ] ) Logical
```

Description

execute executes a program or DOS command. The argument *programName* specifies the program or DOS command. The argument *wait* (optional) specifies whether ObjectPAL suspends execution until you close the program. *displayMode* (optional), one of the ExecuteOptions, specifies the video display mode to use when executing the command.

If the directory where *programName* resides is not in your path, you specify its path. Use double backslashes (\\) in path names.

▪

execute example

Launches the Windows Clock application with the default window style and waits for you to close it before resuming execution.

```
; showClock::pushButton
method pushButton(var eventInfo Event)
   execute("clock.exe", Yes, ExeShowNormal)  ; execute Windows Clock
endMethod
```

▪

executeString method

See also Example System Type

Converts a string into an ObjectPAL script and runs it.

Syntax

```
executeString ( const scriptText String [, const otherText String] ) AnyType
```

Description

executeString converts a string into an ObjectPAL script and runs it. This method inserts the string into the script's built-in **run** method. You need not include the **run** method header or footer. You can declare types, constants, and variables within the string. The optional *otherText* argument allows you to include ObjectPAL constructs, such as procedures or a Uses clause. The *otherText* argument refers to constructs included before the script's built-in **run** method.

To return a value from **executeString**, use **formReturn**.

If the string contains syntax errors, the Script window is left on the Desktop.

▪

executeString example

This example calls a routine from Windows and runs it.

```
method run(var eventInfo Event)
var
   msgText, usesText  string
endvar

msgText = "MessageBoxA(0,
                    \"A Message\",      ; Note the backslash char
                    \"Hello World\,"    ; protects quotes inside
                    1)"                 ; the quoted string

usesText = "Uses USER32
      MessageBoxA(hwnd CLONG,
                  str1 CPTR,
                  str2 CPTR,
                  boxType CLONG) CLONG
         endUses"
   ;// Now display the message box
executeString(scriptText, usesText)
endMethod
```

▪

exit procedure

See also　　Example　　System Type
Beginner

Closes Paradox and exits to Windows.

Syntax
```
exit ( )
```

Description
exit prompts you to save a Paradox application that has changed.

▪

exit example

Creates an Exit button you click to close Paradox that asks for confirmation before closing.

```
; btnExit::pushButton
method pushButton(var eventInfo Event)
   var
      stQuit  String
   endVar

   stQuit  = msgYesNoCancel("Exit", "Do you want to quit?")
   if stQuit = "Yes" then
      exit() ; If user chooses Yes, then exit.
   endIf
endMethod
```

▪

fail procedure

See also Example System Type

Causes a method to fail.

Syntax

```
fail ( [ const errorNumber SmallInt, const errorMessage String ] )
```

Description

Executing **fail** in the **onFail** section of a **try...onFail** block forces a jump to the next highest block, if one exists; then to the implicit **try...onFail** block that ObjectPAL wraps around every method. Use an Errors constant or a user-defined error constant to specify a value for *errorNumber*, which specifies an error code on failure. *errorMessage* (optional) specifies a displayed error message.

enumRTLErrors creates a list of error codes and error messages.

-

fail example

See the example for **errorCode.**

▪

fileBrowser procedure

See also Example1 Example2 System Type

Displays the Paradox File Browser and returns the names of any file(s) you select.

Syntax

```
1. fileBrowser ( var selectedFile String [ , var browserInfo
FileBrowserInfo ] ) Logical
2. fileBrowser ( var selectedFiles Array[ ] String [ , var browserInfo
FileBrowserInfo ] ) Logical
```

Description

fileBrowser suspends ObjectPAL execution until you close the Browser. This method returns True if you select at least one file; otherwise, returns False (even if you click OK to close the dialog box).

Use syntax 1 to return one file name in *selectedFile.* Use syntax 2 to return an array of file names in the resizeable array *selectedFiles*.

For either syntax, you can provide an optional record that specifies the data that the Browser displays. For example, you can make the Browser display Paradox tables only, forms only, forms and reports, and so on. ObjectPAL provides a special predefined record called FileBrowserInfo that you use only with the **fileBrowser** procedure. FileBrowserInfo is an instance of the Record Type with the following structure:

```
TYPE FileBrowserInfo =
  Record
    Title          String      ; title on the TitleBar
    Options        LongInt     ; enables the options
    AllowableTypes LongInt     ; Use FileBrowserFileTypes constants
    SelectedType   LongInt     ; one of the AllowableTypes
    FileFilters    String      ; the filespec in edit box
    CustomFilters  String      ; customized filespec
    Alias          String      ; alias or drive name
    Path           String      ; path relative to Alias
    Drive          String      ; designates the drive
    DefaultExt     String      ; default extension
    PathOnly       Logical     ; return path only, no file name
    NewFileOnly    Logical     ; bring up "Save As" dialog, if True
  endRecord
ENDTYPE
```

This record structure is predefined and built into ObjectPAL, so you just declare a variable of type FileBrowserInfo and assign values to its fields.

After the call to **fileBrowser**, the Alias, Path, and FileFilters fields are filled in with the values that were in the File Browser dialog box. In this way you can find out what you entered into the Browser.

The AllowableTypes field specifies what appears in the drop-down edit list for the Types panel in the File Browser. The SelectedType field indicates which of the AllowableTypes is currently selected. Use FileBrowserFileTypes constants for values in the SelectedType and AllowableTypes fields.

The **fileBrowser** procedure looks only at the field names in the structure. You can pass to it a different, simpler record structure that contains only the fields you are interested in, and it finds only those fields.

▪

fileBrowser example 1

Calls **fileBrowser** twice: the first time, it returns one file name; if that is a table name, it opens a Table window. The second time, it returns an array of file names (selected by Shift-clicking) and displays the array in a dialog box.

```
; fileBrowserButton::pushButton
method pushButton(var eventInfo Event)
var
   oneFile           String
   manyFiles Array[] String
   tView             TableView
endVar
fileBrowser(oneFile)    ; display the File Browser, and wait
                        ; for you to choose one file
                        ; variable oneFile stores the file name chosen
if isTable(oneFile) then
   tView.open(oneFile) ; open a Table window for the chosen file
endIf

fileBrowser(manyFiles) ; let you select multiple files and store
                        ; the file names in an array
manyFiles.view()        ; displays your choices
endMethod
```

▪

fileBrowser example 2

Uses a FileBrowserInfo record to pass information. Attach the following code to a button's built-in **pushButton** method. When it executes, it displays the Browser and waits for you to choose a file. Then, it displays information about your choice in the status area.

```
method pushButton(var eventInfo Event)

var
   fbi FileBrowserInfo ; Declare a variable that uses the predefined
                       ; FileBrowserInfo record structure
   selectedFile String
endVar
```

The following statements assign values to fields in the record of file browser information

```
fbi.Alias = ":WORK:" ; Search the current working directory
fbi.AllowableTypes = fbTable + fbForm ; Search for tables and forms
fbi.CustomFilter = "(Bitmap image) *.bmp|*.bmp|(Other graphics files)
*.jpg;*.pcx|*.jpg;*.pcx||"

; Display the Browser and process your selection
if fileBrowser(selectedFile, fbi) then
   message("You selected ", selectedFile)
else
   message("You selected cancel")
endIf

endMethod
```

▪

formatAdd procedure

See also Example System Type

Adds a format.

Syntax

`formatAdd ( const formatName String, const formatSpec String ) Logical`

Description

formatAdd adds a format. It creates the format *formatName* described by *formatSpec,* which is available to the current session. This method returns True if successful; otherwise, returns False.

Note: Does not save Field width (W*n*), Alignment (AR, AL, AC), and Case specifiers (CU, CL, CC) with a new format definition; does, however, save decimal precision (W.*n*). See **format** in the String type for a complete description of format specifiers.

formatAdd example

Adds a new format specification to the session, then sets the default Currency format to the new format.

```
; addAFormat::pushButton
method pushButton(var eventInfo Event)
var
  someNum Currency
endVar
; first, add a currency format with 4 decimal digits and
; a floating dollar sign (windows dollar sign)
formatAdd("FourCurrency", "W.4, E$W")
; then, set the default format for Currency to the new format
formatSetCurrencyDefault("FourCurrency")
someNum = 41324.09876
someNum.view()                   ; appears as $41,324.0988
endMethod
```

formatDelete procedure

See also Example System Type

Deletes a format.

Syntax

```
formatDelete ( const formatName String ) Logical
```

Description

formatDelete deletes the format specified with the argument *formatName* from the current session.

▪

formatDelete example

Deletes the custom format named *FourCurrency*, if it exists.

```
; deleteAFormat::pushButton
method pushButton(var eventInfo Event)
if formatExist("FourCurrency") then
  formatDelete("FourCurrency")
else
  msgInfo("FYI", "Format was not found.")
endIf
endMethod
```

▪

formatExist procedure

See also Example System Type

Reports whether a format exists.

Syntax

`formatExist` `( const` ***`formatName`*** `String` **`)`** `Logical`

Description

formatExist checks whether the format *formatName* is available for the current session. Returns True if the format is available; otherwise, returns False.

▪

formatExist example

Checks whether a custom format *FourCurrency* exists; if not, adds the format and displays a number formatted as *FourCurrency*.

```
; addCurrFormatExist::pushButton
method pushButton(var eventInfo Event)
var
  someNum Currency
endVar
; check if custom format exists already
if NOT formatExist("FourCurrency") then
  ; if not, add a currency format with 4 decimal digits and
  ; a floating dollar sign (windows dollar sign)
  msgInfo("FYI", "Format does not exist. Adding it now.")
  formatAdd("FourCurrency", "W.4, E$W")
else
  msgInfo("FYI", "Format already exists.")
endIf
; set the default format for Currency to the new format
formatSetCurrencyDefault("FourCurrency")
someNum = 41324.09876
someNum.view()        ; displays number as $41324.0988, because
                      ; someNum is a variable of Currency type
endMethod
```

▪

formatGetSpec procedure

See also Example System Type

Returns the format specification for a named format.

Syntax

`formatGetSpec ( const formatName String ) String`

Description

formatGetSpec returns the format specification for a named format. The argument *formatName* specifies the format whose specification is returned. The return value can be passed to **formatStringToDate** and **formatStringToNumber** to format a string into a date or number.

■

formatGetSpec example

formatGetSpec and **formatStringToDate** assign a date to a variable of type Date in the Windows Long format, and display the new value.

```
;Btn :: pushButton
method pushButton(var eventInfo Event)
   var
      d  Date
   endVar

   d = formatStringToDate("Friday, January 08, 1965", formatGetSpec("Windows
Long"))
   d.view()
endMethod
```

▪

formatSetCurrencyDefault procedure

See also Example System Type

Sets the default display format for Currency values.

Syntax

formatSetCurrencyDefault (const ***formatName*** String **)** Logical

Description

formatSetCurrencyDefault sets the default display format for Currency values. This setting remains in effect for the duration of the session.

▪

formatSetCurrencyDefault example

See the example for **formatExist**.

▪

formatSetDateDefault procedure

See also Example System Type

Sets the default display format for Date values.

Syntax

```
formatSetDateDefault ( const formatName String ) Logical
```

Description

formatSetDateDefault sets the default display format for Date values.This setting remains in effect for the duration of the session.

formatSetDateDefault example

The **pushButton** method for the button *setDateFormat* sets the default display format for Date values to the Windows Long format, then displays a date in the new format.

```
; setDateFormat::pushButton
method pushButton(var eventInfo Event)
var
  someDate Date
endVar
if formatExist("Windows Long") then
  formatSetDateDefault("Windows Long")
  someDate = date("9/15/92")
  someDate.view()              ; displays "Tuesday, September 15, 1992"
else
  msgStop("Stop", "Requested format does not exist.")
endIf
endMethod
```

▪

formatSetDateTimeDefault procedure

See also Example System Type

Sets the default display format for DateTime values.

Syntax

formatSetDateTimeDefault (const ***formatName*** String **)** Logical

Description

formatSetDateTimeDefault sets the default display format for DateTIme values. This setting remains in effect for the duration of the session.

▪

formatSetDateTimeDefault example

The **pushButton** method for the button *setDateTimeFormat* sets the default display format for DateTime values, then uses **view** to display a DateTime value in the new format.

```
setDateTimeFormat::pushButton
method pushButton(var eventInfo Event)
var
  someDateTime DateTime
endVar
if formatExist("h:m:s am m/d/y") then
  formatSetDateTimeDefault("h:m:s am m/d/y")
  someDateTime = DateTime("11:45:25 am 11/24/61")
  someDateTime.view()                   ; displays 11:45:25 AM 11/24/61
else
  msgInfo("Status", "Requested format does not exist.")
endIf
endMethod
```

▪

formatSetLogicalDefault procedure

See also Example System Type

Sets the default display format for Logical values.

Syntax

```
formatSetLogicalDefault ( const formatName String ) Logical
```

Description

formatSetLogicalDefault sets the default display format for Logical values. This setting remains in effect for the duration of the session.

formatSetLogicalDefault example

The **pushButton** method for the button *setLogicalFormat* sets the default display format for Logical values to the Male/Female format, then displays a logical value in the new format.

```
; setLogicalFormat::pushButton
method pushButton(var eventInfo Event)
var
  someLogical  Logical
endVar
if formatExist("Male/Female") then
  formatSetLogicalDefault("Male/Female")
  someLogical = True
  someLogical.view()                ; displays Male
else
  msgStop("Stop", "Requested format does not exist.")
endIf
endMethod
```

▪

formatSetLongIntDefault procedure

See also Example System Type

Sets the default display format for LongInt values.

Syntax

`formatSetLongIntDefault ( const formatName String ) Logical`

Description

formatSetLongIntDefault sets the default display format for LongInt values. This setting remains in effect for the duration of the session.

■

formatSetLongIntDefault example

The **pushButton** method for the button *setIntegerFormat* sets the default display format for LongInt values to the Integer format, then displays a long integer in the new format.

```
; setIntegerFormat::pushButton
method pushButton(var eventInfo Event)
var
  someInt  LongInt
endVar
if formatExist("Integer") then
  formatSetLongIntDefault("Integer")
  someInt = 238756
  someInt.view()                        ; displays 238756
else
  msgStop("Stop", "Requested format does not exist.")
endIf
endMethod
```

▪

formatSetNumberDefault procedure

See also Example System Type

Sets the default display format for Number values.

Syntax

`formatSetNumberDefault ( const formatName String ) Logical`

Description

formatSetNumberDefault sets the default display format for Number values. This setting remains in effect for the duration of the session.

formatSetNumberDefault example

The **pushButton** method for the button *setNumberFormat* sets the default display format for Number values to the Scientific format; then displays a number in the new default format.

```
; setNumberFormat::pushButton
method pushButton(var eventInfo Event)
var
  someNum Number
endVar
if formatExist("Scientific") then
  formatSetNumberDefault("Scientific")
  someNum = 3489.283
  someNum.view()             ; Displays 3.489283e+3.
else
  msgStop("Stop", "Requested format does not exist.")
endIf
endMethod
```

▪

formatSetSmallIntDefault procedure

See also   Example   System Type

Sets the default display format for SmallInt values.

Syntax

`formatSetSmallIntDefault ( ``const `***`formatName`***` String `**`)`**` Logical`

Description

formatSetSmallIntDefault sets the default display format for SmallInt values. This setting remains in effect for the duration of the session.

formatSetSmallIntDefault example

The **pushButton** method for the button *setSmallIntFormat* sets the default display format for SmallInt values to the Integer format; then displays a small integer in the new default format.

```
; setSmallIntFormat::pushButton
method pushButton(var eventInfo Event)
var
  someInt  SmallInt
endVar
if formatExist("Integer") then
  formatSetSmallIntDefault("Integer")
  someInt = 324
  someInt.view()                     ; displays 324
else
  msgStop("Stop", "Requested format does not exist.")
endIf
endMethod
```

▪

formatSetTimeDefault procedure

See also Example System Type

Sets the default display formatfor Time values.

Syntax

formatSetTimeDefault (const ***formatName*** String **)** Logical

Description

formatSetTimeDefault sets the default display format for Time values. This setting remains in effect for the duration of the session.

formatSetTimeDefault example

The **pushButton** method for the button *setTimeFormat* sets the default display format for Time values to the format *hh:mm:ss am*; then displays a time in the new default format.

```
; setTimeFormat::pushButton
method pushButton(var eventInfo Event)
var
  someTime Time
  someStr  String
endVar
if formatExist("hh:mm:ss am") then
  formatSetTimeDefault("hh:mm:ss am")
  someTime = time("12:22:45 pm")
  someTime.view()                    ; displays 12:22:45 PM
else
  msgInfo("Status", "Requested format does not exist.")
endIf
endMethod
```

■

formatStringToDate procedure

See also Example System Type

Uses a format specification to translate a String value to a Date value.

Syntax

formatStringToDate (*dateString* String, ***formatSpec*** String **)** Date

Description

formatStringToDate uses a format specification to translate a String value to a Date value. It translates *dateString*, a string value that represents a date, to a value of type Date, according to the format specification in *formatSpec*. This method returns the Date value and leaves the String value unchanged.

formatSpec must be the format specification of a named format; it cannot be the format name itself. To get the format specification of a named format, use **formatGetSpec**.

formatStringToDate example

Sometimes you need to convert an invalid date stored in a string to a valid date. This example gets a String value that you enter and formats it as a valid date, if possible. The code is attached to an Alpha field's built-in **changeValue** method; it executes when you type a value and then leave the field (for example, by pressing Enter).

If this field object were bound to a Date field (instead of an Alpha field), you could let Paradox validate the date without writing ObjectPAL code.

```
method changeValue(var eventInfo ValueEvent)
   var
      stUserDate   String
      daValidDate  Date
   endVar

   doDefault

   ; Assume user enters "09-94-23" into this Alpha field object.
   stUserDate = self.Value

   try
      ; Format your value as a valid date.
      daValidDate = formatStringToDate(stUserDate, "DO(%M-%Y-%D)")

      ; formatStringToDate does not change the String value.
      ; It returns a Date value. The following statement displays
      ;         You entered: 09-94-23
      ;         Valid date: 09/23/94

      msgInfo("You entered: " + stUserDate,
            "Valid date: " + String(daValidDate))

   onFail
      ; If user's value cannot be formatted as a date,
      ; display a message.
      msgStop(stUserDate, "Cannot format that value as a Date.")
   endTry

endMethod
```

▪

formatStringToDateTime method

[See also](#) [Example](#) [System Type](#)

Translates a String value to a DateTime value.

Syntax

formatStringToDateTime (const ***dateTimeString*** String, const ***formatSpec*** String **)** DateTime

Description

formatStringToDateTime translates *dateTimeString* to a DateTime value, using the format specification in **formatSpec**. If successful, **formatStringToDateTime** returns a DateTime value and leaves the *dateTimeString* value unchanged. The value of *formatSpec* must be the format specification of a named format; it cannot be the format name. To get the format specification of a named format, use **formatGetSpec**.

■

formatStringToDateTime example

The following example converts the specified string to the DateTime data type and displays it.

```
view( formatStringToDateTime( "23:59:59, 3/23/99", "TH1O(%H:%M:%S, %D)" ) )
```

▪

formatStringToNumber procedure

See also Example System Type

Uses a format specification to translate a String value to a Number value.

Syntax

`formatStringToNumber ( `*`numberString`*` String, `*`formatSpec`*` String ) Number`

Description

formatStringToNumber translates *numberString*, a string value that represents a number, to a Number value, using the format specification in *formatSpec*. This procedure returns the Number value and leaves the String value unchanged.

The value of *formatSpec* must be the format specification of a named format; it cannot be the format name itself. To get the format specification of a named format, use **formatGetSpec**.

▪

formatStringToNumber example

In the following example, two strings are concatenated to form a number in scientific notation format. Then **formatStringToNumber** is used to assign the value to a Number variable. Finally, a dialog box displays the formatted and unformatted values. Note that the value of the String variable is unchanged; the formatted value is assigned to a Number variable.

```
;btnScientific :: pushButton
method pushButton(var eventInfo Event)
   var
      st1,
      st2,
      stSciNot  String
      nuResult  Number
   endVar

   st1 = "1.e"
   st2 = "+2"
   stSciNot = st1 + st2
   nuResult = formatStringToNumber(stSciNot, "S-4")

   ; The following statement displays
   ; Before format: 1.e+2
   ; After format: 100.00
   msgInfo("Before format: " + stSciNot,
          "After format: " + String(nuResult))
endMethod
```

▪

formatStringToTime method

See also Example System Type

Translates a String value to a Time value.

Syntax

formatStringToTime (const ***timeString*** String, const ***formatSpec*** String **)** Time

Description

formatStringToTime translates *timeString* to a Time value, using the format specification in *formatSpec*. If successful, **formatStringToTime** returns a Time value and leaves the String value unchanged. The value of *formatSpec* must be the format specification of a named format; it cannot be the format name. To get the format specification of a named format, use **formatGetSpec**.

-

formatStringToTime example

The following example will convert the specified string to the Time data type and display it.

```
view( formatStringToTime( "23:59:59", "TH1O(%H:%M:%S)" ) )
```

▪

getDefaultPrinterStyleSheet procedure

See also Example System Type

Returns the name of the default printer style sheet used by documents designed for the printer.

Syntax

```
getDefaultPrinterStyleSheet ( ) String
```

Description

getDefaultPrinterStyleSheet returns the name of the default printer style sheet used by documents designed for the printer. If the style sheet is in :WORK: (the working directory), returns the file name and extension, if any (for example, BORLAND.FP). Otherwise, returns the full path (for example, C:\PDOXWIN\BORLAND.FP).

Individual forms and reports may use different style sheets. Use **getStyleSheet** and **setStyleSheet** to work with style sheets for specific forms and reports.

Use **getDefaultScreenStyleSheet** to get the name of the default screen style sheet, used whenever you create design documents that are designed for the screen.

▪

getDefaultPrinterStyleSheet example

See the example for **setDefaultPrinterStyleSheet.**

getDefaultScreenStyleSheet procedure

See also Example System Type

Returns the name of the default screen style sheet used by design documents that are designed for the screen.

Syntax

```
getDefaultScreenStyleSheet ( ) String
```

Description

getDefaultScreenStyleSheet returns the file name of the default style sheet for screen documents. If the style sheet is in :WORK: (the working directory), returns the file name and extension, if any (for example, BORLAND.FT). Otherwise, returns the full path (for example, C:\PDOXWIN\BORLAND.FT).

Individual forms and reports may use different style sheets. Use **getStyleSheet** and **setStyleSheet** to work with style sheets for specific forms and reports.

Use **getDefaultPrinterStyleSheet** to get the name of the default printer style sheet, used whenever you create design documents that are designed for the printer.

▪

getDefaultScreenStyleSheet example

See the example for **setDefaultScreenStyleSheet.**

▪

getDesktopPreference procedure

See also Example System Type

Gets a desktop preference.

Syntax

```
getDesktopPreference (const section AnyType, const name AnyType) AnyType
```

Description

getDesktopPreference returns the value of the desktop preference specified with the *section* and *name* arguments. The *value* returned corresponds to one of the DesktopPreferenceTypes Constants.

getDesktopPreference example

Displays the sets the title name preference, then gets the name and displays it.

```
method pushButton(var eventInfo Event)
setDesktopPreference( PrefProjectSection, prefTitleName,"Paradox pour
Windows" )

x = getDesktopPreference( PrefProjectSection, prefTitleName )

x.view()
endmethod
```

getLanguageDriver procedure

See also Example System Type

Returns the name of the default language driver for the system.

Syntax

```
getLanguageDriver ( ) String
```

Description

getLanguageDriver returns the name of the default language driver for the system.

getLanguageDriver example

Displays the name of the system language driver on the status bar.

```
;btnDefaultDriver :: pushButton
method pushButton(var eventInfo Event)
  message(getLanguageDriver())
endmethod
```

▪

getMouseScreenPosition procedure

See also Example System Type

Returns the mouse position as a Point data type.

Syntax

```
getMouseScreenPosition ( ) Point
```

Description

getMouseScreenPosition returns the coordinates (in twips) of the pointer relative to the screen (not to the Desktop). Use Point type methods (for example **x** and **y**) to get more information.

This method gets the mouse position at the time of an event; the current mouse position may be different.

▪

getMouseScreenPosition example

The mouse jumps one inch down and one inch to the left when you click the *nervousMouse* button.

```
; nervousMouse::pushButton
method pushButton(var eventInfo Event)
var
  mouseP,
  newMouseP Point
endVar
mouseP = getMouseScreenPosition()
newMouseP = mouseP + Point(1440, 1440)
setMouseScreenPosition(newMouseP)    ; move mouse pointer 1 inch down and
                                     ; 1 inch to the right
endMethod
```

▪

getRegistryValue method

See also Example System Type

Gets a value from the registry.

Syntax

getRegistryValue (const ***key*** String, const ***value*** String , const ***rootKey*** LongInt **)** AnyType

Description

getRegistryValue retrieves data from a specified *key* and *value* in the registry. The registry value is returned as an AnyType if **getRegistryValue** is successful. If not successful, it returns an empty string.

key is entered as a path similar to a file path. However, unlike a file path, wildcards are not expanded. *key* cannot contain a single backslash and cannot be empty. The size of *key* is limited to 65,534 bytes. The *value* is a string that is limited to 65,534 bytes. *value* can contain backslashes and can be empty. *rootKey* is analogous to a directory drive. The *rootKey* should be set with the predefined constants:

regKeyCurrentUser

regKeyClassesRoot

regKeyLocalMachine

regKeyUser

getRegistryValue example

The following example gets the current ObjectPAL Level from the registry and displays it.

```
var
   strLevel   String
endvar

   strLevel = getRegistryValue( "Software\\Borland\\Paradox\\7.0\\Pdoxwin\
\Properties", "Level",
      RegKeyCurrentUser )
   strLevel.view()
```

getUserLevel procedure

Example System Type

Returns your ObjectPAL level property setting, either Advanced or Beginner.

Syntax

```
getUserLevel ( ) String
```

Description

Use **setUserLevel** to change this setting.

Note: The ObjectPAL level property setting *does not* affect how code executes; it only affects the ObjectPAL language elements that are displayed in the user interface.

▪

getUserLevel example

See the example for **setUserLevel.**

▪

helpOnHelp procedure

See also Example System Type

Displays information about how to use the Windows Help system, and opens Windows Help if necessary.

Syntax

helpOnHelp ( ) Logical

Description

By default, the **helpOnHelp** opens the WINHLP32.HLP file that comes with Windows. To specify another Help file to use instead of WINHLP32.HLP:

1. Use a text editor to open the Help project file.
2. Add a SetHelpOnFile macro to the [CONFIG] section, specifying the Help file you want to use for How to Use Help.
3. Compile the Help file.

For example, the following macro placed in the [CONFIG] section of the Help project file changes the Help file from WINHELP32.HLP (the default) to HOWHELP.HLP:

```
[CONFIG]
SetHelpOnFile("howhelp.hlp")
```

helpOnHelp example

Opens a Help file when you choose Help|Help On Help from a custom menu (not the built-in Paradox menu).

```
method menuAction(var eventInfo MenuEvent)
   var
      siMenuChoice SmallInt
   endVar

   siMenuChoice = eventInfo.id()

   switch
      case siMenuChoice = UserMenu + MenuHelpOnHelp :
           helpOnHelp()
      ; Handle other cases here
   endSwitch

endmethod
```

▪

helpQuit procedure

See also Example System Type

Notifies the Help application that it is no longer needed by the current application.

Syntax

helpQuit (const ***helpFileName*** String **)** Logical

Description

helpQuit notifies the Windows Help application (WINHELP.EXE) that the Help file *helpFileName* is no longer needed by the current Paradox application. If the directory where *helpFileName* resides is not in your path, you specify its full path. If no other applications require the Help application, Windows closes it.

helpQuit example

Executes when you choose an item from a custom menu (not the built-in Paradox menu). If you choose File|Close Form, notifies the Help application that it is no longer needed; then closes the current form.

```
method menuAction(var eventInfo MenuEvent)
   const
      ; Typically, menu choice constants are defined elsewhere,
      ; with the rest of the menu-building code. The following
      ; constant is defined here so the example will compile.
      kMyMenuFileCloseForm = 104
   endConst

   var
      siMenuChoice   SmallInt
      stHelpFileName String
   endVar

   siMenuChoice = eventInfo.id()
   stHelpFileName = "c:\\pdoxapps\\ordentry\\ordentry.hlp"

   switch
      case siMenuChoice = UserMenu + kMyMenuFileCloseForm :
           helpQuit(stHelpFileName) ; Tell Help we don't need it any more.
           close() ; Close the form.
   ; Handle other cases here
   endSwitch

endMethod
```

▪

helpSetIndex procedure

See also Example System Type

Sets the Help contents topic (index).

Syntax

helpSetIndex (**const ***helpFileName* String, const ***indexId*** LongInt **)** Logical

Description

helpSetIndex sets the Help contents topic (index). It instructs the Windows Help application (WINHELP.EXE) to set the current Contents topic (called *index* in early versions of Windows) to the topic in *helpFileName* specified by *indexID*. If the directory where *helpFileName* resides is not in your path, you specify its full path.

When you open a Help file, WinHelp displays the Contents topic by default. When you create a Help file, you specify the Contents topic using the Contents option in the [CONFIG] section of the Help project file. For example, the following SetContents macro placed in the project file's [CONFIG] section sets the Contents topic for a help file to topic number 100 in the file CWH.HLP.

```
[CONFIG]
SetContents("cwh.hlp", 100)
```

If you do not use the SetContents option, the Contents topic is the first topic in the first file listed in the [FILES] section.

You can use **helpSetIndex** to specify a Contents topic from within an application.

helpSetIndex example

The following is a custom method that sets the Contents topic for a Help file to the topic in the file ORDENTRY.HLP with the context number 100.

```
method setHelpContents() Logical
   return helpSetIndex("c:\\pdoxapps\\ordentry\\ordentry.hlp", 100)
endMethod
```

▪

helpShowContext procedure

See also Example System Type

Displays the help topic in *helpFileName* specified by *helpId*.

Syntax

`helpShowContext ( const helpFileName String, const helpId LongInt ) Logical`

Description

helpShowContext instructs the Windows Help application to search *helpFileName* for the topic identified by the *helpId*; and to display the topic. If the directory where *helpFileName* resides is not in your path, you specify its full path.

In a Help source file, each topic is identified by a context ID, a string defined using a # footnote. The context ID is mapped to an integer value in the [MAP] section of the Help project (.HPJ) file. **helpShowContext** uses this integer value to locate the help topic.

helpShowContext example

Instructs the Windows Help application to display context-sensitive help for the object in a form that is active. (Assume that the form contains three buttons and two field objects.) The code is attached to a button whose TabStop property is set to False (otherwise, the button would become active when clicked).

```
helpButton::pushButton
const
; These integer values must also be listed
; in the [MAP] section of the Help project file.
   kNewOrdBtn    = LongInt(1020)
   kEditOrdBtn   = LongInt(1021)
   kDelOrdBtn    = LongInt(1022)
   kCustNameFld = LongInt(2020)
   kOrderNoFld   = LongInt (2021)
endConst

method pushButton(var eventInfo Event)

var
   stObjName,
   stHelpFileName String
   liContextId    LongInt
endVar

   stObjName = active.name ; Get the name of the active object.
   stHelpFileName = "c:\\pdoxapps\\ordentry\\ordentry.hlp"

   switch
      case stObjName = "newOrdBtn"     : liContextId = kNewOrdBtn
      case stObjName = "editOrdBtn"    : liContextId = kEditOrdBtn
      case stObjName = "delOrdBtn"     : liContextId = kDelOrdBtn
      case stObjName = "custNameFld"   : liContextId = kCustNameFld
      case stObjName = "orderNoFld"    : liContextId = kOrderNoFld
   endSwitch

   if not helpShowContext(stHelpFileName, liContextId) then
      errorShow("Could not display Help topic.")
   endIf

endMethod
```

▪

helpShowIndex procedure

See also Example System Type

Displays the index (contents topic) of a specified Help file.

Syntax

```
helpShowIndex ( const helpFileName String ) Logical
```

Description

helpShowIndex instructs the Windows Help application (WINHELP.EXE) to display the Contents topic (called *index* in early versions of Windows) in the Help file specified by *helpFileName*. If the directory where *helpFileName* resides is not on your path, you specify its full path.

When you open a Help file, WinHelp displays the Contents topic by default. When you create a Help file, you specify the Contents topic using the Contents option in the [CONFIG] section of the Help project file. For example, the following SetContents macro in the [CONFIG] section sets the Contents topic for a help file to the topic in the file CWH.HLP with the context number 100.

```
[CONFIG]
SetContents("cwh.hlp", 100)
```

If you do not use the Contents option, the Contents topic is the first topic in the first file listed in the [FILES] section.

helpShowIndex example

Executes when you choose an item from a custom menu (not the built-in Paradox menu). If you choose Help|Contents, instructs the Help application to display the Contents topic for the specified Help file.

```
method menuAction(var eventInfo MenuEvent)
   const
      ; Typically, menu choice constants are defined elsewhere,
      ; with the rest of the menu-building code. The following
      ; constant is defined here so the example will compile.
      kMyMenuHelpContents = 501
   endConst

   var
      siMenuChoice   SmallInt
      stHelpFileName String
   endVar

   siMenuChoice = eventInfo.id()
   stHelpFileName = "c:\\pdoxapps\\ordentry\\ordentry.hlp"

   switch
      case siMenuChoice = UserMenu + kMyMenuHelpContents :
           helpShowIndex(stHelpFileName) ; Display the Contents topic.
      ; Handle other cases here
   endSwitch

endMethod
```

▪

helpShowTopic procedure

See also Example System Type

Displays help for a specified context.

Syntax

helpShowTopic (const ***helpFileName*** String, const ***topicKey*** String **)** Logical

Description

helpShowTopic instructs the Windows Help application to search the file *helpFileName* for the topic associated with *topicKey*, and to display the topic. If the directory where *helpFileName* resides is not on your path, you specify its full path. *topicKey* must match a keyword defined using a K footnote in the Help source file; otherwise, the search fails and the Windows Help application displays an error message.

helpShowTopic example

Prompts you for a word or phrase, then searches for the text in the specified Help file.

```
method pushButton(var eventInfo Event)
   var
      stHelpFileName,
      stTopicKey,
      stPromptText    String
   endVar

   stHelpFileName = "c:\\pdoxapps\\ordentry\\ordEntry.hlp"
   stPromptText   = "Enter a word or phrase here."
   stTopicKey     = stPromptText

   stTopicKey.view("Enter text to search for.")
   if stTopicKey <> stPromptText then
      helpShowTopic(stHelpFileName, stTopicKey)
   endIf
endMethod
```

helpShowTopicInKeywordTable procedure

See also Example System Type

Displays help for a topic identified by a keyword in an alternate keyword table.

Syntax

helpShowTopicInKeywordTable (const ***helpFileName*** String, const ***keyTableLetter*** String, const ***topicKey*** String **)** Logical

Description

helpShowTopicInKeywordTable instructs the Windows Help application to search the file *helpFileName* for the topic associated with *keyTableLetter* and *topicKey*, and to display the topic. If the directory where *helpFileName* resides is not in your path, you specify its full path. The value of *keyTableLetter* must match a multikey index specified in the [OPTIONS] section of the Help project file. For example, if a Help project file included the following lines, you would assign a value of "L" to *keyTableLetter*.

```
[OPTIONS]
MULTIKEY=L
```

The value of *topicKey* must match a keyword defined using a multikey index footnote (the L in the previous example) in the Help source file. Otherwise, the search fails and the Windows Help application displays an error message.

helpShowTopicInKeywordTable example

Prompts you to enter either PARADOX or dBASE, then searches for the text "field types" in the specified keyword table of the specified Help file. Assume that an application is handling a user's request for help on the topic "field types."

```
method pushButton(var eventInfo Event)
   var
      stHelpFileName,
      stPromptText,
      stUserChoice,
      stTopicKey,
      stKeyTableLetter    String
   endVar

   stHelpFileName   = "c:\\pdoxapps\\ordentry\\ordEntry.hlp"
   stPromptText     = "Enter PARADOX or dBASE here."
   stUserChoice     = stPromptText
   stTopicKey       = "field types"

   stUserChoice.view("Do you want Paradox Help or dBASE Help?")
   if stUserChoice <> stPromptText then
      switch
         case stUserChoice = "PARADOX" : stKeyTableLetter = "P"
         case stUserChoice = "dBASE"   : stKeyTableLetter = "D"
         otherwise : return
      endSwitch

      helpShowTopicInKeywordTable(stHelpFileName, stKeyTableLetter,
stTopicKey)
   endIf
endMethod
```

▪

isErrorTrapOnWarnings procedure

See also Example System Type

Reports whether this session is handling warning errors as critical errors.

Syntax

```
isErrorTrapOnWarnings ( ) Logical
```

Description

isErrorTrapOnWarnings reports whether this session is handling warning errors as critical errors. This method returns True if this session is handling warning errors as critical errors; otherwise, returns False.

isErrorTrapOnWarnings example

The **pushButton** method for *btnToggleWarning* toggles between warning errors being treated as critical, and not being treated as critical.

```
; btnToggleWarning :: pushButton
method pushButton(var eventInfo Event)
  errorTrapOnWarnings(not isErrorTrapOnWarnings())
  msgInfo("Warning errors are critical", isErrorTrapOnWarnings())
endmethod
```

▪

isMousePersistent method

See also Example System Type

Reports if mouse persistence is on.

Syntax

`isMousePersistent ( ) Logical`

Description

isMousePersistent reports if mouse persistence in set to on. **isMousePersistent** returns True, if mouse persistence is on and False if mouse persistence on off. To set mouse persistence, use **setMouseShape** or **setMouseShapeFromFile**.

▪

isMousePersistent example

In the following example, a form has two buttons: btnNonPersistent and btnPersistent. The pushButton method of each button uses setMouseShape() to set the mouse shape of the cursor; the first with persistence set to false, the second with persistence set to true. The second button, btnPersistent also contains a mouseEnter method which will use isMousePersistent() to evaluate the persistency of the mouse cursor and revert it to its original state. When the first button is pressed, the mouse cursor is changed. However, when the mouse cursor is moved off the button, the mouse cursor will revert to its original setting. When the second button is pressed, the mouse cursor is changed and will remain unchanged until the mouse cursor is moved back over the second button. This will trigger the mouseEnter method of the second button and revert the mouse cursor back to its original state.

The following code is attached to the pushButton method for btnNonPersistent:

```
; btnNonPersistent::pushButton
method pushButton(var eventInfo Event)
   ;// Set the shape to international symbol for No - non-persistent
   setMouseShape(MouseNo,FALSE)
endMethod
```

The following code is attached to the pushButton method for btnPersistent:

```
; btnPersistent::pushButton
method pushButton(var eventInfo Event)
   ;// Set the shape to international symbol for No - persistent
   setMouseShape(MouseNo,TRUE)
endMethod
```

The following code is attached to the mouseEnter method for btnPersistent:

```
; btnPersistent::mouseEnter
method mouseEnterpushButton(var eventInfo MouseEvent)
   if isMousePersistent() then
      ;// If its persistent, set it back to the arrow cursor
      setMouseShap(MouseArrow,FALSE)
   endIf
endMethod
```

▪

message procedure

See also Example System Type
Beginner

Displays in the status line a message composed of up to six strings.

Syntax

```
message ( const message String [ , const message String ] * )
```

Description

message displays in the status line a message composed of up to six strings.

message example

Writes a message to the status line:

```
; showMessage::pushButton
method pushButton(var eventInfo Event)
var
  lastName, firstName String
endVar
lastName = "Borland"
firstName = "Frank"
message("Hello, my name is ", firstName, " ", lastName, ".")
endMethod
```

▪

msgAbortRetryIgnore procedure

See also Example System Type

Displays a dialog box containing a message and three buttons: Abort, Retry, and Ignore.

Syntax

msgAbortRetryIgnore (**const ***caption* String, const ***text*** String **)** String

Description

msgAbortRetryIgnore displays a three-button dialog box, where *caption* specifies the text in the title bar, and *text* specifies the message displayed. The return value is a string, in mixed upper- and lower-case, that corresponds to the button you click: "Abort", "Retry", or "Ignore". The button labels were changed in version 5.0.

msgAbortRetryIgnore example

The *showAbortRetryIgnore* button warns you that an operation may take a long time, and asks you whether to Abort, Retry, or Ignore.

```
; showAbortRetryIgnore::pushButton
method pushButton(var eventInfo Event)
var
  doThis String
endVar
doThis = msgAbortRetryIgnore("Note", "This may take a long time.
Do you want to stop?")  ; This message spans 2 lines.

doThis.view() ; Display your choice.

; Display a message based on your choice.
switch
   case doThis = "Abort"  : message("Aborting operation.")
   case doThis = "Retry"  : message("Retrying operation.")
   case doThis = "Ignore" : message("Ignoring problem.")
endSwitch
endMethod
```

msgInfo procedure

See also Example System Type
Beginner

Displays a one-button dialog box containing the information icon, a caption and message, and an OK button.

Syntax

msgInfo (const ***caption*** String, const ***text*** String **)**

Description

msgInfo displays *caption* in the title bar, and *text* in the box itself. You click OK or press Esc to close the box. This procedure does not return a value.

▪

msgInfo example

The **msgInfo** method displays a message.

```
; showMsgInfo::pushButton
method pushButton(var eventInfo Event)
msgInfo("Trivia", "The capital of Oregon is Salem.")
endMethod
```

msgQuestion procedure

See also Example System Type
Beginner

Displays a dialog box containing a caption and message, a question mark icon, and Yes and No buttons.

Syntax
`msgQuestion ( const caption String, const text String ) String`

Description
msgQuestion displays a dialog box containing a caption and message, a question mark icon, and Yes and No buttons. It displays *caption* in the title bar, and *text* in the box itself. This procedure returns your selection: "Yes" or "No" in mixed upper- and lower-case.

▪

msgQuestion example

Asks you whether to change the desktop title. If you choose Yes, the desktop title is changed, then restored.

```
; showMsgQuestion::pushButton
method pushButton(var eventInfo Event)
var
  userChoice String
  thisApp    Application
endVar
userChoice = msgQuestion("Confirm", "Are you sure you want to
change the title to 'Custom Application'?")
switch
   case userChoice = "Yes" :
     thisApp.setTitle("Custom Application")  ; Change desktop title.
     sleep(2000)                              ; Pause.
     thisApp.setTitle("Paradox for Windows") ; Restore it.
   case userChoice = "No"  :
     message("Application title not changed.")
endSwitch
endMethod
```

▪

msgRetryCancel procedure

See also Example System Type

Displays a dialog box containing a caption, a message and two buttons: Retry and Cancel.

Syntax

msgRetryCancel (**const ***caption* String**,** const ***text*** String **)** String

Description

msgRetryCancel displays a dialog box containing a caption, a message and two buttons: Retry and Cancel. The argument *caption* specifies the text in the dialog box title bar, and *text* specifies the message displayed. This procedure returns your selection: "Retry" or "Cancel". If you press Esc or select Close, returns "Cancel". Return values are in mixed upper- and lower-case.

▪

msgRetryCancel example

Poses a question in response to a problem; then confirms your selection on the status line .

```
; showMsgRetryCancel::pushButton
method pushButton(var eventInfo Event)
var
  confirm String
endVar
confirm = msgRetryCancel("Dilemma", "What will you do?")
switch
  case confirm = "Retry"  : message("Retrying.")
  case confirm = "Cancel" : message("Giving up.")
endSwitch
endMethod
```

▪

msgStop procedure

See also   Example   System Type

Displays a dialog box containing a stop sign icon, a caption and message, and an OK button.

Syntax

msgStop **(** const ***caption*** String, const ***text*** String **)**

Description

msgStop displays a dialog box containing a stop sign icon, a caption and message, and an OK button. It displays *caption* in the title bar, and *text* and a Stop icon in the box itself. Click OK or press Esc to close the box. This procedure does not return a value.

msgStop example

The **pushButton** method for *showMsgStop* alerts you to a potentially dangerous action.

```
; showMsgStop::pushButton
method pushButton(var eventInfo Event)
msgStop("Stop!", "If you do that, changes to the form will not be saved.")
endMethod
```

▪

msgYesNoCancel procedure

See also Example System Type

Displays a dialog box containing a caption, a message and three buttons: Yes, No, and Cancel.

Syntax

`msgYesNoCancel ( const caption String, const text String ) String`

Description

msgYesNoCancel displays a dialog box containing a caption, a message and three buttons: Yes, No, and Cancel. The argument *caption* specifies the text in the dialog box title bar, and *text* specifies the message displayed. This procedure returns your selection: "Yes", "No", or "Cancel" in mixed upper- and lower-case. If you press Esc or click the Close box, returns "Cancel".

msgYesNoCancel example

msgYesNoCancel asks you whether to save data before quitting; discard the data; or cancel the quit operation and stay in the application.

```
; showMsgYesNoCancel::pushButton
method pushButton(var eventInfo Event)
var
  theChoice String
endVar
theChoice = msgYesNoCancel("Quit", "Save data before quitting?")
switch
   case theChoice = "Yes"    : message("Saving data.")
   case theChoice = "No"     : message("Discarding data.")
   case theChoice = "Cancel" : message("Remaining in application.")
endSwitch
endMethod
```

pixelsToTwips procedure

See also   Example   System Type

Converts the screen coordinates specified in pixels to twips.

Syntax

`pixelsToTwips ( const pixels Point ) Point`

Description

pixelsToTwips converts the screen coordinates specified in pixels to twips.

▪

pixelsToTwips example

Shows the position of the button (using the object variable self) first in twips, then in pixels. Displays the screen resolution (in pixels); then uses that information to open a window in the center of the display.

```
; convertTwipsPixels::pushButton
method pushButton(var eventInfo Event)
var
  selfP,
  sysTwips  Point
  thisSys   DynArray[] AnyType
  x, y      SmallInt
  custForm  Form
endVar
selfP = self.Position
selfP.view("Position of this button in twips")
selfP = twipsToPixels(selfP)
selfP.view("Position of this button in pixels")
; open a 2" by 2" form exactly in the center of the screen
sysInfo(thisSys)                ; fill a dynamic array with system
information
sysTwips = Point(thisSys["FullWidth"], thisSys["FullHeight"])
sysTwips = pixelsToTwips(sysTwips)
x = int(sysTwips.x()/2) - 1440  ; calculate x-coordinate 1 inch left of
center
y = int(sysTwips.y()/2) - 1440  ; calculate y-coordinate 1 inch above center
custForm.open("Customer.fsl", WinStyleDefault, x, y, 2880, 2880)

endMethod
```

▪

play procedure

See also   Example   System Type
Beginner

Plays a standalone script.

Syntax

`play ( const scriptName String ) AnyType`

Description

play executes *scriptName*, just as if you called the Script method **run.** To return a value from a script, you call **formReturn** from within the script.

You can think of a standalone script as a special kind of custom method. For more information, refer to the Script type.

-

play example

Plays a script TESTSCR.SSL, which is assumed to be in the working directory.

```
; playAScript::pushButton
method pushButton(var eventInfo Event)
play("Testscr.ssl")
endMethod
```

printerGetInfo procedure

See also Example System Type

Gets information about the current printer in your system.

Syntax

```
printerGetInfo ( var printInfo PrinterInfo ) Logical
```

Description

printerGetInfo assigns printer information to *printInfo*, a record you declare using a special ObjectPAL data type PrinterInfo. Its structure:

Field	Type	Description
DriverName	String	Name of the printer driver for the current printer. Example: PSCRIPT.DRV.
DeviceName	String	Name that identifies the printer type. Example: Apple LaserWriter Plus.
PortName	String	Name of the port used by the current printer. Example: LPT1:.
DefaultPrinter	Logical	If True, the current printer is the default printer; otherwise, it is not the default printer.

This procedure returns True if successful; otherwise returns False.

▪

printerGetInfo example

See the example for **printerSetOptions**.

▪

printerGetOptions procedure

See also Example System Type

Gets information about settings and options for the printer currently attached to your system.

Syntax

```
1. printerGetOptions ( var printOptions PrinterOptionInfo ) Logical
2. printerGetOptions ( var printerInfo DynArray[] AnyType ) Logical
```

Description

printerGetOptions assigns printer information to *printInfo*, a record you declare as an ObjectPAL data type PrinterOptionInfo.

Syntax 2 fills the array *printerInfo* with only those options supported by the printer.

This procedure returns True if successful; otherwise returns False.

printerGetOptions example

Sets the current printer settings and checks whether the printer is using a large format paper source.

```
method pushButton(var eventInfo Event)
   var
      recUserOptions,
      recMyOptions   PrinterOptionInfo
   endVar

   ; Get the current printer settings.
   printerGetOptions(recUserOptions)
   if recUserOptions.DefaultSource = prnLargeFmt then
      return
   endIf

   ; Specify new printer settings. prnLargeFmt is a PrintSources constant.
   recMyOptions.DefaultSource = prnLargeFmt

   if printerSetOptions(recMyOptions) then
      message("Printer setup complete.")
   else
      errorShow()
   endIf
endMethod
```

PrinterOptionInfo record structure

Field	Type	Description
Orientation	LongInt	Paper orientation: portrait or landscape. Use a PrinterOrientation constant to test the value.
PaperSize	LongInt	Paper size. Use a PrinterSizes constant to test the value.
PaperWidth	LongInt	Custom paper width in twips (maximum of 64K twips). This value is converted internally to the tenths of a millimeter required by Windows.
PaperLength	LongInt	Custom paper length in twips (maximum of 64K twips). This value is converted internally to the tenths of a millimeter required by Windows.
Scale	LongInt	Scaling factor in percent. A scale value of 50 means a reduction to one-half the original size. A value of 200 means twice the original size. This only works with printers that support scaling for all functions, graphics, and fonts. Postscript printers and the Microsoft Windows Printing System are examples of such printers.
Copies	LongInt	Number of copies for the printer to make. Generally works only with page printers such as laser printers where the full page can be held in printer memory. Some printer drivers can support this feature on printers that cannot do full page printing. This setting is equivalent to unchecking the Collate button in the Print File dialog box. Output will not be collated. This operation is typically much faster than repeatedly sending the full document to the printer, but does require hand sorting afterwards.
DefaultSource	LongInt	Bin, tray, or feeder to be used by default. Use a PrintSources constant to test the value.
PrintQuality	LongInt	Higher print qualities are used for final output, lower for faster or draft output. Lower quality may differ significantly from the preview appearance of the document. Use a PrintQuality constant to test the value.
Color	LongInt	Sets color printers to color or monochrome printing. Monochrome is usually faster. Use a PrintColor constant to test the value.
Duplex	LongInt	For printers which support it, sets double-sided printing. Note that some printer drivers can support double-sided printing on otherwise single-sided printers by making two passes over the document. Use a PrintDuplex constant to test the value.

▪

printerSetCurrent procedure

See also Example System Type

Sets the current (active) printer on your system.

Syntax

**`printerSetCurrent ( `*`printerInfo`*`** String **)** Logical`

Description

printerSetCurrent sets the current (active) printer on your system. The argument *printerInfo* specifies the printer name, driver name, and printer port separated by commas. For example, if the printer name is Postscript Printer, the driver is PSCRIPT.DRV, and the port is LPT1:.

```
PostScript Printer,pscript,LPT1:
```

This procedure returns True if successful; otherwise returns False.

printerSetCurrent example

See the example for **enumPrinters**.

▪

printerSetOptions procedure

See also Example System Type

Specifies settings for the current printer attached to your system.

Syntax

```
1. printerSetOptions ( PrintOptions PrinterOptionInfo ) Logical
2. printerSetOptions ( var printerInfo DynArray[] AnyType [const overRide Logical] ) Logical
```

Description

printerSetOptions specifies settings for the current printer attached to your system. You declare *printerSettings,* a record of the special ObjectPAL data type PrinterOptionInfo. You don't have to supply values for every field in a PrinterOptionInfo record; the printer uses its current setting for any value you don't supply.

Syntax 2 uses the array *printerInfo* (obtained with **printerGetOptions**) to send the printer settings for only those options that the printer actually has. The optional *overRide* argument tells **printerSetOptions** to override any printer settings specified in the Form or the Report level.

printerSetOptions returns True if successful, and False otherwise. If you set a value that doesn't apply to the current printer, returns False.

printerSetOptions example

Prompts you to enter the number of copies of a report to print, sets up the printer, and prints the copies.

```
method pushButton(var eventInfo Event)
   var
      siNCopies   SmallInt
      stPrompt    String
      prnOptions  PrinterOptionInfo
      reOrders    Report
   endVar

   siNCopies = 0
   stPrompt  = "Print how many copies?"

   siNCopies.view(stPrompt)
   if siNCopies > 0 then
      prnOptions.Copies = siNCopies
   else
      return
   endIf

; Use constant to specify lower paper tray.
   prnOptions.DefaultSource = prnLower

; Use constant to specify landscape (long) orientation.
   prnOptions.Orientation = prnLandscape

; Use constant to specify high quality print.
   prnOptions.PrintQuality = prnHigh

   if printerSetOptions(prnOptions) then
      reOrders.print("orders")
   else
      errorShow("Could not set printer options.")
   endIf

endMethod
```

▪

projectViewerClose procedure

See also Example System Type

Closes the Project Viewer window.

Syntax

```
projectViewerClose ( ) Logical
```

Description

projectViewerClose closes the Project Viewer window. This procedure returns True if successful; otherwise returns False.

▪

projectViewerClose example

Calls **projectViewerIsOpen** to check whether the Project Viewer window is open. If it is open, closes it.

```
method open(var eventInfo Event)
   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:

   else
      ;// This code executes only for the form:
      if projectViewerIsOpen() then
         projectViewerClose()
      endIf
   endIf
endMethod
```

▪

projectViewerIsOpen procedure

See also   Example   System Type

Tells whether the Project Viewer window is open.

Syntax

```
projectViewerIsOpen ( ) Logical
```

Description

projectViewerIsOpen tells whether the Project Viewer window is open. This procedure returns True if the Project Viewer window is open; otherwise returns False.

▪

projectViewerIsOpen example

See the example for **projectViewerClose**.

▪

projectViewerOpen procedure

See also Example System Type

Opens the Project Viewer window.

Syntax

projectViewerOpen ( ) Logical

Description

projectViewerOpen opens the Project Viewer window. This procedure returns True if successful; otherwise returns False.

▪

projectViewerOpen example

Calls **projectViewerIsOpen** to check whether the Project Viewer window is open. If it is not, opens it.

```
method open(var eventInfo Event)
   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:

   else
      ;// This code executes only for the form:
      if not projectViewerIsOpen() then
         projectViewerOpen()
      endIf
   endIf
endMethod
```

▪

readEnvironmentString procedure

See also Example System Type

Reads an item from the Paradox copy of the DOS environment.

Syntax

`readEnvironmentString ( const key String ) String`

Description

readEnvironmentString returns a string containing information about the DOS environment variable specified by *key*. When Paradox starts it makes a copy of the DOS environment and this method reads that copy. Changes made to DOS environment variables after Paradox starts will not be read by this procedure.

Environment variables are assigned values by the DOS command SET. They control how DOS and some batch files and programs appear and function. Commonly used environment variables include PATH, PROMPT, and COMSPEC. For more information, consult your DOS manuals, especially the SET command.

▪

readEnvironmentString example

Uses **readEnvironmentString** to get, and **writeEnvironmentString** to change, the value of the PATH environment variable.

```
; changeEnvironmentStr::pushButton
method pushButton(var eventInfo Event)
var
   fs               FileSystem
   thePath, myDir   String
   pathArr Array[] String
endVar
; fs.getDir() currently returns some high-ANSI char--not a meaningful string
myDir = getaliaspath(fs.getDir())       ; get the current directory
myDir.view("Current directory")
thePath = readEnvironmentString("PATH") ; read the path environment var
thePath.breakApart(pathArr, ";")        ; break on semicolon
pathArr.view("An array of paths")       ; view the results
if NOT pathArr.contains(myDir) then     ; if current dir not in path
   msgInfo("FYI", "Adding current directory to path.")
   writeEnvironmentString("PATH", thePath + ";" + myDir)  ; add it
endIf
thePath = readEnvironmentString("PATH") ; read the changed environment var
thePath.view()
thePath.breakApart(pathArr, ";")        ; break it up
pathArr.view("An array of paths")       ; view the results
endMethod
```

▪

readProfileString procedure

See also Example System Type

Returns a particular value from a section of a file that you specify.

Syntax

`readProfileString ( ``const `***`fileName`***` String, const `***`section`***` String, const `***`key`***` String `**`)`**` String`

Description

readProfileString searches the WINDOWS directory by default. Typically, you use this method to read your WIN.INI file, so *fileName* would be WIN.INI.

Each section header in WIN.INI is bounded by square brackets on a line by itself; for example, [windows]. When you specify a *section*, however, you omit the brackets; for example, use "windows" to specify the [windows] section. Within each section, a value marker is followed by an equal sign (for example, Beep =); don't include the = when you specify *key*.

▪

readProfileString example

Uses **readProfileString** to get, and **writeProfileString** to change, the setting for the Windows beep.

```
; changeProfileStr::pushButton
method pushButton(var eventInfo Event)
var
   myBeep String
   winDir String
endVar
winDir = windowsDir()
myBeep = readProfileString(winDir + "\\win.ini", "windows", "Beep")
msgInfo("Beep?", myBeep) ; displays yes or no, depending on user's settings
if myBeep <> "yes" then
  msgInfo("Alert", "Changing profile string for Beep to yes.")
  writeProfileString(winDir + "\\win.ini", "windows", "Beep", "yes")
  beep()
else
  msgInfo("Alert", "Changing profile string for Beep to no.")
  writeProfileString(winDir + "\\win.ini", "windows", "Beep", "no")
  beep()
endIf
endMethod
```

▪

resourceInfo procedure

See also Example System Type

Lists the system resources.

Syntax

```
resourceInfo ( var info DynArray[ ] AnyType )
```

Description

resourceInfo writes system resource data to *info*, a DynArray that you declare and pass as an argument. The information returned in *info*:

Index	Definition
DiskAvail	Available disk space on the current drive
DiskTotal	Total disk space on the current dirve
FreeSpace	Total amount of free Windows memory
InternalVersion	Paradox internal BDE engine version
MemoryLoad	Percent of memory in use
MemPhysicalTotal	Total amount of physical memory
MemPhysicalAvail	Amount of available physical memory
MemPageFileTotal	Total amount of page/file memory
MemPageFileAvail	Amount of available page/file memory
MemVirtualTotal	Total amount of virtual memory
MemVirtualAvail	Amount of available virtual memory

resourceInfo example

Writes resource information to a dynamic array *dyn*, then displays *dyn* in a View dialog box.

```
; btnResourceInfo::pushButton
method pushButton(var eventInfo Event)
  var
    dynResources  Dynarray[] String
  endVar

  resourceInfo(dynResources)
  dynResources.view()
endmethod
```

runExpert procedure

See also Example System Type

Runs one of the Paradox experts.

Syntax

```
runExpert ( const expertType String, const expertName String )
```

Description

runExpert runs one of the registered Paradox experts. The *expertName* argument specifies the expert to run. The *expertType* parameter is used to determine the type of experts to list. ObjectPAL provides ExpertTypes constants for this purpose.

-

runExpert example

This example enumerates the available experts, determines if expertForm is in the list, and runs it if it is available.

```
method pushButton(var eventInfo Event)
var
   da dynarray[] anytype
   expert string
endvar

expertForm = "Form"
enumExperts(  expTypeDocument, da )

if da.contains( expertForm ) then
   runExpert( expTypeDocument, expertForm )
else
   msgStop( "Error", "Unable to run the expert :" + expertForm )
endif
endmethod
```

▪

SearchRegistry method

See also Example System Type

Searches the registry for a value.

Syntax

searchRegistry (const *key* String, const *searchStr* String, const *rootKey* LongInt, const *searchMode* LongInt, const *inMem* TCursor) Logical

Description

searchRegistry searches the registry for the value in *searchStr*. Only registry string data types are searched. Searches are case insensitive. The results of the search are placed in *inMem*, an in-memory TCursor. Returns True, if successful, and False, if unsuccessful.

key is entered as a path similar to a file path. If *key* is not blank then the search starts from the path specified in *key*, otherwise the search starts from the *rootKey*. *searchStr* is the value of the object you want to search in the registry. Only strings are searched in the registry. Registry DWORD or Binary types are not searched. If *searchStr* is blank, **searchRegistry** returns an error. *rootKey* can be set with the predefined constants or it can be zero. If *rootKey* is zero, then all the rootKeys will be searched.

searchMode defines which registry objects you want to search in the registry. Registry objects are keys, value names and data. The following table describes the *searchMode* flags:

searchMode	Registry objects searched
0	All
1	Keys
2	Value names
3	Data
4	Keys and value names
5	Keys and Data
6	Value names and Data

The *inMem* TCursor has three fields that are limited to A255. The values in these fields can end up being truncated if the key returned is greater than 255 characters. **searchRegistry** returns a warning if the field limit is reached. *inMem* is an in-memory TCursor that has the following structure:

Field Name	Field Type	Description
RegistryType	A12	The registry object type: "Key", "ValueName" or "Data".
RootKeyConst	A25	The predefined rootKey constant as a string.
KeyPath	A255	The full path of the key.
ValueName	A255	The full value name.
Data	A255	The full Data.

searchRegistry example

The following example searches the entire registry for all Keys with the string "Borland" and display the results in a TableView window.

```
;// Search the registry for keys that have the string "Borland" in them, and
write
;// the results to a table
var
   tc TCursor
endVar
searchRegistry( "", "Borland", 0, 1, tc )  ; Search the registry
                                           ; for keys that have
                                           ; "Borland" in them
if NOT tc.isEmpty() then
   tc.instantiateView("keytab.db")    ; write the results to a table
endif
tc.close()
```

The following example searches the entire registry for all Keys with the word "Pdoxwin" and display the results in a TableView window.

```
var
   tc   Tcursor
   tv   TableView
endvar

searchRegistry( "", "Pdoxwin", 0, 1, tc )
tc.instantiateView( ":priv:keysreg" )
tv.open( ":priv:keysreg" )
tv.wait()
tv.close()
```

▪

sendKeys procedure

See also Example System Type

Sends one or more keystrokes to the active window as if they had been entered at the keyboard. The active window need not be Paradox.

Syntax

```
sendkeys ( const keyText String [ , const wait Logical ] ) Logical
```

Description

sendKeys sends one or more keystrokes to the active window as if they had been entered at the keyboard. The active window need not be Paradox. The argument *keyText* specifies one or more keys to send. *wait* (optional) specifies whether to continue executing keystroke sequences in the message loop without waiting. Returns False if an error occurs and it cannot send any of the keys; in this case **errorCode**returns one of the following:

Error code	Error message
peskMissingCloseBrace	"Missing closing brace."
peskInvalidKey	"The key name is not correct."
peskMissingCloseParen	"Missing closing parentheses."
peskInvalidCount	"The repeat count is not correct."
peskStringTooLong	"The keys string is too long."
peskCantInstallHook	"Could not install Windows journalling hook."

Note: **sendKeys** can only send keystrokes to an application designed to run in Microsoft Windows; it cannot send the print screen (prtsc) key to any application.

The keyText argument

Each key is represented by one or more lowercase characters. For example, to represent the letter A, use "a" for *keyText*. To represent more than one character, string them together. For example, to send the letters a, b, and c, use "abc" for *keyText*. The plus sign (+),caret (^), percent sign (%), tilde (~), and parentheses ( ) have special meanings to **sendKeys**. To specify one of these characters, enclose it inside braces. For example, to specify the plus sign, use{+}.

To send brace characters, enclose each brace in braces: {{} and {}}.

To specify non-printing characters (such as enter or tab) and keys that represent actions rather than characters, use the codes listedbelow:

Key	Codes
backspace	{backspace}, {bs}, {bksp}, {vk_back}
break	{break}, {vk_break}
caps lock	{capslock}, {vk_captial}
clear	{clear}, {vk_clear}
del	{delete}, {del}, {vk_delete}
down arrow	{down}, {vk_down}
end	{end}, {vk_end}
enter	{enter}, {return}, {vk_return} (the character ~)
esc	{escape}, {esc}, {vk_escape}
help	{help}, {vk_help}
home	{home}, {vk_home}

ins	{insert}, {vk_insert}
left arrow	{left}, {vk_left}
num lock	{numlock}, {vk_numlock}
page down	{pgdn}, {vk_next}
page up	{pgup}, {vk_prior}
print screen	{prtsc}, {vk_snapshot}
right arrow	{right}, {vk_right}
scroll lock	{scrolllock}, {vk_scroll}
space bar	{vk_space}
tab	{tab}, {vk_tab}
up arrow	{up}, {vk_up}
F1	{F1}, {vk_F1}
F2	{F2}, {vk_F2}
F3	{F3}, {vk_F3}
F4	{F4}, {vk_F4}
F5	{F5}, {vk_F5}
F6	{F6}, {vk_F6}
F7	{F7}, {vk_F7}
F8	{F8}, {vk_F8}
F9	{F9}, {vk_F9}
F10	{F10}, {vk_F10}
F11	{F11}, {vk_F11}
F12	{F12}, {vk_F12}
F13	{F13}, {vk_F13}
F14	{F14}, {vk_F14}
F15	{F15}, {vk_F15}
F16	{F16}, {vk_F16}

The ~ character represents the Enter key.

For example, `sendKeys("abc~")` types the letters abc and the carriage return.

To specify keys combined with any combination of Shift, Ctrl, and Alt keys, precede the regular key code with one or more of the following codes:

Key	**Code**
Shift	+
Control	^
Alt	%

For example, to show the File menu list in Paradox: `sendKeys("%f")`;then move down 3 menu items: `sendKeys("{down 3}");` then pick the item: `sendKeys("~")`. To combine these three steps into one: `sendKeys("%f{down 3}~")`

To specify that Shift, Ctrl, and/or Alt be held down while one or more keys are pressed, enclose the key codes in parentheses. For example, Shift is pressed while `a` and `b` are pressed, use `"+(ab)"`. If Shift is pressed while `a` is pressed, followed by `b` being pressed without Shift, use `"+ab"`.

To specify repeating keys, enclose a string and a number in braces {*key number*} (put a space between the key and the number). For example, {`left 42`} means press the left arrow key 42 times; and {`h 9`} means press h 9 times.

Special commands

Following are special commands you can include as part of the *keyText* argument:

- **{delay *value*}**
- **{action *integervalue*}**
- **{cmt *comment*}**
- **{beginexact} *text* {endexact}**
- **{menu *integervalue*}**

{delay *value*}

delay sets the delay (in milliseconds) between keystrokes. `{delay 1000} wait`s 1 second between keystrokes; this is approximate and may vary if Shift, Alt, or Ctrl are set. If the actual time to execute the command is longer, you may see additional delays.

delay is mainly used to let dialog boxes display. Without it the keys are sent at full speed, and Windows processes the keys too quickly to paint the dialog box on the screen. **delay** remains in effect until another **delay** or a **sendKeys** statement executes; it does not affect action commands.

{action *integervalue*}

action sends an action to the object in the form that issued the **sendKeys** statement. It allows you to gain control while **sendKeys** executes, to inspect the state of forms or dialog boxes. *integervalue* is a value between 0 and 2047. Do not call any methods or procedures that wait for user input, and do not open a form or report.

{cmt *comment*}

cmt lets you insert comments. *comment* represents your remarks; all characters are allowed.

{beginexact} text {endexact}

sendKeys normally ignores carriage returns and line feeds, and assigns meanings to certain characters. To bypass this processing, enclose the text with `{beginexact}` and `{endexact}`. Once a `{beginexact}` is encountered, all text is processed exactly as is until the `{endexact}`.

If you call **sendKeys** while another **sendKeys** statement is executing, Paradox adds the new key sequence to the end of the event queue.

{menu *integervalue*}

This sends a menu command to the active object. `integervalue` represents a value from the menu command constants.

The wait argument

wait specifies whether to wait after keys are sent, or to continue ObjectPAL execution. The recommended setting is False. Windows sometimes stops responding to **sendKeys** if the **wait** parameter is True, such as when keys are sent to nested dialogs. Always set **wait** to False when changing the working directory or the private directory.

Note: sendKeys statements are not portable across language barriers.

▪

sendKeys example

The system procedure **execute** runs the Windows Notepad application; then **sendKeys** twice sends keystrokes to Notepad and saves the file as TWOLINES.TXT.

```
method pushButton(var eventInfo event)

   execute("notepad.exe")          ; run Notepad.
   sleep(1000)
   ; write a short note.

   sendKeys("this is the first line of a 2-line note.~")
   sendKeys("this is the second line of a 2-line note.")

   ; send alt+f, s to choose File|Save.

   sendKeys("%fs")

   ; send a file name to the dialog box, and
   ; send enter to save the file.

 sendKeys("twolines.txt~")

   ; send Alt+f4 to close Notepad.

   sendKeys("%{f4}")

endMethod
```

sendKeysActionID method

Example System Type

Lets the **sendKeys** procedure notify you that the **sendKeys** queue is empty.

Syntax

```
sendKeysActionID ( const id SmallInt )
```

Description

sendKeysActionID lets the **sendKeys** procedure notify you that the **sendKeys** queue is empty. The argument *id* is a user-defined action constant whose value is between the IdRanges constants UserAction and UserActionMax. *id* is sent to the active object of the form (or to the form itself if there is no active object) that issued the **sendKeys** method.

Typically, you put the code to process **sendKeysActionID** at the form level. If there is an active object, it receives the ID in its **action** method. The default, however, is to bubble the action ID to the form.

▪

sendKeysActionID example

Specifies the action ID value sent when the queue is empty. Suppose a form contains an unbound field and a button. The following code is attached to the form's Const window.

```
const
   kMyCustomAction = 1
endConst
```

The following code is attached to the form's built-in **action** method.

```
method action(var eventInfo ActionEvent)
  if eventInfo.id() = UserAction + kMyCustomAction then
     message("sendKeys has finished sending")
  endIf
endMethod
```

The following code is attached to a button's built-in **pushButton** method.

```
method pushButton(var eventInfo Event)
   ; Send keys but do not wait.
   sendKeys("This is some text", FALSE)

   ; Set the action id to send when the queue is empty.
   sendKeysActionID(UserAction + kMyCustomAction)
endMethod
```

■

setDefaultPrinterStyleSheet procedure

See also   Example   System Type

Specifies a default printer style sheet.

Syntax

```
setDefaultPrinterStyleSheet ( const fileName String )
```

Description

setDefaultPrinterStyleSheet sets the default Paradox style sheet, specified by *fileName*, for documents designed for the printer. If *fileName* does not specify a full path, **setDefaultPrinterStyleSheet** searches the working directory.

Note: Printer style sheet files have the extension .FP, and screen style sheet files the extension .FT. You cannot use an .FT file as a printer style sheet, nor an .FP file as a screen style sheet.

Any UIObjects created in forms and reports while the style sheet is active will have the properties and methods of the corresponding prototype objects in the style sheet.

This procedure does not change the properties or methods of existing UIObjects. It has no effect on UIObjects in forms and reports that use style sheets other than the default.

Individual forms and reports may use style sheets other than the default. Use **getStyleSheet** and **setStyleSheet** to work with style sheets for specific forms and reports.

Use **setDefaultScreenStyleSheet** to specify the name of the default screen style sheet, used whenever you create design documents that are designed for the screen.

▪

setDefaultPrinterStyleSheet example

Calls **getDefaultPrinterStyleSheet** to check the current default style sheet. If the style sheet is not BORLAND.FT, the code calls **setDefaultPrinterStyleSheet** to set it, then calls **getDefaultPrinterStyleSheet** again to make sure it was reset successfully.

Note: **setDefaultPrinterStyleSheet** requires double backslashes in the path, but **getDefaultPrinterStyleSheet** returns single backslashes.

```
method pushButton(var eventInfo Event)
   ; Get and set the system style sheet.
   if getDefaultPrinterStyleSheet() <> "c:\\pdoxwin\\borland.fp" then
      setDefaultPrinterStyleSheet("c:\\pdoxwin\\borland.fp")
      if getDefaultPrinterStyleSheet() <> "c:\\pdoxwin\\borland.fp" then
         msgStop("Problem", "Could not set the style sheet.")
      endIf
   endIf
endMethod
```

▪

setDefaultScreenStyleSheet procedure

See also		Example		System Type

Specifies a default screen style sheet.

Syntax

`setDefaultScreenStyleSheet ( const fileName String )`

Description

`setDefaultScreenStyleSheet` sets the default Paradox style sheet, specified by *fileName*, for documents designed for the screen. If *fileName* does not specify a full path, **`setDefaultScreenStyleSheet`** searches the working directory.

Note: Screen style sheet files have the extension .FT, and printer style sheet files the extension .FP. You cannot use an .FP file as a screen style sheet, nor an .FT file as a printer style sheet.

Any UIObjects created in forms and reports while the style sheet is active will have the properties and methods of the corresponding prototype objects in the style sheet.

This procedure does not change the properties or methods of existing UIObjects. It has no effect on UIObjects in forms and reports that use style sheets other than the default.

Individual forms and reports may use style sheets other than the default. Use **getStyleSheet** and **setStyleSheet** to work with style sheets for specific forms and reports.

Use **setDefaultPrinterStyleSheet** to specify the name of the default printer style sheet, used whenever you create design documents that are designed for the printer.

setDefaultScreenStyleSheet example

Calls **getDefaultScreenStyleSheet** to check the current system style sheet. If it is not BORLAND.FT, **setDefaultScreenStyleSheet** sets it, then **getDefaultScreenStyleSheet** makes sure it was set successfully.

```
method pushButton(var eventInfo Event)
   ; Get and set the system style sheet.
   if getDefaultScreenStyleSheet() <> "c:\\pdoxwin\\borland.ft" then
      setDefaultScreenStyleSheet("c:\\pdoxwin\\borland.ft")
      if getDefaultScreenStyleSheet() <> "c:\\pdoxwin\\borland.ft" then
         msgStop("Problem", "Could not set the style sheet.")
      endIf
   endIf
endMethod
```

▪

setDesktopPreference procedure

See also Example System Type

Sets a desktop preference.

Syntax

setDesktopPreference (const ***section*** AnyType, const ***name*** AnyType, const ***value*** AnyType **)** Logical

Description

setDesktopPreference sets the value of the desktop preference specified by the *section* and *name* arguments. The *value* argument corresponds to one of the DesktopPreferenceTypes Constants.

setDesktopPreference example

Displays the sets the title name preference, then gets the name and displays it.

```
method pushButton(var eventInfo Event)
setDesktopPreference( PrefProjectSection, prefTitleName,"Paradox pour
Windows" )

x = getDesktopPreference( PrefProjectSection, prefTitleName )

x.view()
endmethod
```

▪

setMouseScreenPosition procedure

See also Example System Type

Displays the mouse pointer at a specified position.

Syntax

```
1. setMouseScreenPosition ( const mousePosition Point )
2. setMouseScreenPosition ( const x LongInt, const y LongInt )
```

Description

setMouseScreenPosition displays the mouse pointer at the point specified in *mousePosition* (syntax 1); or at the coordinates specified in twips by *x* and *y* (syntax 2).

Use Point type methods such as **x** and **y** to get more information.

▪

setMouseScreenPosition example

See the example for **getMouseScreenPosition**.

setMouseShape procedure

See also Example System Type

Sets the shape of the mouse pointer.

Syntax

```
setMouseShape ( const mouseShapeId LongInt [,const persist Logical] ) LongInt
```

Description

setMouseShape sets the shape of the mouse pointer. The argument *mouseShapeId* specifies the shape of the mouse pointer. ObjectPAL provides MouseShapes constants for this purpose.

If **persist** is true then the mouse cursor will be persistent (will not change shape) to objects that implicitly change the shape of the mouse (for instance, button objects and field objects). **persist** will not affect where the ObjectPAL developer has explicity changed the shape of the mouse. For example, in a **mouseEnter** method of an object, **setMouseShape** will override mouse persistence. **persist** will not affect OCX or Native windows controls.

▪

setMouseShape example

In the following example, a form has two buttons: btnNonPersistent and btnPersistent. The pushButton method of each button uses **setMouseShape** to set the mouse shape of the cursor; the first with persistence set to false, the second with persistence set to true. The second button, btnPersistent also contains a mouseEnter method which will use **isMousePersistent** to evaluate the persistency of the mouse cursor and revert it to its original state.

When the first button is pressed, the mouse cursor is changed. However, when the mouse cursor is moved off the button, the mouse cursor will revert to its original setting. When the second button is pressed, the mouse cursor is changed and will remain unchanged until the mouse cursor is moved back over the second button. This will trigger the mouseEnter method of the second button and revert the mouse cursor back to its original state.

The following code is attached to the pushButton method for btnNonPersistent:

```
; btnNonPersistent::pushButton
method pushButton(var eventInfo Event)
   ;// Set the shape to international symbol for No - non-persistent
   setMouseShape(MouseNo,FALSE)
endMethod
```

The following code is attached to the pushButton method for btnPersistent:

```
; btnPersistent::pushButton
method pushButton(var eventInfo Event)
   ;// Set the shape to international symbol for No - persistent
   setMouseShape(MouseNo,TRUE)
endMethod
```

The following code is attached to the mouseEnter method for btnPersistent:

```
; btnPersistent::mouseEnter
method mouseEnter(var eventInfo MouseEvent)
   if isMousePersistent() then
      ;// If its persistent, set it back to the arrow cursor
      setMouseShape(MouseArrow,FALSE)
   endIf
endMethod
```

■

setMouseShapeFromFile method

See also Example System Type

Specifies the shape of the mouse pointer.

Syntax

`setMouseShapeFromFile ( const fileName String [,const persist Logical] ) LongInt`

Description

setMouseShapeFromFile sets the shape of the mouse pointer based on data in *fileName*. *fileName* can be a *.CUR or *.ANI file. Paths and alias's are supported. If *fileName* does not exist, a warning is generated. setMouseShapeFromFile returns a LongInt handle to the mouse shape.

If *persist* is true then the mouse cursor will be persistent (will not change shape) to objects that implicitly change the shape of the mouse (for instance, button objects and field objects). *persist* will not affect where the ObjectPAL developer has explicity changed the shape of the mouse. For example, in a **mouseEnter** method of an object, **setMouseShape** will override mouse persistence. *persist* will not affect OCX or Native windows controls.

setMouseShapeFromFile example

In the following example, a form has two buttons: btnNonPersistent and btnPersistent. The pushButton method of each button uses **setMouseShapeFromFile** to set the mouse shape of the cursor to an animated cursor provided with Windows 95 and Windows NT; the first with persistence set to false, the second with persistence set to true. The second button, btnPersistent also contains a mouseEnter method which will use **isMousePersistent** to evaluate the persistency of the mouse cursor and revert it to its original state.

When the first button is pressed, the mouse cursor is changed. However, when the mouse cursor is moved off the button, the mouse cursor will revert to its original setting. When the second button is pressed, the mouse cursor is changed and will remain unchanged until the mouse cursor is moved back over the second button. This will trigger the mouseEnter method of the second button and revert the mouse cursor back to its original state. Each pushButton method will check to see which operating system its running under to determine where to find the animated cursor file.

The following code is attached to the pushButton method for btnNonPersistent:

```
; btnNonPersistent::pushButton
method pushButton(var eventInfo Event)
var
   sysDyn       DynArray[] AnyType
   mouseHandle  LongInt
endVar
   sysInfo(sysDyn)
   if sysDyn["WindowsPlatform"] = "WIN95" then
                    ;// if Windows 95
      mouseHandle = setMouseShapeFromFile( windowsDir() +
                    "\\CURSORS\\HOURGLAS.ANI", FALSE)
   else
         ;// if Windows NT
      mouseHandle = setMouseShapeFromFile( windowsSystemDir() +
                    "\\HOURGLAS.ANI", FALSE)
   endIf
endMethod
```

The following code is attached to the pushButton method for btnPersistent:

```
; btnPersistent::pushButton
method pushButton(var eventInfo Event)
var
   sysDyn       DynArray[] AnyType
   mouseHandle  LongInt
endVar
   sysInfo(sysDyn)
   if sysDyn["WindowsPlatform"] = "WIN95" then
         ;// if Windows 95
      mouseHandle = setMouseShapeFromFile( windowsDir() +
                    "\\CURSORS\\HOURGLAS.ANI", TRUE)
   else
         ;// if Windows NT
      mouseHandle = setMouseShapeFromFile( windowsSystemDir() +
                    "\\HOURGLAS.ANI", TRUE)
   endIf
endMethod
```

The following code is attached to the mouseEnter method for btnPersistent:

```
; btnPersistent::mouseEnter
method mouseEnterpushButton(var eventInfo MouseEvent)
   if isMousePersistent() then
         ;// If its persistent, set it back to the arrow cursor
      setMouseShap(MouseArrow,FALSE)
   endIf
endMethod
```

■

setRegistryValue method

See also Example System Type

Sets a value in the registry.

Syntax

```
setRegistryValue ( const key String, const value String, const data AnyType, const rootKey LongInt ) Logical
```

Description

setRegistryValue writes data to a specified value of a registry key. If the *key* or *value* do not exist, then they will be created. If *data* is empty then only *key* is created. If *value* is empty, then *key* and *data* are created.

The *key* is entered as a path similar to a file path. However, unlike a file path, wildcards are not expanded. *key* cannot contain a single backslash and cannot be empty. The *value* is a string that is limited to 65,534 bytes. *value* can contain backslashes and *value* can be empty. **setRegistryValue** returns True, if successful, and False, if unsuccessful.

data can accept the following types:

ObjectPAL Type	Registry type	Size limitation
Currency	String	32k
Date	String	32k
DateTime	String	32k
Logical	String	32k
LongInt	DWORD	32k
Memo	String	32k
Number	String	32k
Point	String	32k
SmallInt	DWORD	32k
String	String	32k
Time	String	32k

rootKey is analogous to a directory drive. The *rootKey* should be set with the predefined constants:

regKeyCurrentUser
regKeyClassesRoot
regKeyLocalMachine
regKeyUser

▪

setRegistryValue example

The following example sets the current ObjectPAL Level in the Registry.

```
var
   strLevel   String
endvar

;// create key, value and data in regCurrentUser
setRegistryValue( "Software\\Borland\\Myapp\\Settings", "ObjectValue", "An
object", regKeyCurrentUser )
```

setUserLevel procedure

See also Example System Type

Sets your ObjectPAL level, either Beginner or Advanced. Beginner restricts the methods displayed for each object in the Integrated Development Environment (IDE) to those a new ObjectPAL user would likely need; Advanced displays all methods.

Note: The advanced setting is highly recommended.

Syntax

```
setUserLevel ( const level String )
```

Description

Use **getUserLevel** to return the current setting.

Note: The ObjectPAL level setting *does not* affect how code executes; it only affects what is displayed in the user interface.

▪

setUserLevel example

Uses **getUserLevel** to see if the ObjectPAL user level is set to Beginner; if so, **setUserLevel** sets it to Advanced. Next, if the ObjectPAL user level is already set to Advanced, sends a message stating this to the status bar.

```
;setToAdvanced::pushButton
method pushButton(var eventInfo Event)
   if getUserLevel() = "Beginner" then
      setUserLevel("Advanced")
      message("ObjectPAL level is now set to Advanced")
   else
      message("ObjectPAL level was already set to Advanced")
   endIf
endmethod
```

■

sleep procedure

See also Example System Type
Beginner

Produces a delay of a specified duration.

Syntax

`sleep ( [ const numberOfMilliSeconds LongInt ] )`

Description

sleep disables the currently executing form, rendering it incapable of receiving keystrokes, mouse events, or focus for the number of milliseconds specified in *numberofMilliseconds*. It does not disable the Desktop or stop timer events.

Note: When called with no argument, **sleep** does *not* disable the form. It causes the current method to yield to Windows to let a single pending message be processed.

■

sleep example

Displays a message in the status line, then waits five seconds before displaying a second message.

```
; goToSleep::pushButton
method pushButton(var eventInfo Event)
var
   yourTurn SmallInt
endVar
yourTurn = 5000
beep()
message("Next message in 5 seconds.")
sleep(yourTurn)                     ; waits for 5 seconds
message("5 seconds have elapsed.")
endMethod
```

▪

sound procedure

See also   Example   System Type
Beginner

Creates a sound of specified frequency and duration.

Syntax

```
sound ( const freqHertz, const durationMilliSecs LongInt )
```

Description

sound creates a sound of frequency *freqHertz* (in Hertz) for a time *durationMilliSecs* (in milliseconds) . Frequency values can range from 1 to 50,000 Hertz.

▪

sound example

The **pushButton** method for *makeMusic* first declares a number of constants for frequency values in a scale. These notes are used to specify the *frequency* argument in the calls to the **sound** method. After playing a few bars from a tune, the method demonstrates the calculation for notes in a chromatic scale (proceeds by half notes).

```
; makeMusic::pushButton
method pushButton(var eventInfo Event)
var
   quarterNote, octave, note LongInt
   power                     Number
endVar
; frequency values for notes in a scale
const
  noteA1  = 110
  noteA#1 = 116
  noteB1  = 123
  noteC1  = 130
  noteC#1 = 138
  noteD1  = 146
  noteD#1 = 155
  noteE1  = 164
  noteF1  = 174
  noteF#1 = 184
  noteG1  = 195
  noteG#1 = 207
  noteA2  = 220
  noteA#2 = 234
  noteB2  = 249
  noteC2  = 265
  noteC#2 = 282
  noteD2  = 300
endConst
; several bars from Peter and the Wolf
sound(noteA1, 200)
sound(noteD1, 150)
sound(noteF#1, 50)
sound(noteA2, 100)
sound(noteB2, 100)
sound(noteA2, 150)

sound(noteF#1, 50)
sound(noteA2, 100)
sound(noteB2, 100)
sound(noteC#2, 150)
sound(noteD2, 50)
sound(noteA2, 100)
sound(noteF#1, 100)
sound(noteD1, 100)
sleep(1000)

; play a few chromatic scales
quarterNote = 120
for octave from 0 to 1
   for note from 0 to 11
      sound(int(pow(2, octave + note / 12.0) * 110), quarterNote)
   endFor
endFor
sound(int(pow(2, 2) * 110), quarterNote) ; finish out the scale
endMethod
```

▪

sysInfo procedure

See also Example System Type Changes

Creates a dynamic array of information about the system running Paradox.

Syntax

```
sysInfo ( var info DynArray[ ] AnyType )
```

Description

sysInfo creates a dynamic array of information about the system running Paradox. Declare the DynArray **info** before calling this method. **info** contains indexes for system attributes and their values, described in the table below:

System Attribute Index	Definition
AnsiCodePage	The ANSI (Windows) code page loaded by Windows.
AreMouseButtonsSwapped	Reports whether functions of the left and right mouse buttons are reversed
CodePage	Reports which code page is currently loaded by Windows
CPU	Processor type
Edition	Paradox edition (for example, Standard)
EngineDate	Creation date of database engine
EngineLanguageID	The language used for BDE messages and QBE keywords, shown in the list of language identifiers
EngineVersion	Version number of database engine
FullHeight	Vertical working area (in pixels) in a maximized window
FullWidth	Horizontal working area (in pixels) in a maximized window
IconHeight	Height of icons (in pixels)
IconWidth	Width of icons (in pixels)
KeyboardFNKeys	Number of function keys
KeyboardLayoutID	The layout name for the currently loaded keyboard; usually a language ID.
KeyboardSubType	Keyboard subtype is an OEM-dependent value
KeyboardType	Type and manufacturer of the keyboard
LanguageDriver	Default language (drivers for Paradox tables)
LocalShare	Reports whether Local Share is active
Memory	Available memory, including swap file (if present) in bytes
Mouse	The number of mice attached to the system
NetDir	The path to PDOXUSRS.NET
NetProtocol	Network protocol
NetShare	Reports whether Net Share is active
NetType	Network type
ParadoxSystemDir	The path of the Paradox directory
ScreenHeight	Total height of screen (in pixels)
ScreenWidth	Total width of screen (in pixels)
StartupDir	The full path (including the drive ID letter) to your start-up directory; that

	is, the directory from which Paradox was started
SystemDefaultLCID	The system default locale ID; a 32-bit value which is the combination of a language ID and a sort ID.
UserDefaultLCID	The user default locale ID
UserName	Network user name
WindowsBuild#	The internal build number
WindowsDir	Path to the WINDOWS directory
WindowsPlatform	Chicago, NT, or WIN32s
WindowsSystemDir	Path to the WINDOWS\SYSTEM directory
WindowsText	Arbitrary information
WindowsVersion	Version number of Windows

sysInfo example

Writes system information to a dynamic array *userSys*, then displays *userSys* in a View dialog box. (See also pixelsToTwips.)

```
; showSysInfo::pushButton
method pushButton(var eventInfo Event)
var
   userSys DynArray[] AnyType
endVar
sysInfo(userSys)    ; fill the array with system information
userSys.view()      ; show the array
endMethod
```

Changes to System::sysInfo

In version 7, several fields were added to the sysInfo information dynarray.

The following fields were added in version 7:

Field name	Type	Description
EngineLanguageID		The language used for BDE messages and QBE keywords, shown in the list of language identifiers
AnsiCodePage		The ANSI (Windows) code page loaded by Windows.
WindowsBuild#		The internal build number
WindowsPlatform		Chicago, NT, or WIN32s
WindowsText		Arbitrary information
KeyboardLayoutID		The layout name for the currently loaded keyboard; usually a language ID.
SystemDefaultLCID		The system default locale ID; a 32-bit value which is the combination of a language ID and a sort ID.
UserDefaultLCID		The user default locale ID

Note: In version 7, the ConfigFile, MathCoprocessor, NumTasks, Protected fields were deleted from the sysInfo dynarray.

▪

Language identifiers

The language identifier consists of the the primary language ID and the sub_language ID.

The primary language IDs are as follows:

Code	Language	Code	Language
0x0401	Arabic	0x0415	Polish
0x0402	Bulgarian	0x0416	Brazilian Portuguese
0x0403	Catalan	0x0417	Rhaeto-Romanic
0x0404	Traditional Chinese	0x0418	Romanian
0x0405	Czech	0x0419	Russian
0x0406	Danish	0x041A	Croato-Serbian (Latin)
0x0407	German	0x041B	Slovak
0x0408	Greek	0x041C	Albanian
0x0409	U.S. English	0x041D	Swedish
0x040A	Castilian Spanish	0x041E	Thai
0x040B	Finnish	0x041F	Turkish
0x040C	French	0x0420	Urdu
0x040D	Hebrew	0x0421	Bahasa
0x040E	Hungarian	0x0804	Simplified Chinese
0x040F	Icelandic	0x0807	Swiss German
0x0410	Italian	0x0809	U.K. English
0x0411	Japanese	0x080A	Mexican Spanish
0x0412	Korean	0x080C	Belgian French
0x0413	Dutch	0x0C0C	Canadian French
0x0414	Norwegian - Bokml	0x100C	Swiss French
0x0810	Swiss Italian	0x0816	Portuguese
0x0813	Belgian Dutch	0x081A	Serbo-Croatian(Cyrillic)
0x0814	Norwegian - Nynorsk		

tracerClear procedure

See also Example System Type

Clears the Tracer window.

Syntax

```
tracerClear ( )
```

Description

tracerClear clears the Tracer window. The Tracer window can be opened by the **tracerOn** procedure at run time, or by selecting the ObjectPAL Editor menu's View|Tracer command.

tracerClear example

Clears the Tracer window. Assume that the Tracer window is open and contains information.

```
; wipeTracer::pushButton
method pushButton(var eventInfo Event)
tracerClear()                    ; clear the Tracer window
endMethod
```

▪

tracerHide procedure

See also Example System Type

Hides the Tracer window.

Syntax

```
tracerHide ( )
```

Description

tracerHide hides the Tracer window. It makes the Tracer window invisible, but does not clear nor close it. To make the Tracer window visible again, use **tracerShow.**

tracerHide example

Hides the Tracer window, pauses, then displays it again. Assume that the Tracer window is already open.

```
; toggleTracerWin::pushButton
method pushButton(var eventInfo Event)
tracerHide()                     ; make the Tracer window invisible
message("Hiding Tracer window. Pausing...")
sleep(2000)
message("Showing Tracer window.")
tracerShow()                     ; make the Tracer window visible again
tracerToTop()                    ; bring it to the top
endMethod
```

▪

tracerOff procedure

See also Example System Type

Closes the Tracer window.

Syntax

```
tracerOff ( )
```

Description

tracerOff closes the Tracer window. It stops writing code traces to the Tracer window. You resume tracing code with the **tracerOn** procedure. Tracing is on by default after the Tracer window is opened.

tracerOff example

This code turns off code tracing.

```
; stopTracer::pushButton
method pushButton(var eventInfo Event)
tracerOff()                          ; close the Tracer window
endMethod
```

tracerOn procedure

See also Example System Type

Activates code tracing.

Syntax

```
tracerOn ( )
```

Description

tracerOn activates code tracing. It resumes writing code traces to the Tracer window.

▪

tracerOn example

Reactivates code tracing.

```
; startTracer::pushButton
method pushButton(var eventInfo Event)
tracerOn()                      ; reactivate the Tracer window
endMethod
```

tracerSave procedure

See also Example System Type

Saves the contents of the Tracer window to a file *fileName*.

Syntax

tracerSave (const ***fileName*** String **)**

Description

tracerSave saves the contents of the Tracer window to the file specified with *fileName*.

■

tracerSave example

Saves the contents of the Tracer window to a file MYTRACE.TXT.

```
; saveTracerToFile::pushButton
method pushButton(var eventInfo Event)
tracerSave("mytrace.txt")          ; save the Tracer window to a file
endMethod
```

▪

tracerShow procedure

See also Example System Type

Makes the Tracer window visible.

Syntax

```
tracerShow ( )
```

Description

tracerShow makes the Tracer window visible. You make the Tracer window invisible with the **tracerHide** procedure.

▪

tracerShow example

See the example for **tracerHide**.

tracerToTop procedure

See also Example System Type

Makes the Tracer window the topmost window on the desktop.

Syntax

`tracerToTop ( )`

Description

tracerToTop takes the Tracer window the topmost window on the desktop.

▪

tracerToTop example

See the example for **tracerWrite**.

▪

tracerWrite procedure

See also   Example   System Type

Writes a message to the Tracer window.

Syntax

tracerWrite (const ***message*** String [, const ***message*** String] * **)**

Description

tracerWrite writes a message to the Tracer window.

▪

tracerWrite example

Logs a message to the Tracer window, then brings the Tracer window to the top layer of the desktop.

```
; logTracerMsg::pushButton
method pushButton(var eventInfo Event)
tracerWrite("Tracer hit by " + String(self.Name) +
            " at " + String(time()))              ; log a message
tracerToTop()            ; make the Tracer window the top-layer window
endMethod
```

twipsToPixels procedure

See also Example System Type

Converts screen coordinates from twips to pixels.

Syntax

twipsToPixels (const ***twips*** Point **)** Point

Description

twipstToPixels converts the screen coordinates specified by *twips* from twips to pixels.

▪

twipsToPixels example

See the example for **pixelsToTwips.**

▪

version procedure

[See also](#) [Example](#) [System Type](#)

Returns the version number of the currently used Paradox installation.

Syntax

`version ( ) String`

Description

version returns the version number of the currently used Paradox installation.

version example

The **pushButton** method for *showVersion* shows which Paradox version is in use.

```
; showVersion::pushButton
method pushButton(var eventInfo Event)
   msgInfo("FYI", "You are running version " + version() + ".")
endMethod
```

▪

winGetMessageID procedure

[See also](#) [Example](#) [System Type](#)

Returns the ID of a Windows message.

Syntax

`winGetMessageID ( const msgName String ) SmallInt`

Description

winGetMessageID returns the integer value of the Windows message represented by the string specified in *msgName*. Some often-used messages are WM_CLOSE (sent as a signal that a window or application should terminate); and WM_ACTIVATE (sent when a window is activated or deactivated).

Returns 0 if *msgName* is not recognized as a Windows message. Windows, not Paradox, determines which values of *msgName* are recognized. See your Windows programming documentation for more information.

Note: **winGetMessageID** should only be used by Windows programmers who understand how to work with Windows messages.

winGetMessageID example

Displays the integer value of the Windows message WM_LBUTTONDOWN.

```
method pushButton(var eventInfo event)
  var
    smMsgID   SmallInt
    stMsgName   String
  endVar

  stMsgName = "WM_LBUTTONDOWN"
  smMsgID = winGetMessageID(stMsgName)
  smMsgID.view(stMsgName) ; Displays 513 in Win32.
  ; The value may be different in other versions of Windows.
endMethod
```

▪

winPostMessage procedure

See also Example System Type

Posts a message to Windows.

Syntax

```
winPostMessage ( const hWnd LongInt, const msg LongInt, const wParam LongInt,
const lParam LongInt ) Logical
```

Description

winPostMessage posts a message to Windows. Unlike **winSendMessage,** which dispatches its message immediately, this method adds its message to the end of the Windows message queue; it is dispatched after messages (if any) ahead of it.

Valid arguments to this method are determined by Windows, not by Paradox. See your Windows programming documentation for more information.

Note: **winPostMessage** should only be used by Windows programmers who understand how to work with Windows messages.

▪

winPostMessage example

See the example for **winSendMessage**.

▪

winSendMessage procedure

See also Example System Type

Sends a message to Windows.

Syntax

`winSendMessage ( const hWnd LongInt, const msg LongInt, const wParam LongInt, const lParam LongInt ) LongInt`

Description

winSendMessage sends a message to Windows. Valid arguments to this method are determined by Windows, not by Paradox. See your Windows programming documentation for more information.

Note: **winSendMessage** should only be used by Windows programmers who understand how to work with Windows messages.

▪

winSendMessage example

Send a command to NOTEPAD.EXE, a text editor that comes with Windows. First, opens Notepad and calls **enumWindowNames** to create a table of data about windows currently open in your system. Then searches the table for the record of information about Notepad, and gets the handle for that window. Next, calls **winGetMessageID** to get the integer value of the command represented by the string "WM_CLOSE". Then calls **winSendMessage** with the window handle and command value as arguments. Dispatches the message to Windows, and closes Notepad. (You could call **winPostMessage** instead of **winSendMessage** to add the message to the end of the Windows message queue.)

```
method pushButton(var eventInfo Event)
  var
    tcOpenWin   TCursor
    tbOpenWin   Table
    stTbName    String
    siWinHandle,
    siWinMsgID  SmallInt
  endVar

  stTbName = ":PRIV:openWin"

  execute("Notepad.exe", No, ExeShowNormal)           ; Run Notepad.
  sleep(1000)                  ;  Pause so you can see what happens.

  enumWindowNames(stTbName)                        ; List open windows.

  tcOpenWin.open(stTbName)
  ; Locate the Notepad window in the list of names.
  if tcOpenWin.locatePattern("WindowName", "Notepad..") then
    ; Get the Windows handle for the Notepad window.
    siWinHandle = tcOpenWin."Handle"
    ; Get the Windows message ID for WM_CLOSE to close the window.
    siWinMsgID = winGetMessageId("WM_CLOSE")
    ; Send the specified message to the specified window.
    winSendMessage(siWinHandle, siWinMsgID, 0, 0)
  else
    errorShow()
  endIf
endmethod
```

▪

writeEnvironmentString procedure

See also   Example   System Type

Sets a variable in the Paradox copy of the DOS environment.

Syntax

`writeEnvironmentString ( const key String, const value String ) Logical`

Description

writeEnvironmentString sets a variable in the Paradox copy of the DOS environment. When Paradox starts it makes a copy of the DOS environment and this procedure writes to that copy. Changes made by this procedure to Paradox copy of the DOS environment are not written to the DOS environment.

Environment variables are assigned values using the DOS command SET. They control how DOS and some batch files and programs appear and work. Commonly used environment variables include PATH, PROMPT, and COMSPEC. For more information, consult your DOS manuals, especially the SET command.

▪

writeEnvironmentString example

See the example for **readEnvironmentString**.

writeProfileString procedure

See also Example System Type

Writes information about your system to a file.

Syntax

`writeProfileString (` `const` ***`fileName`*** `String, const` ***`section`*** `String, const` ***`key`*** `String, const` ***`value`*** `String` **`)`** `Logical`

Description

writeProfileString writes a value to a specified section of a file on your system. If you specify a file name without a path, this method searches for the file in the WINDOWS directory, not in :WORK: (the working directory).

Typically, you use this method to modify your WIN.INI file, so *fileName* would be WIN.INI. A section in WIN.INI is marked by a string bounded by square brackets on a line by itself (for example, [windows]). When you specify a *section*, however, you omit the brackets; that is, to specify the [windows] section, use "windows." Within a section, a string followed by = specifies *key* (for example, "Beep = "), but don't include the = when you specify *key*. Specify *value* by writing a string after the equal sign (=) in the key.

writeProfileString example

See the example for **readProfileString**.

Table type

Changes

A Table variable represents a description of a table. It is distinct from a TCursor, which is a pointer to the data, and from a table frame and a TableView, which are objects that display the data.

Using Table objects, you can add, copy, create, and index tables, do column calculations, get information about a table's structure, and more, but you can't edit records. Use a TCursor or a table frame (UIObject) for that. Some table operations require Paradox to create temporary tables. Paradox creates these tables in the private directory.

The **create**, **index**, and **sort** structures are basic language elements, not methods or procedures, but they're listed here because they operate on Table variables.

Methods for the Table type

fieldNo
fieldType
getGenFilter
getRange
index
isAssigned
isEmpty
isEncrypted
isShared
isTable
lock
nFields
nKeyFields
nRecords
protect
reIndex
reIndexAll
rename
setExclusive
setGenFilter
setIndex
setRange
setReadOnly
showDeleted
sort
subtract
tableRights
type
unAttach
unlock
unProtect
usesIndexes

Changes to Table type methods

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
createIndex	cCount
dropGenFilter	create
getGenFilter	enumFieldStruct
getRange	enumIndexStruct
setGenFilter	enumRefIntStruct
setRange	enumSecStruct
	fieldType
	setFilter was replaced by **setRange** which offers enhanced functionality and performance. Code that calls **setFilter** will continue to execute as before.

▪

add method/procedure

See also Example Table Type
Beginner

Adds the data in one table to another table.

Syntax

```
1. add ( const destTableName String [ , const append Logical [ , const update
Logical ] ] ) Logical
2. add ( const destTableVar Table [ , const append Logical [ , const update
Logical ] ] ) Logical
```

Description

add adds the data in a table to a destination table, which you can specify using a String (*destTableName* in syntax 1) or a Table variable (*destTableVar* in syntax 2). If the destination table does not exist, this method creates it. The source table and the destination table can be the same type or different types; in any case, the tables must have compatible field structures.

Arguments *append* and *update* can be True or False. When True, *append* adds records at the end of a non-indexed destination table, or at the appropriate place in an indexed destination table. When True, *update* compares records in both tables, and where key values match, replaces the data in the destination table. When both are True, records with matching key values are updated, and others are appended. These arguments are optional, but if you specify *update*, you must also specify *append.* If omitted, both are True. For example,

```
myTable.add(yourTable, False, True) ; specifies update
myTable.add(yourTable)              ; specifies update
and append by default
```

Key violations (including validity check violations), if any, are listed in KEYVIOL.DB in the user's private directory. This method overwrites an existing KEYVIOL.DB or creates one, if necessary. Following are some example statements.

When tables are keyed, **add** uses the keyed fields to determine which records to update and which to append. When the destination table is not keyed, **add** fails if *update* is True. Also, if the destination table is not keyed, the structure of the entire record in the source table must match the record structure in the destination table.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. add ( const sourceTableName String, const destTableName String [ , const
append Logical [ , const update Logical ] ] ) Logical
2. add ( const sourceTableName String, const destTableVar Table [ , const
append Logical [ , const update Logical ] ] ) Logical
```

▪

add example

For the following example, the **pushButton** method for *updateCust* runs a query from an existing file, then adds records from the *Answer* table to the *Customer* table.

```
; updateCust::pushButton
method pushButton(var eventInfo Event)
var
  newCust Query
  ansTbl Table
  destTbl String
endVar
destTbl = "Customer.db"

newCust.readFromFile("newCust.qbe")

if newCust.executeQBE() then         ; if the query succeeds
  ansTbl.attach(":PRIV:Answer.db")

  ; attempt to add Answer.db records to Customer.db
  if isTable(destTbl) then
    if NOT ansTbl.add(destTbl) then
      errorShow()
    endIf
  else
    msgStop("Error", "Can't find " + destTbl + ".")
  endIf
else
  errorShow("Query failed.")
endIf

endMethod
```

■

attach method

See also Example Table Type
Beginner

Associates a Table variable with a table on disk.

Syntax

```
1. attach ( const tableName String ) Logical
2. attach ( const tableName String, const db Database ) Logical
3. attach ( const tableName String, const tableType String ) Logical
4. attach ( const tableName String, const tableType String, const db Database
) Logical
```

Description

attach associates a Table variable with the table in *tableName*. Optional arguments *tableType* and *db* specify a table type ("Paradox" or "dBASE") and a database, respectively. If you don't specify *tableType*, ObjectPAL tries to determine the table type from the table name's file extension. If you don't specify *db*, ObjectPAL works in the default database.

This method fails if the value of *tableName* is not valid (for example, because the table name is invalid, or because the table name doesn't match the table type, or because of a conflict between the table name and the database name). This method returns True if successful; otherwise, it returns False.

Note: **attach** does not verify that *tableName* is a table, or even that the file exists. Use the **isTable** method to verify its existence.

To free a Table variable completely, use **unAttach**. To associate the Table variable with another table, just use **attach** again; the **unAttach** happens automatically.

attach example

In the following example, the *westTable* Table variable is attached to *Orders* so that **cSum** can be used with that Table variable. This example uses **isTable** to determine whether *Orders* exists in the default database before performing a calculation on the table.

```
; getWestTotal::pushButton
method pushButton(var eventInfo Event)
var
  westTable Table
  westTotal Number
endVar

if isTable("Orders.db")      then

   ; attach to Paradox table Orders in the default database
  westTable.attach("Orders", "Paradox")
   ; get total of Total Invoice field and store result in westTotal
  westTotal = westTable.cSum("Total Invoice")
   ; display total invoices
  msgInfo("Total Invoices", westTotal)

else
  msgInfo("Status", "Can't find Orders.db table.")
endIf

endMethod
```

▪

cAverage method/procedure

See also Example Table Type
Beginner

Returns the average value of a field (column) in a table.

Syntax

```
1. cAverage ( const fieldName String ) Number
2. cAverage ( const fieldNum SmallInt ) Number
```

Description

cAverage returns the average of values in the column of fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) **cAverage** handles blank values as specified by the **blankAsZero** setting for the session. This method respects the limits of restricted views set by **setRange** or **setGenFilter.**

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cAverage ( const tableName String, const fieldName String ) Number
2. cAverage ( const tableName String, const fieldNum SmallInt ) Number
```

cAverage example

The following example uses **cAverage** to calculate the average order size in the *Orders* table. This code is attached to the **pushButton** method for the *getAvgSales* button.

```
; getAvgSales::pushButton
method pushButton(var eventInfo Event)
var
  ordTbl   Table
  avgSales Number
endVar

ordTbl.attach("Orders.db")
avgSales = ordTbl.cAverage("Total Invoice") ; store average invoice total
                                            ; in avgSales
msgInfo("Average Order size", avgSales)     ; display avgSales in a dialog

endMethod
```

▪

cCount method/procedure

See also Example Table Type
Beginner

Returns the number of nonblank values in a field (column) of a table.

Syntax

```
1. cCount ( const fieldName String ) LongInt
2. cCount ( const fieldNum SmallInt ) LongInt
```

Description

cCount returns the number of values in the column (field) specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) **cCount** works for all field types. If the field is numeric, this method handles blank values as specified in the **blankAsZero** setting for the session. If the field is non-numeric, **cCount** returns the number of nonblank values in the column of fields. In version 5.0, this method was changed to return a LongInt instead of a Number.

This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cCount ( const tableName String, const fieldName String ) Number
2. cCount ( const tableName String, const fieldNum SmallInt ) Number
```

cCount example

For the following example, the **pushButton** method for *lineItemInfo* uses **cAverage** and **cCount** to perform calculations on the Qty field in LINEITEM.DB. The example attempts to place a write lock on the table so that another user on a network can not make changes to the table between the calls to **cAverage** and **cCount**. If the lock is unsuccessful, this code aborts the operation.

```
; lineItemInfo::pushButton
method pushButton(var eventInfo Event)
var
  lineTbl Table
  avgQty Number
  numItems LongInt
endVar
if lineTbl.attach("Lineitem.db") then
  if lineTbl.lock("Write") then           ; if write lock succeeds
    avgQty = lineTbl.cAverage("Qty")
    numItems = lineTbl.cCount(4)          ; assumes Qty is field 4
    lineTbl.unLock("Write")               ; unlock the table
    msgInfo("Average quantity",
            String(avgQty, "\nbased on ", numItems, " items."))
  else
    errorShow("Can't lock Lineitem table.")
  endIf
else
  errorShow("Can't attach to Lineitem table.")
endIf

endMethod
```

cMax method/procedure

[See also] [Example] [Table Type]
Beginner

Returns the maximum value of a field (column) in a table.

Syntax

```
1. cMax ( const fieldName String ) Number
2. cMax ( const fieldNum SmallInt ) Number
```

Description

cMax returns the maximum value in the column of fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) **cMax** respects the limits of restricted views set by **setRange** or **setGenFilter. cMax** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cMax ( const tableName String, const fieldName String ) Number
2. cMax ( const tableName String, const fieldNum SmallInt ) Number
```

cMax example

The following code displays the greatest amount in the Total Invoice field of the *Orders* table.

```
; showMaxOrder::pushButton
method pushButton(var eventInfo Event)
var
  orderTbl Table
endVar

if orderTbl.attach("Orders.db") then
  ; display maximum order in a dialog box
  msgInfo("Biggest Order in History", orderTbl.cMax("Total Invoice"))
else
  msgStop("Sorry", "Can't open Orders table.")
endIf

endMethod
```

▪

cMin method/procedure

See also Example Table Type
Beginner

Returns the minimum value of a field (column) in a table.

Syntax

```
1. cMin ( const fieldName String ) Number
2. cMin ( const fieldNum SmallInt ) Number
```

Description

cMin returns the minimum value in the column of fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views set by **setRange** or **setGenFilter.** **cMin** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cMin ( const tableName String, const fieldName String ) Number
2. cMin ( const tableName String, const fieldNum SmallInt ) Number
```

cMin example

The following code displays the smallest amount in the Total Invoice field of the *Orders* table.

```
; showMinOrder::pushButton
method pushButton(var eventInfo Event)
var
  orderTbl Table
endVar

if orderTbl.attach("Orders.db") then
    ; display smallest order in a dialog box
    msgInfo("Smallest Order in History", orderTbl.cMin("Total Invoice"))

else
  msgStop("Sorry", "Can't open Orders table.")
endIf

endMethod
```

■

cNpv method/procedure

[See also](#) [Example](#) [Table Type](#)
Beginner

Returns the net present value of a field (column), based on a specified discount or interest rate.

Syntax

```
1. cNpv ( const fieldName String, const discRate AnyType ) Number
2. cNpv ( const fieldNum SmallInt, const discRate AnyType ) Number
```

Description

cNpv returns the net present value of the column of fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views set by **setRange** or **setGenFilter.** **cNpv** handles blank values as specified in the **blankAsZero** setting for the session.

The calculation is based on *discRate*, expressed as a decimal (for example, 0.12 for 12 percent). This method calculates net present value using the following formula:

cNpv = *sum*(*p* = 1 to *n*) of *Vp* / (1 + *r*)*p*

where

n = number of periods, *Vp* = cash flow in *p*th period, and *r* = rate per period.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cNpv ( const tableName String, const fieldName String, const discRate
AnyType ) Number
2. cNpv ( const tableName String, const fieldNum SmallInt, const discRate
AnyType ) Number
```

cNpv example

The following defines a Table variable for the *GoodFund* table, then calculates the net present value for the Expected Return field. For this example, the net present value is calculated based on a monthly interest rate.

```
; calcNPV::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  goodFundNPV, apr Number
endVar
apr = .125                     ; annual percentage rate

tbl.attach("GoodFund.db")

; calculate net present value based on monthly interest rate
goodFundNPV = tbl.cNpv("Expected Return", (apr / 12))
msgInfo("Net present value", goodFundNPV)

endMethod
```

▪

compact method

See also Example Table Type
Beginner

Removes deleted records from a table.

Syntax
`compact ( [ const regIndex Logical ] ) Logical`

Description
Deleted records are not immediately removed from a dBASE table. Instead, they are flagged as deleted and kept in the table. **compact** removes deleted records. The optional argument *regIndex* specifies whether to regenerate indexes associated with the table, or just to update them. When *regIndex* is True, this method regenerates all indexes associated with the table: indexes specified by **usesIndexes,** and the .MDX index whose name matches the table name. When *regIndex* is False, indexes are not regenerated. If omitted, *regIndex* is True by default.

When records are deleted from a Paradox table, they can no longer be retrieved; they are permanently deleted. However the table file (and associated index files) contain "dead" space where the record was originally stored. **compact** removes dead space from Paradox files. When you use **compact** with a Paradox table, the *regIndex* argument is ignored: indexes are always regenerated.

This method fails if any locks have been placed on the table, or the table is open. This method returns True if successful; otherwise, it returns False.

▪

compact example

The following example demonstrates how **compact** affects indexes specified by **usesIndexes**. In this example, the *ordTbl* Table variable is attached to ORDERS.DBF and *salesTbl* is attached to SALES.DBF. Because *ordTbl* uses INDEX1.NDX and INDEX2.NDX (specified by **usesIndexes**), **compact** regenerates INDEX1.NDX and INDEX2.NDX if *regIndex* is True. For this example, *regIndex* is set to False, so **compact** affects only ORDERS.NDX.

```
; compactTbls::pushButton
method pushButton(var eventInfo Event)
var
  ordTbl, salesTbl Table
endVar

ordTbl.usesIndexes("index1.ndx", "index2.ndx")
ordTbl.attach("Orders.dbf")
ordTbl.compact(False)
  ; removes deleted records and fixes Orders.mdx

salesTbl.usesIndexes("index3.mdx")
salesTbl.attach("Sales.dbf")
salesTbl.compact()
  ; removes deleted records and regenerates all indexes

endMethod
```

▪

copy method/procedure

See also Example Table Type
Beginner

Copies a table.

Syntax

```
1. copy ( const destTable String ) Logical
2. copy ( const destTable Table ) Logical
```

Description

copy copies the records from a source table to a destination table specified in *destTable*. The data from the source table completely replaces the data in destination table. The source table and destination table can be different types of tables. If the destination table is open, the method fails.

This method tries, for the duration of the retry period, to place a write lock on the source table, and a full (exclusive) lock on the destination table. If either lock cannot be placed, the method fails.

For information, see Copying to a different table type in the User's Guide help.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. copy ( const sourceTable String, const destTable String ) Logical
2. copy ( const sourceTable String, const destTable Table ) Logical
```

▪

copy example

For the following example, the **pushButton** method for *backupCust* copies the *Customer* table to *CustBak*. If *CustBak* already exists in the current directory, this method asks the user for confirmation before overwriting it.

```
; backupCust::pushButton
method pushButton(var eventInfo Event)
var
  srcTbl  Table
  destTbl String
endVar
destTbl = "CustBak.db"
srcTbl.attach("Customer.db")

if isTable(destTbl) then      ; if "CustBak.db" exists
                              ; ask for confirmation
  if msgQuestion("Copy table", "Overwrite " + destTbl + "?") = "Yes" then
    return
  endIf
endIf
srcTbl.copy(destTbl)    ; this copies Customer.db to CustBak.db
; Does not copy .VAL file if all it contains is RI information.
endMethod
```

▪

create keyword

See also Example1 Example2 Table Type

Creates a table.

Syntax

```
create tableName [ as tableType ] [ database db ]
    [ [ like likeObject ]
      [ with fieldName : type [ , fieldName : type ] * ]
      [ where fieldID is newname [ , fieldID is newname ] * ]
      [ without fieldID [ , fieldID ] * ]
      [ struct fieldStructTable ]
      [ indexStruct indexStructTable ]
      [ refIntStruct refIntStructTable ]
      [ secStruct secStructTable ]
      [ languageDriver driverName ]
      [ versionLevel versionNumber ]
      ] *
    [ key fieldID [ , fieldID ] * ]
endCreate
```

Description

create creates a table, where *tableName* is the file name of the table to create. Unless an **as** clause explicitly specifies a table type (see below), **create** infers the table type from the *tableName* extension, if given. (.DB is a Paradox table and .DBF is a dBASE table.) For example, given "Orders.dbf" for *tableName*, **create** creates a dBASE table. If *tableName* does not include an extension, **create** creates a Paradox table.

If *tableName* exists, **create** tries, for the duration of the retry period, to place a full lock on *tableName*. If the lock cannot be placed, **create** fails.

The following clauses specify table attributes. They are optional, and can appear in any order within the **create** structure. The clauses are executed in the order they appear in the structure.

As *tableType* specifies the table format. Example:

```
AS "Paradox"
```

If **as** is omitted, **create** creates a Paradox table by default (unless the table resides on a SQL server. See the discussion of the **database** clause, below).

database *db* specifies a database variable (opened before creating the new table) that determines where the table will reside. If the database is on an SQL server, the table will be of a type appropriate for the server. If omitted, the table is created in the default database. For example:

```
DATABASE megaData
```

like *likeObject* specifies an opened TCursor, table name, or Table variable from which to borrow field names, field types, language driver, and version level. The **like** clause does not borrow validity checks, primary or secondary indexes, referential integrity information, or security information. (Use **struct**, **indexStruct**, **refIntStruct**, and **secStruct** options to borrow more detailed information.)

For example:

```
LIKE "Sales.dbf"        ; table name as a string
LIKE ordersTC           ; a TCursor variable pointing to ORDERS.DB
LIKE ordersTB           ; a Table variable pointing to ORDERS.DB
```

with "*fieldName*" : "*type*" adds one or more fields to the table structure. For example:

```
with "Last name" : "A20", "First name" : "A15", "Quantity" : "N"
```

Specify in *type* the field type for *fieldName*. Valid values for *type* vary depending on the type of table you are creating. The following tables list valid field specifications for Paradox and dBASE tables.

Paradox tables	**3.5 and earlier**	**4.5**	**5.0**

Alpha	A*nnn*	A*nnn*	A*nnn*
Number	N	N	N
Money	$	$	$
Date	D	D	D
Short	S	S	S
Memo	(none)	M*nnn*	M*nnn*
Formatted Memo	(none)	F*nnn*	F*nnn*
Binary	(none)	B*nnn*	B*nnn*
Graphic	(none)	G*nnn*	G*nnn*
OLE	(none)	O*nnn*	O*nnn*
Logical	(none)	(none)	L
Long Integer	(none)	(none)	I
Time	(none)	(none)	T
Timestamp	(none)	(none)	@
BCD	(none)	(none)	#
Autoincrement	(none)	(none)	+
Bytes	(none)	(none)	Y

dBASE tables	**III+**	**IV**	**V**
Character	C*nnn*	C*nnn*	C*nnn*
Number	N*nnn*	N*nnn*	N*nnn*
Date	D	D	D
Logical	L	L	L
Memo	M	M	M
Float	(none)	F*nnn.d*	F*nnn.d*
OLE	(none)	(none)	O
Binary	(none)	(none)	B

where *fieldID* **is** "*newName*" changes the name of one or more fields *fieldID* (name or number) to "*newName*" (String). Example:

```
where "Last name" IS "Customer last name", 2 IS "Customer first name"
```

without *fieldID* removes one or more fields (specified by name or number) from the structure. Example:

```
without 4, "Country code"
```

struct specifies in *fieldStructTable* an opened TCursor, table name, or Table variable from which to borrow the field-level structure. Unlike the **like** clause, **struct** borrows all validity check and primary key information. Use **enumFieldStruct** to generate *fieldStructTable* (or create it manually) before executing **create**. For example:

```
struct "CustFlds.db"
```

indexStruct specifies in *indexStructTable* an opened TCursor, table name, or Table variable from which to borrow secondary index information. Use **enumIndexStruct** to generate *indexStructTable* (or create it manually) before executing **create**. For example:

```
indexStruct "CustIndx.db"
```

refIntStruct specifies an opened TCursor, table name, or Table variable from which to borrow referential integrity information. Use **enumRefIntStruct** to generate *refIntStructTable* (or create it manually) before executing **create**. For example:

```
refIntStruct "Cust_Ref.db"
```

secStruct specifies in *secStructTable* an opened TCursor, table name, or Table variable from which to borrow security information. Use **enumSecStruct** to generate *secStructTable* (or create your own) before executing **create**. For example:

```
secStruct "Cust_Sec.db"
```

Note: When you use **secStruct**, Paradox automatically protects the table with the master password *secret*. Refer About password security in the User's Guide help for information about master passwords

languageDriver (added in version 5.0) specifies in *driverName* the internal name of a language driver to use with the table. A language driver determines the table's sort order and available character set. Choose an item below to display a list of language drivers.

- Language drivers for Paradox tables
- Language drivers for dBASE tables

versionLevel (added in version 5.0) specifies in *versionNumber* what level of table to create. Valid values for *versionNumber* are listed in the following table.

Table type	**Version number**
Paradox	▪ 3 specifies a level 3 table corresponding to that created for Paradox 3.5 and earlier (Paradox Engine version 2). ▪ 4 specifies a level 4 table corresponding to Paradox for Windows 4.5 and earlier and Paradox for DOS 4.0 and 4.5 (Paradox Engine version 3). ▪ 5 specifies a level 5 table corresponding to Paradox for Windows 5.0.
dBASE	▪ 3 specifies a dBASE III table. ▪ 4 specifies a dBASE IV table. ▪ 5 specifies a dBASE for Windows table.

key *fieldID* specifies one or more key fields. You must specify key fields in order from left to right. For example:

```
key "Last name", "First name"
```

Fields are created in the order you specify them, whether explicitly using a **with** clause, or as implied by one or more **like** clauses. **where** and **without** clauses have no meaning unless preceded by a **like** clause.

Note: **create** is not a method, so dot notation, as in the following statement, is inappropriate:

```
tableVar.create() ; This is inappropriate.
```

Instead, use = to assign the **create** structure to a Table variable.

■

create example 1

The following example creates the Paradox table PARTS.DB. The table has three fields: Part number, Part name, and Quantity. It has one key field: Part number.

```
; createParts::pushButton
method pushButton(var eventInfo Event)
var
  newParts Table
  partsTV TableView
endVar
if isTable("Parts.db") then
  if msgQuestion("Confirm",
                 "Parts.db exists. Overwrite it?") <> "Yes" then
    return
  endIf
endIf

newParts = create "Parts.db"
             WITH "Part number" : "A20",
                  "Part name" : "A20",
                  "Quantity" : "S"
             KEY "Part number"
           endCreate

partsTV.open("Parts.db")      ; Open the new table.
endMethod
```

■

create example 2

The following examples show two ways to create the dBASE table NEWSALES.DBF using the same structure as the dBASE table SALES.DBF.

```
; version 1
var
  newSales Table
endVar
newSales = CREATE "Newsales.dbf"
             LIKE "Sales.dbf"
           ENDCREATE

; version 2
var
  newSales Table
  salesTC TCursor
endVar
salesTC.open("Sales.dbf")
newSales = CREATE
             LIKE salesTC
           ENDCREATE
```

The next example uses the **struct** option to borrow field-level information, including primary keys and validity checks, for use in a new table. (See **enumFieldStruct** for more information.)

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  custTbl, newCustTbl Table
  custTC TCursor
endVar

custTbl.attach("Customer.db")
if custTbl.isTable() then

  if custTbl.enumFieldStruct("CustFlds.db") then

    ; Open a TCursor for CustFlds table.
    custTC.open("CustFlds.db")
    custTC.edit()

    ; This loop scans through the CustFlds table and
    ; changes ValCheck definitions for every field.
    scan custTC :
      custTC."_Required Value" = 1   ; Make all fields required.
    endScan

    ; Now create NEWCUST.DB and borrow field names,
    ; ValChecks and key fields from CUSTFLDS.DB.
    newCustTbl = CREATE "NewCust.db"
                   STRUCT "CustFlds.db"
                 ENDCREATE

    ; NEWCUST.DB requires that all fields be filled

  else
    msgStop("Error", "Can't get field structure for Customer table.")
  endIf

else
  msgStop("Error", "Can't find Customer table.")
endIf

endMethod
```

Language drivers for Paradox tables

The following table shows the language drivers you can use for Paradox tables, along with the code page for each driver. Use the internal name to specify *driverName*.

Note: Internal language driver names are case-sensitive.

Driver name	Internal	Language/DOS Code Page
Paradox 'ascii'	ASCII	English (US)/437
Paradox 'hebrew'	HEBREW	Hebrew
Paradox 'intl'	INTL	International/437
Paradox 'intl850'	INTL850	International/850
Paradox 'nordan'	NORDAN	Danish-Norwegian
Paradox 'turk'	TURK	Turkish
Paradox ANSI 'turk'	ANTURK	Turkish
Paradox ANSI China	ANCHINA	Chinese
Paradox ANSI Cyrillic	ANCYRR	Russian
Paradox ANSI Czech	ANCZECH	Czech
Paradox ANSI Greek	ANGREEK1	Greek
Paradox ANSI HEBREW	ANHEBREW	ANSI Hebrew
Paradox ANSI Hun DC	ANHUNDC	Hungarian
Paradox ANSI Intl	ANSIINTL	ANSI International
Paradox ANSI Intl850	ANSII850	ANSI International 850
Paradox ANSI Korea	ANKOREA	Korean
Paradox ANSI Nordan4	ANSINOR4	ANSI Danish-Norwegian/4
Paradox ANSI Polish	ANPOLISH	Polish
Paradox ANSI Slovene	ANSISLOV	Yugoslavia
Paradox ANSI Spanish	ANSISPAN	ANSI Spanish
Paradox ANSI Swedfin	ANSISWFN	ANSI Swedish-Finnish
Paradox ANSI Thai	ANTHAI	ANSI Thai
Paradox China 437	CHINA	Chinese/437
Paradox Cyrr 866	CYRR	Russian/866
Paradox Czech 852	CZECH	Czech/852
Paradox Czech 867	CSKAMEN	Czech/867
Paradox ESP 437	SPANISH	Spanish/437
Paradox Greek GR437	GRCP437	Greek/437
Paradox Hun 852 DC	HUN852DC	Hungarian/852
Paradox ISL 861	ICELAND	Iceland/861
Paradox Korea 949	KOREA	Korean/949
Paradox NORDAN	NORDAN	Danish-Norwegian/865
Paradox NORDAN40	NORDAN40	Danish-Norwegian/865
Paradox Polish 852	POLISH	Polish/852

Paradox Slovene 852	SLOVENE	Yugoslavia/852
Paradox SWEDFIN	SWEDFIN	Swedish-Finnish/437
Paradox Thai 437	THAI	Thai 437

Language drivers for dBASE tables

The following table shows the language drivers you can use for dBASE tables. Use the internal name to specify *driverName*.

Note: Internal language driver names are case-sensitive.

Driver name	Internal	Language
dBASE CHN pc437	DB437CN0	Chinese
dBASE CSY cp852	DB852CZ0	Czech
dBASE CSY cp867	DB867CZ0	Czech
dBASE DAN cp865	DB865DA0	Danish
dBASE DEU cp437	DB437DE0	German
dBASE DEU cp850	DB850DE0	German
dBASE ELL GR437	DB437GR0	Greek
dBASE ENG cp437	DB437UK0	English (U.K)
dBASE ENG cp850	DB850UK0	English (U.K)
dBASE ENU cp437	DB437US0	English (U.S.)
dBASE ENU cp850	DB850US0	English (U.S.)
dBASE ESP cp437	DB437ES1	Spanish
dBASE ESP cp850	DB850ES0	Spanish
dBASE FIN cp437	DB437FI0	Finnish
dBASE FRA cp437	DB437FR0	French
dBASE FRA cp850	DB850FR0	French
dBASE FRC cp850	DB850CF0	French (Can.)
dBASE FRC cp863	DB863CF1	French (Can.)
dBASE HUN cp852	DB852HDC	Hungarian
dBASE ITA cp437	DB437IT0	Italian
dBASE ITA cp850	DB850IT0	Italian
dBASE KOR cp949	DB949KO0	Korean
dBASE NLD cp437	DB437NL0	Dutch
dBASE NLD cp850	DB850NL0	Dutch
dBASE NOR cp437	DB437NO0	Norwegian
dBASE NOR cp865	DB865NO0	Norwegian
dBASE PLK pc852	DB852PO0	Polish
dBASE PTB cp850	DB850PT0	Portuguese (Bra.)
dBASE PTG cp860	DB860PT0	Portuguese
dBASE RUS cp866	DB866RU0	Russian
dBASE SLO cp852	DB852SL0	Yugoslavian
dBASE SVE cp437	DB437SV0	Swedish
dBASE SVE cp850	DB850SV0	Swedish
dBASE TRK cp857	DB857TR0	Turkish

dBASE TWN cp437	DB437TW0	Taiwanese

▪

createIndex method

See also Example1 Example2 Table Type

Creates an index for a table.

Syntax

`1. createIndex ( const attrib DynArray[ ] AnyType, const fieldNames Array[ ] String ) Logical`

`2. createIndex ( const attrib DynArray[ ] AnyType, const fieldNums Array[ ] SmallInt ) Logical`

Description

createIndex creates an index for a table using attributes specified in the DynArray *attrib* and the field names (or numbers) specified in the Array *fieldNames* (or *fieldNums*). This method is provided as an alternative to the **index** structure, and performs the same task. It can be useful when you don't know the index structure beforehand (for example, when the information is supplied by the user).

Each key of the DynArray must be a string, and the value of each corresponding item is described in the following table. You do not have to include all the keys to use **createIndex**. Any key you omit is assigned the corresponding default value.

String value	Description
MAINTAINED	If True, the index is incrementally maintained. That is, after a table is changed, only that portion of the index affected by the change is updated. If False, Paradox does not maintain the index automatically. Maintained indexes typically result in better performance. Default = False (Paradox tables only).
PRIMARY	If True, the index is a primary index. If False, it's a secondary index. Default = False (Paradox tables only).
CASEINSENSITIVE	If True, the index ignores differences in case. If False, it considers case. Default = False (Paradox tables only).
DESCENDING	If True, the index is sorted in descending order, from highest values to lowest. If False, it is sorted in ascending order. Default = False.
UNIQUE	If True, records with duplicate values in key fields are ignored. If False, duplicates are included and available.
IndexName	A name used to identify this index. No default value, unless you're creating a secondary, case-sensitive index on a single field, in which case the default value is the field name. For dBASE tables, the index name must be a valid DOS file name. If you do not specify an extension, .NDX is added automatically.
TagName	The name of the index tag associated with the index specified in *indexName* (dBASE tables only).

For information on indexes, see About keys and indexes in tables in the User's Guide help.

▪

createIndex example 1

The following example builds a maintained secondary index for a Paradox table named CUSTOMER.DB. If the *Customer* table cannot be found, or cannot be locked, this method aborts the operation.

```
method pushButton(var eventInfo Event)
var
   stTbName      String
   tbCust         Table
   arFieldNames   Array[3] String
   dyAttrib      DynArray[]AnyType
endVar

stTbName = "Customer.db"

arFieldNames[1] = "Customer No"
arFieldNames[2] = "Name"
arFieldNames[3] = "Street"

dyAttrib["PRIMARY"]     = False
dyAttrib["MAINTAINED"] = True
dyAttrib["IndexName"]  = "NumberNameStreet"

if isTable(stTbName) then
  tbCust.attach(stTbName)
   if not tbCust.lock("FULL") then
      errorShow()
      return
   endIf

   if not tbCust.createIndex(dyAttrib, arFieldNames) then
      errorShow()
   endIf

; This createIndex statement has the same effect
; as the following INDEX structure:
{
  INDEX tbCust                  ; Create index for Customer.db.
     MAINTAINED
    ON "Customer No", "Name", "Street"
  ENDINDEX
}

else
  errorShow()
endIf

endMethod
```

▪

createIndex example 2

The following example adds a unique index tag named StatProv to the production index for the dBASE table CUSTOMER.DBF.

```
method pushButton(var eventInfo Event)
var
   tbCust          Table
   arFieldNames    Array[1] String
   dyAttrib       DynArray[]AnyType
endVar

arFieldNames[1] = "STATE_PROV"

dyAttrib["UNIQUE"]     = True
dyAttrib["MAINTAINED"] = True

; A dBASE index name must be a valid DOS file name.
; If an extension is omitted, .NDX is appended automatically.

dyAttrib["IndexName"] = "Customer.Mdx"
dyAttrib["TagName"] = "StatProv"

if isTable("Customer.dbf") then
  tbCust.attach("Customer.dbf")

  if not tbCust.createIndex(dyAttrib, arFieldNames) then
       errorShow()
  endIf

; This createIndex statement has the same effect
; as the following INDEX structure:
{
  INDEX tbCust                          ; Create index for Customer.dbf.
       UNIQUE
    ON "STATE_PROV"                     ; Index on this field.
    TAG "StatProv" OF "Customer.dbf"   ; Name the tag "StatProv".
  ENDINDEX
}

else
  errorShow()
endIf

endMethod
```

▪

cSamStd method/procedure

See also   Example   Table Type
Beginner

Returns the sample standard deviation of a field (column) of a table.

Syntax

```
1. cSamStd ( const fieldName String ) Number
2. cSamStd ( const fieldNum SmallInt ) Number
```

Description

cSamStd returns the sample standard deviation for the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered form left to right, beginning with 1). This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cSamStd** handles blank values as specified in the **blankAsZero** setting for the session.

The calculation is based on the sample variance. The sample (as opposed to population) standard deviation is calculated using the formula:

sqrt((*sampVar*) * (*n*/(*n*-1)))

where

sampVar = cSamVar(*tableName*, *fieldName*)

n = cCount(*tableName*, *fieldName*).

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cSamStd ( const tableName String, const fieldName String ) Number
2. cSamStd ( const tableName String, const fieldNum SmallInt ) Number
```

cSamStd example

The following example calculates the sample standard deviation of test scores for the Winter quarter. This code is attached to the **pushButton** method for *showSamStd*.

```
; showSamStd::pushButton
method pushButton(var eventInfo Event)
   const
      kTbName = "winter"
   endConst

   var
      tbWinter   Table
      nuSamStd    Number
   endVar

   tbWinter.attach(kTbName)
   nuSamStd = tbWinter.cSamStd("TestScore")
   nuSamStd.view()
endMethod
```

▪

cSamVar method/procedure

[See also](#) [Example](#) [Table Type](#)
Beginner

Returns the sample variance of a field (column) in a table.

Syntax

```
1. cSamVar ( const fieldName String ) Number
2. cSamVar ( const fieldNum SmallInt ) Number
```

Description

cSamVar returns the sample variance of the column of fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views set by **setRange** or **setGenFilter.** **cSamVar** handles blank values as specified in the **blankAsZero** setting for the session.

The sample (as opposed to population) variance is calculated using the formula:

cVar(*tableName*, *fieldName*) * (*n/*(*n - 1*)) where:

n = cCount(*tableName, fieldName*)

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cSamVar ( const tableName String, const fieldName String ) Number
2. cSamVar ( const tableName String, const fieldNum SmallInt ) Number
```

▪

cSamVar example

The following example uses both forms of the syntax to calculate the sample variance of two different fields in the *Answer* table. This code is attached to the **pushButton** method for *showSamVar*.

```
; showSamVar::pushButton
method pushButton(var eventInfo Event)
var
   empTbl Table
   tblName String
   calcSalary, calcYears Number
endVar
tblName = "Answer"

empTbl.attach(tblName)
calcSalary = empTbl.cSamVar("Salary")  ; get sample variance for Salaries
calcYears  = empTbl.cSamVar(2)         ; assume "Years in service" is field 2
msgInfo("Sample Variance",             ; display info in a dialog box
        "Salaries : " + String(calcSalary,
        "\nYears in service : ", calcYears))

endMethod
```

▪

cStd method/procedure

See also Example Table Type
Beginner

Returns the standard deviation of a field (column) in a table.

Syntax

```
1. cStd ( const fieldName String ) Number
2. cStd ( const fieldNum SmallInt ) Number
```

Description

cStd returns the population standard deviation of the column of fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views set by **setRange** or **setGenFilter.** The calculation is based on the variance; see **cVar**. This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cStd ( const tableName String, const fieldName String ) Number
2. cStd ( const tableName String, const fieldNum SmallInt ) Number
```

cStd example

For the following example, the **pushButton** method for *thisButton* calculates the population standard deviation for two separate fields and displays the results in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myTable Table
  test1, test2 Number
endVar
myTable.attach("scores.db")
test1 = myTable.cStd("Test1")
test2 = myTable.cStd(2)         ; assumes Test2 is field 2
msgInfo("Standard Deviation",
        "Test1 results : " + String(test1) + "\n" +
        "Test2 results : " + String(test2))
endMethod
```

cSum method/procedure

See also Example Table Type
Beginner

Returns the sum of the values in a field (column) of a table.

Syntax

```
1. cSum ( const fieldName String ) Number
2. cSum ( const fieldNum SmallInt ) Number
```

Description

cSum returns the sum of the values in the column of fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cSum** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cSum ( const tableName String, const fieldName String ) Number
2. cSum ( const tableName String, const fieldNum SmallInt ) Number
```

▪

cSum example

For the following example, the **pushButton** method for *sumOrders* uses both forms of **cSum** syntax to calculate totals for two fields in ORDERS.DB.

```
; sumOrders::pushButton
method pushButton(var eventInfo Event)
var
  orderTbl Table
  orderTotal, amtPaid Number
  tblName String
endVar
tblName = "Orders"

orderTbl.attach(tblName)
orderTotal = orderTbl.cSum("Total Invoice")
amtPaid    = orderTbl.cSum(7)    ; assumes Amount Paid is field 7
msgInfo("Order Totals",
        "Total Orders : " + String(orderTotal) + "\n" +
        "Total Receipts : " + String(amtPaid))

endMethod
```

▪

cVar method/procedure

See also Example Table Type
Beginner

Returns the variance of a field in a table.

Syntax

```
1. cVar ( const fieldName String ) Number
2. cVar ( const fieldNum SmallInt ) Number
```

Description

cVar returns the population variance of the column of fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cVar** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. cVar ( const tableName String, const fieldName String ) Number
2. cVar ( const tableName String, const fieldNum SmallInt ) Number
```

cVar example

For the following example, the **pushButton** method for thisButton calculates the population variance deviation for two separate fields and displays the results in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myTable Table
  test1, test2 Number
endVar
myTable.attach("scores.db")
test1 = myTable.cVar("Test1")
test2 = myTable.cVar(2)       ; assumes Test2 is field 2
msgInfo("Population Variance",
        "Test1 results : " + String(test1) + "\n" +
        "Test2 results : " + String(test2))

endMethod
```

▪

delete method/procedure

See also Example Table Type
Beginner

Deletes a table.

Syntax
`delete ( )` `Logical`

Description
delete deletes a table without asking for confirmation. Compare this method to **empty**, which removes data from a table but does not delete it.

This method fails if the table is open or is locked.

DOS
The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax
`delete (` `const` ***`tableName`*** `String` **`)`** `Logical`

▪

delete example

The following code deletes ANSWER.DB from the private directory if it exists.

```
; delAnswer::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  tblName String
endVar

tblName = privDir() + "\\Answer.db"

tbl.attach(tblName)
if tbl.isTable() then
  tbl.delete()
  message(tblName, " deleted.")
else
  message("Can't find ", tblName, ".")
endIf

endMethod
```

▪

dropGenFilter method

See also Example Table Type

Drops (removes) the filter criteria associated with a Table variable.

Syntax

`dropGenFilter ( )` `Logical`

Description

dropGenFilter drops (removes) the filter criteria associated with a Table variable, leaving it unfiltered. Indexes and ranges (if any) remain in effect.

▪

dropGenFilter example

In the following example, a form contains a button called *btnCACustomers*. The **pushButton** method for *btnCACustomers* attaches a Table variable to the *Customer* table, sets a filter criteria, and stores the value in the number variable *nSubTotal*. Then **dropGenFilter** is used to drop the filter and the total number of records is stored into the number variable *nTotal*. Finally, a message information box is displayed showing the number of customers in California compared to the total number of customers.

```
;btnCACustomers :: pushButton
method pushButton(var eventInfo Event)
   var
      tbl            Table
      dyn            DynArray[] AnyType
      nTotal,
      nSubTotal   Number
   endVar

   tbl.attach("CUSTOMER.DB")

   dyn["State/Prov"] = "CA"
   tbl.setGenFilter(dyn)
   nSubTotal = tbl.cCount("State/Prov")   ;Get customers in CA.

   tbl.dropGenFilter()
   nTotal = tbl.nRecords()         ;Get all customers.

   msgInfo("Customer Analysis",  string(nSubtotal) + " out of " +
string(nTotal) + " reside in California.")
endMethod
```

dropIndex method

See also Example Table Type

Deletes an index file associated with a table.

Syntax

```
1. (Paradox tables) dropIndex ( const indexName String ) Logical
2. (dBASE tables) dropIndex ( const indexName String
                        [ , const tagName String ] ) Logical
```

Description

dropIndex deletes a specified index file or index tag.

When working with a Paradox table, *indexName* specifies a secondary index; you can specify an empty string in *indexName* to drop the primary index.

When working with a dBASE table, you can use *indexName* to specify a .NDX file, or use *indexName* and *tagName* to specify a .MDX file and an index tag.

You must obtain exclusive rights to the table (by calling the Table method **setExclusive**) before calling **dropIndex**.

dropIndex fails if the index you're trying to delete is currently being used, or if the table is open.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

▪

dropIndex example

For the following example, the **pushButton** method for *thisButton* deletes the CustName tag from a .MDX file.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  salesTbl Table
endVar

salesTbl.attach("Sales.dbf")          ; Sales.dbf is a dBASE table
if isTable(salesTbl) then             ; if salesTbl is a table

  ; Get exclusive access to the table.
  salesTbl.setExclusive(Yes)
  ; delete CustName tag from index2.mdx file
  if salesTbl.dropIndex("index2.mdx", "CustName") then
    msgInfo("Status", "CustName index deleted.")
  else
    msgInfo("Error", "Can't drop CustName from Index2.")
  endIf

else
  msgStop("Stop!", "Could not find Sales.dbf table.")
endIf

endMethod
```

▪

empty method/procedure

See also Example Table Type
Beginner

Removes all records from a table in a database.

Syntax
```
empty ( ) Logical
```

Description

empty removes all records from a table without asking for confirmation. This operation cannot be undone. This method returns True if it succeeds; otherwise, it returns False.

empty removes information from the table, but does not delete the table itself. Compare this method to **delete**, which does delete the table.

This method first tries to gain exclusive rights to the table. If exclusive rights are not possible, **empty** tries to place a write lock on the table.

If exclusive rights are possible, this method deletes all records in the table at once. If only a write lock is possible, **empty** must delete each record one at a time. (This can be slow for large tables.)

If **empty** is able to gain exclusive rights to a dBASE table, all records are deleted and the table is compacted (records are permanently removed). If only a write lock is possible, this method flags all records as deleted, but does not remove them from the table. (Records can be undeleted from a dBASE table if they have not been removed with the **compact** method.)

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax
```
empty ( const tableName String ) Logical
```

▪

empty example

The following example prompts the user for confirmation before deleting all records from the *Scratch* table.

```
; tblEmpty::pushButton
method pushButton(var eventInfo Event)
var
  tblName String
  tblVar Table
endVar
tblName = "Scratch.db"

tblVar.attach(tblName)
if isTable(tblName) then
  if msgQuestion("Empty?", "Empty " + tblName + " ?") = "Yes" then

    if tblVar.empty() then
      message("All " + tblName + " records have been deleted.")
    else
      errorShow()
    endIf

  endIf
else
  errorShow()
endIf
endMethod
```

▪

enumFieldNames method

See also Example Table Type

Fills an array with the names of fields in a table.

Syntax

`enumFieldNames ( var fieldArray Array[ ] String ) Logical`

Description

enumFieldNames fills *fieldArray* with the names of the fields in a table. You must declare *fieldArray* as a resizeable array before calling this method. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

enumFieldNames example

For the following example, the **pushButton** method for the *btnEnumFields* button stores field names in a resizeable array, then uses **view** to display the contents of the array.

```
; btnEnumFields::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  arFieldNames Array[] AnyType
endVar

tbl.attach("Sales.dbf")
if tbl.isTable() then
  tbl.enumFieldNames(arFieldNames)
  arFieldNames.view()
else
  errorShow()
endIf

endMethod
```

▪

enumFieldNamesInIndex method

[See also](#) [Example](#) [Table Type](#)

Fills an array with the names of fields in a table's index.

Syntax

```
1. (Paradox tables) enumFieldNamesInIndex ( [ const indexName String, ] var
fieldArray Array[ ] String ) Logical
2. (dBASE tables) enumFieldNamesInIndex ( [ const indexName String, [ const
tagName String, ] ]
var fieldArray Array[ ] String ) Logical
```

Description

enumFieldNamesInIndex fills *fieldArray* with the names of the fields in a table's index, as specified in *indexName*. You must declare *fieldArray* as a resizeable array before calling this method. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

When working with a dBASE table, the argument *tagName* is required to specify an index tag within a .MDX file.

If omitted, *indexName* corresponds to the index currently being used.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

enumFieldNamesInIndex example

For the following example, the **pushButton** method for the *showIndexFlds* button stores field names in a resizeable array, then uses **view** to display the contents of the array.

```
; showIndexFlds::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  fieldNames Array[] String
endVar

tbl.attach("Sales.dbf")
if tbl.isTable() then
  tbl.enumFieldNamesInIndex("DateIndx", "byDate", fieldNames)
  ; display the index field names for byDate in DateIndx
  fieldNames.view()
else
  msgStop("Stop", "Couldn't find Sales.dbf.")
endIf

endMethod
```

▪

enumFieldStruct method

See also Example Table Type

Lists the field structure of a table.

Syntax

```
1. enumFieldStruct ( const tableName String ) Logical
2. enumFieldStruct ( inMem TCursor ) Logical
```

Description

enumFieldStruct lists the field structure of a Table variable. Syntax 1 creates a Paradox table; syntax 2 (added in version 5.0) stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* exists, this method overwrites it without confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory. You can supply *tableName* to the **struct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In syntax 2 (added in version 5.0), the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Using syntax 2 may result in faster performance because the information is stored in system memory without the overhead of disk access.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field name	Field type	Description
Field Name	A31	Name of field.
Type	A31	Data type of field.
Size	S	Size of field.
Dec	S	Number of decimal places, or 0 if field type doesn't support decimal places.
Key	A1	Is it a key field? * = key field, blank = not key field.
_Required Value	A1	Is the field required? T = required, N (or blank) = Not required.
_Min Value	A255	Field's minimum value, if specified; otherwise blank.
_Max Value	A255	Field's maximum value, if specified; otherwise blank.
_Default Value	A255	Field's default value, if specified; otherwise blank.
_Picture Value	A175	Field's picture, if specified; otherwise blank.
_Table Lookup	A255	Name of lookup table. Includes full path if the lookup table is not in :WORK:
_Table Lookup Type	A1	Type of lookup table. 0 (or blank) = no lookup table, 1 = Current field + private 2 = All corresponding + no help 3 = Just current field + help and field 4 = All corresponding + help
_Invariant Field ID	S	Field's ordinal position in table (first field = 1, second field = 2, etc.)

Once *tableName* is created, you can modify values in the table, then use it with the **struct** option in the **create** command.

enumFieldStruct example

For the following example, assume that you want a new table called *NewCust* that is similar to the *Customer* table. However, you want all of the fields in *NewCust* to be required fields. To accomplish this, the following code uses **enumFieldStruct** to load a new table (CUSTFLDS.DB) with the field-level information from *Customer*. The code then scans through *CustFlds* and modifies the field definitions so that each record describes a field that will be required. *CustFlds* is then supplied in the **struct** clause of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  custTbl, newCustTbl Table
  custTC TCursor
endVar

custTbl.attach("Customer.db")
if custTbl.isTable() then

  if custTbl.enumFieldStruct("CustFlds.db") then

    ; Open a TCursor for CustFlds table.
    custTC.open("CustFlds.db")
    custTC.edit()

    ; This loop scans through the CustFlds table and
    ; changes ValCheck definitions for every field .
    scan custTC :
      custTC."_Required Value" = 1     ; Make all fields required.
    endScan

    ; Now create NEWCUST.DB and borrow field names,
    ; ValChecks and key fields from CUSTFLDS.DB.
    newCustTbl = CREATE "NewCust.db"
                   STRUCT "CustFlds.db"
                 endCreate

    ; NEWCUST.DB requires that all fields be filled.

  else
    msgStop("Error", "Can't get field structure for Customer table.")
  endIf

else
  msgStop("Error", "Can't find Customer table.")
endIf

endMethod
```

▪

enumIndexStruct method

See also Example Table Type

Lists the structure of a table's secondary indexes.

Syntax

```
1. enumIndexStruct ( const tableName String ) Logical
2. enumIndexStruct ( inMem TCursor ) Logical
```

Description

enumIndexStruct lists the structure of a table's secondary indexes. Syntax 1 creates a Paradox table; syntax 2 (added in version 5.0) stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. For dBASE tables, this method lists the structure of the indexes associated with the table by the **usesIndexes** method. If *tableName* exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory. You can supply *tableName* to the **indexStruct** option in a **create** statement to borrow secondary indexes for use in the new table.

In syntax 2, the structure information is stored in a TCursor variable *inMem* that you pass as an argument. Using syntax 2 may result in faster performance, because the information is stored in system memory without the overhead of disk access.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field Name	Field Type	Description
infoHeader	A1	If Y, this record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field. If N, this record is not a header.
szName	A255	Index name, including path.
szTagName	A31	Tag name, no path (dBASE only).
szFormat	A31	Optional index type, e.g., BTREE, HASH
bPrimary	A1	Y if the index is primary; otherwise blank.
bUnique	A1	Y if the index is unique; otherwise blank.
bDescending	A1	Y if the index is descending; otherwise blank.
bMaintained	A1	Y if the index is maintained; otherwise blank.
bCaseInsensitive	A1	Y if the index is not case-sensitive; otherwise blank.
bSubset	A1	Y if the index is a subset index (dBASE only); otherwise blank.
bExpIdx	A1	Y if the index is an expression index (dBASE only); otherwise blank.
iKeyExpType	N	Key type of index expression (dBASE only).
szKeyExp	A220	Key expression for expression index (dBASE only).
szKeyCond	A220	Subset condition for subset index (dBASE only).
FieldNo	N	Ordinal position of key field in table.
FieldName	A31	Name of key field.

For dBASE tables, *tableName* includes information for indexes which would be used if the table were open. To specify which indexes to associate to a Table variable, use the **usesIndexes** method, then call **enumIndexStruct** to create a table that list those indexes.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

▪

enumIndexStruct example

For the following example, assume that you want a new table called *NewCust* that is similar to the *Customer* table. However, you don't want to borrow referential integrity or security information. To accomplish this, the following code uses **enumFieldStruct** and **enumIndexStruct** to generate two tables: CUSTFLDS.DB and CUSTINDX.DB. *CustFlds* and *CustIndx* are then supplied to the **struct** and **indexStruct** clauses of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  custTbl, newCustTbl Table
  custTC TCursor
endVar

custTbl.attach("Customer.db")
if custTbl.isTable() then

  custTbl.enumFieldStruct("CustFlds.db")
  custTbl.enumIndexStruct("CustIndx.db")

  ; Now create NEWCUST.DB.
  ; Borrow field names, ValChecks, and key fields from CUSTFLDS.DB.
  ; Borrow secondary indexes from CUSTINDX.DB.
  newCustTbl = CREATE "NewCust.db"
                 STRUCT "CustFlds.db"
                 INDEXSTRUCT "CustIndx.db"
               ENDCREATE

else
  msgStop("Error", "Can't find Customer table.")
endIf

endMethod
```

▪

enumRefIntStruct method

See also Example Table Type

Lists a table's referential integrity information.

Syntax

```
1. enumRefIntStruct ( const tableName String ) Logical
2. enumRefIntStruct ( inMem TCursor ) Logical
```

Description

enumRefIntStruct lists referential integrity information for a Table variable. Syntax 1 creates a Paradox table; syntax 2 (added in version 5.0) stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory. You can supply *tableName* to the **refIntStruct** option in a **create** statement to borrow referential integrity information for use in the new table.

In syntax 2, the structure information is stored in a TCursor variable *inMem* that you pass as an argument. Using syntax 2 may result in faster performance because the information is stored in system memory without the overhead of disk access.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field name	Type	Description
infoHeader	A1	If Y, this record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field. If N, this record is not a header.
RefName	A31	A name to identify this referential integrity constraint.
OtherTable	A255	Name (including path) of the other table in the referential integrity relationship.
Slave	A1	Y if the table is slave, not master (i.e., table is dependent); otherwise blank.
Modify	A1	Specifies update rule: Y = Cascade, blank = Prohibit.
Delete	A1	Specifies delete rule: blank = Prohibit. Paradox does not support cascading deletes for Paradox or dBASE tables.
FieldNo	N	Ordinal position of the field in this table involved in a referential integrity relationship.
aiThisTabField	A31	Name of the field in this table involved in a referential integrity relationship.
Other FieldNo	N	Ordinal position of the field in the other table involved in a referential integrity relationship.
aiOthTabField	A31	Name of the field in the other table involved in a referential integrity relationship.

enumRefIntStruct example

The following example uses **enumRefIntStruct** to write CUSTOMER.DB referential integrity information to the *CustRef* table. Then, the code supplies *CustRef* to the **refIntStruct** clause in a **create** statement.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
   var
      tb1, tb2 Table
   endVar

   tb1.attach("Customer.db")
   tb1.enumRefIntStruct("CustRef.db")
   tb1.enumFieldStruct("CustFlds.db")

   try
      tb2 = CREATE "NewCust.db"
                     STRUCT "CustFlds.db"
                     REFINTSTRUCT "CustRef.db"
                  ENDCREATE
   onFail
      errorShow()
   endTry

endMethod
```

▪

enumSecStruct method

See also Example Table Type

Lists a table's security information.

Syntax

```
1. enumSecStruct ( const tableName String ) Logical
2. enumSecStruct ( inMem TCursor ) Logical
```

Description

enumSecStruct lists the security information (access rights) of a Table variable. Syntax 1 creates a Paradox table; syntax 2 (added in version 5.0) stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory. You can supply *tableName* to the **secStruct** option in a **create** statement to borrow security information for use in the new table.

In syntax 2, the structure information is stored in a TCursor variable *inMem* that you pass as an argument. Using syntax 2 may result in faster performance because the information is stored in system memory without the overhead of disk access.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field name	Type	Description
infoHeader	A1	If Y, this record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field. If N, this record is not a header.
iSecNum	N	Number to identify security description (first description = 1).
eprvTable	N	Table privilege value.
eprvTableSym	A10	Table privilege name.
iFamRights	N	Family rights value.
iFamRightsSym	A10	Family rights name.
szPassword	A31	Password.
fldNum	N	Ordinal position of field in table.
aprvFld	N	Field privilege value.
aprvFldSym	A10	Field privilege name.

▪

enumSecStruct example

The following example creates a new table based on the security information associated with the *Secrets* table. The code uses **enumSecStruct** to write security information to the *SecInfo* table, then uses the table to create the *MySecrts* table.

```
; getSecrets::pushButton
method pushButton(var eventInfo Event)
var
  tb1, tb2 Table
endVar

tb1.attach("Secrets.db")
tb1.enumSecStruct("SecInfo.db")

tb2 = CREATE "MySecrts.db"
        LIKE "Secrets.db"
        SECSTRUCT "SecInfo.db"
      ENDCREATE

endMethod
```

Privilege values and names for Table::enumSecStruct

The following table lists numeric values and symbolic names for table and field privileges.

Value	Name	Description
0	None	No privileges.
1	ReadOnly	Read-only field or table.
3	Modify	Read and modify field or table.
7	Insert	Insert + all of the above privileges (table only).
15	InsDel	Delete + all of the above privileges (table only).
31	Full	Full rights (table only).
255	Unknown	Privileges unknown.

Family rights values and names for Table::enumSecStruct

The following table lists numeric values and symbolic names for family rights.

Value	Name	Description
0	NoFamRights	No family rights.
1	FormRights	Can change forms only.
2	RptRights	Can change reports only.
4	ValRights	Can change val checks only.
8	SetRights	Can change image settings.
15	AllFamRights	All of the above.

▪

familyRights method

See also Example Table Type

Tests for a user's ability to create or modify objects in a table's family.

Syntax

```
familyRights ( const rights String) Logical
```

Description

familyRights returns True if you have rights to the type of object specified in *rights*; otherwise, it returns False. *rights* is a single-letter string▪either "F" (form), "R" (report), "S" (image settings), or "V" (validity checks)

▪that indicates the type of object you are interested in. This method preserves functionality required by Paradox 3.5 tables, which had the concept of a table family. The concept does not apply to tables created in versions of Paradox after 3.5.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
familyRights( const tableName String, rights AnyType ) Logical
```

▪

familyRights example

The following example indicates in a dialog box whether you have "F" rights to CUSTOMER.DB.

```
; showFRights::pushButton
method pushButton(var eventInfo Event)
var
  custTB Table
endVar

custTB.attach("Orders.db")
if custTB.isTable() then
  msgInfo("Rights", "Form Rights: " +
          String(custTB.familyRights("F")))
  ;displays True if you have Form rights to Orders.db
else
  msgStop("Error", "Can't find Orders.db.")
endIf

endMethod
```

▪

fieldName method/procedure

See also Example Table Type

Returns the name of a field in a table, given a field number.

Syntax

```
fieldName ( const fieldNum SmallInt ) String
```

Description

fieldName returns the name of the field specified in *fieldNum.* If *fieldNum* is greater than the number of fields in the table, **fieldName** returns an empty string.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
fieldName ( const tableName String, const fieldNum SmallInt ) String
```

▪

fieldName example

The following example uses **fieldName** to display the name of field number two in the *Answer* table. This code is attached to the built-in **pushButton** method of a button.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  fldName, tblName String
  fldNum SmallInt
endVar
tblName = "Answer.db"
fldNum = 2

tbl.attach(tblName)
if isTable(tbl) then
  fldName = tbl.fieldName(fldNum)    ; store name of field 2 in fldName
  msgInfo("The name of field " + String(fldNum) + " is:", fldName)
else
  msgStop("Sorry", "Can't find " + tblName + " table.")
endIf

endMethod
```

fieldNo method/procedure

See also   Example   Table Type

Returns the position of a field in a table.

Syntax

`fieldNo ( const fieldName String ) SmallInt`

Description

fieldNo returns the position of *fieldName* in a table, or 0 if *fieldName* is not found. Fields are numbered from left to right, beginning with 1.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

`fieldNo ( const tableName String, const fieldName String ) SmallInt`

▪

fieldNo example

This code displays the field number of the Date field if it exists in the Orders table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ord Table
  fldNo SmallInt
endVar

ord.attach("Orders.db")
fldNo = ord.fieldNo("Date")

if fldNo = 0 then
  msgInfo("Orders table", "Date is not a field in this table.")
else
  msgInfo("Orders table", "Date is field number " + String(fldNo))
endIf

endMethod
```

■

fieldType method/procedure

See also Example Table Type

Returns the type of a field in a table.

Syntax

```
1. fieldType ( const fieldName String ) String
2. fieldType ( const fieldNum SmallInt ) String
```

Description

fieldType returns the data type of a field. If the field is not found, this method returns "Unknown". The following tables (updated for version 5.0) list the possible return values for Paradox and dBASE tables:

Paradox field type	Return value
Alpha	ALPHA
Autoincrement	AUTOINCREMENT
BCD	BCD
Binary	BINARY
Bytes	BYTES
Date	DATE
Formatted Memo	FMTMEMO
Graphic	GRAPHIC
Logical	LOGICAL
Long Integer	LONG
Memo	MEMO
Money	MONEY
Number	NUMBER
OLE	OLE
Short	SHORT
Time	TIME
Timestamp	TIMESTAMP

dBASE field type	Return value
BINARY	BINARY
CHARACTER	CHARACTER
DATE	DATE
FLOAT	FLOAT
LOGICAL	LOGICAL
MEMO	MEMO
NUMBER	NUMERIC
OLE	OLE

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. **fieldType (** const ***tableName*** String, const ***fieldName*** String **)** String
2. **fieldType (** const ***tableName*** String, const ***fieldNum*** SmallInt **)** String

fieldType example

The following example uses a dynamic array to store the type of each field in the *BioLife* table, then displays the contents of the dynamic array in a dialog box.

```
; showFldTypes::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  i SmallInt
  fldTypes DynArray[] AnyType
  tblName String
endVar
tblName = "BioLife.db"

if isTable(tblName) then
  tblVar.attach(tblName)
    ; This FOR loop loads the DynArray with BioLife.db field types.
  for i from 1 to tblVar.nFields()
    fldTypes[tblVar.fieldName(i)] = tblVar.fieldtype(i)
  endFor
    ; Now show the contents of the DynArray.
  fldTypes.view(tblName + " field types")
else
  msgStop("Sorry", "Can't find " + tblName + " table.")
endIf
endMethod
```

▪

getGenFilter method

See also Example1 Example2 Table Type

Retrieves the filter criteria associated with a Table variable.

Syntax

1. getGenFilter (*criteria* DynArray[ ] AnyType **)** Logical

2. getGenFilter (*criteria* Array[ ] AnyType [**, *fieldName*** Array[ ] AnyType] **)** Logical

3. getGenFilter (*criteria* String **)** Logical

Description

getGenFilter retrieves the filter criteria associated with a Table variable. This method does not return values directly; instead, it assigns them to a DynArray variable (syntax 1) or to two Array variables (syntax 2) that you declare and include as arguments.

In syntax 1, the DynArray *criteria* lists fields and filtering conditions as follows: the index is the field name or number (depending on how it was set), and the item is the corresponding filter expression.

In syntax 2, the Array *criteria* lists filtering conditions, and the optional Array *fieldName* lists corresponding field names. If you omit *fieldName*, conditions apply to fields in the order they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to the second field, and so on).

If the arrays used in syntax 2 are resizeable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

In syntax 3, the filter criteria is assigned to a String variable *criteria* that you must declare and pass as an argument.

getGenFilter example 1

In this example, the **pushButton** method for a button named *btnShowFilter* uses **getGenFilter** to fill a DynArray *dyn* with a table's filter criteria. Then it checks the DynArray to see if the current criteria filters the State/Prov field with a value of "CA", and resets the filter if necessary.

```
;btnShowFilter :: pushButton
method pushButton(var eventInfo Event)
   var
      custTb   Table
      dyn            DynArray[] AnyType
      keysAr      Array[] AnyType
      stFilterFld,
      stCriteria   String
   endVar

   stFilterFld = "State/Prov"
   stCriteria  = "CA"
   custTb.attach("Customer")

   custTb.getGenFilter(dyn)   ; Get filter information.

   dyn.getKeys(keysAr)
   if keysAr.contains(stFilterFld) then
         if dyn[stFilterFld] = stCriteria then
               return                 ; Filter is set correctly.
         endIf
   else
         dyn.empty()               ; Set filter criteria correctly.
         dyn[stFilterFld] = stCriteria
         custTb.setGenFilter(dyn)
   endIf
endMethod
```

▪

getGenFilter example 2

In this example, a form contains a custom method called *cmGetOrders*. This custom method is used by a button called *btnViewOrders* to set a filter and then return the number of records in the filter. Following is the code attached to the form.

```
;Form :: cmGetOrders
method cmGetOrders(var tbl Table) Number
   var
      dynCurrent   DynArray[] AnyType
      dynNew      DynArray[] AnyType
   endVar

   dynNew["Ship Via"] = "UPS"      ;Set filter criteria.
   dynNew["Total Invoice"] = "> 10000"
   tbl.getGenFilter(dynCurrent)   ;Get the current criteria.

   if dynCurrent <> dynNew then   ;If current criteria is not
      tbl.setGenFilter(dynNew)      ;the same as new criteria,
   endIf                     ;then set new criteria.

   return(tbl.cCount("Order No"))   ;Return number of orders.
endMethod
```

Following is the code attached to the button. It associates a Table variable with a table, then calls the custom method attached to the form to operate on the data.

```
;btnViewOrders :: pushButton
method pushButton(var eventInfo Event)
   var
      tbl   Table
   endVar

   tbl.attach("ORD_JUN.DB")
   view(cmGetOrders(tbl), "UPS orders over $10,000 in June")

   tbl.attach("ORD_JUL.DB")
   view(cmGetOrders(tbl), "UPS orders over $10,000 in July")
endMethod
```

getRange method

See also Example Table Type

Retrieves the values that specify a range for a Table variable.

Syntax

`getRange ( var rangeVals Array[ ] String ) Logical`

Description

getRange retrieves the values that specify a range for a Table variable. This method does not return values directly; instead, it assigns them to an Array variable that you declare and include as an argument. The array values describe the range criteria, as listed in the following table.

Number of array items	Range specification
No items (empty array)	No range criteria associated with the Table variable.
One item	Specifies a value for an exact match on the first field of the index.
Two items	Specifies a range for the first field of the index.
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size *n*, specify exact matches on the first *n*-2 fields of the index. The last two array items specify a range for the *n*-1 field of the index.

If the array is resizeable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

▪

getRange example

In the following example, **getRange** is used on a Table variable *tbl* to test if the current range criteria is the same as the new range criteria. If it is not, then the new range is set using **setRange**.

```
;btnSetRange :: pushButton
method pushButton(var eventInfo Event)
   var
      arGet         Array[2] Anytype
      arSet         Array[2] Anytype
   endVar

   arSet[1] = "A"
   arSet[2] = "B"

   ;The following assumes a Table variable
   ;is declared and used elsewhere.

   tbl.getRange(arGet)      ;Get the current range.
   if arGet <> arSet then   ;Compare current range with new.
      tbl.setRange(arSet)      ;Show records starting with A.
   endIf
endMethod
```

▪

index keyword

See also Example1 Example2 Table Type

Creates an index on the specified fields of a table.

Syntax

```
1. index
      [ maintained ] tableDesc on fieldID
    endIndex
2. index tableDesc
      [ maintained ]            (Paradox)
      [ primary ]               (Paradox)
      [ caseInsensitive ]       (Paradox)
      [ descending ]            (Paradox and dBASE)
      [ unique ]                (dBASE)
      on
        { fieldDesc [ , fieldDesc ] [ to indexName ]
          |
        { keyExp
          to ndxFileName|tag tagName [ of mdxFileName ]
            |
          for condition } }
    endIndex
```

Description

index generates a primary or secondary index on the specified fields of a table. Paradox uses the index to speed up queries and searches that access those fields.

For Paradox tables, the keywords **maintained**, **primary**, and **caseInsensitive** are available. The **primary** keyword specifies a primary index (key), which is required before you can create any secondary indexes. If the table already has a primary index and you create another one, the new primary index replaces the old one. A primary index must be declared on one or more consecutive fields, beginning with the first field in the table. Memo fields, formatted memo fields, OLE fields, and Graphic fields cannot be indexed.

Secondary indexes can be either maintained (created using the **maintained** keyword) or non-maintained. Paradox updates a maintained index as records are added, deleted, or changed. A non-maintained index is only updated when it is in use. If you use the **maintained** keyword for Paradox tables and specify a non-keyed table to index, **index** fails. For dBASE tables, all opened index files are automatically maintained.

The **caseInsensitive** keyword makes an index ignore case. A primary index must be case-sensitive (in other words, you cannot use the **caseInsensitive** keyword when creating a primary index), and it must be declared on one or more consecutive fields, beginning with the first field. For Paradox tables, a case-sensitive maintained index on a single field must have the same name as that field. A case-*in*sensitive maintained index on a single field must *not* have the same name as that field.

The **on** clause, which specifies which fields to use, has two forms: one for Paradox tables, and one for dBASE tables.

For Paradox tables, use

on *fieldDesc* [, *fieldDesc*] to *indexName*

where *fieldDesc* specifies one or more field names or field numbers, and *indexName* specifies the name of the index file. Other methods use this name to refer to the index.

For dBASE tables, use

keyExp to *ndxFileName*|tag *tagName* [of *mdxFileName*]

which lets you choose between a .NDX file or a tag in a .MDX file. If *mdxFileName* is omitted, the default

.MDX file name is the same as the table. A dBASE table can only be indexed on one field or expression.

In multiuser applications, **index** automatically places a full lock on the table while it is being indexed. If the table has been locked by another user or application, the command is continuously retried for the duration of the currently set retry period. If the lock cannot be obtained by the end of the period, a **index** fails. You can use the **lock** method to make certain that you can lock the table *before* you use the **index** command.

It can be convenient to develop your applications without worrying about indexes, then introduce them where appropriate to speed up queries and searches.

In the following situations, the index command will not successfully complete:

- Too many indexes already exist (maximum of 255 for a single table).
- An index being defined is already in use.

index is not a method, so dot notation, as in

```
tableVar.index()
```

is inappropriate. Instead, you create an index structure to specify how to index the table.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

▪

index example 1

The following example builds a primary index for a Paradox table named CUSTOMER.DB. If the Customer table can not be found, or cannot be locked, this method aborts the **index** operation. If this code successfully indexes the table, the code enumerates indexed fields to an array and displays the contents of the array in a dialog box.

```
; newCustKeys::pushButton
method pushButton(var eventInfo Event)
var
  tblToIndex String
  tblVar Table
  indexedFlds Array[] String
endVar
tblToIndex = "Customer.db"

if isTable(tblToIndex) then
  tblVar.attach(tblToIndex)
  if not tblVar.lock("Full") then
    msgStop("Stop!", "Can't lock " + tblToIndex + " table.")
    return
  endIf
  INDEX tblVar              ; create new primary index for Customer.db
    PRIMARY
    ON "Customer No", "Name", "Street"
  ENDINDEX

    ; now display Customer's keyed fields in a dialog box
  tblVar.enumFieldNamesInIndex(indexedFlds)
  indexedFlds.view("Primary key fields for " + tblToIndex)

else
  msgStop("Stop!", "Can't find " + tblToIndex + " table.")
endIf

endMethod
```

▪

index example 2

The following example builds a maintained secondary index named *CityState* for the Paradox table, CUSTOMER.DB. If successful, this code enumerates the indexed field names to an array and displays them in a dialog box.

```
; cityStateIndex::pushButton
method pushButton(var eventInfo Event)
var
  tblToIndex String
  tblVar Table
  indexedFlds Array[] String
  tv TableView
endVar
tblToIndex = "Customer.db"

if isTable(tblToIndex) then
  tblVar.attach(tblToIndex)
  if not tblVar.lock("Full") then
    msgStop("Stop!", "Can't lock " + tblToIndex + " table.")
    return
  endIf

  INDEX tblVar              ; create secondary index for Customer.db
    MAINTAINED              ; maintain index incrementally
    ON "City", "State/Prov" ; index on these two fields
    TO "CityState"          ; name the index "CityState"
  ENDINDEX

    ; now display Customer's keyed fields in a dialog box
  tblVar.enumFieldNamesInIndex("CityState", indexedFlds)
  indexedFlds.view("Fields in the CityState index")

else
  msgStop("Stop!", "Can't find " + tblToIndex + " table.")
endIf

endMethod
```

▪

isAssigned method

See also Example Table Type

Reports whether a Table variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

isAssigned returns True if a Table variable has an assigned value; otherwise, it returns False. You can assign a value to a Table variable using **create** or **attach**.

Note: A return value of True does not guarantee that the table exists. For example, the following code displays True in a dialog box:

```
var tb Table endVar
tb.attach("zxcv.qw")                    ; attach to some nonsense file name
msgInfo("Assigned?", tb.isAssigned()) ; displays True
displays True in the dialog box.
```

isAssigned example

The following example tests whether the *tblVar* Table variable is assigned before attaching to a table. The following code goes in the Var window for the *thisForm* form:

```
; thisForm::var
var
  tblVar
endVar
```

The following code is attached to the **pushButton** method for the *thisButton* button. In this code, if *tblVar* is not already assigned, it is attached to the *Orders* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

if NOT tblVar.isAssigned() then
  tblVar.attach("Orders.db")
else
  msgStop("Error", "Can't attach tblVar to Orders.db")
endIf

endMethod
```

▪

isEmpty method/procedure

See also Example Table Type

Reports whether a table contains any records.

Syntax

```
isEmpty ( ) Logical
```

Description

isEmpty returns True if there are no records in a table; otherwise, it returns False.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
isEmpty ( const tableName String ) Logical
```

isEmpty example

For the following example, the **pushButton** method for the *rptRecNo* button displays the number of records in the *Orders* table. If *Orders* is empty, this code alerts the user that the table is empty.

```
; rptRecNo::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar
tblName = "Orders.db"

if isTable(tblName) then
  tblVar.attach(tblName)
  if tblVar.isEmpty() then     ; if Orders.db table is empty
    msgStop("Hey!", tblName + " table is empty!")
  else
    msgInfo(tblName + " table has", String(tblVar.nRecords()) + " records")
  endIf
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

▪

isEncrypted method/procedure

See also Example Table Type

Reports whether a table is encrypted.

Syntax

```
isEncrypted ( [ const tableName String ] ) Logical
```

Description

isEncrypted returns True if a table is password-protected; otherwise, it returns False. A TCursor can't be opened on an encrypted table until the password is presented, either interactively or using the Session type method **addPassword.** This method does not report whether a user has access rights to the table▪use **tableRights** for that.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
isEncrypted ( const tableName String ) Logical
```

▪

isEncrypted example

The following example uses **isEncrypted** to determine whether the *Secrets* table is protected (encrypted); if it is the code prompts the user to enter a password.

```
method pushButton(var eventInfo Event)
   const
      kTbName = "Secrets"
   endConst

   var
      tbSecret Table
      tvSecret TableView
   endvar

   tbSecret.attach(kTbName)

   ; If the table is encrypted, prompt the
   ; user for the password.

   if tbSecret.isEncrypted() then
      menuAction(MenuFileTablePasswords)
   endIf

   if not tvSecret.open(kTbName) then
      errorShow("Could not open " + kTbName)
   endIf

endMethod
```

isShared method/procedure

See also Example Table Type

Reports whether a table is currently shared.

Syntax

```
isShared ( ) Logical
```

Description

isShared returns True if a table is being shared by another user on a network; otherwise, it returns False. **isShared** does not report whether a table is being shared by another session.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
isShared ( const tableName String ) Logical
```

▪

isShared example

In the following example, a Table variable is attached to the Customer table. This code uses **setExclusive** to give the user exclusive rights to *Customer* then uses **isShared** to demonstrate the effect **setExclusive** has on tables in a multiuser environment.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar
tblName = "Customer.db"

tblVar.attach(tblName)

tblVar.setExclusive(True)   ; give user exclusive rights to Customer.db
if tblVar.isShared() then   ; this is never True!
                            ; exclusive tables can't be shared
  msgStop("", "This message will never appear!")
else
  msgInfo("Multiuser Status", tblName + " is not shared.")
endIf

endMethod
```

▪

isTable method/procedure

See also Example Table Type

Reports whether a table exists in a database.

Syntax

```
isTable ( ) Logical
```

Description

isTable returns True if the Table variable represents a table that can be opened; otherwise, it returns False.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
isTable ( const tableName String ) Logical
```

isTable example

The following example uses **isTable** to verify that the *Customer* table exists before doing anything with the table. If *Customer* exists in the default database, this code stores *Customer* field names in an array, then displays the contents of the array in a dialog box.

```
; showCustFlds::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
  fldNames Array[] AnyType
endVar
tblName = "Customer.db"

tblVar.attach(tblName)
if isTable(tblVar) then
  tblVar.enumFieldNames(fldNames)
  fldNames.view(tblName + " fields")
else
  msgStop("Stop!", "Can't find " + tblName + " table.")
endIf

endMethod
```

▪

lock method

See also Example Table Type
Beginner

Locks a specified table.

Syntax
lock **(** const ***lockType*** String **)** Logical

Description
lock attempts to place a lock on the table, where *lockType* is one of the following String values, listed in order of decreasing strength and increasing concurrency.

String value	Description
Full	The current session has exclusive access to the table. No other session can open the table. Cannot be used with dBASE tables.
Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

If successful, this method returns True; otherwise, it returns False.

▪

lock example

The following example attaches a Table variable to *Customer*, places an exclusive lock on the table, then uses **reIndex** to rebuild the *Phone_Zip* index. Once the index is rebuilt, this code unlocks *Customer* so other users on a network can gain access to the table.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

if isTable(pdoxTbl) then
  tblVar.attach(pdoxTbl)
  if tblVar.lock("Exclusive") then    ; Try to lock the table.
    tblVar.reIndex("Phone_Zip")       ; Rebuild Phone_Zip index.
    tblVar.unLock("Exclusive")        ; Unlock the table.
  else
    msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
  endIf
else
  msgStop("Sorry", "Can't find " + pdoxTbl + " table.")
endIf
endMethod
```

▪

nFields method/procedure

See also Example Table Type

Returns the number of fields in a table.

Syntax

```
nFields ( ) LongInt
```

Description

nFields returns the number of fields in a table.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
nFields ( const tableName String ) LongInt
```

nFields example

For the following example, the **pushButton** method for *thisButton* displays the number of fields in the *BioLife* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
endVar

tblVar.attach("BioLife.db")
msgInfo("BioLife", "BioLife has " +
        String(tblVar.nFields(), " fields."))

endMethod
```

■

nKeyFields method/procedure

See also Example Table Type

Returns the number of fields in the primary or current index for a table.

Syntax

```
nKeyFields ( ) LongInt
```

Description

nKeyFields returns the number of fields in the current index for a table. Use TCursor::**getIndexName** to get the name of the current index.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
nKeyFields ( const tableName String ) LongInt
```

▪

nKeyFields example

The following example reports the number of primary key fields in a Paradox table (ORDERS.DB) and the number of primary key fields in the LastName tag of the SCORES.MDX index for a dBase table (SCORES.DBF).

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  pdoxTbl, dBaseTbl Table
  nkf LongInt
endVar

pdoxTbl.attach("Orders.db")
nkf = pdoxTbl.nKeyFields()        ; number of key fields in the primary index
msgInfo("Orders", "Orders.db has " + String(nkf) + " key fields.")


dBaseTbl.attach("Scores.dbf")
dBaseTbl.setIndex("Scores", "LastName")
nkf = dBaseTbl.nKeyFields() ; key fields in LastName tag
msgInfo("Scores.dbf", "LastName tag has "
                      + String(nkf) + " key fields.")

endMethod
```

▪

nRecords method/procedure

See also Example Table Type

Returns the number of records in a table.

Syntax

```
nRecords ( ) LongInt
```

Description

nRecords returns the number of records in the table associated with a Table variable.

Note: When you call **nRecords** after setting a filter, the returned value does not represent the number of records in the filtered set. To get that information, use **cCount.** When you call **nRecords** after setting a range, the returned value represents the number of records in the set defined by the range.

When working with a dBASE table, **nRecords** counts deleted records if **showDeleted** is turned on. If **showDeleted** is turned off, deleted records are not counted.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
nRecords ( const tableName String ) LongInt
```

▪

nRecords example

The following example prompts the user for confirmation before deleting all records from the *Scratch* table. If the user does not confirm the action, this code uses **nRecords** to indicate how many records exist in SCRATCH.DB.

```
; tblEmpty::pushButton
method pushButton(var eventInfo Event)
var
  tblName String
  tblVar Table
endVar
tblName = "Scratch.db"

if isTable(tblName) then
  tblVar.attach(tblName)
  if msgYesNoCancel("Confirm", "Empty " + tblName + " table?") = "Yes" then
    tblVar.empty()
    message("All " + tblName + " records have been deleted.")
  else
    message(tblname + " has " + String(tblVar.nRecords()) + " records.")
  endIf
else
  msgInfo("Error", "Can't find " + tblName + " table.")
endIf
endMethod
```

▪

protect method/procedure

See also   Example   Table Type

Encrypts and assigns an owner password to a table.

Syntax

`protect ( const password String ) Logical`

Description

protect assigns an owner password to a table. The maximum length of a password is 31 characters. A protected table is encrypted and cannot be accessed without presenting the password specified in *password*. If the table already has a password, **protect** fails.

Once a table is protected, you can use the **addPassword** method to present the password of a protected table, and the **removePassword** method to withdraw the password and reprotect the table. *password* is case-sensitive; a table protected with "Sesame" won't open for "SESAME".

Do not confuse **protect** with **lock**: **protect** encrypts tables, while **lock** controls simultaneous access to tables.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

`protect ( const tableName String, const password String ) Logical`

▪

protect example

For the following example, the **pushButton** method for *protectSecrets* password-protects the *Secrets* table in the default database.

```
; protectSecrets::pushButton
method pushButton(var eventInfo Event)
var
  secretData Table
endVar

secretData.attach("Secrets.db")
if not secretData.isEncrypted() then
  secretData.protect("Get007") ; Password-protect table with "Get007"
endIf

endMethod
```

reIndex method

See also Example Table Type

Rebuilds specified index files.

Syntax

`1. (Paradox tables) **reIndex ( const *indexName* String ) Logical`**

`2. (dBASE tables) **reIndex ( const *indexName* String [ const *tagName* String ] ) Logical`**

Description

reIndex rebuilds an index (or index tag) that is not automatically maintained. When working with a Paradox table, use *indexName* to specify an index. When working with a dBASE table, use *indexName* to specify a .NDX file, or *indexName* and *tagName* to specify an index tag in a .MDX file. This method requires exclusive access to the table.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

▪

reIndex example

The following example attaches a Table variable to Customer (a Paradox table), places an exclusive lock on the table, then uses **reIndex** to rebuild the *Phone_Zip* index.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

tblVar.attach(pdoxTbl)
if tblVar.lock("Exclusive") then  ; Try to lock the table.
  tblVar.reIndex("Phone_Zip")     ; Rebuild Phone_Zip index.
  tblVar.unLock("Exclusive")      ; Unlock the table.
else
  msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
endIf

endMethod
```

▪

reIndexAll method

See also Example Table Type

Rebuilds all index files associated with a table.

Syntax

```
reIndexAll ( ) Logical
```

Description

reIndexAll rebuilds all index files associated with a table. This method requires exclusive rights to the table to rebuild a maintained index, and it requires a write lock to rebuild a non-maintained index.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

reIndexAll example

For the following example, the **pushButton** method for a button attempts to place an exclusive lock on the *Customer* table. If **lock** is successful, this code rebuilds all indexes for the *Customer* table then unlocks the table.

```
; reindexAllCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

tblVar.attach(pdoxTbl)
if tblVar.lock("Exclusive") then    ; attempt to lock Customer.db
  tblVar.reIndexAll()               ; rebuild all Customer.db indexes
  tblVar.unLock("Exclusive")        ; unlock the table
else
  msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
endIf

endMethod
```

▪

rename method/procedure

See also   Example   Table Type
Beginner

Renames a table.

Syntax

```
rename ( const destTableName String ) Logical
```

Description

rename changes the name of a table to *destTableName*. If the table named by *destTableName* exists, an error results.

This method tries, for the duration of the retry period, to place a full lock on the table. If the lock cannot be placed, an error results.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
rename ( const tableName String, const destTableName String ) Logical
```

▪

rename example

The following code renames CUSTOMER.DB to OLDCUST. If *OldCust* exists, this example offers the user an opportunity to abort the operation.

```
; renameCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  oldName, newName String
endVar

oldName = "Customer.db"
newName = "OldCust.db"

tblVar.attach(oldName)
if tblVar.isTable() then
  if isTable(newName) then
    if msgQuestion("Confirm", newName + " exists. Overwrite it?") <> "Yes"
then
      message("Operation canceled.")
      return
    endIf
  endIf
  tblVar.rename(newName)
  message(oldName + " renamed to " + newName)
else
  msgStop("Stop!", "Can't find " + oldName + " table.")
endIf

endMethod
```

■

setExclusive method

See also Example Table Type

Specifies whether to give the user exclusive rights to a table when it is opened.

Syntax

```
setExclusive ( [ const yesNo Logical ] )
```

Description

setExclusive specifies in *yesNo* whether to open a table with shared or exclusive rights. This method does not place any locks on the table■an exclusive lock (more powerful than a full lock) is placed on the table only when it is opened.

By default, tables are opened in shared mode. Optional argument *yesNo* specifies whether to set exclusive rights: a value of Yes requests exclusive rights so that no other user can read or write to the table, a value of No allows the table to be opened in shared mode. If omitted, *yesNo* defaults to Yes.

setExclusive example

The following example demonstrates how **setExclusive** affects access rights to a table. The code defines a Table variable for the *Customer* table, then calls **setExclusive** so *Customer* is opened exclusively. Then, a TCursor is opened for *Customer*. If the TCursor is successfully opened, it has exclusive rights to the table and the code calls the TCursor method **lockStatus** to indicate that an exclusive lock has been placed on *Customer*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tc     TCursor
endvar

tblVar.attach("Customer.db")
if tblVar.isTable() then
  ; set exclusive rights for the Table variable
  tblVar.setExclusive()

  ; attempt to open a TCursor on Customer.db
  ; if successful, tc has exclusive rights to Customer.db
  if tc.open(tblVar) then

    ; if tc.open was successful, this message indicates
    ; that tc has 1 exclusive lock on Customer.db
    msgInfo("Lock Status", tc.lockStatus("Exclusive"))

  else
    ; else open failed
    msgInfo("Status", "Can't open Customer.db")
  endIf

else
  msgInfo("Status", "Can't find Customer.db table.")
endIf

if tc.isAssigned() then    ; if the TCursor was opened
  tc.close()               ; close tc
now Customer.db is not
                           ; locked and can be opened by another user
endIf

endMethod
```

▪

setGenFilter method

See also Example1 Example2 Table Type

Specifies conditions for including records in a TCursor opened on a Table variable.

Syntax

```
1. setGenFilter ( criteria DynArray[ ] AnyType ) Logical
2. setGenFilter ( criteria Array[ ] AnyType [ , fieldId Array[ ] AnyType ] )
Logical
```

Description

setGenFilter specifies conditions for including records in a TCursor opened on a Table variable. Records that meet all the specified conditions are included; records that don't are filtered out, creating a restricted view of the table.

In syntax 1, the DynArray *criteria* specifies fields and filtering conditions as follows: the index is the field name or field number, and the item is the filter expression.

For example, the following code specifies criteria based on the values of three fields.

```
criteriaDA[1]      = "Widget" ; The value of the first field in the
; table is Widget.
criteriaDA["Size"] = "> 4"    ; The value of the field named Size is
; greater than 4.
criteriaDA["Cost"] = ">= 10.95, < 22.50" ; The value of the field
; named Cost is greater than or equal to 10.95 and less than 22.50.
```

If the DynArray is empty, any existing filter criteria are removed.

In syntax 2, the Array *criteria* specifies filtering conditions, and the optional Array *fieldId* specifies field names and/or field numbers. If you omit *fieldID*, conditions are applied to fields in the order they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to the second field, and so on). The following example fills arrays for syntax 2 to specify the same criteria as the example for syntax 1.

```
criteriaAR[1] = "Widget"
criteriaAR[2] = "> 4"
criteriaAR[3] = ">= 10.95, < 22.50"
fieldAR[1] = 1
fieldAR[2] = "Size"
fieldAR[3] = "Cost"
```

If the Array is empty, any existing filter criteria are removed.

setGenFilter example 1

In this example, the built-in **run** method for a script attaches a Table variable to the *Customer* table, then sets filter criteria on the *State* field to equal "CA".

```
;Script :: run
method run(var eventInfo Event)
   var
      tb         Table
      dyn       DynArray[] AnyType
   endVar

   dyn["State/Prov"] = "CA"

   tb.attach("CUSTOMER.DB")
   tb.setGenFilter(dyn)

endMethod
```

▪

setGenFilter example 2

In the following example, a form contains a button called *btnBalanceStatus*. The **pushButton** method for *btnBalanceStatus* attaches a Table variable to the *Orders* table and sets a filter criteria to view only the records with a positive balance. Then, **cCount** gets the number of records, **cAverage** gets the average balance due, and **cSum** gets the total balance due. Finally, a dialog box displays the values.

```
;btnBalanceStatus
method pushButton(var eventInfo Event)
   var
      tbl         Table
      dyn         DynArray[] AnyType
      s1,
      s2,
      s3            String
   endVar

   tbl.attach("ORDERS")
   Dyn["Balance Due"] = "> 0"
   tbl.setGenFilter(Dyn)

   s1 = string(tbl.cCount("Balance Due"))
   s2 = string(tbl.cAverage("Balance Due"))
   s3 = string(tbl.cSum("Balance Due"))

   msgInfo("Outstanding balances", "There are " + s1 + " orders with an
average balance due of " + s2 + ", totaling " + s3 + ".")
endMethod
```

▪

setIndex method

See also Example Table Type

Specifies an index for a table.

Syntax

```
1. (Paradox tables) setIndex ( const indexName String ) Logical
2. (dBASE tables) setIndex ( const indexName String [ , const tagName String ] ) Logical
```

Description

setIndex specifies an index to use when a table is opened.

When working with a Paradox table, use *indexName* specify an index. When working with a dBASE table, you can use *indexName* to specify a .NDX file, or *indexName* and *tagName* to specify an index tag in a .MDX file.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

■

setIndex example

In the following example, assume the Paradox Customer table has a secondary index named CityState. The following code specifies CityState with **setIndex** to set up for a call to **setRange**. When the designated filter is set for *Customer*, this example loads a DynArray with information from the filtered table then displays the contents of the DynArray in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  custTbl Table
  tc TCursor
  dy DynArray[] Anytype
endVar

custTbl.attach("Customer.db")
if isTable(custTbl) then

  ; now use the secondary index named CityState
  custTbl.setIndex("CityState")

  ; filter out everything but St. Thomas
  custTbl.setRange("St. Thomas", "St. Thomas")

  ; open a TCursor for the filtered Customer table
  if tc.open(custTbl) then

    ; scan the table and load the DynArray with
    ; company names (Name) and phone numbers
    scan tc:
      dy[tc.Name] = tc.Phone
    endScan
    ; display contents of the DynArray
    dy.view("St. Thomas Phone Numbers")

  else
    msgStop("Error", "Can't open TCursor.")
  endIf

else
  msgStop("Error", "Can't find Customer.db")
endIf
endMethod
```

▪

setRange method

See also Example1 Example2 Table Type

Specifies a range of records to associate with a Table variable.

Syntax

```
1. setRange ( [ const exactMatchVal AnyType] * [ , const minVal AnyType,
const maxVal AnyType ] ) Logical
2. setRange ( rangeVals Array[ ] AnyType ) Logical
```

Description

setRange specifies conditions for associating a contiguous range of records with a Table variable. Records that meet the conditions are included when the table is opened; records that don't are excluded. **setRange** compares the criteria you specify with values in the corresponding fields of a table's index; it fails if the table is not indexed. Calling **setRange** without any arguments resets the range criteria to include the entire table.

Note This method replaces **setFilter** included in earlier versions: functionality is enhanced. Code that calls **setFilter** will continue to execute as before.

In syntax 1, to set a range based on the value of the first field of the index, specify values in *minVal* and *maxVal*. For example, the following statement checks values in the first field of the index of each record:

```
tableVar.setRange(14, 88)
```

If a value is less than 14 or greater than 88, that record is filtered out. To specify an exact match on the first field of the index, assign *minVal* and *maxVal* the same value. For example, the following statement filters out all values except 55:

```
tableVar.setRange(55, 55)
```

You can set a range based on the values of more than one field. To do so, specify exact matches *except* for the last one in the list. For example, the following statement looks for exact matches on "Borland" and "Paradox" (assuming they are the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field:

```
tableVar.setRange("Borland", "Paradox", 100, 500)
```

In syntax 2, you can pass an array of values to specify the range criteria, as listed in the following table.

Number of array items	Range specification
No items (empty array)	Resets range criteria to include the entire table.
One item	Specifies a value for an exact match on the first field of the index.
Two items	Specifies a range for the first field of the index.
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size *n*, specify exact matches on the first *n*-2 fields of the index. The last two array items specify a range for the *n*-1 field of the index.

setRange example 1

In the following example, assume that Lineitem's key field is Order No. and you want to know the total for order number 1005. The following code attaches a Table variable to the *Lineitem* table, limits the range of records to those with 1005 in the first field of the primary index, then uses **cSum** to calculate the total for order 1005.

```
; getDetailSum::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar
tblName = "LineItem.db"
tblVar.attach(tblName)

  ; this limits TCursor's view to records that have
  ; 1005 in the first field of the primary index
tblVar.setRange(1005, 1005)

  ; now display the total for Order No. 1005
msgInfo("Total for Order 1005", tblVar.cSum("Total"))

endMethod
```

setRange example 2

This example shows how to call **setRange** with a criteria array that contains more than three items. The following code sets a range to include orders from a person with a specific first name, middle initial, and last name, and an order quantity ranging from 100 to 500 items. Then it counts the number of records in this range and displays the value in a dialog box. This example assumes that the *PartsOrd* table is indexed on the FirstName, MiddleInitial, LastName, and Qty fields.

```
; setQtyRange::pushButton
method pushButton(var eventInfo Event)
   var
      tbPartsOrd   Table
      arRangeInfo   Array[5] AnyType
      nuCount      Number
   endVar

   arRangeInfo[1] = "Frank"      ; FirstName (exact match)
   arRangeInfo[2] = "P."         ; MiddleInitial (exact match)
   arRangeInfo[3] = "Borland"    ; LastName (exact match)
   arRangeInfo[4] = 100          ; Minimum qty value
   arRangeInfo[5] = 500          ; Maximum qty value

   if tbPartsOrd.attach("PartsOrd") then
         tbPartsOrd.setRange(arRangeInfo)
         nuCount = tbPartsOrd.cCount(1)
         nuCount.view("Number of big orders by Frank P. Borland:")
   else
         errorShow("Can't open the table.")
   endIf
endMethod
```

▪

setReadOnly method

See also Example Table Type

Specifies whether to give the user read-only rights to a table when it is opened.

Syntax

```
setReadOnly ( [ const yesNo Logical ] )
```

Description

setReadOnly specifies whether to give the user read-only rights to a table when it is opened. This method fails if the table has been locked by another user or if the table is open.

Optional argument *yesNo* specifies whether to set read-only rights: a value of Yes grants read-only rights, a value of No allows full rights to the table. If omitted, *yesNo* is Yes by default.

setReadOnly example

The following code attaches a Table variable to the Orders table, issues **setReadOnly** to limit user rights, then opens a TCursor for Orders.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tc TCursor
endVar

errorTrapOnWarnings()
tblVar.attach("Orders.db")   ; attach Table var to Orders.db
tblVar.setReadOnly()         ; set Table to read-only
tc.open(tblVar)              ; open a TCursor for Orders.db
tc.edit()

endMethod
```

▪

showDeleted method

See also   Example   Table Type

Specifies whether to display deleted records in a dBASE table.

Syntax

```
showDeleted ( [ const yesNo Logical ] ) Logical
```

Description

Records deleted from a dBASE table aren't immediately removed. Instead, they are flagged for deletion and removed later. **showDeleted** specifies whether to display these records when the table is opened. **showDeleted** is relevant only for dBASE table.

Optional argument *yesNo* specifies whether to show deleted records (a value of Yes) or hide deleted records (a value of No). If omitted, *yesNo* is Yes by default. If you don't call this method before using the Table variable associated with the table, deleted records are not shown.

showDeleted example

For the following example, the **pushButton** method attached to the *showDeletedRecs* button instructs a Table variable's deleted records be shown.

```
; showDeletedRecs::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
endVar

tblVar.attach("Orders.dbf")
if isTable(tblVar) then

  ; show deleted records in Orders.dbf
  tblVar.showDeleted(Yes)

  ; display sum of deleted and undeleted records
  msgInfo("Total # of Records", tblVar.nRecords())
else
  msgStop("Error", "Can't find Orders table.")
endIf

endMethod
```

▪

sort keyword

[See also] [Example] [Table Type]

Sorts a table.

Syntax

```
sort sourceTable [ on fieldNameList [ D ] ] [ to destTable ]    endSort
```

Description

sort fills in a sort form for the table specified in *sourceTable* and performs the sort.

sourceTable can be of type Table, TCursor, or String. *destTable* can be of type Table or String. However, you can't sort a TCursor onto itself.

If you include the optional **on** clause, the table is sorted on the first field specified in *fieldNameList*. Each subsequent field is used, in turn, to settle ties in the preceding fields. An optional **D** after a field name specifies a sort in descending order. If you omit the **on** clause, records are sorted in ascending order, moving from left to right across the fields.

If you include the optional **to** clause, the result of the sort is written to the table described by *destTable*. If that table already exists, it is overwritten without asking for confirmation. If you omit the **to** clause, the sorted records are placed back *sourceTable* (this fails if the table is open). You must specify the **to** clause if the source table is keyed.

sort automatically places a full lock on tables being sorted if the result will be written to the same table. Otherwise, a write lock is required for the source table and a full lock for the destination table.

sort is not a method, so dot notation, as in

```
tableVar.sort()
```

is inappropriate. Instead, you create a structure to specify how to sort the table.

▪

sort example

The following example sorts *Customer* on the Last Name and First Name fields, and places the results in the *CustSort* table.

```
; sortCustTable::pushButtton
method pushButton(var eventInfo Event)
var
  custTbl Table
  tv TableView
endVar

custTbl.attach("Customer.db")

sort custTbl
  on "Country" D, "Name" D     ; sort in descending order
  to "CustSort.db"
endSort

tv.open("CustSort.db")                  ; open the sorted table

endMethod
```

▪

subtract method/procedure

See also Example Table Type
Beginner

Subtracts the records in one table from another table.

Syntax

```
1. subtract ( const destTableName String ) Logical
2. subtract ( const destTableName Table ) Logical
```

Description

subtract checks whether any records in the source table are also in *destTableName*. If so, **subtract** deletes them from *destTableName* without asking for confirmation.

If *destTable* is keyed, **subtract** deletes all records with keys that exactly match values in corresponding key fields in the source table. If *destTable* is not keyed, **subtract** deletes all records that exactly match any record in the source table. Whether tables are keyed or not, this method considers only fields that *could* be keyed (based on data type, not position). For example, numeric fields are considered, but formatted memos are not. This method requires read/write access to both tables.

Note: If the destination table is not keyed, this operation can be time-consuming.

This method tries, for the duration of the retry period, to place a full lock on both tables. If locks cannot be placed, an error results.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
1. subtract ( const sourceTableName String, const destTableName String )
Logical
2. subtract ( const sourceTableName String, const destTableName Table )
Logical
```

■

subtract example

The following code subtracts from Customer matching records found in the *Inserted* table in the private directory.

```
; subtractCust::pushButton
method pushButton(var eventInfo Event)
var
  insTbl, CustTbl Table
  fs FileSystem
  tblName String
endVar
tblName = privDir() + "\\Inserted.db"

insTbl.attach(tblName)
if insTbl.isTable() then
  insTbl.subtract(custTbl)    ; remove from custTbl matching records in
insTbl
else
  msgInfo("Sorry", "Can't find " + tblName + " table.")
endIf

endMethod
```

▪

tableRights method/procedure

See also Example Table Type

Specifies whether the user has rights to perform certain operations on a table.

Syntax

```
tableRights ( const rights String ) Logical
```

Description

tableRights reports about a user's rights to a table, where *rights* is one of the following:

Value	Description
"ReadOnly"	Read from the table, but not change it.
"Modify"	Enter or change data.
"Insert"	Add new records.
"InsDel"	Add and delete records.
"Full" or "All"	Perform all of the above operations.

This method returns True if the user has the specified rights; otherwise, it returns False.

DOS

The following procedure is provided as a convenience to DOS PAL programmers. You can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
tableRights ( const tableName String, const rights String )
```

tableRights example

The following example reports whether the user has "All" rights to the Orders table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myRights Logical
  ordTbl   Table
endVar

ordTbl.attach("Orders.db")
if ordTbl.isTable() then
  myRights = ordTbl.tableRights("All")

  ; this displays True if you have All rights to Orders.db
  msgInfo("All Rights?", myRights)

else
  message("Can't find Orders table.")
endIf
endMethod
```

▪

type method

See also Example Table Type

Returns the type of a table.

Syntax

```
type ( ) String
```

Description

type returns the string value "PARADOX" or "DBASE" to report the type of a table.

▪

type example

The following code compacts (removes deleted records from) the *Orders* table if **type** returns DBASE; otherwise, a message informs the user.

```
; compactButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
endVar
tblVar.attach("Orders")
if tblVar.type() = "DBASE" then
  tblVar.compact()
else
  msgStop("Stop!", "Orders is a " + tblVar.type() + " table.")
endIf

endMethod
```

▪

unAttach method

See also Example Table Type

Ends the association between a Table variable and a table description.

Syntax

`unAttach ( ) Logical`

Description

unAttach ends the association (created using **attach** or **create**) between a Table variable and a table description. You don't have to end the association between a Table variable and a table to attach the same variable to another table▪**unAttach** is automatically called when a Table variable is assigned to a different table.

▪

unAttach example

In the following example, a single Table variable is used to summarize sales information from two different tables. Once the Table variable (*tableVar*) is no longer needed this code calls **unAttach** to end the association between *tableVar* and the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tableVar Table
  q1, q2   Number
  msg      String
endVar

tableVar.attach("q1_sales.db")  ; attach to q1_sales table
q1 = tableVar.cSum("Amount")    ; get a summary

tableVar.attach("q2_sales.db")  ; no need to unattach
q2 = tableVar.cSum("Amount")    ; get summary from q2_sales

tableVar.unAttach()             ; we don't need tableVar anymore
                                ; so end the association to q2_sales

switch
   case q2 < q1 : msg = "Sales are down."
   case q2 = q1 : msg = "Sales are flat."
   case q2 > q1 : msg = "Sales are up."
endSwitch

msgInfo("Sales", msg)

endMethod
```

▪

unlock method

See also Example Table Type
Beginner

Unlocks a specified table.

Syntax
`unlock ( const lockType String ) Logical`

Description

unlock attempts to remove locks explicitly placed on the table associated with a Table variable, where *lockType* is one of the following String values, listed in order of decreasing strength and increasing concurrency.

String value	Description
Full	The current session has exclusive access to the table. No other session can open the table. Cannot be used with dBASE tables.
Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

unlock removes locks explicitly placed by a particular user or application using **lock**; it has no effect on locks placed automatically by Paradox. Each time you lock a table explicitly, be sure to unlock it as soon as you no longer need the explicit lock. This ensures maximum concurrent availability of tables. Also, when you lock a table twice, you must unlock it twice. You can use **lockStatus** (defined for the TCursor and UIObject types) to determine how many explicit locks you have placed on a table. **unlock** returns False if you try to unlock a table that isn't locked or cannot be unlocked.

If successful, this method returns True; otherwise, it returns False.

▪

unlock example

For the following example, the **pushButton** method for *updateCust* runs a query from an existing file, then adds records from the *Answer* table to the *Customer* table. This code attempts to place a write lock on the *Customer* table before adding records to it. If the lock succeeds, this code proceeds to add *Answer* records, then uses **unlock** to unlock *Customer*.

```
; updateCust::pushButton
method pushButton(var eventInfo Event)
var
  newCust Query
  ansTbl Table
  destTbl String
endVar
destTbl = "Customer.db"

newCust.readFromFile("getCust.qbe")

if newCust.executeQBE() then            ; If the query succeeds,
  ansTbl.attach(":PRIV:Answer.db")
  if destTbl.lock("Write") then         ; try to write lock the table.
    ansTbl.add(destTbl)                 ; Add records from Answer.db.
    destTbl.unLock("Write")             ; Unlock the table.
  else
    msgStop("Stop", "Can't write lock " + destTbl + " table.")
  endIf
else
  msgStop("Stop!", "Query failed.")
endIf

endMethod
```

▪

unProtect method/procedure

See also   Example   Table Type

Decrypts and removes an owner password from a table.

Syntax

1. (Procedure) **unProtect (** const ***tableName*** String [**,** const ***Password*** String] **)**

2. (Method) **unProtect (** [const ***password*** String] **)**

Description

unProtect permanently removes an owner password from a table. A protected table is encrypted and cannot be accessed without presenting the password specified in *password*. If you have already issued the master password for a table, *password* is not necessary.

unProtect example

The following example permanently removes password protection from the *Secrets* table.

```
; decrypt::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar

tblName = "Secrets.db"
tblVar.attach(tblName)
if tblVar.isEncrypted() then
  tblVar.unprotect("Get007")    ; permanently remove password
                                ; this assumes Get007 is the master password
endIf

endMethod
```

usesIndexes method

See also Example Table Type

Specifies index files to use and maintain with a dBASE table.

Syntax

usesIndexes (const ***indexFileName*** String [, const ***indexFileName*** String] * Logical

Description

usesIndexes specifies in *indexFileName* one or more index files (.NDX and .MDX) to maintain while you use a dBASE table. This method does not open the table, but specifies index files to open when the table is opened. Don't use this method to open production files (such as the .MDX file with the same name as the table) for a dBASE table▪these files are opened automatically.

This method fails if any of the specified index files does not exist.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

usesIndexes example

The following example calls **usesIndexes** to specify two different indexes in the *Orders* table, then opens a TCursor for the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tc     TCursor
endvar

tblVar.attach("Orders.dbf")
if tblVar.isTable() then

  ; specify NameStat and Ord_Name indexes
  tblVar.usesIndexes("NAMESTAT.NDX", "ORD_NAME.NDX")

  ; now attempt to open the table, using the specified indexes
  if tc.open(tblVar) then
    if tc.locate("State", "FL", "Contact", "Simons") then
      msgInfo("Order Date", tc."Order Date")
    else
      msgStop("Error", "Can't find values.")
    endIf
  endIf
else
  msgStop("Error", "Can't find Orders.dbf table.")
endIf
endMethod
```

Using ranges and filters

See also

In broad terms, both ranges and filters enable you to select a subset of the records in a Table variable, a TCursor, or a UIObject. However, ranges and filters operate differently.

A range is based on the fields in an index, so applying a range to a table results in a subset of records that are contiguous and consecutive. Therefore, a range generally gives faster performance than a filter.

A filter offers greater flexibility in selecting fields and specifying criteria. A filter can be based on any field in a table; it is not restricted to fields in an index. Also, a filter can use expressions to specify criteria, but a range cannot. For example, a filter could select records in which the Quantity field has values of 125, 200, and 350, but a range could only specify values ranging from 125 to 350.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

TableView type

A TableView object displays the data in a table in its own window. A TableView object is distinct from a table frame, which is a UIObject placed in a form, and from a TCursor, a programmatic construct that points to the data in a table.

When you declare a TableView variable, then open a TableView object to that variable, you create a handle to the TableView window, something you can refer to in your code to manipulate the TableView object.

TableView methods are a subset of the methods for the Form type. You can use them to control the Table window's size, position, and appearance. Although you can start and end Edit mode for a table view, you cannot use ObjectPAL to directly edit the data in a table view. You can use ObjectPAL to manipulate TableView properties in these main areas:

- The TableView object as a whole
- for example, background color, grid style, number of records, and the value of the current record
- The field-level data in the table (TVData)
- for example, font, color, and display format
- The table view heading (TVHeading)
- for example, font, color, and alignment

The TableView type includes several derived methods from the Form type.

Methods for the TableView type

Form	TableView
bringToTop	**action**
getPosition	**close**
getTitle	**moveToRecord**
hide	**open**
isMaximized	**wait**
isMinimized	
isVisible	
maximize	
minimize	
setPosition	
setTitle	
show	
windowHandle	

▪

action method

See also Example TableView Type

Performs an action command.

Syntax

```
action ( const actionID SmallInt ) Logical
```

Description

action performs the action represented by the constant *actionId*, where *actionId* is a constant in one of the following action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use **action** to send a user-defined action constant to a built-in **action** method. User-defined action constants are simply integers that don't interfere with any of ObjectPAL's constants. You can use them to signal other parts of an application. For instance, assume that the Const window for a form declares a constant named *myAction*. In the built-in **action** method for a page on the form, you might check the value of every incoming ActionEvent (with the **id** method); if the value is equal to *myAction*, you can respond to that action accordingly. Paradox's default response for user-defined action constants is simply to pass the action to the **action** method. For more information on defining constants, see the *Guide to ObjectPAL*.

This **action** method is distinct from the built-in **action** method for a TableView or for any form or UIObject. The built-in **action** method for an object responds to an action event; this method causes an ActionEvent.

action example

The following example opens a table view for the *Orders* table, moves the cursor to the end of the table, starts Edit mode, and inserts a new blank record. This code is attached to the **pushButton** method for a button named *startEditInsert*.

```
; startEditInsert::pushButton
method pushButton(var eventInfo Event)
var
  orderTV  TableView
endVar
if orderTV.open("Orders") then
  orderTV.action(DataEnd)            ; move to the end of the table
  orderTV.action(DataBeginEdit)      ; start Edit mode
  orderTV.action(DataInsertRecord)   ; insert a new blank record
  orderTV.wait()                     ; wait until TableView object is closed
  orderTV.close()                    ; close when return
else
  msgStop("Status", "Could not find Orders table.")
endIf
endMethod
```

▪

close method

See also   Example   TableView Type

Closes a table window.

Syntax

```
close ( )
```

Description

close closes a table window, equivalent to choosing Close from the Control menu.

close example

In the following example, the **open** method for a form opens a TableView object for the *Customer* table to the global variable *custTV*. When the form closes, the **close** method for the form closes the *custTV* table view. This code is attached to the **close** method for the form:

```
; thisForm::close
method close(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    custTV.close()      ; close the Customer table that was
                        ; opened by thisForm's open method
endIf
endMethod
```

This is the code for the form's Var window.

```
; thisForm::Var
Var
  custTV  TableView    ; global to form, the TableView object is opened by
                       ; form's open method
endVar
```

This is the code for the form's **open** method.

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    custTV.open("Customer")  ; open the Customer table view
endIf
endMethod
```

■

moveToRecord method

See also Example TableView Type

Moves to a specific record in a table.

Syntax

```
moveToRecord ( const tc TCursor ) Logical
```

Description

moveToRecord sets the current record to the record pointed to by the TCursor *tc*. This method can be very slow for dBASE tables; use the **RecNo** property instead.

▪

moveToRecord example

The following example uses a TCursor to search for a customer named Jones, then calls **moveToRecord** to make a table view display that record. The following code is attached to a button's built-in **pushButton** method.

```
method pushButton (var eventInfo Event)
var
   custTC TCursor
   custTV TableView
endVar

custTC.open ("customer.db")
custTV.open ("customer.db")

if custTC.locate ("Last Name", "Jones") then
  custTV.moveToRecord (custTC)
else
   msgInfo("Search failed", "Couldn't find Jones.")
endIf

endMethod
```

▪

open method

See also Example TableView Type

Opens a table window.

Syntax

```
1. open ( const tvName String [ , const windowStyle LongInt ] ) Logical
2. open ( const tvName String, const windowStyle LongInt, const x SmallInt,
const y SmallInt, const w SmallInt, const h SmallInt ) Logical
```

Description

open displays the table specified in *tvName* in a table window. Optional arguments specify (in twips) the location of the upper left corner of the form (*x* and *y*), the width and height (*w* and *h*), and style (*windowStyle*). The *windowStyle* argument is ignored, but required for syntax 2. If you want to specify a size and position, you can use a window style constant of WinStyleDefault.

■

open example

In the following example, the **pushButton** method for a button named *openWaitOrders* opens the *Orders* table, then waits until the user closes the table.

```
; openWaitOrders::pushButton
method pushButton(var eventInfo Event)
var
  ordersTV  TableView
endVar
if ordersTV.open("Orders", WinStyleDefault, 100, 100,
                 1440*5, 1440*4) then
  ordersTV.wait()    ; wait for user to close
  ordersTV.close()   ; close Orders table
endIf
endMethod
```

▪

wait method

See also Example TableView Type

Suspends execution of a method.

Syntax

`wait ( )`

Description

wait suspends execution of a method. Execution resumes when the TableView object is closed. Note that you must follow a **wait** with a **close**. When a TableView object has been called by **wait**, the calling method suspends execution until the TableView object is closed by the user.

-

wait example

See the example for **open**.

TCursor type

Changes

A TCursor is a pointer to the data in a table, enabling you to manipulate data without having to display the table. It is not a clone or a copy of the table editing records in a TCursor changes the underlying table, and any locks on the table affect the TCursor. A TCursor can point to an entire table, or to a subset of the records in a table (for example, as specified by a restricted view, detail set, filter, or range).

For information about related objects, refer to the Table, TableView, and UIObject types.

Some table operations require Paradox to create temporary tables. Paradox creates these tables in the private directory.

Methods for the TCursor type

TCursor

add

aliasName

atFirst

atLast

attach

attachToKeyViol

bot

cancelEdit

cAverage

cCount

close

cMax

cMin

cNpv

compact

copy

copyFromArray

copyRecord

copyToArray

createIndex

cSamStd

cSamVar

cStd

cSum

currRecord

cVar

deleteRecord

didFlyAway

dmAttach

dropGenFilter
dropIndex
edit
empty
end
endEdit
enumFieldNames
enumFieldNamesInIndex
enumFieldStruct
enumIndexStruct
enumLocks
enumRefIntStruct
enumSecStruct
enumTableProperties
eot
familyRights
fieldName
fieldNo
fieldRights
fieldSize
fieldType
fieldUnits2
fieldValue
forceRefresh
getGenFilter
getIndexName
getLanguageDriver
getLanguageDriverDesc
getRange
home
initRecord
insertAfterRecord
insertBeforeRecord
insertRecord
instantiateView
isAssigned
isEdit
isEmpty
isEncrypted
isOpenOnUniqueIndex

isRecordDeleted
isShared
isShowDeletedOn
isValid
isView
locate
locateNext
locateNextPattern
locatePattern
locatePrior
locatePriorPattern
lock
lockRecord
lockStatus
moveToRecord
moveToRecNo
nextRecord
nFields
nKeyFields
nRecords
open
postRecord
priorRecord
qLocate
recNo
recordStatus
reIndex
reIndexAll
seqNo
setBatchOff
setBatchOn
setFlyAwayControl
setFieldValue
setGenFilter
setRange
showDeleted
skip
subtract
switchIndex
tableName

Changes to TCursor type methods

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
createIndex	attach
dmAttach	enumFieldStruct
getGenFilter	enumIndexStruct
getIndexName	enumRefIntStruct
getRange	enumSecStruct
instantiateView	fieldType
isView	nRecords
setGenFilter	open
setRange	recNo
	seqNo

setFilter was replaced by **setRange** which offers enhanced functionality and performance. Code that calls **setFilter** will continue to execute as before.

The following table lists new methods for version 7.

New	Changed
aliasName	none

▪

add method

See also Example TCursor Type

Adds the records of one table to another.

Syntax

```
1. add ( const destTable String [ , const append Logical [ , const update
Logical ] ] ) Logical
2. add ( const destTable Table [ , const append Logical [ , const update
Logical ] ] ) Logical
3. add ( const destTable TCursor [ , const append Logical [ , const update
Logical ] ] ) Logical
```

Description

add adds the records pointed to by a TCursor to the destination table specified in *destTable*. If the destination does not exist, this method creates it. The source table and the destination table can be the same type or different types; in any case, the tables must have compatible field structures.

Arguments *append* and *update* can be True or False. When True, *append* adds records at the end of a non-indexed table, or at the appropriate places in an indexed table. When True, *update* compares records in both tables, and where key values match, replaces the data in the destination table. When both are True, records with matching key values are updated, and others are appended. These arguments are optional, but if you specify *update*, you must also specify *append.* If omitted, both are True. Here are some example statements:

```
myTCursor.add("yourTable", False, True) ; specifies update
myTCursor.add("yourTable") ; specifies update and append by default
```

When tables are indexed, **add** uses the indexed fields to determine which records to update and which to append. When the destination table is not indexed, **add** fails if *update* is True. Key violations (including validity check violations), if any, are listed in KEYVIOL.DB in the user's private directory. This method overwrites an existing KEYVIOL.DB or creates one, if necessary. **add** respects the limits of restricted views set by **setRange** or **setGenFilter.**

This method tries, for the duration of the retry period, to place write locks on the source table and the destination table. If either lock cannot be placed, the method fails.

add example

In the following example, assume the *OldCust* and *NewCust* tables exist in the current directory. The following code associates a TCursor with each of the tables, adds *NewCust* records to *OldCust*, then adds all records to a table named *MyCust*. If *MyCust* does not exist in the current directory, **add** creates it. This code is attached to a button's **pushButton** method.

```
; getMyCust::pushButton
method pushButton(var eventInfo Event)
var
  dTC, sTC TCursor
endVar

if sTC.open("oldCust.db") and
   dTC.open("newCust.db") then  ; if both TCursors can be associated
   dTC.add(sTC, True)           ; append oldCust records to newCust records
                                ; ▪now sTC has records from both tables

   sTC.add("myCust.db", True)   ; add sTC to myCust table

   sTC.close()                  ; close both TCursors
   dTC.close()
else
   msgStop("Stop!", "Could not open one or more tables.")
endIf

endMethod
```

aliasName method

See also Example TCursor Type

Returns the alias of the TCursor.

Syntax

```
aliasName ( ) String
```

Description

aliasName returns a string containing the alias of the tcursor. Only tcursors that were opened with an alias will return an alias name. If the tcursor was not opened with an alias, **aliasName** will return an empty string.

▪

aliasName example

The following example uses **aliasName** to determine the value of the OPEN MODE property for the open tcursor.

```
method pushButton(var eventInfo Event)
var
   tc Tcursor
   tabName, propName, expectedProp, actualProp String
endVar

;// initialize variables
propName = "OPEN MODE"
expectedProp = "READ/WRITE"
tabName = ":Musetto:GOODEATS"

if tc.open( tabName ) then
   ;// get property value and compare with expected value
   actualPropVal = getAliasProperty( tc.aliasName(), propName )

   if actualProp = expectedProp  or actualProp.isBlank() then
      doSomething() ;// continue processing
      return
    else
       ;// try to set to the desired property
       setAliasProperty( tc.aliasName(), expectedProp )
    endif
  endif
endMethod
```

atFirst method

See also Example TCursor Type

Reports whether the TCursor is pointing to the first record of a table.

Syntax

```
atFirst ( ) Logical
```

Description

atFirst returns True if the TCursor is pointing to the first record of a view of a table; otherwise, it returns False.

atFirst example

The following example assumes a form has a button named *moveToFirst*, and a multi-record object bound to ORDERS.DB. The code attached to the **pushButton** method for *moveToFirst* uses **atFirst** to determine if the TCursor is at the first record. If it isn't, this code moves to the first record.

```
; moveHome::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

tc.attach(orders)         ; orders is a multi-record object
if not tc.atFirst() then  ; if not at the first record
  tc.home()               ; move to it
  orders.moveToRecord(tc) ; move highlight to first record
else
  msgStop("Currently on record " + String(tc.recNo()),
          "You're already at the top of the list!")
endIf
endMethod
```

atLast method

See also   Example   TCursor Type

Reports whether the TCursor is pointing to the last record of a table.

Syntax

```
atLast ( ) Logical
```

Description

atLast returns True if the TCursor is pointing to the last record of a view of a table; otherwise, it returns False.

atLast example

The following example assumes a form has a button named *moveToLast*, and a multi-record object bound to ORDERS.DB. The code attached to the **pushButton** method for *moveToLast* uses **atLast** to determine if the TCursor is on the last record. If it isn’t, this code moves to the last record.

```
; moveToLast::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

tc.attach(ORDERS)
if not tc.atLast() then   ; if not on the last record
  tc.end()                ; move TCursor to the last record
  orders.moveToRecord(tc) ; move highlight to the last record
else
  msgStop("Currently on record " + String(tc.recNo()),
          "You're already at the last record!")
endIf
endMethod
```

▪

attach method

See also Examples TCursor Type

Associates a TCursor with a table.

Syntax

```
1. attach ( const object UIObject ) Logical
2. attach ( const srcTCursor TCursor ) Logical
3. attach ( const tv TableView ) Logical
```

Description

attach associates a TCursor with a specified table. The data (including filters, indexes, and edit mode) comes from the underlying table▪the TCursor gets no data from records that have not been committed (for example, because the record is being edited or has just been inserted).

Syntax 1 associates a TCursor with the table displayed in the UIObject *object*.

Syntax 2 associates the TCursor with the table represented by another TCursor, *srcTCursor*.

Syntax 3 associates the TCursor with the TableView object *tv*.

attach returns True if successful; otherwise, it returns False and adds the following warning to the error stack: "You have tried to access a document that is not open." (Warning added in version 5.0.)

▪

attach examples

In the following example, assume a form contains a table frame bound to ORDERS.DB, and another table frame bound to LINEITEM.DB. The *Orders* table has a one-to-many link to *LineItem*. A button named *findDetails* is also on the form. Suppose you want the user to be able to search through the entire *LineItem* table▪not just those records linked to the current order. In this case, the **pushButton** method for *findDetails* searches for orders that include the current part number.

This code is attached to the Var window for the *findDetails* button:

```
; findDetails::Var
Var
  lineTC TCursor  ; instance of LINEITEM for searching
endVar
```

The code that follows is attached to the **open** method for the *findDetails* button. This code associates the *lineTC* TCursor with LINEITEM.DB.

```
; findDetails::open
method open(var eventInfo Event)
   lineTC.open("LineItem.db")
endMethod
```

The following code is attached to the **pushButton** method for *findDetails*:

```
; findDetails::pushButton
method pushButton(var eventInfo Event)
var
   stockNum,
   orderNum   Number
   orderTC      TCursor
endVar
; Get Stock No from current LineItem record.
stockNum = LINEITEM.Stock_No

; LineTC was declared in Var window and opened by open method.
if NOT lineTC.locateNext("Stock No", stockNum) then
  lineTC.locate("Stock No", stockNum)
endIf

orderTC.attach(ORDERS)       ; Attach TCursor to table frame.
orderTC.locate("Order No", lineTC."Order No")
ORDERS.moveToRecord(orderTC) ; Move to CUSTOMER and
                             ; resynchronize with TCursor.
LINEITEM.moveTo()            ; Move TCursor to LINEITEM detail.

; Move TCursor to matching record.
LINEITEM.locate("Stock No", stockNum)
endMethod
```

This code is attached to the **close** method for *findDetails*:

```
; findDetails::close
method close(var eventInfo Event)
lineTC.close()   ; Close the TCursor to LineItem.
endMethod
```

▪

attachToKeyViol method

See also Example TCursor Type

Attaches a TCursor to the existing record that has the same key as the record you attempted to post.

Syntax

`attachToKeyViol ( const oldTC TCursor ) Logical`

Description

After a key violation occurs, **attachToKeyViol** attaches a TCursor to the existing record▪the record that existed before the key violation occurred. Specify in *oldTC* the TCursor that points to the record that caused the key violation (the new, unposted record).

This method gives you a way to compare conflicting records before replacing or discarding a change to an existing record. *oldTC* must already be pointing to the new (yet unposted) record.

▪

attachToKeyViol example

The following example demonstrates how **attachToKeyViol** can be used after a key violation occurs. The code declares two TCursors: *keyViolTC* and *originalRecTC*. The code opens *keyViolTC* for the *Orders* table, then deliberately inserts a record whose key value conflicts with another record. Then, the example attempts to post the new record to the table, which forces a key violation. At this point, if the user chooses to view the existing record, the code calls **attachToKeyViol**, attaches the second TCursor (*originalRecTC*) to the original record, and displays the record in a **view** dialog box. If the user chooses to update the original record with data from the new record, this example calls **updateRecord** method to do so; otherwise, the code makes no changes.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  keyViolTC, originalRecTC TCursor
  rec DynArray[] AnyType
endvar

keyViolTC.open("Orders.db")          ; open TCursor for Orders
keyViolTC.edit()                     ; put TCursor in Edit mode
keyViolTC.insertRecord()             ; insert a new record
keyViolTC."Order No" = 1011          ; 1011 is a duplicate key

; if this attempt to post the new record fails
if NOT keyViolTC.postRecord() then

  ; attach originalRecTC to the existing record
  originalRecTC.attachToKeyViol(keyViolTC)

  ; give user the option to see the existing record
  if msgQuestion("Key Exists!",
     "Do you want to see the existing record?") = "Yes" then

    originalRecTC.copyToArray(rec)   ; copy existing record to rec
    rec.view("Original Record")      ; display rec in a dialog box

  endIf

  ; give user the option to replace the existing record
  if msgQuestion("Confirm Update",
     "Do you want to replace existing record?") = "Yes" then

    ; force the new record to post
    keyViolTC.updateRecord(True)
  else
    message("Original record left intact.")
    sleep(1500)
  endIf
else
  message("Posted order number 1011.")
endIf

endMethod
```

▪

bot method

See also Example TCursor Type

Tests for a move past the beginning of a table.

Syntax

```
bot ( ) Logical
```

Description

bot returns True if a command attempts to move past the first record of a table; otherwise, it returns False. **bot** is reset by the next move operation.

▪

bot example

The following example moves a TCursor backwards through a table then displays a message. This code is attached to a button's **pushButton** method.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myTable TCursor
endVar
myTable.open("sites.db")
myTable.end()                         ; moves to end of table
while myTable.bot() = False           ; loop until we hit the top
   myTable.priorRecord()              ; move backwards through table
endWhile
msgInfo("The Top", "You're at the beginning.")
msgInfo("At the top?", myTable.bot()) ; displays True
myTable.nextRecord()
msgInfo("At the top?", myTable.bot()) ; displays False
endMethod
```

▪

cancelEdit method

See also Example TCursor Type
Beginner

Ends Edit mode without saving changes to the current record.

Syntax

```
cancelEdit ( ) Logical
```

Description

cancelEdit takes a TCursor out of Edit mode without saving changes to the current record. You must use **cancelEdit** before moving the TCursor from the current record or otherwise committing or unlocking the record. Once you move the TCursor, changes to the record are committed.

▪

cancelEdit example

The code for the following example is attached to the **pushButton** method for the *changeKey* button. This example associates a TCursor with the *Customer* table then attempts to change a value in a keyed field. If the record can not be successfully posted (for example, because of a key violation) this example displays an error message, then calls **cancelEdit** to restore the record to the original values and end Edit mode.

```
; changeKey::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  rec Array[] AnyType
endVar

tc.open("Customer.db")
if tc.locate("Customer No", 1231) then
  tc.edit()
  tc."Customer No" = 1221   ; attempt to change key value
  if not tc.endEdit() then  ; if endEdit fails
    errorShow("Can't complete the operation.")
    tc.cancelEdit()         ; restore record and leave edit mode
    message("Record left intact.")
  else
    message("Key value changed.")
  endIf
else
  errorShow("Can't find Customer 1231")
endIf

endMethod
```

▪

cAverage method

See also Example TCursor Type

Returns the average value of a field (column) in a table.

Syntax

```
1. cAverage ( const fieldName String ) Number
2. cAverage ( const fieldNum SmallInt ) Number
```

Description

cAverage returns the average of values in the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cAverage** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

cAverage example

The following example uses **cAverage** to calculate the average order size in the *Orders* table. This code is attached to the **pushButton** method for the *getAvgSales* button.

```
; getAvgSales::pushButton
method pushButton(var eventInfo Event)
var
  ordTC TCursor
  avgSales Number
endVar

; open TCursor for ORDERS table
ordTC.open("Orders.db")
; store average invoice total in avgSales variable
avgSales = ordTC.cAverage("Total Invoice")
; display avgSales in a dialog
msgInfo("Average Order size", avgSales)

endMethod
```

■

cCount method

See also Example TCursor Type

Returns the number of values in a field (column) of a table.

Syntax

```
1. cCount ( const fieldName String ) LongInt
2. cCount ( const fieldNum SmallInt ) LongInt
```

Description

cCount returns the number of values in the column (field) specified by *fieldName* or *fieldNum*. **cCount** works for all field types. If the field is numeric, this method handles blank values as specified in the **blankAsZero** setting for the session. If the field is non-numeric, **cCount** returns the number of nonblank values in the column of fields. In version 5.0, this method was changed to return a LongInt instead of a Number.

This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

cCount is useful for returning the number of entries used by another column function.

cCount example

The following example opens a TCursor for a table, then uses **cCount** to display the number of records in the TCursor. This code is attached to the **pushButton** method for the *lineItemInfo* button.

```
; lineItemInfo::pushButton
method pushButton(var eventInfo Event)
var
numbersTC TCursor
avgQty Number
numRecs LongInt
endVar
numbersTC.open("Lineitem.db")
avgQty = numbersTC.cAverage("Qty")
numRecs = numbersTC.cCount(4)        ; assumes Quantity is field 4
msgInfo("Average quantity", "Average quantity: " +
String(avgQty) + " \nbased on " + String(numRecs) + " records.")

endMethod
```

close method

See also Example TCursor Type
Beginner

Closes a table.

Syntax

```
close ( ) Logical
```

Description

close closes a TCursor, and makes the TCursor variable unassigned. If the current record cannot be committed, **close** still closes the TCursor, but discards any changes to the record.

close example

The following example opens a TCursor for a table, displays information found in the last record, then closes the TCursor. In this example, the code displays a message indicating whether the TCursor variable is still assigned after the TCursor is closed. This code is attached to the built-in **pushButton** method for *thisButton*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

tc.open("Orders.db")  ; open TCursor for the Orders table
tc.end()              ; move to the end of the table

; display information in the last record
msgInfo("Last Order", "Order number: " + String(tc."Order No") +
        " \nOrder date: " + String(tc."Sale Date"))

tc.close()                                  ; close tc TCursor
msgInfo("Is tc Assigned?", tc.isAssigned())  ; displays False

endMethod
```

▪

cMax method

See also Example TCursor Type

Returns the maximum value of a field (column) in a table.

Syntax

```
1. cMax ( const fieldName String ) Number
2. cMax ( const fieldNum SmallInt ) Number
```

Description

cMax returns the maximum value in the column of fields specified by *fieldName* or *fieldNum*. If the field is numeric, this method handles blank values as specified in the **blankAsZero** setting for the session. This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

▪

cMax example

In the following example, assume a form has a button, *getMaxBalance*, and a table frame bound to the *Orders* table. In this code, the **pushButton** method for *getMaxBalance* associates the table frame with a TCursor then locates the highest balance due in the *Orders* table:

```
; getMaxBalance::pushButton
method pushButton(var eventInfo Event)
var
  ordTC TCursor
endVar

ordTC.attach(ORDERS)    ; ORDERS is a table frame on the form

; now locate the maximum value in the "Balance Due" field
ordTC.locate("Balance Due", ordTC.cMax("Balance Due"))
; synchronize the table frame to the TCursor
ORDERS.moveToRecord(ordTC)

endMethod
```

▪

cMin method

See also		Example		TCursor Type

Returns the minimum value of a field (column) in a table.

Syntax

```
1. cMin ( const fieldName String ) Number
2. cMin ( const fieldNum SmallInt ) Number
```

Description

cMin returns the minimum value in the column of fields specified by *fieldName* or *fieldNum*. If the field is numeric, this method handles blank values as specified in the **blankAsZero** setting for the session. This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

cMin example

The following example uses both forms of the syntax to calculate minimum values in the ORDERS.DB table:

```
; showMinimums::pushButton
method pushButton(var eventInfo Event)
var
  OrdTC TCursor
  minBalDue, minOrder Number
endVar
OrdTC.open("Orders.db")
minBalDue = ordTC.cMin("Balance Due")  ; get minimum balance due
minOrder  = ordTC.cMin(6) ; assumes "Total Invoice" is field 6

; display results in a dialog box
msgInfo("Minimums", "Minimum balance due: " +
String(minBalDue) + "\n" +
                    "Minimum order : " + String(minOrder))
endMethod
```

▪

cNpv method

See also Example TCursor Type

Returns the net present value of a field (column), based on a specified discount or interest rate.

Syntax

```
1. cNpv ( const fieldName String, const discRate Number ) Number
2. cNpv ( const fieldNum SmallInt, const discRate Number ) Number
```

Description

cNpv returns the net present value of the entries in a column of fields. The calculation is based on the interest or discount rate *discRate*, where *discRate* is a decimal number (for example, 12 percent is expressed as .12). This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails. This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

This method calculates net present value using the following formula:

cNpv = sum(*p*=1 to *n*) of *Vp*/(1+*i*)*p*

where *n* = number of periods, *Vp* = cash flow in *p*th period, and

i = interest rate per period.

cNpv example

The following example associates a TCursor with the *GoodFund* table, then calculates the net present value for the *Expected Return* field. In this example, the net present value is calculated based on a monthly interest rate. This code is attached to the **pushButton** method for the *calcNPV* button.

```
; calcNPV::pushButton
method pushButton(var eventInfo Event)
var
  SavingsTC TCursor
  goodFundNPV, apr Number
endVar
SavingsTC.open("GoodFund.db")  ; associate TCursor with Savings table
apr = .125                     ; annual percentage rate

; now calculate net present value based on monthly interest rate
goodFundNPV = SavingsTC.cNpv("Expected Return", (apr / 12))
msgInfo("Net present value", goodFundNPV)

endMethod
```

▪

compact method

See also Example TCursor Type

Removes deleted records from a dBASE table.

Syntax

```
compact ( [ const regIndex Logical ] ) Logical
```

Description

compact removes deleted records from a dBASE table. Deleted records are not immediately removed from a dBASE table. Instead, they are flagged as deleted and kept in the table. This method returns True if successful; otherwise, it returns False. The optional argument *regIndex* is used to specify whether to regenerate indexes associated with the table, or simply to update them. When *regIndex* is True, this method regenerates all indexes associated with the TCursor and frees any unused space in the indexes. When *regIndex* is False, indexes are not regenerated. If omitted, *regIndex* is True by default.

This method fails if any locks have been placed on the table, or if the table is open. If the table has maintained indexes, this method requires exclusive access; otherwise it requires a write lock.

The **compact** method defined for the TCursor type does not work with Paradox tables. To pack a Paradox table, use the **compact** method defined for the Table type.

▪

compact example

The following example removes deleted records from the dBASE table named OLDDATA.DBF. This code is attached the **pushButton** method for the *purgeTable* button.

```
; purgeTable::pushButton
method pushButton(var eventInfo Event)
var
tb Table
tc TCursor
endVar
tb.attach("OldData.dbf")
   tb.setExclusive()            ; Get exclusive rights to the table.

   tc.open(tb)                  ; Associate TCursor with OldData table.

   if tc.compact() then         ; Remove all deleted records.
tc.close()
message("Old records purged.")
else
errorShow()
endIf
endMethod
```

▪

copy method

See also Example TCursor Type

Copies a table.

Syntax

```
1. copy ( const tableName String ) Logical
2. copy ( const tableName Table ) Logical
```

Description

copy copies a table to the destination table *tableName*. If *tableName* does not exist, **copy** creates it. If *tableName* already exists, **copy** overwrites it without asking for confirmation.

This method tries, for the duration of the retry period, to place a write lock on the source table and a full lock on the destination table. This method fails if either lock cannot be placed, or if the destination table is open.

This method does not respect the limits of restricted views.

See Copying to a different table type in the User's Guide help for information.

copy example

The following example copies the *Customer* table to the *NewCust* table.

This code uses the **isTable** method (from the DataBase type) to test whether *NewCust* exists; if it does, the user is prompted to confirm the action before *NewCust* is overwritten:

```
; copyCust::pushButton
method pushButton(var eventInfo Event)
var
  sourceTC TCursor
  destTb Table
endVar
destTb.attach("NewCust.db")
sourceTC.open("Customer.db")

; if NewCust.db exists, ask for confirmation
if isTable(destTb) then
  if msgYesNoCancel("Copy table", "Overwrite Newcust.db?") = "Yes" then

    ; copy Customer.db records to NewCust.db
; If .VAL file contains only RI info, it is not copied.
    sourceTC.copy(destTb)
  endIf
endIf

endMethod
```

▪

copyFromArray method

See also		Example		TCursor Type

Copies data from an array to the fields of the current record.

Syntax

```
1. copyFromArray ( const ar Array[ ] AnyType ) Logical
2. copyFromArray ( const ar DynArray[ ] AnyType ) Logical
```

Description

copyFromArray copies the elements of an array or a DynArray to the record pointed to by a TCursor, which must be in Edit mode.

Syntax 1 uses an array *ar*. The first element of the array is copied to the first field, the second element to the second field, and so on until the array is exhausted or the record is full.

Syntax 2 uses a DynArray *ar*, where each index is a field name or a field number, and the corresponding item is the field value.

The method fails if an attempt is made to copy an unassigned array element or if the structures do not match. (This can never happen if the array was created by **copyToArray**, because **copyToArray** assigns a blank value if a field is blank.) If there are more elements in the array than fields in the record, the extra elements are ignored. To copy a new record into an empty table, use **insertRecord** to insert a blank record before using **copyFromArray**.

■

copyFromArray example

In the following example, suppose CUSTNAME.DB has three fields: Last Name, A20; First name, A20; and Telephone, A12. This method associates a TCursor with the *CustName* table, creates an array with three elements, inserts a new record in the table, then uses **copyFromArray** to copy data from the array to the new record.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  aa Array[3] AnyType
endVar
aa[1] = "Borland"
aa[2] = "Frank"
aa[3] = "555-1212"
if tc.open("CustName.db") then  ; open table
  tc.edit()                     ; copyFromArray works only in Edit mode
  tc.insertRecord()             ; insert new record
  tc.copyFromArray(aa)          ; copy from array to table
  tc.endEdit()
else
  msgStop("Stop", "Couldn’t open CustName.db.")
endIf
endMethod
```

▪

copyRecord method

See also Example TCursor Type

Copies a record from one TCursor into another TCursor.

Syntax

`copyRecord ( const sourceTC TCursor ) Logical`

Description

copyRecord copies the record pointed to by one TCursor into the record pointed to by another TCursor. For example, the following code copies a record from the *sourceTC* TCursor into the *destinationTC* TCursor:

```
destinationTC.copyRecord(sourceTC)
```

The TCursor specified in *sourceTC* does not have to be in Edit mode; the TCursor you're copying to (the destination TCursor) does. This method fails if any field in the source record cannot be converted to the data type of the corresponding field in the destination record. This method returns True if it succeeds; otherwise, it returns False.

Note: You cannot use **copyRecord** to copy a record into an empty table. To copy a new record into an empty table, use **insertRecord**.

▪

copyRecord example

The following example uses a TCursor to scan the *Orders* table for sales posted in the last 10 days and copies them to the *NewOrdrs* table in the current directory. This code is attached to the **pushButton** method for the *getNewOrders* button.

```
; getNewOrders::pushButton
method pushButton(var eventInfo Event)
var
  ordTC,
   newOrdTC    TCursor
  tvNewOrds   TableView
endVar

ordTC.open("Orders.db")
newOrdTC.open("NewOrdrs.db")
newOrdTC.edit()                ; copyRecord only works in Edit mode.

; Scan Orders.db table for records posted in the last ten days.
scan ordTC for ordTC."Sale Date" >= (today() - 10) and
               ordTC."Sale Date" <= today() :
  newOrdTC.insertRecord()      ; Insert a new record in NewOrdrs.db.
  newOrdTC.copyRecord(ordTC)   ; Copy from Orders.db into NewOrdrs.db.
endScan
newOrdTC.endEdit()             ; End Edit mode for TCursor.

tvNewOrds.open("NewOrdrs.db")  ; Display the table.
endMethod
```

▪

copyToArray method

See also Example TCursor Type

Copies the fields of the current record to an array.

Syntax

```
1. copyToArray ( var ar Array[ ] AnyType ) Logical
2. copyToArray ( var ar DynArray[ ] AnyType ) Logical
```

Description

copyToArray copies the fields of the current record to the elements of an array specified in *ar*. You must declare the array to be of type AnyType, or of a type that matches every field in the table.

In syntax 1, where *ar* is a fixed or resizable array, the value of the first field is copied to the first element of the array, the value of the second field to the second element, and so on. If the array is resizeable, it grows automatically to hold the number of fields in the record. If the array is not resizeable, it holds as many fields as it can, and the rest are discarded.

If syntax 2, where *ar* is a DynArray, index values correspond to the field names and DynArray values correspond to field values:

ar [fieldName] = fieldValue

The size of the array is equal to the number of fields in the record (unless ar is a fixed array). The record number field and any display-only or calculated fields that appear in a form window of the table are not copied to the array.

copyToArray example

In the following example, assume a form has a table frame, CUSTOMER, bound to CUSTOMER.DB. When the user attempts to delete a CUSTOMER record, this code (attached to the built-in **action** method) uses **copyToArray** and **copyFromArray** to copy the record to an archive table, CUSTARC.DB. If CUSTARC.DB cannot be opened, this method informs the user and does not delete the record.

```
; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
  tcOrig, tcArc TCursor
  arcRec Array[] AnyType
endVar

if eventInfo.id() = DataDeleteRecord then ; when user deletes a record
  if thisForm.Editing = True then         ; if form is in Edit mode
    disableDefault                        ; don't delete the record

                                          ; ask for confirmation
    if msgQuestion("Confirm", "Delete record?") = "Yes" then

      tcOrig.attach(CUSTOMER)             ; sync TCursor to UIObject
      tcOrig.copyToArray(arcRec)          ; store the record in arcRec
      if tcArc.open("CustArc.db") then    ; True if tcArc can open CustArc
        tcArc.edit()                      ; copyFromArray requires Edit
        tcArc.insertAfterRecord()         ; create a new record
        tcArc.copyFromArray(arcRec)       ; copy arcRec to new record
        enableDefault                     ; delete the record in Customer
      else                                ; can't open Customer TCursor
        msgStop("Stop!", "Sorry, Can't archive record.")
      endIf
    else                                  ; user didn't confirm dialog box
      message("Record not deleted.")
    endIf

  else                                    ; not in Edit mode
    msgStop("Stop!", "Press F9 to edit data.")
  endIf
endIf
endMethod
```

▪

createIndex method

See also Example1 Example2 TCursor Type

Creates an index for a table.

Syntax

```
1. createIndex ( const attrib DynArray[ ] AnyType, const fieldNames Array[ ]
String ) Logical
2. createIndex ( const attrib DynArray[ ] AnyType, const fieldNums Array[ ]
SmallInt ) Logical
```

Description

createIndex creates an index for a table using attributes specified in the DynArray *attrib* and the field names (or numbers) specified in the Array *fieldNames* (or *fieldNums*). This method is provided as an alternative to the **index** structure, and performs the same task. It can be useful when you don't know the index structure beforehand (for example, when the information is supplied by the user).

Each key of the DynArray must be a string, and the value of each corresponding item is described in the following table. You do not have to include all the keys to use **createIndex**. Any key you omit is assigned the corresponding default value.

String value	Description
MAINTAINED	If True, the index is incrementally maintained. That is, after a table is changed, only that portion of the index affected by the change is updated. If False, Paradox does not maintain the index automatically. Maintained indexes typically result in better performance. Default = False (Paradox tables only).
PRIMARY	If True, the index is a primary index. If False, it's a secondary index. Default = False (Paradox tables only).
CASEINSENSITIVE	If True, the index ignores differences in case. If False, it considers case. Default = False (Paradox tables only).
DESCENDING	If True, the index is sorted in descending order, from highest values to lowest. If False, it is sorted in ascending order. Default = False.
UNIQUE	If True, records with duplicate values in key fields are not allowed. If False, duplicates are allowed.
IndexName	A name used to identify this index. No default value, unless you're creating a secondary, case-sensitive index on a single field, in which case the default value is the field name. For dBASE tables, the index name must be a valid DOS file name. If you do not specify an extension, .NDX is added automatically.
TagName	The name of the index tag associated with the index specified in *indexName* (dBASE tables only).

For information on indexes, see About keys and indexes in tables in the User's Guide help.

createIndex example 1

The following example builds a maintained secondary index for a Paradox table named CUSTOMER.DB. If the *Customer* table cannot be found, or cannot be locked, this method aborts the operation.

```
method pushButton(var eventInfo Event)
var
   tbCust         Table
   stTbName         String
   tcCust         TCursor
   arFieldNames   Array[3] String
   dyAttrib         DynArray[]AnyType
endVar

stTbName        = "Customer.db"

arFieldNames[1] = "Customer No"
arFieldNames[2] = "Name"
arFieldNames[3] = "Street"

dyAttrib["PRIMARY"]     = False
dyAttrib["MAINTAINED"] = True
dyAttrib["IndexName"]  = "NumberNameStreet"

if isTable(stTbName) then
   tbCust.attach(stTbName)
   tbCust.setExclusive()

   if tcCust.open(tbCust) = FALSE then
      msgStop("Stop!", "Can't lock " + stTbName + " table.")
      return
   endif

   if not tcCust.createIndex(dyAttrib, arFieldNames) then
      errorShow()
   endif

; This createIndex statement has the same effect
; as the following INDEX structure:

 {

  INDEX "Customer.db"
    MAINTAINED
    ON "Customer No", "Name", "Street"
    TO "NumberNameStree"
  ENDINDEX

  }

else
  msgStop("Stop!", "Can't find " + stTbName + " table.")
endIf

endMethod
```

▪

createIndex example 2

The following example builds a unique index named CITYSTAT.NDX for the dBASE table CUSTOMER.DBF.

```
; cityStateIndex::pushButton
method pushButton(var eventInfo Event)
var
   tbCust      Table
   stTbName      String
   tcCust         TCursor
   arFieldNames   Array[1] String
   dyAttrib      DynArray[]AnyType
endVar

stTbName        = "Cust.dbf"

arFieldNames[1] = "CITY"

dyAttrib["UNIQUE"]     = True
dyAttrib["MAINTAINED"] = True

; A dBASE index name must be a valid DOS file name.
; If an extension is omitted, .NDX is appended automatically.

dyAttrib["IndexName"] = "City"


if isTable(stTbName) then
   tbCust.attach(stTbName)
   tbCust.setExclusive()
   if tcCust.open(tbCust) = False then
      msgStop("Stop!", "Can't lock " + stTbName + " table.")
      return
     endif

  tcCust.createIndex(dyAttrib, arFieldNames)
; This createIndex statement has the same effect
; as the following INDEX structure:
{
  INDEX "Cust.dbf"
       UNIQUE
    ON "CITY", "STATE_PROV"
    TO "CityStat"
  ENDINDEX
}

else
  msgStop("Stop!", "Can't find " + stTbName + " table.")

endif

endMethod
```

▪

cSamStd method

See also Example TCursor Type

Returns the sample standard deviation of a field (column) of a table.

Syntax

```
1. cSamStd ( const fieldName String ) Number
2. cSamStd ( const fieldNum SmallInt ) Number
```

Description

cSamStd returns the sample standard deviation of values in a column of numeric fields. This method respects the limits of restricted views set by **setRange** or **setGenFilter.** The returned value is based on the sample variance. This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

The sample standard deviation (as opposed to population) is calculated using this formula:

sqrt(*TCursor*.cVar(*FieldName*) * (*n*/(*n*-1)))

where

variance = *TCursor*.cVar(*fieldName*)

n = TCursor.cCount(fieldName)

cSamStd example

The following example uses both forms of the syntax to calculate the sample standard deviation of two different fields in the *Answer* table. This code is attached to the **pushButton** method for *showSamStd*:

```
; showSamStd::pushButton
method pushButton(var eventInfo Event)
var
  empTC TCursor
  tblName String
  CalcSalary, CalcYears Number
endVar
tblName = "Answer"
if empTC.open(tblName) then
  CalcSalary = empTC.cSamStd("Salary")  ; get sample std deviation for
salaries
  CalcYears  = empTC.cSamStd(2)         ; assume "Years in service" is field
2
  msgInfo("Sample Std Deviation",       ; display info in a dialog box
          "Salaries : " + String(CalcSalary) + "\n" +
          "Years in service : " + String(CalcYears))
else
  msgInfo("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

▪

cSamVar method

See also Example TCursor Type

Returns the sample variance of a field (column) in a table.

Syntax

```
1. cSamVar ( const fieldName String ) Number
2. cSamVar ( const fieldNum SmallInt ) Number
```

Description

cSamVar returns the sample variance of the values in a column of fields. This method respects the limits of restricted views set by **setRange** or **setGenFilter.** This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

The sample variance (as opposed to population) is calculated using this formula:

TCursor.cVar(*fieldName*) * (*n*/(*n*-1))

where

n = TCursor.cCount(fieldName)

▪

cSamVar example

The following example uses both forms of the syntax to calculate the sample variance of two different fields in the *Answer* table. This code is attached to the **pushButton** method for *showSamVar*.

```
; showSamVar::pushButton
method pushButton(var eventInfo Event)
var
  empTC TCursor
  tblName String
  CalcSalary, CalcYears Number
endVar
tblName = "Answer"
if empTC.open(tblName) then
  CalcSalary = empTC.cSamVar("Salary") ; get sample variance for salaries
  CalcYears  = empTC.cSamVar(2)        ; assume "Years in service" is field 2
  msgInfo("Sample Variance",           ; display info in a dialog box
          "Salaries : " + String(CalcSalary) + "\n" +
          "Years in service : " + String(CalcYears))
else
  msgInfo("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

▪

cStd method

See also Example TCursor Type

Returns the standard deviation of a field (column) in a table.

Syntax

```
1. cStd ( const fieldName String ) Number
2. cStd ( const fieldNum SmallInt ) Number
```

Description

cStd returns the population standard deviation of the values in a column of fields. The calculation is based on the variance. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

▪

cStd example

In the following example, the **pushButton** method for *thisButton* calculates the population standard deviation for two separate fields and displays the results in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  test1, test2 Number
endVar
tc.open("scores.dbf")
test1 = tc.cStd("Test1")
test2 = tc.cStd(2)             ; assumes Test2 is field 2

; show results in a dialog
msgInfo("Standard Deviation",
        "Test1 results : " + String(test1) + "\n" +
        "Test2 results : " + String(test2))

endMethod
```

▪

cSum method

See also Example TCursor Type

Returns the sum of the values in a field (column) of a table.

Syntax

```
1. cSum ( const fieldName String ) Number
2. cSum ( const fieldNum SmallInt ) Number
```

Description

cSum returns the sum of the values in a column of fields. This method respects the limits of restricted views set by **setRange** or **setGenFilter.** This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

cSum example

In the following example, the **pushButton** method for *sumOrders* uses both forms of **cSum** syntax to calculate totals for two fields in ORDERS.DB:

```
; sumOrders::pushButton
method pushButton(var eventInfo Event)
var
  orderTC TCursor
  orderTotal, amtPaid Number
  tblName String
endVar
tblName = "Orders"
if orderTC.open(tblName) then
  orderTotal = orderTC.cSum("Total Invoice")  ; get sum for Total Invoice
field
  amtPaid    = orderTC.cSum(7)                ; assumes Amount Paid is field
7
  msgInfo("Order Totals",
          "Total Orders : " + String(orderTotal) + "\n" +
          "Total Receipts : " + String(amtPaid))
else
  msgInfo("Sorry", "Can’t open " + tblName + " table.")
endIf
endMethod
```

▪

currRecord method

See also Example TCursor Type

Reads the current record into the record buffer.

Syntax

`currRecord ( ) Logical`

Description

currRecord reads values of the current record from the underlying table into the record buffer. Any unposted changes to the TCursor are canceled. This method ensures you’re working with the most recently updated version of the record, particularly on a network.

▪

currRecord example

The following example is part of a system that processes ticket orders for concerts. It finds out which artist the customer wants to see, then finds out how many seats the customer needs.

```
; updateSeats::pushButton
method pushButton(var eventInfo Event)
   var
      tcConcert      TCursor
      siSeatsNeeded,
      siCustSeats      SmallInt
      stArtist      String
   endVar

   ; Call a custom method to find out which artist
   ; the customer wants to see.
   stArtist = getArtistName()
   tcConcert.open("concerts")
   tcConcert.locate("Artist", stArtist)

   if tcConcert.SoldOut = True then
         msgStop("Sorry", "Sold out")
         return
   else

      ; Call a custom method to find out how many seats
      ; the customer needs (this may take awhile).
      siCustSeats = getCustSeats()

      ; Meanwhile, other customers may have ordered seats for this
      ; concert, so read current values into the record buffer.
      tcConcert.currRecord()

      if tcConcert.Seats >= siCustSeats then
            processOrder() ; Call a custom method to process the order.
      else
            notEnoughSeats() ; Call a custom method.
      endIf

endMethod
```

■

cVar method

See also Example TCursor Type

Returns the variance of a field (column) in a table.

Syntax

```
1. cVar ( const fieldName String ) Number
2. cVar ( const fieldNum SmallInt ) Number
```

Description

cVar returns the population variance of values in a column of numeric fields. This method respects the limits of restricted views set by **setRange** or **setGenFilter.** **cVar** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a read lock on the table. If a lock cannot be placed, the method fails.

▪

cVar example

In the following example, the **pushButton** method for *thisButton* calculates the population variance deviation for two separate fields and displays the results in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myTable TCursor
  test1, test2 Number
endVar
myTable.open("scores.dbf")
test1 = myTable.cVar("Test1")      ; get Test1 cVar
test2 = myTable.cVar(2)            ; assumes Test2 is field 2
msgInfo("Population Variance",
        "Test1 results : " + String(test1) + "\n" +
        "Test2 results : " + String(test2))
endMethod
```

▪

deleteRecord method

See also Example TCursor Type
Beginner

Deletes the record pointed to by a TCursor.

Syntax

```
deleteRecord ( ) Logical
```

Description

deleteRecord deletes the record pointed to by a TCursor without prompting for confirmation. The operation cannot be undone for Paradox tables; but it can be undone for dBASE tables. The table must be in Edit mode.

If the record is locked or has already been deleted by another user (in a dBASE table), this method fails.

▪

deleteRecord example

In the following example, the **pushButton** method for the *checkIOU* button determines whether a particular debt has been marked as paid; if it has, this code uses **deleteRecord** to delete the record:

```
; checkIOU::pushButton
method pushButton(var eventInfo Event)
var
  iou TCursor
  searchName String
endVar
searchName = "Hall"
iou.open("iou.db")
iou.edit()
if iou.locate("Name", searchName) then
  if iou."paid" = "Yes" then
    iou.deleteRecord()           ; delete the current record
    message(searchName + " deleted")
  else
    sendBill()                   ; run custom procedure
  endIf
else
   msgStop("Stop", "Couldn’t find " + searchName)
endIf
endMethod
```

didFlyAway method

See also Example TCursor Type

Reports whether the current record moved to a different position as the result of a key value change.

Syntax

```
didFlyAway ( ) Logical
```

Description

didFlyAway returns True if the most recent call to **unlockRecord** caused the record to move to a different position in the table; otherwise, it returns False. This method is relevant only if the **setFlyAwayControl** method has been set to True; otherwise, **didFlyAway** returns False (even if the record moved to its sorted position).

▪

didFlyAway example

The following example demonstrates how **setFlyAwayControl** affects the position of a TCursor after a call to **unlockRecord** and under what circumstances **didFlyAway** returns True.

```
; demoButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

tc.open("MyTable.db")

; Assume that MyTable.db has the following
; values in its only key field, "Customer No" :
;  Record#  Customer No
;     1        110
;     2        120   ; the code below changes this value to 145
;     3        130
;     4        140
                     ; which moves the record to this position
;     5        150

tc.setFlyAwayControl(Yes) ; Enable flyaway tracking.

if tc.locate("Customer No", 120) then
  tc.edit()

  ; Change the key value so that the record
  ; changes relative position.
  tc."Customer No" = 145

  tc.unlockRecord()       ; Unlock the record.

  ; The dialog box displays True because the new key value
  ; changes the record’s relative position in the table.
  msgInfo("Did 145 fly away?", tc.didFlyAway())

else
  message("120 not found.")
endIf

endMethod
```

dmAttach method

See also Example TCursor Type

Associates a TCursor with a table in the data model.

Syntax

```
dmAttach ( const dmTableName String ) Logical
```

Description

dmAttach associates a TCursor with the table specified in *dmTableName*. The table must be in the data model. This method returns True if it succeeds; otherwise, it returns False.

dmAttach example

The following example demonstrates using **dmAttach** to open a TCursor to a table in the data model. The TCursor respects the restricted view of the data model. **cSum** is used to gather information stored in the string variables *s1, s2,* and *s3.* Finally, a dialog box displays the information to the user.

```
;btnCustomerSummary :: pushButton
method pushButton(var eventInfo Event)
var
      tc   TCursor
      s1   String
      s2   String
      s3   String
endVar
tc.dmAttach("Orders.db")
s1 = string(tc.cSum("Total Invoice"))
s2 = string(tc.cSum("Amount Paid"))
s3 = string(tc.cSum("Balance Due"))

msgInfo("Customer Summary",
"Total Orders = " + s1 +
"\nTotal Paid = " + s2 +
"\nTotal Due = " + s3)
endMethod
```

▪

dropGenFilter method

See also Example1 Example2 TCursor Type

Drops (removes) the filter criteria associated with a TCursor.

Syntax

```
dropGenFilter ( ) Logical
```

Description

dropGenFilter drops (removes) the filter criteria associated with a TCursor, leaving it unfiltered. Indexes and tags (if any) remain in effect.

dropGenFilter example 1

The following example attaches a TCursor to a table frame bound to the *Orders* table. This method calculates the average total invoice amount for the entire table. To do so, it first calls **dropGenFilter** to remove any filter criteria that may have been set by the user (or by code executing elsewhere in the application). The call to **dropGenFilter** operates on the TCursor only; it does not affect the table frame.

```
; btnCalAvgInvoice::pushButton
method pushButton(var eventInfo Event)
var
      ordersTC   TCursor
      nuAvgInvoice   Number
endVar
   ordersTC.attach(Orders)     ; Attach to the Orders table frame.
   ordersTC.dropGenFilter()    ; Remove any filters on the TCursor.

nuAvgInvoice = ordersTC.cAverage("Total Invoice")
nuAvgInvoice.view("Average Total Invoice:")
endMethod
```

▪

dropGenFilter example 2

In the following example, a form contains a button called *btnCascadeDelete*. The **pushButton** method for *btnCascadeDelete* attaches a TCursor to a child table, uses **dropGenFilter** to make sure the TCursor can see all the child records, moves the TCursor to the first record, and puts it in edit mode. Then a **while** loop is used to delete all the child records. Finally, the form is put into edit mode and the parent record is deleted.

```
;btnCascadeDelete::pushButton
method pushButton(var eventInfo Event)
var
      tc                TCursor
      siCounter   SmallInt
endVar
   tc.attach(LINEITEM)      ;Attach to detail table.
   tc.dropGenFilter()      ;Drop any user set filters.
   tc.home()         ;Make sure TCursor is on first record.

tc.edit()
   while not tc.eot()      ;If there are any child
      tc.deleteRecord()      ;records, delete all of them.
endWhile
   edit()               ;Make sure form is in edit mode.
   Order_No.deleteRecord()   ;Delete the parent record.
endMethod
```

▪

dropIndex method

See also Example TCursor Type

Deletes an index file associated with a table.

Syntax

`1. (Paradox tables) dropIndex ( const indexName String ) Logical`

`2. (dBASE tables) dropIndex ( const indexName String [ , const tagName String ] ) Logical`

Description

dropIndex deletes a specified index file or index tag. You can't delete an index that's in use.

When working with a Paradox table, *indexName* is required. It specifies a secondary index; you can't use a TCursor to drop the primary index of a Paradox table.

When working with a dBASE table, you can use *indexName* to specify a .NDX file, or use *indexName* and *tagName* to specify a .MDX file and an index tag.

Note: You must obtain exclusive rights to the table (by opening the TCursor on a Table variable that has called the Table method **setExclusive** *before* opening the table) before calling **dropIndex**.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

dropIndex example

In the following example, the **pushButton** method for a button deletes the *CustName* tag from the .MDX file for a dBASE table.

```
method pushButton(var eventInfo Event)
var
  tc1 TCursor
  tb1 Table
endVar

if isTable("Sales.dbf") then
  tb1.attach("Sales.dbf") ; Sales.dbf is a dBASE table
  tb1.setExclusive (Yes)
  tc1.open(tb1)
; delete CustName tag from index2 file
  if tc1.dropIndex("index2.mdx", "CustName") then
     msgInfo("", "custname dropped")
   else
      errorShow("Could not drop index."))
else
  msgStop("Stop!", "Could not find Sales.dbf table.")
endIf

endMethod
```

■

edit method

See also Example TCursor Type
Beginner

Puts a TCursor into Edit mode.

Syntax
```
edit ( ) Logical
```

Description

edit puts a TCursor into Edit mode so changes can be made to the current record. After editing, if you want to stay in Edit mode, move off the record or use **postRecord** or **unlockRecord** to accept changes to the record. If you want to leave Edit mode, use **cancelEdit** to cancel changes to the record or use **endEdit** to accept changes.

▪

edit example

The following example creates an array and uses **copyFromArray** to copy the contents of the array to a new record in the *CustName* table. Because the TCursor must be in Edit mode before the new record is inserted, this code uses **edit** to start editing the table. After the new record is inserted, this code uses **endEdit** to end Edit mode and accept changes.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  aa Array[3] AnyType
endVar
aa[1] = "Borland"
aa[2] = "Frank"
aa[3] = "555-1212"
if tc.open("custname.db") then  ; open table
  tc.edit()                      ; put TCursor in Edit mode
  tc.insertRecord()              ; insert new record
  tc.copyFromArray(aa)           ; copy from array to table
  tc.endEdit()                   ; end Edit mode
else
  msgStop("Stop", "Couldn’t open Custname.db.")
endIf
endMethod
```

▪

empty method

See also Example TCursor Type

Deletes all records from a table.

Syntax

```
empty ( ) Logical
```

Description

empty deletes all records from a table without prompting for confirmation. If the TCursor is associated with a dBASE table, the records are flagged as deleted, and then the table is compacted (packed), if possible. In any case, the TCursor does not have to be in Edit mode, but a write lock (at least) is required. This operation cannot be undone.

empty removes information from the table, but does not delete the table itself. Compare this method to Table::**delete**, which does delete the table. This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

This method first tries to gain exclusive rights to the table. If exclusive rights are not possible, **empty** tries to place a write lock on the table.

If exclusive rights are possible, this method deletes all records in the table at once. If only a write lock is possible, **empty** must delete each record one at a time. (This can be slow for large tables.)

If **empty** is able to gain exclusive rights to a dBASE table, all records are deleted and the table is compacted (records are permanently removed). If only a write lock is possible, this method flags all records as deleted, but does not remove them from the table. (Records can be undeleted from a dBASE table if they have not been removed with the **compact** method.)

▪

empty example

The following example prompts the user for confirmation before deleting all records from the *Scratch* table. If the user does not confirm the action, this code uses **nRecords** to indicate how many records exist in SCRATCH.DB.

```
; tblEmpty::pushButton
method pushButton(var eventInfo Event)
var
  tblName String
  tc TCursor
endVar

tblName = "Scratch.db"
if isTable(tblName) then
  tc.open(tblName)
  if msgQuestion("Confirm", "Empty " + tblName + " table?") = "Yes" then
    tc.empty()
    message("All " + tblName + " records have been deleted.")
  else
    message(tblname + " has " + String(tc.nRecords()) + " records.")
  endIf
else
  msgInfo("Error", "Can’t find " + tblName + " table.")
endIf
endMethod
```

■

end method

See also Example TCursor Type
Beginner

Moves a TCursor to the last record in a table.

Syntax
`end ( ) Logical`

Description
end sets the current record (and the record buffer) to the last record in a table.

▪

end example

The following example uses **end** to move a TCursor to the last record in the *Orders* table, then displays in a dialog box information in the last record.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.db")         ; open tc for Orders table
tc.end()                     ; move to the last record in the table
                             ; display info in last record
msgInfo("Customer No " + tc."Customer No",
        "Outstanding balance: " + tc."Balance Due")

endMethod
```

▪

endEdit method

See also	Example	TCursor Type
Beginner

Exits Edit mode and accepts changes to the current record.

Syntax

`endEdit ( ) Logical`

Description

endEdit accepts changes to the current record and exits Edit mode. It does not close the TCursor. (Changes to previous records are committed or canceled as the user navigates through the table.)

endEdit example

The following example creates an array and uses **copyFromArray** to copy the contents of the array to a new record in the *CustName* table. Because *CustName* must be in Edit mode before the new record is inserted, this code uses **edit** to start editing the table. After the new record is inserted, this code uses **endEdit** to exit Edit mode.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  aa Array[3] AnyType
endVar
aa[1] = "Borland"
aa[2] = "Frank"
aa[3] = "555-1212"
if tc.open("custname.db") then  ; open table
  tc.edit()                     ; put TCursor in Edit mode
  tc.insertRecord()             ; insert new record
  tc.copyFromArray(aa)          ; copy from array to table
  tc.endEdit()                  ; end Edit mode
else
  msgStop("Stop", "Couldn't open Custname.db.")
endIf
endMethod
```

▪

enumFieldNames method

See also Example TCursor Type

Fills an array with the names of fields in a table.

Syntax

`enumFieldNames ( const fieldArray Array[ ] String ) Logical`

Description

enumFieldNames fills *fieldArray* with the names of the fields in a table. If *fieldArray* is resizeable, it grows automatically to hold the field names; if it is not resizeable, it holds as many as it can, and discards the rest. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

enumFieldNames example

For the following example, the **pushButton** method for the *enumFields* button stores field names in a resizeable array, then uses **view** to display the contents of the array.

```
; enumFields::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fieldNames Array[] String     ; field names for tables are always strings
endVar
if tc.open("orders.db") then
  tc.enumFieldNames(fieldNames) ; load fieldNames with names of Orders.db
fields
  fieldNames.view()             ; display field names in a dialog box
else
  msgStop("Stop", "Couldn't open Orders.db.")
endIf

endMethod
```

▪

enumFieldNamesInIndex method

See also Example TCursor Type

Fills an array with the names of fields in a table’s index.

Syntax

1. (Paradox tables) **enumFieldNamesInIndex (** [const ***indexName*** String,] var ***fieldArray*** Array[ ] String **)** Logical

2. (dBASE tables) **enumFieldNamesInIndex (** [const ***indexName*** String [, const ***tagName*** String,] var ***fieldArray*** Array[ ] String **)** Logical

Description

enumFieldNamesInIndex fills *fieldArray* with the names of the fields in a table’s index, as specified in *indexName*. If *indexName* is omitted, this method uses the current index. If *fieldArray* is resizeable, it grows automatically to hold the field names; if it is not resizeable, it holds as many as it can, and discards the rest. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

When working with a dBASE table, you can use the optional argument *tagName* to specify an index tag within a .MDX file.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

enumFieldNamesInIndex example

In the following example, the **pushButton** method for the *enumIndex* button stores field names in a resizeable array, then uses **view** to display the contents of the array:

```
; enumIndex::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fieldNames Array[] String
endVar
if tc.open("Sales.dbf") then
  ; load fieldNames array with field names in the byDate index
  tc.enumFieldNamesInIndex("DateIndx.MDX", "byDate", fieldNames)
  ; display the index field names for byDate in DateIndx
  fieldNames.view()
else
  msgStop("Stop", "Couldn't open Sales.dbf.")
endIf
endMethod
```

▪

enumFieldStruct method

See also   Example   TCursor Type

Lists the field structure of a TCursor.

Syntax

```
1. enumFieldStruct ( const tableName String ) Logical
2. enumFieldStruct ( inMem TCursor ) Logical
```

Description

enumFieldStruct lists the field structure of a TCursor. Syntax 1 creates a Paradox table; syntax 2 (added in version 5.0) stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* exists, this method overwrites it without confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory. You can supply *tableName* to the **struct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In syntax 2 (added in version 5.0), the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Using syntax 2 may result in faster performance because the information is stored in system memory without the overhead of disk access.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field name	Field type	Description
Field Name	A31	Name of field.
Type	A31	Data type of field.
Size	S	Size of field.
Dec	S	Number of decimal places, or 0 if field type doesn't support decimal places.
Key	A1	Is it a key field? * = key field, blank = not key field.
_Required Value	A1	Is the field required? T = required, N (or blank) = Not required.
_Min Value	A255	Field's minimum value, if specified; otherwise blank.
_Max Value	A255	Field's maximum value, if specified; otherwise blank.
_Default Value	A255	Field's default value, if specified; otherwise blank.
_Picture Value	A175	Field's picture, if specified; otherwise blank.
_Table Lookup	A255	Name of lookup table. Includes full path if the lookup table is not in :WORK:
_Table Lookup Type	A1	Type of lookup table. 0 (or blank) = no lookup table, 1 = Paradox
_Invariant Field ID	S	Field's ordinal position in table (first field = 1, second field = 2, etc.)

■

enumFieldStruct example

For the following example, assume that you want a new table called *NewCust* that is similar to the *Customer* table. However, you want all of the fields in *NewCust* to be required fields. To accomplish this, the following code uses **enumFieldStruct** to load a new table (CUSTFLDS.DB) with the field-level information from *Customer*. The code then scans through *CustFlds* and modifies the field definitions so that each record describes a field that will be required. *CustFlds* is then supplied in the **struct** clause of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  newCustTbl Table
  tc   TCursor
  structName, sourceName String
endVar

structName = "CustFlds.db"
sourceName = "Customer.db"

if tc.open(sourceName) then

  tc.enumFieldStruct(structName)

  ; Point the TCursor to the CustFlds table.
  tc.open(structName)
  tc.edit()

  ; This loop scans through the CustFlds table and
  ; changes ValCheck definitions for every field.
  scan tc :
    tc."_Required Value" = 1    ; Make all fields required.
  endScan

  ; Now create NEWCUST.DB and borrow field names,
  ; ValChecks and key fields from CUSTFLDS.DB.
  newCustTbl = CREATE "NewCust.db"
                 STRUCT structName
               ENDCREATE

  ; NEWCUST.DB requires that all fields be filled.

else
  msgStop("Error", "Can’t get field structure for Customer table.")
endIf

endMethod
```

▪

enumIndexStruct method

[See also](#) [Example](#) [TCursor Type](#)

Lists the structure of a TCursor's secondary indexes.

Syntax

```
1. enumIndexStruct ( const tableName String ) Logical
2. enumIndexStruct ( inMem TCursor ) Logical
```

Description

enumIndexStruct lists the structure of a table's secondary indexes. Syntax 1 creates a Paradox table; syntax 2 (added in version 5.0) stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. For dBASE tables, this method lists the structure of the indexes associated with the table by the **usesIndexes** method. If *tableName* exists, this method prompts the user for confirmation before overwriting the table. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory. You can supply *tableName* to the **indexStruct** option in a **create** statement to borrow secondary indexes for use in the new table.

In syntax 2, the structure information is stored in a TCursor variable *inMem* that you pass as an argument. Using syntax 2 may result in faster performance, because the information is stored in system memory without the overhead of disk access.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field name	Field type	Description
infoHeader	A1	If Y, this record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field. If N, this record is not a header.
szName	A255	Index name, including path.
szTagName	A31	Tag name, no path (dBASE only).
szFormat	A31	Optional index type, e.g., BTREE, HASH
bPrimary	A1	Y if the index is primary; otherwise blank.
bUnique	A1	Y if the index is unique; otherwise blank.
bDescending	A1	Y if the index is descending; otherwise blank.
bMaintained	A1	Y if the index is maintained; otherwise blank.
bCaseInsensitive	A1	Y if the index is not case-sensitive; otherwise blank.
bSubset	A1	Y if the index is a subset index (dBASE only); otherwise blank.
bExpIdx	A1	Y if the index is an expression index (dBASE only); otherwise blank.
iKeyExpType	N	Key type of index expression (dBASE only).
szKeyExp	A220	Key expression for expression index (dBASE only).
szKeyCond	A220	Subset condition for subset index (dBASE only).
FieldNo	N	Ordinal position of key field in table.
FieldName	A31	Name of key field.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

▪

enumIndexStruct example

For the following example, assume that you want a new table called *NewCust* that is similar to the *Customer* table. However, you don't want to borrow referential integrity or security information. To accomplish this, the following code uses **enumFieldStruct** and **enumIndexStruct** to generate two tables: CUSTFLDS.DB and CUSTINDX.DB. *CustFlds* and *CustIndx* are then supplied to the **struct** and **indexStruct** clauses of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  newcustTC Table
  custTC    TCursor
endVar

if custTC.open("Customer.db") then

  ; write field level information to CUSTFLDS.DB
  custTC.enumFieldStruct("CustFlds.db")

  ; write secondary index information to CUSTINDX.DB
  custTC.enumIndexStruct("CustIndx.db")

  ; now create NEWCUST.DB▪
  ; borrow field names, ValChecks, and key fields from CUSTFLDS.DB
  ; borrow secondary indexes from CUSTINDX.DB
  newcustTC = CREATE "NewCust.db"
                STRUCT "CustFlds.db"
                INDEXSTRUCT "CustIndx.db"
              ENDCREATE

else
  msgStop("Error", "Can't find Customer table.")
endIf

endMethod
```

▪

enumLocks method

See also Example TCursor Type

Creates a Paradox table listing the locks currently applied to a table.

Syntax

```
enumLocks ( const tableName String ) LongInt
```

Description

enumLocks creates the Paradox table specified in *tableName*, and the return value indicates the number of locks on the specified table. *tableName* lists the locks currently applied to the table pointed to by a TCursor. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. For dBASE tables, this method lists only the lock you've placed (not all locks currently on the table). You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName*, which changed in version 5.0, is listed below:

Field name	Field type	Description
UserName	A15	User name.
LockType	A32	String describing type of lock, for example, Table Write Lock.
NetSession	N	Net level session number.
Session	N	BDE session number, if the lock was placed by BDE.
RecordNumber	N	Record number, if the lock is a record lock; otherwise 0.

enumLocks example

In the following example, the built-in **pushButton** method for the *showOrdersLcks* button creates a table listing the locks currently applied to ORDERS.DB and opens the newly created table.

```
; showOrdersLcks::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tv TableView
endVar
if tc.open("Orders.db") then
  tc.enumLocks("OrderLck.db")  ; store Orders.db locks in OrderLck.db
  tv.open("OrderLck.db")       ; open OrderLck.db
else
  msgStop("Stop!", "Can't open Orders.db table")
endIf

endMethod
```

▪

enumRefIntStruct method

See also Example TCursor Type

Lists referential integrity information for a TCursor.

Syntax

```
1. enumRefIntStruct ( const tableName String ) Logical
2. enumRefIntStruct ( inMem TCursor ) Logical
```

Description

enumRefIntStruct lists referential integrity information for a TCursor. Syntax 1 creates a Paradox table; syntax 2 (added in version 5.0) stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory. You can supply *tableName* to the **refIntStruct** option in a **create** statement to borrow referential integrity information for use in the new table.

In syntax 2, the structure information is stored in a TCursor variable *inMem* that you pass as an argument. Using syntax 2 may result in faster performance because the information is stored in system memory without the overhead of disk access.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field name	Type	Description
infoHeader	A1	If Y, this record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field. If N, this record is not a header.
RefName	A31	A name to identify this referential integrity constraint.
OtherTable	A255	Name (including path) of the other table in the referential integrity relationship.
Slave	A1	Y if the able is slave, not master (i.e., table is dependent); otherwise blank.
Modify	A1	Specifies update rule: Y = Cascade, blank = Prohibit.
Delete	A1	Specifies delete rule: blank = Prohibit. Paradox does not support cascading deletes for Paradox or dBASE tables.
FieldNo	N	Ordinal position of the field in this table involved in a referential integrity relationship.
aiThisTabField	A31	Name of the field in this table involved in a referential integrity relationship.
Other FieldNo	N	Ordinal position of the field in the other table involved in a referential integrity relationship.
aiOthTabField	A31	Name of the field in the other table involved in a referential integrity relationship.

▪

enumRefIntStruct example

The following example uses **enumRefIntStruct** to write CUSTOMER.DB referential integrity information to the *CustRef* table. Then, the code supplies *CustRef* to the **refIntStruct** clause in a **create** statement.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc  TCursor
  tbl Table
endVar

tc.open("Customer.db")

; Write referential integrity information to CustRef.
tc.enumRefIntStruct("CustRef.db")

; Write field level information to CustFlds.
tc.enumFieldStruct("CustFlds.db")

; Now create NEWCUST.DB.
; Borrow field level information from CUSTFLDS.DB.
; Borrow referential integrity information from CUSTREF.DB.
tbl = CREATE "NewCust.db"
        STRUCT "CustFlds.db"
        REFINTSTRUCT "CustRef.db"
      ENDCREATE

endMethod
```

▪

enumSecStruct method

See also Example TCursor Type

Lists security information of a TCursor.

Syntax

```
1. enumSecStruct ( const tableName String ) Logical
2. enumSecStruct ( inMem TCursor ) Logical
```

Description

enumSecStruct lists the security information (access rights) of a TCursor. Syntax 1 creates a Paradox table; syntax 2 (added in version 5.0) stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory. You can supply *tableName* to the **secStruct** option in a **create** statement to borrow security information for use in the new table.

In syntax 2, the structure information is stored in a TCursor variable *inMem* that you pass as an argument. Using syntax 2 may result in faster performance because the information is stored in system memory without the overhead of disk access.

The structure of the table (syntax 1) or TCursor (syntax 2) is listed in the following table:

Field name	Type	Description
infoHeader	A1	If Y, this record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field. If N, this record is not a header.
iSecNum	N	Number to identify security description (first description = 1).
eprvTable	N	Table privilege value.
eprvTableSym	A10	Table privilege name.
iFamRights	N	Family rights value.
iFamRightsSym	A10	Family rights name.
szPassword	A31	Password.
fldNum	N	Ordinal position of field in table.
aprvFld	N	Field privilege value.
aprvFldSym	A10	Field privilege name.

▪

enumSecStruct example

The following example creates a new table based on the security information associated with the *Secrets* table. The code uses **enumSecStruct** to write security information to the *SecInfo* table, then uses the table to create the *MySecrts* table.

```
method pushButton(var eventInfo Event)
var
  tc  TCursor
  tbl Table
endVar

; Associate tc with SECRETS.DB.
tc.open("Secrets.db")
; Write security information to SECINFO.DB.
tc.enumSecStruct("SecInfo.db")

; Now create MYSECRTS.DB.
; Borrow field names and types from SECRETS.DB.
; Borrow security information from SECINFO.DB.
tbl = CREATE "MySecrts.db"
        LIKE "Secrets.db"
        SECSTRUCT "SecInfo.db"
      ENDCREATE
endMethod
```

Privilege values and names for TCursor::enumSecStruct

The following table lists numeric values and symbolic names for table and field privileges. The concept of family rights applies to tables created in Paradox version 3.5 or earlier.

Value	Name	Description
0	None	No privileges.
1	ReadOnly	Read-only field or table.
3	Modify	Read and modify field or table.
7	Insert	Insert + all of the above privileges (table only).
15	InsDel	Delete + all of the above privileges (table only).
31	Full	Full rights (table only).
255	Unknown	Privileges unknown.

Family rights values and names for TCursor::enumSecStruct

The following table lists numeric values and symbolic names for family rights.

Value	Name	Description
0	NoFamRights	No family rights.
1	FormRights	Can change forms only.
2	RptRights	Can change reports only.
4	ValRights	Can change val checks only.
8	SetRights	Can change image settings.
15	AllFamRights	All of the above.

enumTableProperties method

See also Example TCursor Type

Writes the properties of a TCursor to a Paradox table.

Syntax

```
enumTableProperties ( const tableName String ) Logical
```

Description

enumTableProperties writes the properties of a table associated with a TCursor to the table specified in *tableName*. If *tableName* exists, this method prompts the user for confirmation before overwriting the table. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is listed in the following table.

Field name	Field type	Description
TableName	A32	Name of table (name only: no path, no extension).
PropertyName	A64	Name of property. The following properties are valid for Paradox and dBASE tables: Name, Type, FieldCount, LogicalRecordSize, PhysicalRecordSize, KeySize, IndexCount, ValCheckCount, ReferentialCount, BookMarkSize, StableBookMarks, OpenMode, ShareMode.
PropertyValue	A255	Value of corresponding property.

enumTableProperties example

The following example uses **enumTableProperties** to write ORDERS.DB properties to ORDPROPS.DB. If ORDPROPS.DB exists, this code prompts the user for confirmation before overwriting the table.

```
; showTblProps::pushButton
method pushButton(var eventInfo Event)
var
  tblName, propTbl String
  tc TCursor
  tv TableView
endVar
tblName = "Orders.db"
propTbl = "OrdProps.db"

if tc.open(tblName) then
  if isTable(propTbl) then
    if msgYesNoCancel("Confirm",
       propTbl + " exists. Overwrite it?") <> "Yes" then
      return
    endIf
  endIf
  ; Write Orders.db properties to OrdProps.db.
  tc.enumTableProperties(propTbl)
  ; Open newly created OrdProps.db table.
  tv.open(propTbl)
else
  msgStop("Stop!", "Can't open " + tblName + " table.")
endIf

endMethod
```

▪

eot method

See also Example TCursor Type

Tests for a move past the end of a table.

Syntax

`eot ( ) Logical`

Description

eot returns True if a command attempts to move past the last record of a table; otherwise, it returns False. **eot** is reset by the next move operation.

eot (and **bot**) also return True if a command forces the TCursor to point to a nonexistent record. For example, suppose the *Customer* table has values in the first key field that range from 1,000 to 10,000. If you call **setRange** such that the TCursor points to key values from 1 to 10 (outside the possible range of *Customer* values), the TCursor points to a nonexistent record. The following code fragment demonstrates how **setRange** can affect **eot** and **bot**:

```
var tc TCursor endvar
tc.open("Customer.db")
; Suppose values in field 1 range from 1,000 to 10,000.
tc.setRange(1, 10)           ; filter ranges from 1 to 10
  ; tc.eot() and tc.bot() are True at this point
```

Similarly, if a call to **setGenFilter** forces the TCursor to point to a nonexistent record, **eot** and **bot** methods return True.

▪

eot example

In the following example, a **while** loop controls a TCursor’s movement through the *Orders* table. When code within the loop attempts to move beyond the end of the table, **eot** returns True and the loop terminates.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tblName String
  fldVal AnyType
endVar
tblName = "Customer.db"
if tc.open(tblName) then
  while tc.eot() = False      ; While subsequent commands do not
                              ; move past end of the table,
    message(tc."Customer No") ; display value in Customer No field,
    sleep(250)                ; pause for the message,
    tc.nextRecord()           ; move to the next record.
  endWhile
  msgInfo("End", "That’s all, folks!")
else
  msgStop("Stop!", "Can’t open " + tblName + " table.")
endIf
endMethod
```

▪

familyRights method

See also Example TCursor Type

Tests for a user’s ability to create or modify objects in a table’s family.

Syntax

```
familyRights ( const rights String ) Logical
```

Description

familyRights returns True if you have rights to the type of object specified in *rights*; otherwise, it returns False. *rights* is a single-letter string▪either "F" (form), "R" (report), "S" (image settings), or "V" (validity checks)

▪that indicates the type of object you are interested in. This method preserves functionality required by Paradox 3.5 tables, which had the concept of a table family. The concept does not apply to tables created in versions of Paradox after 3.5.

familyRights example

The following example indicates in a dialog box whether you have "F" rights to CUSTOMER.DB.

```
; showFRights::pushButton
method pushButton(var eventInfo Event)
var
  custTC TCursor
endVar

custTC.open("Customer.db")
msgInfo("Rights", "Form Rights: " + String(custTC.familyRights("F")))
; Displays True if you have Form rights to CUSTOMER.DB.

endMethod
```

▪

fieldName method

See also   Example   TCursor Type

Returns the name of a field.

Syntax

```
fieldName ( const fieldNum SmallInt ) String
```

Description

fieldName returns the name of field *fieldNum*. Fields are numbered from left to right, beginning with 1.

fieldName example

The following example uses **fieldName** to display the name of field number two in the *Answer* table. This code is attached to the built-in **pushButton** method of a button.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fldName, tblName String
  fldNum SmallInt
endVar
tblName = "Answer.db"

if tc.open(tblName) then
  fldName = tc.fieldName(2)       ; store name of field 2 in fldName
  msgInfo("Field Name",           ; display field 2 field name
          "Field name for field 2 is\n" + fldName)
else
  msgStop("Sorry", "Can’t open " + tblName + " table.")
endIf

endMethod
```

▪

fieldNo method

See also Example TCursor Type

Returns the position of a field in a table.

Syntax

```
fieldNo ( const fieldName String ) SmallInt
```

Description

fieldNo returns the position of the field *fieldName* in a table. Fields are numbered from left to right, beginning with 1.

▪

fieldNo example

The following code is attached to the **pushButton** method for *thisButton*. When you press *thisButton*, this example uses **fieldNo** to display *Common Name*'s field position in the *BioLife* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fldNum SmallInt
endVar

if tc.open("biolife.db") then
  fldNum = tc.fieldNo("Common Name")    ; store field number in fldNum
  msgInfo("Field Number",
          "Common Name field is\n field number " + String(fldNum))
else
  msgInfo("Sorry", "Can't open BioLife.db table.")
endIf

endMethod
```

▪

fieldRights method

See also Example TCursor Type

Reports whether a user has rights to read or modify a field in a table.

Syntax

```
1. fieldRights ( const fieldName String, const rights String ) Logical
2. fieldRights ( const fieldNum SmallInt, const rights String ) Logical
```

Description

fieldRights returns True if the user has *rights* to the field specified in *fieldName* or *fieldNum*; otherwise, it returns False. The value of *rights* must be an expression that evaluates to one of the following strings: ReadOnly, ReadWrite, or All. Rights are obtained using the Session type method **addPassword**; rights cannot be acquired after the table is opened.

fieldRights example

The following example uses **fieldRights** to determine whether a TCursor has adequate field rights before attempting to modify the field's value.

```
; updateCust::pushButton
method pushButton(var eventInfo Event)
var
  custTC TCursor
endVar
custTC.open("Customer.db")
if custTC.locate("Name", "Unisco") then
  ; if we don't have sufficient rights to change the Name field
  if NOT custTC.fieldRights("Name", "ReadWrite") then
    ; display error message and abort operation
    msgStop("Error!", "Insufficient rights to change Name field")
  else
    ; otherwise, we have rights to make changes to the field
    custTC.edit()
    custTC.Name = "Unisco Worldwide, Inc."
    message("Changed Unisco to Unisco Worldwide, Inc.")
    custTC.endEdit()
  endIf
else
  msgStop("Error", "Can't find Unisco")
endIf

endMethod
```

▪

fieldSize method

See also		Example		TCursor Type

Returns the size of a field.

Syntax

```
1. fieldSize ( const fieldName String ) SmallInt
2. fieldSize ( const fieldNum SmallInt ) SmallInt
```

Description

fieldSize returns the size of a field as defined when the table was created. The return value can represent the maximum number of characters a field can contain; for example, given a field defined as Alpha20, **fieldSize** returns 20. Or, the return value can represent the maximum amount of data to display. For example, when you create a table and define a Memo field, you can specify a number of characters to display. **fieldSize** would return that number.

Numeric fields in dBASE tables can specify the number of digits to display on either side of the decimal point; for example, a field defined as Number 8.2 could display up to 8 digits total, with 6 digits to the left of the decimal and 2 digits to the right. **fieldSize** returns the first part of the definition; that is, the number of digits to the left of the decimal. To get the second part, use **fieldUnits2**.

For field types that do not display characters or numbers (such as OLE, binary, graphic, and so on), this method returns 0.

▪

fieldSize example

The following example uses a dynamic array to store the size of each field in the *BioLife* table, then displays the contents of the dynamic array in a dialog box.

```
; showFldSizes::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  i SmallInt
  fldSizes DynArray[] AnyType
  tblName String
endVar
tblName = "BioLife.db"

if tc.open(tblName) then
   ; this FOR loop loads the DynArray with BioLife.db field sizes
  for i from 1 to tc.nFields()
    fldSizes[tc.fieldName(i)] = tc.fieldSize(i)
  endFor
   ; now show the contents of the DynArray
  fldSizes.view(tblName + " field sizes.")
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

fieldType method

See also Example TCursor Type

Returns the data type of a field.

Syntax

```
1. fieldType ( const fieldName String ) String
2. fieldType ( const fieldNum SmallInt ) String
```

Description

fieldType returns the data type of a field. It returns "unknown" if the field is not found. The following tables (updated for version 5.0) list the possible return values for Paradox and dBASE tables:

Paradox field type	Return value
Alpha	ALPHA
Autoincrement	AUTOINCREMENT
BCD	BCD
Binary	BINARY
Bytes	BYTES
Date	DATE
Formatted Memo	FMTMEMO
Graphic	GRAPHIC
Logical	LOGICAL
Long Integer	LONG
Memo	MEMO
Money	MONEY
Number	NUMBER
OLE	OLE
Short	SHORT
Time	TIME
Timestamp	TIMESTAMP

dBASE field type	Return value
BINARY	BINARY
CHARACTER	CHARACTER
DATE	DATE
FLOAT	FLOAT
LOGICAL	LOGICAL
MEMO	MEMO
NUMBER	NUMERIC
OLE	OLE

fieldType example

The following example uses a dynamic array to store the type of each field in the *BioLife* table, then displays the contents of the dynamic array in a dialog box.

```
; showFldTypes::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  i SmallInt
  fldTypes DynArray[] AnyType
  tblName String
endVar
tblName = "BioLife.db"

if tc.open(tblName) then
   ; this FOR loop loads the DynArray with BioLife.db field types
  for i from 1 to tc.nFields()
    fldTypes[tc.fieldName(i)] = tc.fieldtype(i)
  endFor
   ; now show the contents of the DynArray
  fldTypes.view(tblName + " field types.")
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

fieldUnits2 method

See also Example TCursor Type

Returns the number of decimal places defined for a numeric field in a dBASE table.

Syntax

```
1. fieldUnits2 ( const fieldName String ) SmallInt
2. fieldUnits2 ( const fieldNum SmallInt ) SmallInt
```

Description

fieldUnits2 returns the number of decimal places defined for a numeric field in a dBASE table. For a Paradox table (or any other driver or field type that does not require field units to be specified), this method returns 0. Numeric fields in dBASE tables can specify the number of digits to display on either side of the decimal point; for example, a field defined as Number 8.2 could display up to 8 digits total with 6 characters to the left of the decimal and 2 digits to the right. **fieldUnits2** returns the second part of the definition; that is, the number of digits to the right of the decimal. To get the first part, use **fieldSize**. This method returns 0 for non-numeric field types such as alphanumeric, boolean, date, and so on.

■

fieldUnits2 example

For the following example, the **pushButton** method for *thisButton* concatenates values returned from **fieldSize** and **fieldUnits2** such that both sides of the decimal point are expressed in a single number. For example, if a field's size is 11 and is defined with 2 decimal places, this method concatenates the values to 11.2. This code uses a DynArray to store concatenated values for each field in SCORES.DBF then displays the contents of the array in a dialog box.

```
; showFldSizes::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  i SmallInt
  fldSizes DynArray[] AnyType
  tblName String
  totalSize Number
endVar
tblName = "Scores.dbf"

if tc.open(tblName) then
   ; This FOR loop loads the DynArray with the full field spec.
   ; For example if fieldSize(1) = 11 and fieldUnits2(1) = 2,
   ; one value in the DynArray would be 11.2
  for i from 1 to tc.nFields()
    totalSize = numVal(String(tc.fieldsize(i)) + "." +
                       String(tc.fieldUnits2(i)))
    fldSizes[tc.fieldName(i)] = totalSize
  endFor
   ; now show the contents of the DynArray
  fldSizes.view(tblName + " total field sizes.")
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

▪

fieldValue method

See also Example TCursor Type

Reads the value of a specified field.

Syntax

```
1. fieldValue ( const fieldName String, var result AnyType ) Logical
2. fieldValue ( const fieldNum SmallInt, var result AnyType ) Logical
```

Description

fieldValue gets the value of a specified field (*fieldName* or *fieldNum*) in the current record and assigns it to the variable *result*. This method returns True if successful; otherwise, it returns False.

You can get the same information using dot notation. For example, this statement uses dot notation to assign the *myPrice* variable with data from the Last Bid field:

```
myCost = tcVar."Last Bid"
```

The following statement uses **fieldValue** to achieve the same results:

```
tcVar.fieldValue("Last Bid", myCost)
```

▪

fieldValue example

For the following example, assume a form has at least one field, one of which is named *paymentField*. When you right-click on the *paymentField*, the code in this example presents a PopUpMenu listing possible values for the field. When you choose a menu item from the list, that value is inserted into the field.

The following code is attached to the field's Var window:

```
; paymentField::Var
Var
  lkupTbl String
  menuArray Array[] String
  fldVal AnyType
  p1 PopUpMenu
  tc TCursor
endVar
```

The following code is attached to the field's **open** method. When the field opens, this code scans through the *PayMethd* table and loads the *menuArray* array with values from the *Pay Method* field.

```
; paymentField::open
method open(var eventInfo Event)

lkupTbl = "PayMethd.db"
tc.open(lkupTbl)
scan tc :                            ; scan through table
  tc.fieldValue("Pay Method", fldVal) ; store field value in fldVal
  menuArray.addLast(fldVal)           ; add new element to menuArray
endScan
p1.addStaticText("Possible Values")   ; put static text at top of menu
p1.addSeparator()                     ; add a horizontal bar below static
text
p1.addArray(menuArray)                ; add array to the menu

endMethod
```

The following code is attached to the field's **mouseRightUp** method. When you right-click the field, this code presents a PopUpMenu and the field takes the value of your menu choice.

```
; paymentField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)

disableDefault              ; don't show the default menu
choice = p1.show()          ; show the pop-up menu
if NOT isBlank(choice) then ; if user did not press Esc
  self.value = choice       ; enter choice into the field
endIf

endMethod
```

forceRefresh method

See also Example TCursor Type

Makes TCursor point to the current data in the underlying table.

Syntax

`forceRefresh ( ) Logical`

Description

forceRefresh empties a TCursor's record buffer and refreshes it with the current data from the underlying table. The record position is maintained, provided the record still exists in the table. On an SQL server, a call to **forceRefresh** forces a read from the server. This is the only way to get a refresh from the server; it may be a time-consuming operation. **forceRefresh** only works on an SQL table if the table has a unique index.

■

forceRefresh example

The following example opens a TCursor onto the Orders table and executes two scan loops to perform two calculations. The first calculation returns the total quantity of orders from California. Then, the code calls **forceRefresh** to get the latest data from the table before executing the second scan loop to calculate the total quantity of orders from Florida.

```
method pushButton(var eventInfo Event)
   var
      tc         TCursor
      tName,
      fName,
      fVal_1,
      fVal_2     String
      caQty,
      flQty      LongInt
   endVar

   ; initialize variables
   tName  = "orders" ; assign table name
   fName  = "State"  ; assign field name
   caQty  = 0        ; assign CA quantity
   flQty  = 0        ; assign FL quantity
   fVal_1 = "CA"     ; assign 1st field value
   fVal_2 = "FL"     ; assign 2nd field value

   tc.open(tName)

   scan tc for tc.State = fVal_1:
      caQty = caQty + tc.Qty
   endScan

; during the first scan, other users may
; change data in the underlying table

   tc.forceRefresh() ; Get latest data from table

   scan tc for tc.State = fVal_2:
      flQty = flQty + tc.Qty
   endScan

   msgInfo("CA Qty and FL Qty:",
           "CA = " + String(caQty) + "\n" + "FL = " + String(flQty))

endMethod
```

▪

getGenFilter method

See also Example TCursor Type

Retrieves the filter criteria associated with a TCursor.

Syntax

```
1. getGenFilter ( criteria DynArray[ ] AnyType ) Logical
2. getGenFilter ( criteria Array[ ] AnyType [ , fieldName Array[ ] AnyType ]
) Logical
3. getGenFilter ( criteria String ) Logical
```

Description

getGenFilter retrieves the filter criteria associated with a TCursor. This method does not return values directly; instead, it assigns them to a DynArray variable (syntax 1) or to two Array variables (syntax 2) that you declare and include as arguments.

In syntax 1, the DynArray *criteria* lists fields and filtering conditions as follows: the index is the field name, and the item is the corresponding filter expression.

In syntax 2, the Array *criteria* lists filtering conditions, and the optional Array *fieldName* lists corresponding field names. If you omit *fieldName*, conditions apply to fields in the order they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to the second field, and so on).

If the arrays used in syntax 2 are resizeable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

In syntax 3, the filter criteria is assigned to a String variable *criteria* that you must declare and pass as an argument.

getGenFilter example

In this example, the **pushButton** method for a button named *btnShowFilter* uses **getGenFilter** to fill a DynArray *dyn* with a TCursor's filter criteria. Then it checks the DynArray to see if the current criteria filters the State/Prov field with a value of "CA", and resets the filter if necessary.

```
;btnShowFilter :: pushButton
method pushButton(var eventInfo Event)
var
      custTC      TCursor
      dyn         DynArray[] AnyType
      keysAr      Array[] AnyType
stFilterFld,
stCriteria   String
endVar
stFilterFld = "State/Prov"
stCriteria  = "CA"
custTC.open("Customer")

custTC.getGenFilter(dyn)   ; Get filter info.
dyn.getKeys(keysAr)
if keysAr.contains(stFilterFld) then
if dyn[stFilterFld] = stCriteria then
return               ; Filter is set correctly.
endIf
else
dyn.empty()              ; Set filter criteria correctly.
dyn[stFilterFld] = stCriteria
custTC.setGenFilter(dyn)
endIf
endMethod
```

▪

getIndexName method

See also Example TCursor Type

Retrieves the name of the current index for a table.

Syntax

`1.` (Paradox tables) **getIndexName (*indexName*** String **)** Logical

`2.` (dBASE tables) **getIndexName (*indexName*** String [**,** ***tagName*** String] **)** Logical

Description

getIndexName retrieves the name of the current index (and, optionally, the current tag for dBASE tables). This method does not return values directly; instead, it assigns them to String variables you declare and provide as arguments.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

getIndexName example

The following code gets and displays the name of the index associated with the *Orders* table.

```
method pushButton(var eventInfo Event)
   var
      ordersTC TCursor
      indexName String
   endVar

   ordersTC.open("orders")

   ; Get the index name and assign the value to the String variable
indexName.
   ordersTC.getIndexName(indexName)

   if indexName.isAssigned() then
      indexName.view("Current index")
   else
      msgInfo("indexName", "No value for indexName.")
   endIf
endMethod
```

■

getLanguageDriver method

See also Example TCursor Type

Returns the name of the current language driver for a table.

Syntax

`getLanguageDriver ( ) String`

Description

getLanguageDriver returns a String value indicating the language driver for a table.

getLanguageDriver example

The following example displays in a dialog box the language driver for the *Customer* table:

```
; getDriver::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Customer.db")
msgInfo("", tc.getLanguageDriver())  ; displays "ascii"
endMethod
```

▪

getLanguageDriverDesc method

See also Example TCursor Type

Returns the name of the current language driver description for a table.

Syntax

```
getLanguageDriverDesc ( ) String
```

Description

getLanguageDriverDesc returns a String value indicating the language driver description for a table.

getLanguageDriverDesc example

The following example displays in a dialog box the language driver description for the *Customer* table.

```
; getDriverDesc::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Customer.db")
msgInfo("", tc.getLanguageDriverDesc())  ; displays "Paradox ascii"
endMethod
```

▪

getRange method

See also Example TCursor Type

Retrieves the values that specify a range for a TCursor.

Syntax

`getRange ( var rangeVals Array[ ] String ) Logical`

Description

getRange retrieves the values that specify a range for a TCursor. This method does not return values directly; instead, it assigns them to an Array variable that you declare and include as an argument. The array values describe the range criteria, as listed in the following table.

Number of array items	Range specification
No items (empty array)	No range criteria associated with the TCursor.
One item	Specifies a value for an exact match on the first field of the index.
Two items	Specifies a range for the first field of the index.
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size *n*, specify exact matches on the first *n*-2 fields of the index. The last two array items specify a range for the *n*-1 field of the index.

If the array is resizeable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

▪

getRange example

In the following example, a button on a form is used to display the number of orders for any customer number per any month. Assume a form with the *Orders* table in its data model contains at least a Customer_No field, a Month field, and a button called *btnCustOrdersByMonth*. You could use **setGenFilter** to accomplish the same task. However, in this example a secondary index called *secCustomerMonth,* **getRange**, **getIndexName**, **switchIndex** and **setRange** is used to speed up the task.

```
;btnCustOrdersByMonth :: pushButton
method pushButton(var eventInfo Event)
var
      tc                TCursor
      nuCustomer   Number
arGet,
arSet         Array[2] AnyType
stMonth,
stActiveInd,
stDisplay   String
endVar
   nuCustomer = Customer_No.value   ;Customer field on form.
   nuCustomer.view("Customer #:")   ;Allow user to alter cust #.

   stMonth = Month.value       ;Month field on form.
   stMonth.view("Month:")      ;Allow user to alter month.

   arSet[1] = nuCustomer       ;Set array to range criteria.
arSet[2] = stMonth
   tc.attach(Customer_No)      ;Attach tc to Customer field.

   tc.getIndexName(stActiveInd)    ;Get the active index name.
if stActiveInd = "secCustomerMonth" then
      tc.getRange(arGet)          ;Get the current range.
      if arGet <> arSet then      ;Compare current range.
tc.setRange(nuCustomer, stMonth, stMonth)
endIf
else
;You must create a secondary index called secCustomerMonth
;for this example to work.
tc.switchIndex("secCustomerMonth")
tc.setRange(nuCustomer, stMonth, stMonth)
endIf
stDisplay = String(nuCustomer) + " had "
•   String(tc.nRecords()) +
" orders in " + stMonth
msgInfo("Orders in a month", stDisplay)
endMethod
```

▪

home method

See also Example TCursor Type
Beginner

Moves to the first record of a table.

Syntax

```
home ( ) Logical
```

Description

home sets the current record (and the record buffer) to the first record in a table.

home example

For the following example, the **pushButton** method associates a *TCursor* with the *Orders* table, then loads an array with field values in a **scan** loop. After the loop terminates, the TCursor is positioned at the last record in the table. This method uses **home** to move the TCursor back to the first record of the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fldArray Array[] AnyType
  fldVal AnyType
endVar
tc.open("Orders.db")
fldArray.grow(tc.nRecords())
; scan table and store order numbers in fldArray
scan tc:
  tc.fieldValue(1, fldVal)
  fldArray[tc.recNo()] = tc.fldVal
endScan
; TCursor is on the last record after the scan loop

fldArray.view()            ; display contents of array

tc.home()                  ; move TCursor to the first record
endMethod
```

▪

initRecord method

See also Example TCursor Type

Empties the record buffer.

Syntax

```
initRecord ( ) Logical
```

Description

initRecord initializes the record buffer by filling it with blanks (*not* spaces). If default values have been set for fields in the table, **initRecord** initializes those fields with the default.

initRecord example

This example uses **initRecord** to undo changes made while editing a record in a table frame. Assume that a form contains a table frame *custTF* bound to the *Customer* table.

```
method menuAction(var eventInfo MenuEvent)
var
tc TCursor
endVar
if eventInfo.id() = UserMenu + kMenuUndoRecordEdit then
tc.attach(custTF)
tc.initRecord()
custTF.resync(tc)
endIf
endMethod
```

insertAfterRecord method

See also Example TCursor Type

Inserts a record into a table after the current record.

Syntax

`insertAfterRecord ( [ const pointer TCursor ] ) Logical`

Description

insertAfterRecord inserts a record after the current record. This method is useful for inserting a new record after the last record of a table. You can use the optional argument *pointer* to insert the record pointed to by a different TCursor, or omit the argument to insert a blank record.

If the table is indexed, the record is placed in its sorted position when the record is committed; otherwise it is inserted after the current record.

This method fails if the table is not in Edit mode. Also, this method fails if the current record cannot be committed (for example, because of a key violation).

▪

insertAfterRecord example

For the following example, assume a form has a table frame, *CUSTOMER*, bound to CUSTOMER.DB. When the user attempts to delete a record, the built-in action method for *CUSTOMER* moves the record to CUSTARC.DB before deleting the record from *CUSTOMER*. You could use **copyFromArray** and **copyToArray** to accomplish the same thing, but if you use **insertAfterRecord** you don't have to store the record in an array in order to copy it. As this code demonstrates, you can use the optional argument *pointer* to insert the record pointed to by a TCursor.

```
; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
  tcCust, tcArc TCursor
endVar
if eventInfo.id() = DataDeleteRecord then ; if user attempts to delete a
record
  if thisForm.Editing = True then         ; if form is in Edit mode
    disableDefault                        ; don't process DataDeleteRecord
yet

    if msgYesNoCancel("Confirm",          ; if user confirms delete
       "Delete the current record?") = "Yes" then
      tcCust.attach(CUSTOMER)             ; sync TCursor to CUSTOMER pointer
      if tcArc.open("CustArc.db") then
        tcArc.edit()
        tcArc.end()                       ; move to end of table
        tcArc.insertAfterRecord(tcCust)   ; insert current CUSTOMER record
                                          ; after last record in CustArc.db
        doDefault                         ; process DataDeleteRecord now
      else
        msgStop("Stop!", "Sorry, Can't archive record.")
      endIf
    else                                  ; else user didn't confirm delete
      message("Record not deleted.")
    endIf
  else                                    ; else form is not in Edit mode
    msgStop("Stop!", "Press F9 to edit data.")
  endIf
endIf
endMethod
```

insertBeforeRecord method

See also   Example   TCursor Type

Inserts a record into a table before the current record.

Syntax

insertBeforeRecord ([const ***pointer*** TCursor] **)** Logical

Description

insertBeforeRecord inserts a record before the current record (the same as **insertRecord**). You can use the optional argument *pointer* to insert the record pointed to by another TCursor, or omit the argument to insert a blank record.

If the table is indexed, the record is placed in its sorted position when the record is committed; otherwise, it is inserted before the current record.

This method fails if the table is not in Edit mode. Also, this method fails if the current record cannot be committed (for example, because of a key violation).

▪

insertBeforeRecord example

For the following example, assume a form has a table frame, *CUSTOMER*, bound to CUSTOMER.DB. When the user attempts to delete a record, the built-in **action** method for *CUSTOMER* moves the record to CUSTARC.DB before deleting the record from *CUSTOMER*. You could use **copyFromArray** and **copyToArray** to accomplish the same thing, but if you use **insertBeforeRecord** you don’t have to store the record in an array in order to copy it. As this code demonstrates, you can use the optional argument *pointer* to insert the record pointed to by a second TCursor.

```
; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
  tcCust, tcArc TCursor
endVar
if eventInfo.id() = DataDeleteRecord then ; if user attempts to delete a
record
  if thisForm.Editing = True then          ; if form is in Edit mode
    disableDefault                         ; don’t process DataDeleteRecord
yet

    if msgYesNoCancel("Confirm",           ; if user confirms delete
       "Delete the current record?") = "Yes" then
      tcCust.attach(CUSTOMER)              ; sync TCursor to CUSTOMER pointer
      if tcArc.open("CustArc.db") then
        tcArc.edit()
        tcArc.insertBeforeRecord(tcCust)   ; insert current CUSTOMER record
                                           ; before current record in
CustArc.db
        doDefault                          ; process DataDeleteRecord now
      else
        msgStop("Stop!", "Sorry, Can’t archive record.")
      endIf
    else                                   ; else user didn’t confirm delete
      message("Record not deleted.")
    endIf
  else                                     ; else form is not in Edit mode
    msgStop("Stop!", "Press F9 to edit data.")
  endIf
endIf
endMethod
```

▪

insertRecord method

See also Example TCursor Type
Beginner

Inserts a record into a table.

Syntax
insertRecord ([const ***pointer*** TCursor] **)** Logical

Description

insertRecord inserts a record into a table before the current record (the same as **insertBeforeRecord**). You can use the optional argument *pointer* to insert the record pointed to by another TCursor, or omit the argument to insert a blank record.

If the table is indexed, the record is placed in its sorted position when the record is committed; otherwise, it is inserted before the current record.

This method fails if the table is not in Edit mode. Also, this method fails if the current record cannot be committed (for example, because of a key violation).

▪

insertRecord example

For the following example, assume a form has a table frame, *CUSTOMER*, bound to CUSTOMER.DB. When the user attempts to delete a record, the built-in **action** method for *CUSTOMER* moves the record to CUSTARC.DB before deleting the record from *CUSTOMER*. You could use **copyFromArray** and **copyToArray** to accomplish the same thing, but if you use **insertRecord** you don't have to store the record in an array in order to copy it. As this code demonstrates, you can use the optional argument *pointer* to insert the record pointed to by another TCursor.

```
; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
  tcCust, tcArc TCursor
endVar
if eventInfo.id() = DataDeleteRecord then ; if user attempts to delete a
record
  if thisForm.Editing = True then         ; if form is in Edit mode
    disableDefault                        ; don't process DataDeleteRecord
yet

    if msgYesNoCancel("Confirm",          ; if user confirms delete
       "Delete the current record?") = "Yes" then
      tcCust.attach(CUSTOMER)             ; sync TCursor to CUSTOMER pointer
      if tcArc.open("CustArc.db") then
        tcArc.edit()
        tcArc.insertRecord(tcCust)        ; insert current CUSTOMER record
                                          ; before current record in
CustArc.db
        doDefault                         ; process DataDeleteRecord now
      else
        msgStop("Stop!", "Sorry, Can't archive record.")
      endIf
    else                                  ; else user didn't confirm delete
      message("Record not deleted.")
    endIf
  else                                    ; else form is not in Edit mode
    msgStop("Stop!", "Press F9 to edit data.")
  endIf
endIf
endMethod
```

▪

instantiateView method

See also Example TCursor Type

Copies an in-memory TCursor to a physical table and makes the TCursor point to it.

Syntax

```
1. instantiateView ( const tableName String ) Logical
2. instantiateView ( const tableVar Table ) Logical
```

Description

instantiateView copies an in-memory TCursor to a physical table and makes the TCursor point to it. This method returns True if it succeeds; otherwise, it returns False.

Syntax 1 creates the table using the name specified in *tableName*.

Syntax 2 associates the table with the Table variable specified in *tableVar*.

Use this method after executing a query that generates a TCursor onto a live query view. **instantiateView** copies the data from the live query view to a table on disk and makes the TCursor point to it▪then you can use that TCursor to manipulate the data. The resulting table has no relationship to the underlying tables in the query.

For information on live query views, see Live query views in the User's Guide help.
You can also use **instantiateView** with TCursors created by ObjectPAL methods.

instantiateView example

This example executes a query to a TCursor and determines if the result is a live query view. If so, it calls **instantiateView** to write the view to a physical table, then displays the table in a Table window.

```
method pushButton(var eventInfo Event)
const
kName = "salary"
endConst
var
      qbeVar      Query
      tcAnswer   TCursor
      tvAnswer   TableView
endVar
qbeVar.readFromFile(kName)
qbeVar.executeQBE(tcAnswer)

if tcAnswer.isView() then
tcAnswer.instantiateView(kName)
tvAnswer.open(kName)
else
return
endIf
endMethod
```

isAssigned method

See also Example TCursor Type

Reports whether a TCursor variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

isAssigned returns True if a TCursor variable has a value assigned using open or attach; otherwise, it returns False.

isAssigned example

The following example associates a TCursor with a table, displays information found in the last record, then closes the TCursor. In this example, the code displays a message indicating whether the TCursor variable is still assigned after the TCursor is closed. This code is attached to the built-in **pushButton** method for *thisButton*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.db")         ; open a TCursor for Orders.db
tc.end()                     ; move to end of the table

; display information in last record
msgInfo("Last Order", "Order number: " +
String(tc."Order No") + " \nOrder date: " + String(tc."Sale Date"))

tc.close()                   ; attempt to close TCursor

; if close is successful, this displays False (tc is no longer assigned)
; otherwise, it displays True (tc is still assigned if close fails)
msgInfo("Is tc Assigned?", tc.isAssigned())

endMethod
```

▪

isEdit method

See also Example TCursor Type

Reports whether a TCursor is in Edit mode.

Syntax

```
isEdit ( ) Logical
```

Description

isEdit returns True if the TCursor is in Edit mode; otherwise, it returns False. If you attach a TCursor to a display manager (such as a UIObject or a TableView), and that object is in Edit mode, the TCursor will be in Edit mode as well.

▪

isEdit example

For the following example, assume a form has a table frame bound to the *Customer* table and a button. The code attached to the **pushButton** method for *thisButton* attaches a TCursor to the table frame, then uses **isEdit** to determine whether the TCursor is in Edit mode. If the table frame was in Edit mode when the TCursor was attached, the TCursor will also be in Edit mode.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

; attach to the table frame
tc.attach(CUSTOMER)

; if CUSTOMER was in Edit mode, tc will be in Edit mode too

if NOT tc.isEdit() then    ; test whether tc is in Edit mode
  tc.edit()
endIf

if tc.locate("Name", "Action Club") then
  tc.phone = "808-555-1234"
else
  msgStop("Sorry", "Can't find Action club")
endIf

endMethod
```

▪

isEmpty method

See also Example TCursor Type

Determines whether a table contains any records.

Syntax

`isEmpty ( ) Logical`

Description

isEmpty returns True if there are no records in the table associated with the TCursor; otherwise, it returns False.

isEmpty example

For the following example, the **pushButton** method for the *rptRecNo* button displays the number of records in the *Orders* table. If the table is empty, this code alerts the user that the table is empty.

```
; rptRecNo::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tblName String
endVar
tblName = "Orders.db"

if tc.open(tblName) then
  if tc.isEmpty() then                          ; if Orders.db is empty
    msgStop("Hey!",
             tblName + " table is empty!")
  else
    msgInfo(tblName + " table has",              ; report number of records
            String(tc.nRecords()) + " records")
  endIf
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

▪

isEncrypted method

See also Example TCursor Type

Reports whether a table is password-protected.

Syntax

```
isEncrypted ( ) Logical
```

Description

isEncrypted returns True if a table is password-protected; otherwise, it returns False. A TCursor can't be opened on an encrypted table until you use the Session type method **addPassword** to present the required password. This method does not report whether a user has access rights to the table▪use **tableRights** for that.

isEncrypted example

The following example tests whether the *Customer* table is encrypted.

```
; thisButton::pushButton
method open(var eventInfo Event)
var
  tc TCursor
endvar

if tc.open("Customer.db") then
  if tc.isEncrypted() then
    msgInfo("Table is protected", "An acceptable password has been
presented.")
  endif
else
  msgStop("Error", "Can't open the Customer table.")
endIf

endMethod
```

▪

isInMemoryTCursor method

See also Example TCursor Type

Reports whether a TCursor points to a table in system memory or a physical table.

Syntax

```
isInMemoryTCursor ( ) Logical
```

Description

isInMemoryTCursor returns True if the TCursor is associated with a table in system memory (for example, a table generated by anObjectPAL method that enumerates information to a TCursor); otherwise, it returns False.

By default, when you execute a query, Paradox tries to create a live query view. In some cases, however, a live query view is not possible. Use **isInMemoryTCursor** to find out whether the query resulted in a live query view (in which case changes made to the TCursor affect the underlying tables) or an in-memory answer table (in which case the underlying tables are not affected). By default, if the query results in a live query view, **isInMemoryTCursor** returns False (and **isView** returns True). However, you can use **wantInMemoryTCursor** to specify how to create a TCursor resulting from a query.

isInMemoryTCursor example

This example reads a query from a file and executes it, then uses a **scan** loop to increase the salary of each employee by 12 percent. There's no way to know before running the query if it will result in a live query view, so the call to **isInMemoryTCursor** is required to prevent changes to the actual employee salary data.

```
method pushButton(var eventInfo Event)
   var
      qbeVar    Query
      tcAnswer  TCursor
   endVar

   ; Read the query from a file.
   qbeVar.readFromFile("Salary.qbe")

   ; We don't know if this query will generate a live
   ; query view, so use isInMemoryTCursor to find out.
   if qbeVar.executeQBE(tcAnswer) then

      ; If it is in memory (i.e., not live), then
      ; see the effects of a 12% raise for all employees.
      if tcAnswer.isInMemoryTCursor() then
         nuOldTotalPayroll = tcAnswer.cSum("Salary")

         tcAnswer.edit()
         scan tcAnswer :
            tcAnswer.Salary = tcAnswer.Salary * .15
         endScan
         tcAnswer.endEdit()

         nuNewTotalPayroll = tcAnswer.cSum("Salary")

         msgInfo("Before raise: " + String(nuOldTotalPayroll),
                 "After raise: " + String(nuNewTotalPayroll))

      else
         ; If it is live, inform user and quit the method.
         msgStop("Live query view",
             "Edits would affect the underlying table.")
         return
      endIf
   else
      errorShow()
   endIf

endmethod
```

isOnSQLServer method

See also Example TCursor Type

Reports whether a TCursor is associated with a table on a SQL server.

Syntax

```
isOnSQLServer ( ) Logical
```

Description

isOnSQLServer returns True if the TCursor is associated with a table on a SQL server; otherwise, returns False.

isOnSQLServer example

The following example is a custom method that uses **isOnSQLServer** to find out whether a TCursor is associated with a remote table. If **isOnSQLServer** returns True, the code displays a msgQuestion dialog box and prompts the user to confirm the decision to lock the remote table.

```
method confirmRemoteLock(const tc TCursor) Logical

   if tc.isOnSQLServer() then

      ; you might not want to lock remote tables
      if msgQuestion("Lock table?",
                     "Lock a remote table?") = "Yes" then
         return True
      else
         return False
      endIf
   endIf
endMethod
```

▪

isOpenOnUniqueIndex method

See also Example TCursor Type

Reports whether a TCursor is open on a unique index.

Syntax

```
isOpenOnUniqueIndex ( ) Logical
```

Description

isOpenOnUniqueIndex returns True if a TCursor is open on a unique index (that is, an index that does not allow duplicate key values); otherwise, returns False. This method is useful for working with remote tables, because operations that update the table (for example, editing data or deleting records) may fail unless the TCursor is opened on a unique index.

isOpenOnUniqueIndex example

The following example is a custom method that calls **isOpenOnUniqueIndex** before putting the TCursor into Edit mode.

```
method editIfUniqueIndex(const tc TCursor) Logical
   if tc.isOpenOnUniqueIndex() then
      return tc.edit()
   else
      return False
   endIf
endMethod
```

isRecordDeleted method

See also Example TCursor Type

Reports whether the current record has been deleted (dBASE tables only).

Syntax

```
isRecordDeleted ( ) Logical
```

Description

isRecordDeleted reports whether the current record has been deleted. **isRecordDeleted** works only for dBASE tables because deleted Paradox records can't be displayed. This method returns True if the current record has been deleted; otherwise, it returns False.

Deleted records in a dBASE table are not shown by default. For **isRecordDeleted** to work correctly, you must call **showDeleted** to show deleted records in the table; otherwise, deleted records are not visible to **isRecordDeleted**.

▪

isRecordDeleted example

The following example opens a TCursor for the SCORES.DBF dBASE table, then uses **showDeleted** to display all deleted records. Then, the code attempts to locate a specific record in the table. This example uses **isRecordDeleted** to determine whether the record has been deleted; if it has, it is undeleted with **undeleteRecord**. The following code is attached to the **pushButton** method for *thisButton*:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Scores.dbf")                  ; open TCursor on a dBASE table
tc.showDeleted()                       ; show deleted records
if tc.locate("Name", "Jones") then     ; if locate finds Jones in Name field
  if tc.isRecordDeleted() then         ; if the record has been deleted
    tc.edit()                          ; begin Edit mode
    tc.undeleteRecord()                ; undelete the record
    message("Jones record undeleted")
  endIf
else
  msgStop("Error", "Can't find Jones.")
endIf
endMethod
```

▪

isShared method

See also Example TCursor Type

Reports whether a table is currently shared.

Syntax

```
isShared ( ) Logical
```

Description

isShared returns True if another user has opened the table pointed to by a TCursor; otherwise, it returns False.

▪

isShared example

For the following example, a form's built-in **open** method determines whether CUSTOMER.DB is currently being shared by another user; if it is, the user is warned and given the option to continue or abort.

```
; thisPage::open
method open(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Customer.db")              ; open a TCursor for Customer
if tc.isShared() then               ; if table is currently shared
  if msgYesNoCancel("Continue?",    ; ask for confirmation
     "Customer table is currently being shared.\n" +
     "Continue anyway?") <> "Yes" then

    close()                         ; close this form
  endIf
endIf
endMethod
```

▪

isShowDeletedOn method

See also Example TCursor Type

Reports whether deleted records in a dBASE table are shown.

Syntax

```
isShowDeletedOn ( ) Logical
```

Description

isShowDeletedOn reports whether the table pointed to by a TCursor currently shows deleted records. You can use the **showDeleted** method to specify whether or not to show deleted records, then use **isShowDeletedOn** to determine states. **isShowDeletedOn** is valid only for dBASE tables.

isShowDeletedOn example

In the following example, if **isShowDeletedOn** returns False, the code calls **showDeleted** to show deleted records in ORDERS.DBF.

```
; showDeletedRecs::pushButton
method pushButton(var eventInfo Event)
var
  dbfTC TCursor
endVar
if dbfTC.open("Orders.dbf") then
  if NOT dbfTC.isShowDeletedOn() then   ; if deleted records are not shown
    dbfTC.showDeleted(Yes)              ; show deleted records
  endIf
else
  msgStop("Sorry", "Can't open Orders.dbf table.")
endIf
endMethod
```

▪

isValid method

See also Example TCursor Type

Reports whether the contents of a field are legitimate and complete.

Syntax

```
1. isValid ( const fieldName String, const value AnyType ) Logical
2. isValid ( const fieldNum SmallInt, const value AnyType ) Logical
```

Description

isValid reports whether the value specified in *value* conforms with field type and validity checks for the field specified in *fieldNum* or *fieldName*. This method gives you an opportunity to check whether a new value is valid for a field before you attempt to post the record.

isValid returns True if *value* conforms to field type and validity checks; otherwise, it returns False.

isValid example

The following example uses **isValid** to test whether a given value is valid for a Date field. If the value is not valid, this code warns the user of the error; otherwise the value is entered into the field. The following code is attached to the **pushButton** method for *thisButton*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tryValue String
endVar
tryValue = "100/5/1994" ; Invalid date.
tc.open("Orders.db")
if NOT tc.isValid("Sale Date", tryValue) then
  msgStop("Error",
  String(tryValue) + " is not valid for this field.")
else                  ; this condition is never met
  tc."Sale Date" = tryValue
  tc.postRecord()
endIf

endMethod
```

▪

isView method

See also Example TCursor Type

Reports whether a TCursor is associated with a live query view.

Syntax

```
isView ( ) Logical
```

Description

isView returns True if the TCursor is associated with a live query view; otherwise, it returns False.

When **isView** is True, **isInMemoryTCursor** returns False.

▪

isView example

See the example for **instantiateView**.

▪

locate method

See also Example TCursor Type
Beginner

Searches for a specified field value.

Syntax

```
1. locate ( const fieldName String, const exactMatch AnyType [ , const
fieldName String, const exactMatch AnyType ] * ) Logical
2. locate ( const fieldNum SmallInt,  const exactMatch AnyType [ , const
fieldNum SmallInt, const exactMatch AnyType ] * ) Logical
```

Description

locate searches a table for records whose values exactly match the criteria specified in one or more field/value pairs. Specify the value to search for in *exactMatch* and the field to search in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). This method guarantees that the first value matching *exactMatch* is found, given the current view of the records. If the TCursor is using a secondary index, **locate** finds the first record in secondary index order▪regardless of that record's primary index order.

The search always starts from the beginning of the table, but if no match is found, the TCursor returns to the original record. If a match is found, the TCursor moves to that record. This operation fails if the current record cannot be posted (for example, because of a key violation).

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is set to True.

locate example

In the following example, the **pushButton** method for the *fixSpelling* button searches for a value in the *Name* field of the *Customer* table. If **locate** is successful, this method replaces the name with a new value and informs the user of the change.

```
; fixSpelling::pushButton
method pushButton(var eventInfo Event)
var
  ordTC TCursor
endVar

ordTC.open("Customer.db")
; if locate finds "Proffessional Divers, Ltd." in the Name field
if ordTC.locate("Name", "Proffessional Divers, Ltd.") then
  ; begin Edit mode
  ordTC.edit()
  ; correct spelling (Professional)
  ordTC.Name = "Professional Divers, Ltd."
  msgInfo("Success", "Corrected spelling error.")
else
  msgInfo("Search Failed",
          "Couldn't find \nProfessional Divers, Ltd.")
endIf
ordTC.endEdit()
endMethod
```

▪

locateNext method

See also Example TCursor Type
Beginner

Searches for a specified field value.

Syntax

```
1. locateNext ( const fieldName String, const exactMatch AnyType [ , const fieldName String, const exactMatch AnyType ] * ) Logical
2. locateNext ( const fieldNum SmallInt, const exactMatch AnyType [ , const fieldNum SmallInt, const exactMatch AnyType ] * ) Logical
```

Description

locateNext searches a table for records whose values exactly match the criteria specified in one or more field/value pairs. Specify the value to search for in *exactMatch* and the field to search in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). This method guarantees that the next value matching *exactMatch* is found, given the current view of the records. If the TCursor is using a secondary index, **locateNext** finds the next record in secondary index order▪regardless of that record's primary index order.

The search begins with the record after the current record. If a match is found, the TCursor moves to that record. If no match is found, the TCursor returns to the original record. To start a search from the beginning of a table, use **locate**.

This operation fails if the current record cannot be posted (for example, because of a key violation).

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is set to True.

locateNext example

The following example uses **locate** and **locateNext** to count the number of records that have "FL" in the State/Prov field of the *Customer* table. The following code is attached to the *findFL* **pushButton** method.

```
; findFL::pushButton
method pushButton(var eventInfo Event)
var
  CustTC TCursor
  numFound LongInt
endVar
custTC.open("Customer.db")

if custTC.locate("State/Prov", "FL") then
  numFound = 1
  while custTC.locateNext("State/Prov", "FL")
    numFound = numFound + 1
  endWhile
  msgInfo("Records Found", String("Found ", numFound, " companies in FL"))
else
  msgInfo("Sorry", "Can't find FL in State/Prov field.")
endIf

endMethod
```

■

locateNextPattern method

See also		Example		TCursor Type

Locates the next record containing a field that has a specified pattern of characters.

Syntax

```
1. locateNextPattern ( [ const fieldName String, const exactMatch AnyType ] * const fieldName String, const pattern AnyType ) Logical
2. locateNextPattern ( [ const fieldNum SmallInt, const exactMatch AnyType ] * const fieldNum SmallInt, const pattern AnyType ) Logical
```

Description

locateNextPattern finds sub-strings (for example, "comp" in "computer"). The search begins with the record after the current record. If a match is found, the TCursor moves to that record. If no match is found, the TCursor returns to the original record. If the TCursor is using a secondary index, **locateNextPattern** finds the next record in secondary index order▪regardless of that record's primary index order.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search at the beginning of a table, use **locatePattern.**

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance), and specify a pattern of characters in *pattern*.

You can include the standard pattern operators **@** and **..** in the *pattern* argument. The **..** operator stands for any string of characters (including none at all); **@** stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildCardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** for more information about advanced match pattern operators.

For example, the following statement checks the values in the first field of each record. If a value is anything except "Borland", **locateNextPattern** returns True.

```
tc.locateNextPattern(1, [^Borland])
```

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the following statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, "database"):

```
tc.locateNextPattern("Name", "Borland", "Product", "Paradox", "Keywords", "data*")
```

For examples, see Sample search strings with wildcards in the User's Guide help.

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is set to True.

▪

locateNextPattern example

In the following example, assume the SOFTWARE.DB table exists in the current directory. Assume further that two of the fields are named Product and Name. This code (attached to the **pushButton** method) searches for records whose Name field contains "Borland" and whose Product field begins with "Par". This code keeps track of the matches found and stores field values in a resizeable array. When the method can't locate any more records that match the criteria, the results are displayed in a dialog box.

```
; findGoodProducts::pushButton
method pushButton(var eventInfo Event)
var
  myNames TCursor
  searchFor String
  numFound SmallInt
  productNames Array[] String
endVar
myNames.open("software.db")
searchFor = "Borland"

; this searches for records with "Borland" in the Name field
; and values starting with "Par" in the Product field
if myNames.locatePattern("Name", searchFor, "Product", "Par..") then
  numFound = 1
  productNames.grow(1)
  productNames[numFound] = myNames.Product

  ; now continue searching through fields with same criteria and
  ; store Product values in myNames TCursor
  while myNames.locateNextPattern("Name", searchFor, "Product", "Par..")
    numFound = numFound + 1
    productNames.addLast(myNames.product)
  endWhile
endIf
if productNames.size() > 0 then
  productNames.view()
endIf
endMethod
```

■

locatePattern method

See also Example TCursor Type

Locates a record containing a field that has a specified pattern of characters.

Syntax

```
1. locatePattern ( [ const fieldName String, const exactMatch AnyType ] *
const fieldName String, const pattern String ) Logical
2. locatePattern ( [ const fieldNum SmallInt, const exactMatch AnyType ] *
const fieldNum SmallInt, const pattern String ) Logical
```

Description

locatePattern finds sub-strings (for example, "comp" in "computer"). The search always starts at the beginning of the table, but if no match is found, the TCursor returns original record. If a match is found, the TCursor moves to that record. If the TCursor is using a secondary index, locate finds the first record in secondary index order▪regardless of that record's primary index order.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search after the current record, use **locateNextPattern**. To start a search before the current record, use **locatePriorPattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance), and specify a pattern of characters in *pattern*.

You can include the standard pattern operators **@** and **..** in the *pattern* argument. The **..** operator stands for any string of characters (including none at all); **@** stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildCardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** for more information about advanced match pattern operators.

For example, the following statement checks values in the first field of each record. If a value is anything except "Borland", **locatePattern** returns True.

```
tc.locatePattern(1, [^Borland])
```

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the following statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, database).

```
tc.locatePattern("Name", "Borland", "Product", "Paradox", "Keywords",
"data*")
```

For examples, see Sample search strings with wildcards in the User's Guide help.

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is set to True.

locatePattern example

In the following example, assume the SOFTWARE.DB table exists in the current directory. Assume further that two of the fields are named Product and Name. This code (attached to the **pushButton** method) searches for records whose Name field contains "Borland" and whose Product field begins with "Par". This code keeps track of the matches found and stores field values in a resizeable array. When the method can't locate any more records that match the criteria, the results are displayed in a dialog box.

```
; findGoodProducts::pushButton
method pushButton(var eventInfo Event)
var
  myNames TCursor
  searchFor String
  numFound SmallInt
  productNames Array[] String
endVar
myNames.open("software.db")
searchFor = "Borland"

; this searches for records with "Borland" in the Name field
; and values starting with "Par" in the Product field
if myNames.locatePattern("Name", searchFor, "Product", "Par..") then
  numFound = 1
  productNames.grow(1)
  productNames[numFound] = myNames.Product

  ; now continue searching through fields with same criteria and
  ; store Product values in myNames TCursor
  while myNames.locateNextPattern("Name", searchFor, "Product", "Par..")
    numFound = numFound + 1
    productNames.addLast(myNames.product)
  endWhile
endIf
if productNames.size() > 0 then
  productNames.view()
endIf
endMethod
```

▪

locatePrior method

See also Example TCursor Type

Searches for a specified field value.

Syntax

```
1. locatePrior ( const fieldName String, const exactMatch AnyType
                 [ , const fieldName String, const exactMatch AnyType ] * )
Logical
2. locatePrior ( const fieldNum SmallInt, const exactMatch AnyType
                 [ , const fieldNum SmallInt, const exactMatch AnyType ] * )
Logical
```

Description

locatePrior searches a table for records whose values exactly match the criteria specified in one or more field/value pairs. Specify the value to search in *searchValue* and the field to search in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). This method guarantees that the previous value matching *exactMatch* is found, given the current view of the records. If the TCursor is using a secondary index, **locatePrior** finds the previous record in secondary index order▪regardless of that record's primary index order.

The search starts from the record before the current record, and searches backwards in the table for the previous match. If a match is found, the TCursor moves to that record; otherwise, it returns to the original record. This operation fails if the current record cannot be committed (for example, because of a key violation). This method returns True if a successful match was made; otherwise, it returns False.

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is set to True.

locatePrior example

In the following example, the **pushButton** method for *showPrior* searches backwards through the *Lineitem* table for records with a certain order number. The *lineTC* variable is declared in the page's Var window, and opened to the *Lineitem* table in the **open** method for the page.

The following code goes in the Var window for *thisPage*:

```
; thisPage::var
Var
  lineTC TCursor
endVar
```

The following code is attached to the **open** method for *thisPage*:

```
; thisPage::open
method open(var eventInfo Event)
  lineTC.open("Lineitem")        ; open a TCursor for LineItem.db
endMethod
```

The following code is attached to the **pushButton** method for the *showPrior* button:

```
; showPrior::pushButton
method pushButton(var eventInfo Event)
var
  rec Array[] AnyType
endVar

if lineTC.locatePrior("Order No", 1005) then
  lineTC.copyToArray(rec)
  rec.view("Record #" + String(lineTC.recNo()))
else
  msgStop("Sorry", "No more records.")
endIf
endMethod
```

▪

locatePriorPattern method

See also Example TCursor Type

Locates the previous record containing a field that has a specified pattern of characters.

Syntax

```
1. locatePriorPattern ( [ const fieldName String, const exactMatch AnyType ]
* const fieldName String, const pattern String ) Logical
2. locatePriorPattern ( [ const fieldNum SmallInt, const exactMatch AnyType ]
* const fieldNum SmallInt, const pattern String ) Logical
```

Description

locatePriorPattern finds sub-strings (for example, "comp" in "computer"). The search begins with the record before the current record. If a match is found, the TCursor moves to that record. If no match is found, the TCursor returns to the original record. If the TCursor is using a secondary index, **locatePriorPattern** finds the previous record in secondary index order▪regardless of that record's primary index order.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search at the beginning of a table, use **locatePattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance), and specify a pattern of characters in *pattern*.

You can include the standard pattern operators **@** and **..** in the *pattern* argument. The **..** operator stands for any string of characters (including none at all); **@** stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildCardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** for more information about advanced match pattern operators.

For example, the following statement checks values in first field of each record. If a value is anything except "Borland," locatePriorPattern returns True.

```
tc.locatePriorPattern(1, [^Borland])
```

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the following statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, "database"):

```
tc.locatePriorPattern("Name", "Borland", "Product", "Paradox", "Keywords",
"data*")
```

For examples, see Sample search strings with wildcards in the User's Guide help.

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is set to True.

locatePriorPattern example

In the following example, the **pushButton** method for *showPriorPtrn* searches backwards through the *Software* table for records with a certain company and product name. The *tc* variable is declared in the page's Var window, and opened to the *Software* table in the **open** method for the page.

The following code goes in the Var window for *thisPage*:

```
; thisPage::var
Var
  tc        TCursor
  searchFor String
endVar
```

The following code is attached to the **open** method for *thisPage*:

```
; thisPage::open
method open(var eventInfo Event)
  tc.open("Software.db")  ; open TCursor for Software.db
  tc.end()                ; move TCursor to the last record
  searchFor = "Borland"
endMethod
```

The following code is attached to the **pushButton** method for the *showPriorPtrn* button:

```
; showPrior::pushButton
method pushButton(var eventInfo Event)
var
  rec Array[] AnyType
endVar

; search for the previous pattern
if tc.locatePriorPattern("Name", searchFor, "Product", "Par..") then
  tc.copyToArray(rec)
  rec.view("Record #" + String(tc.recNo()))
else
  msgStop("Sorry", "No more records.")
endIf
endMethod
```

▪

lock method

See also Example TCursor Type
Beginner

Places specified locks on a specified table.

Syntax
`lock ( const lockType String ) Logical`

Description
lock attempts to place a lock on the TCursor, where *lockType* is one of the following String values, listed in order of decreasing strength and increasing concurrency.

String value	Description
Full	The current session has exclusive access to the table. No other session can open the table. Cannot be used with dBASE tables.
Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

If successful, this method returns True; otherwise, it returns False.

lock example

The following example opens a TCursor for *Customer*, places a full lock on the table, then uses **reIndex** to rebuild the *Phone_Zip* index. Once the index is rebuilt, this code unlocks *Customer* so other users on a network can gain access to the table.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

if tc.open(pdoxTbl) then
  if tc.lock("Full") then      ; attempt to place Full lock
    tc.reIndex("Phone_Zip")    ; rebuild Phone_Zip index
    tc.unLock("Full")          ; unlock the table
    message("Phone_Zip rebuilt.")
  else
    msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
  endIf
endIf
endMethod
```

▪

lockRecord method

See also Example TCursor Type

Puts a write lock on the current record.

Syntax

`lockRecord ( ) Logical`

Description

Paradox places a write lock on a record when you begin to make changes (an implicit record lock). **lockRecord** attempts to place a write lock on the record pointed to by a TCursor (an explicit record lock) and if successful, returns True; otherwise, it returns False.

■

lockRecord example

In the following example, the **pushButton** method for *thisButton* searches for a record in the *Customer* table. If the search is successful, this example attempts to lock the record with **lockRecord**. When the record has been locked, a custom procedure is called to get new customer information from the user. If **lockRecord** is not successful, the user is asked to try again later.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  custTC, myCustTC TCursor
endVar
custTC.open("Customer.db")

; attempt to locate record in Customer.db
if custTC.locatePattern("Name", "Jamaica..") then
  custTC.edit()
  if custTC.lockRecord() then         ; attempt to lock the record
    custTC.initRecord()               ; initialize record to the defaults
    getCustInfo()                     ; call a custom procedure
  else                                ; otherwise record couldn't be locked
    msgStop("Sorry", "Can't lock record. \n Try again later.")
  endIf
else
  msgStop("Sorry", "Can't find record.")
endIf

endMethod
```

▪

lockStatus method

See also Example TCursor Type

Returns the number of times you have placed a lock on a TCursor.

Syntax

```
lockStatus ( lockType String ) SmallInt
```

Description

lockStatus returns the number of times you have placed a lock of type *lockType* on a TCursor, where *lockType* is one of the following String values: Write, Read, Full, or Any.

If you haven't placed any locks of a given type, **lockStatus** returns 0.

If you specify "Any" for *lockType*, **lockStatus** returns the total number of locks you've placed on the table. **lockStatus** reports only on locks you've placed explicitly on the TCursor, not on locks placed by Paradox or by other users or applications.

lockStatus example

This example shows how **lockStatus** reports only on locks you've placed explicitly on a TCursor, not on locks placed by Paradox or by other users or applications. Assume a form contains a button named *thisButton* and a field object named *Balance_Due* bound to the Balance Due field of the *Orders* table.

```
; thisButton::pushButton
const
   kTbName = "locks"
   kStatus = "Any"
endConst

var
   tcOrders    TCursor
   tvLocks    TableView
endVar

proc displayLockInfo()
   tcOrders.enumLocks(kTbName)
   tvLocks.open(kTbName)
   tvLocks.setTitle("Locks on Orders table:")

   siNumLocks = tcOrders.lockStatus(kStatus)
   siNumLocks.view("Locks on TCursor:")
   tvLocks.close()
endProc

method pushButton(var eventInfo Event)

   ; Associate TCursor with a field object bound
   ; to the Balance Due field in the Orders table.
   ; TCursor gets locks from the UIObject.
   tcOrders.attach(Balance_Due)

   displayLockInfo() ; Table is locked, but not TCursor.

   tcOrders.lock("Write") ; Lock TCursor.

   displayLockInfo() ; Table and TCursor are locked,
                     ; but locks are different.
endmethod
```

▪

moveToRecNo method

See also Example TCursor Type

Moves a TCursor to a specific record in a table.

Syntax

`moveToRecNo ( const recordNum LongInt ) Logical`

Description

moveToRecNo sets the current record to the record specified in *recordNum*. It returns an error if *recordNum* is not in the table. Use the **nRecords** method or examine the NRecords property to find out how many records a table contains. This method is recommended only for dBASE tables. When used for a Paradox table, **moveToRecNo** behaves exactly like the **moveToRecord** method.

moveToRecNo example

This example uses **moveToRecNo** to move to a specified record in the dBASE table ORDERS.DBF. Then it displays the value of the SALE_DATE field for that record.

```
method pushButton(var eventInfo Event)
   var
      tcOrders   TCursor
      siRecNo      SmallInt
      daSaleDate   Date
   endVar

   tcOrders.open("orders.dbf")

   siRecNo = 0
   siRecNo.view("Enter a record number:")

   if siRecNo > 0 then
      if   tcOrders.moveToRecNo(siRecNo) then
         daSaleDate = tcOrders."SALE_DATE"
         daSaleDate.view("Sale date: ")
      else
         errorShow("Invalid record number.")
      endIf
   else
      return
   endIf

endMethod
```

moveToRecord method

See also Example TCursor Type

Moves a TCursor to a specific record in a table.

Syntax

```
moveToRecord ( const recordNum LongInt ) Logical
```

Description

moveToRecord sets the current record (and the record buffer) to the record specified in *recordNum*. It returns an error if *recordNum* is greater than the number of records in the table. Use **nRecords** to find out how many records a table contains. This method can be very slow for dBASE tables; use **moveToRecNo** instead.

This operation fails if the current record cannot be committed (for example, because of a key violation).

moveToRecord example

This example uses **moveToRecord** to move to a specified record in the *Orders* table. Then it displays the value of the Sale Date field for that record.

```
method pushButton(var eventInfo Event)
   var
      tcOrders   TCursor
      siRecNo      SmallInt
      daSaleDate   Date
   endVar

   tcOrders.open("orders.db")

   siRecNo = 0
   siRecNo.view("Enter a record number:")

   if siRecNo > 0 then
      if   tcOrders.moveToRecord(siRecNo) then
         daSaleDate = tcOrders."Sale Date"
         daSaleDate.view("Sale date: ")
      else
         errorShow("Invalid record number.")
      endIf
   else
      return
   endIf

endMethod
```

▪

nextRecord method

[See also](#) [Example](#) [TCursor Type](#)
Beginner

Moves to the next record in a table.

Syntax

```
nextRecord ( ) Logical
```

Description

nextRecord sets the current record to the next record in the table. If the table is in Edit mode, **nextRecord** commits changes to the current record before moving. This operation fails if the current record cannot be committed (for example, because of a key violation).

nextRecord returns False if you try to move past the end of the table. Also, the last record of the table becomes the current record, and eot returns True.

nextRecord example

In the following example, the **pushButton** method for *showNextCust* uses **nextRecord** to move a TCursor through the *Customer* table. Each time the TCursor lands on a new record, the code uses **copyToArray** to copy the contents of the record to a DynArray, then displays field values in a dialog box. When **nextRecord** attempts to move beyond the last record in the table, **eot** returns True and the **pushButton** method terminates.

```
; showNextCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  scratch DynArray[] AnyType
  tblName String
endVar
tblName = "Customer.db"

if tc.open(tblName) then

  while NOT tc.eot()           ; True until nextRecord attempts to move
                               ; beyond the end the table
    tc.copyToArray(scratch)    ; copy the record to scratch DynArray
    scratch.view("Record " + String(tc.recNo()))
    if msgQuestion("",
       "Do you want to see the next record?") = "Yes" then
      tc.nextRecord()          ; move down one record
    else
      return
    endIf
  endWhile

  msgStop("That's it!", "No more records.")

else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

nFields method

See also Example TCursor Type

Returns the number of fields in a table.

Syntax

```
nFields ( ) LongInt
```

Description

nFields returns the number of fields in the table associated with a TCursor.

nFields example

In the following example, the **pushButton** method for *thisButton* displays the number of fields in the *BioLife* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
if tc.open("BioLife.db") then
  msgInfo("Number of BioLife fields", tc.nFields())
else
  msgStop("Sorry", "Can't open BioLife.db table")
endIf

endMethod
```

▪

nKeyFields method

See also Example TCursor Type

Returns the number of fields in the current index of a table.

Syntax

```
nKeyFields ( ) LongInt
```

Description

nKeyFields returns the number of fields in the current index of the table associated with a TCursor. Use **getIndexName** to get the name of the current index.

nKeyFields example

The following example reports the number of key fields in a Paradox table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  pdoxTC   TCursor
  nkf      LongInt
  pdoxTbl  String
endVar
pdoxTbl = "Orders.db"

if pdoxTC.open(pdoxTbl) then
  nkf = pdoxTC.nKeyFields() ; Key fields in the primary index
  msgInfo(pdoxTbl,
          pdoxTbl + " has " + String(nkf) + " key fields.")
else
  msgInfo("Sorry", "Can't open " + pdoxTbl + " table.")
endIf

endMethod
```

▪

nRecords method

See also Example TCursor Type
Beginner

Returns the number of records in a table.

Syntax
`nRecords ( ) LongInt`

Description

nRecords returns the number of records in the table associated with a TCursor. This operation can take a long time for dBASE tables and large Paradox tables.

Note: When you call **nRecords** after setting a filter, the returned value does *not* represent the number of records in the filtered set. To get that information, use **cCount.**
When you call **nRecords** after setting a range, the returned value represents the number of records in the set defined by the range. This functionality was added in version 5.0.

When working with a dBASE table, **nRecords** counts deleted records if **showDeleted** is turned on. Otherwise, deleted records are not counted.

nRecords example

In the following example, the **pushButton** method for *thisButton* runs a custom method if there are more than 10,000 records in ORDERS.DB; otherwise, the code displays the current number of records in *Orders*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ordTC TCursor
  nOrders LongInt
endVar
if ordTC.open("Orders.db") then
  nOrders = ordTC.nRecords()
  if nOrders > 10000 then    ; If Orders has more than 10,000 records
    archiveOldOrders()       ; run a custom method.
  else
    msgInfo("Status",
            "Orders table has " + String(nOrders) + " records.")
  endIf
else
  msgStop("Sorry", "Can't open Orders table.")
endIf
endMethod
```

▪

open method

See also Example1 Example2 TCursor Type
Beginner

Opens a table.

Syntax

```
1. open ( const tableName String [ , const db DataBase ] [ , const indexName String ] ) Logical
2. open ( const tableVar Table ) Logical
```

Description

open associates a TCursor with the table named in *tableName*.

In syntax 1, where *tableName* is a String, you can use arguments *db* and *indexName* to specify a database and an index. If *tableName* does not specify a file name extension, Paradox assumes the extension is .DB.

Note: In version 5.0, the *indexName* option is valid for Paradox tables only. To specify an index for a dBASE table, use Table methods **usesIndexes** and **setIndex.**

In syntax 2, where *tableVar* is the name of a Table variable, you can use the Table method **setIndex** to specify an index, and you can specify the database using the Table method **attach.**

open example 1

The following example uses the first syntax to open a TCursor on the *Customer* table in the "SampleTables" database. This code uses the optional *indexName* clause, so the TCursor uses the "NameAndState" index. The following code is attached to the **pushButton** method for *firstButton*:

```
; firstButton::pushButton
method pushButton(var eventInfo Event)
var
  tc1  TCursor
  samp Database
endVar

; Create the SampleTables alias for the default sample directory.
addAlias("SampleTables", "Standard", "c:\\pdoxwin\\sample")

; Associate the samp Database var with SampleTables database.
samp.open("SampleTables")

; Associate tc1 to the Customer table in samp database,
; and use the NameAndState index.
if not tc1.open("Customer.db", samp, "NameAndState") then
   errorShow()
endIf

endMethod
```

▪

open example 2

The following example achieves the same as Example 1, but uses the second form of the syntax where a *tableVar* is used. The following code is attached to the **pushButton** method for *secondButton*:

```
; secondButton::pushButton
method pushButton(var eventInfo Event)
var
  tc1  TCursor
  samp DataBase
  tbl  Table
endVar

; Create the SampleTables alias for the default sample directory.
addAlias("SampleTables", "Standard", "c:\\pdoxwin\\sample")

; Associate the samp DataBase var with SampleTables database.
samp.open("SampleTables")

; Attach the tbl Table handle to Customer in the samp database.
tbl.attach("Customer.db", samp)
; Set the tbl index to the NameAndState index.
tbl.setIndex("NameAndState")

; Now associate tc1 TCursor to Customer table in samp database.
if not tc1.open(tbl) then
   errorShow()
endIf

endMethod
```

▪

postRecord method

See also Example TCursor Type
Beginner

Posts changes to a record.

Syntax

`postRecord ( ) Logical`

Description

postRecord posts changes to a record immediately. The record remains locked. If a key value is changed in an indexed table and the record flies away, the TCursor flies with it and continues to point to the same record. This method returns True if successful; otherwise, it returns False.

▪

postRecord example

For the following example, the **pushButton** method for the *fixName* button attempts to find a misspelled name in the *Customer* table. If the erroneous name is found, the code corrects it, then posts changes with **postRecord**.

```
; fixName::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  badName String
endVar
badName = "Usco"
goodName = "Unisco"

tc.open("Customer.db")
if tc.locate("Name", badName) then ; if the erroneous name is found
  tc.edit()                        ; put TCursor in Edit mode
  tc.Name = goodName               ; correct misspelled name
  if tc.postRecord() then          ; True if record is posted
    message("Changes posted.")
  else                             ; record is not posted (Key violation?)
    msgStop("PostRecord", "Can't post these changes.")
  endIf
  tc.endEdit()                     ; end Edit mode
  ; If the record was committed, endEdit simply ends Edit mode▪the Name
  ; field now stores "Unisco". If the record was not committed, the field
  ; retains its original value ("Usco").

else                               ; can't find "Usco" in Name field
   message("Can't find " + badName)
endIf
endMethod
```

▪

priorRecord method

See also Example TCursor Type
Beginner

Moves to the previous record in a table.

Syntax

```
priorRecord ( ) Logical
```

Description

priorRecord sets the current record to the previous record in a table. If the table is in Edit mode, **priorRecord** commits changes to the current record before moving. It returns False if the TCursor is already at the first record. Also, the first record of the table becomes the current record, and **bot** returns True.

priorRecord may not be appropriate in all databases, because some may not be bi-directional. This operation fails if the current record cannot be committed (for example, because of a key violation).

priorRecord example

In the following example, the **pushButton** method for *showPrevCust* uses **priorRecord** to move a TCursor backwards through the *Customer* table. Each time the TCursor lands on a new record, this code uses **copyToArray** to copy the contents of the record to a DynArray and display field values in a dialog box. When **priorRecord** attempts to move beyond the beginning of the table, **bot** returns True and the **pushButton** method terminates.

```
; showPrevCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  scratch DynArray[] AnyType
  tblName String
endVar
tblName = "Customer.db"

if tc.open(tblName) then

  tc.end()                      ; move to end of table
  while NOT tc.bot()            ; True until priorRecord attempts to move
                                ; beyond the beginning of the table
    tc.copyToArray(scratch)     ; copy the record to scratch DynArray
    scratch.view("Record " + String(tc.recNo()))
    if msgQuestion("",
       "Do you want to see the next record?") = "Yes" then
      tc.priorRecord()          ; move up one record
    else
      return
    endIf
  endWhile

  msgStop("That's it!", "No more records.")

else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

▪

qLocate method

See also Example TCursor Type

Searches an indexed table for a specified field value.

Syntax

`qLocate ( ``const `***`searchValue`***` AnyType [ , const `***`searchValue`***` AnyType ] * `**`)`**
`Logical`

Description

qLocate searches an indexed table for records where values in key fields exactly match the criteria specified in *searchValue*. **qLocate** searches for values in the active index; the first value corresponds to the first field in the index, the second value corresponds to the second field in the index, and so on.

The search always starts from the beginning of the table, but if no match is found, the TCursor returns to the original record. If a match is found, the TCursor moves to that record. This method does not attempt to post the current record. The operation fails if the number of search values exceeds the number of fields in the current index.

qLocate does not clear existing record locks on the Tcursor.   If a lock is present, **qLocate** will fail. To prevent failure, issue an **unLockRecord** before the **qLocate** is called. This could be particularly helpful within a scan loop.

qLocate example

This code uses **qLocate** to find a key value in the *Lineitem* table:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

if tc.open("Lineitem.db") then

  ; if qLocate can find 1002 in the first field of the
  ; index and 1316 in the second field of the index
  if tc.qLocate(1002, 1316) then

    ; make some changes to the record
    tc.edit()
    tc.Qty = 10
    tc.Total = tc."Selling Price" * tc.Qty
    tc.close()
  else
    msgStop("Sorry", "Can't find specified record.")
  endIf
else
  msgStop("Error", "Can't open Lineitem.db")
endIf

endMethod
```

recNo method

See also Example TCursor Type

Returns the record number of the current record.

Syntax

```
recNo ( ) LongInt
```

Description

recNo returns an integer representing the current record's position in the table. For a dBASE table, **recNo** returns the physical position of the record in the table; for an indexed Paradox table, it returns the record's sorted position according to the current index.

Note: When you call **recNo** after setting a filter, the returned value is represented by the ObjectPAL constant peInvalidRecordNumber. This functionality was added in version 5.0.

▪

recNo example

In the following example, the **pushButton** method for *thisButton* searches the *Customer* table for customers residing in Oregon. If any are found, this code stores record numbers in an array, then displays the contents of the array in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  ar Array[] SmallInt
  tblName String
endVar
tblName = "Customer.db"

tc.open(tblName)
if tc.locate("State/Prov", "OR") then
  ar.addLast(tc.recNo())                  ; add record number to array
  while tc.locateNext("State/Prov", "OR")  ; find the next "OR"
    ar.addLast(tc.recNo())                ; add more array elements
  endWhile
  ar.view("Record Numbers")               ; display ar array
else
  msgInfo("Nothing to do!", "Can't find \"OR\" in \"State/Prov\" field")
endIf
endMethod
```

▪

recordStatus method

See also Example TCursor Type

Reports about the status of a record.

Syntax

```
recordStatus ( const statusType String ) Logical
```

Description

recordStatus returns True or False to a question to report about the status of a record. Use the argument *statusType* to specify the status to ask about, where *statusType* is one of the following String values: New, Locked, or Modified.

"New" means the record has just been inserted into the table and is not yet posted to the table. "Locked" means a lock (implicit or explicit) has been placed on the record. "Modified" means at least one of the field values has been changed and is not yet posted to the table.

▪

recordStatus example

The following example tests whether the current record is locked. If the record is not locked, this method uses **lockRecord** to lock the record; otherwise this example informs the user that the record has previously been locked.

```
; lockThisRecord::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("orders.db")
tc.edit()

; if the current record is NOT locked
if tc.recordStatus("Locked") = False then
  ; lock the current record
  tc.lockRecord()

  ; if record is locked, this statement will display True
  msgInfo("Record Status", "recordStatus(\"Locked\") = " +
                        String(tc.recordStatus("Locked")))
else
  message("Current record is already locked.")
endIf

endMethod
```

▪

reIndex method

See also Example TCursor Type

Rebuilds specified index files.

Syntax

```
reIndex ( const IndexName String [ , const TagName String ] ) Logical
```

Description

reIndex rebuilds an index (or index tag) that is not automatically maintained. When working with a Paradox table, use *indexName* to specify an index (the field name, for a single-field index, or the full name of a composite index). When working with a dBASE table, use *indexName* to specify a .NDX file, or *indexName* and *tagName* to specify an index tag in a .MDX file. This method requires exclusive access to the table.

reIndex example

The following example opens a TCursor for *Customer* (a Paradox table), gains exclusive access to the table, then uses **reIndex** to rebuild the *Phone_Zip* index.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tc      TCursor
  pdoxTbl String
  tb      Table
endVar
pdoxTbl = "Customer.db"

tb.attach(pdoxTbl)
tb.setExclusive(Yes)

if tc.open(tb) then
  tc.reIndex("Phone_Zip")          ; rebuild Phone_Zip index
  message("Phone_Zip reindexed.")
else
  msgStop("Sorry", "Can't open " + pdoxTbl + " table.")
endIf

endMethod
```

reIndexAll method

See also Example TCursor Type

Rebuilds all index files for a table.

Syntax

`reIndexAll ( ) Logical`

Description

reIndexAll rebuilds all indexes for the table associated with a TCursor. This method requires exclusive rights to the table to rebuild a maintained index, and it requires a write lock to rebuild a non-maintained index. **reIndexAll** works only with Paradox tables, because any index opened for a dBASE table is maintained automatically.

reIndexAll example

For the following example, the **pushButton** method for a button rebuilds all indexes for the *Customer* table.

```
; reindexAllCust::pushButton
method pushButton(var eventInfo Event)
var
  tc      TCursor
  pdoxTbl String
  tb      Table
endVar
pdoxTbl = "Customer.db"

tb.attach(pdoxTbl)
tb.setExclusive(Yes) ; Need exclusive rights for a  maintained index.

if tc.open(tb) then
  tc.reIndexAll()               ; Rebuild all Customer indexes.
  message("Indexes rebuilt.")
else
  msgStop("Sorry", "Can't open " + pdoxTbl + " table.")
endIf
endMethod
```

seqNo method

See also Example TCursor Type

Returns the record number of the current record.

Syntax

```
seqNo ( ) LongInt
```

Description

seqNo returns an integer representing the current record's position in a table. For dBASE tables, **seqNo** returns the sequential position of a record as viewed by the current index. For Paradox tables, **seqNo** and **recNo** always return the same value.

Note: When you call **seqNo** after setting a filter, the returned value is represented by the ObjectPAL constant peInvalidRecordNumber. This functionality was added in version 5.0.

▪

seqNo example

In the following example, assume SCORES.DBF has three records and the second record has been deleted. The code attached to the **pushButton** method for *testSeqNo* demonstrates the difference between **seqNo** and **recNo** methods.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

  ; Scores.dbf has 3 records and the second record is deleted
tc.open("Scores.dbf")

  ; do not show deleted records
tc.showDeleted(No)

  ; this displays recNo() = 1
  ;               seqNo() = 1
msgInfo("tc Status", "recNo() = " + String(tc.recNo()) + "\n" +
                     "seqNo() = " + String(tc.seqNo()))

  ; move to the last record in the table
tc.end()

  ; this displays   recNo() = 3
  ;                 seqNo() = 2   (record number 2 is deleted)
msgInfo("tc Status", "recNo() = " + String(tc.recNo()) + "\n" +
                     "seqNo() = " + String(tc.seqNo()))

endMethod
```

▪

setBatchOff method

See also Example TCursor Type

Ends the batch processing mode invoked by a call to **setBatchOn**.

Syntax

`setBatchOff ( ) Logical`

Description

setBatchOff ends batch processing mode by removing the restrictions imposed by **setBatchOn**.

-

setBatchOff example

See the example for **setBatchOn**.

setBatchOn method

See also Example1 Example2 TCursor Type

Groups multiple operations to improve performance of table updates in a multiuser environment.

Syntax

`setBatchOn ( ) Logical`

Description

When update operations are performed after executing a **setBatchOn** statement, file I/O and concurrency control are minimized, resulting in improved performance. **setBatchOn** gives you exclusive access to a table for a short period of time. After **setBatchOn** executes, no other user or session can access, open, modify, lock, or read from the table until **setBatchOff** executes. (Other TCursors in the same session can still access the table.) If **setBatchOff** does not execute, the lock remains in effect for the life of the TCursor. **setBatchOn** is useful when several short operations should occur sequentially. **setBatchOn** should be used by advanced developers for serializing operations and improving performance. Most developers will not need this command.

Note: **setBatchOn** is intended to operate for less than two seconds. If another user attempts to update or access the current table, that user's system will freeze. If **setBatchOn** is not followed by a **setBatchOff** statement, the other user's system will be frozen for up to two minutes. After two minutes, the operation that caused the user's system to freeze will fail due to a timeout error, and the user's system will resume operation.

Other users have no way of determining whether **setBatchOn** has been called. Always call **setBatchOff** as soon as possible after calling **setBatchOn** to minimize the chances of interfering with other users.

setBatchOn example 1

Suppose a form's data model contains the *Orders* table and the *Lineitem* table linked 1:M, with *Orders* as the master table. The following code deletes all the records in the current detail set; that is, it deletes all the line items for the current order. In this example, the object referred to as Lineitem is a tableframe or a multirecord object bound to the Lineitem table.

```
method pushButton(var eventInfo Event)
   var
      ordersTC TCursor
   endVar

   ordersTC.attach(Lineitem) ; attach to the detail set
   ordersTC.edit()

   ordersTC.setBatchOn()
   while not ordersTC.eot()
      ordersTC.deleteRecord()
   endWhile
   ordersTC.setBatchOff()

endMethod
```

▪

setBatchOn example 2

Many applications require an autosequence number that must be incremented by each user who attempts to add a record to a table. This example shows how **setBatchOn** and **setBatchOff** could be used to serialize access to such an autosequence number. The following example assumes that the *NumTable* table contains a single numeric field called *Sequence Number*.

In this example, every user who attempts an operation would call the custom method **GetAutoSequence**. The first user to call the method gets the lowest sequence number. The call to **setBatchOn** holds every other user out without locking the table. Every other user who has issued a GetAutoSequence call gains access to the table sequentially.

```
method GetAutoSequence() LongInt
   var
      numTableTC   TCursor
      SequenceVar   LongInt
   endVar

   numTableTC.open("numtable.db")
   numTableTC.edit()

   numTableTC.setBatchOn()
   numTableTC."Sequence Number" = numTableTC."Sequence Number" + 1
   numTableTC.postRecord()
   SequenceVar = numTableTC."Sequence Number"
   numTableTC.setBatchOff()

   return SequenceVar
endMethod
```

▪

setFieldValue method

See also Example TCursor Type

Assigns a value to a specified field.

Syntax

```
1. setFieldValue ( const fieldName String, const value AnyType ) Logical
2. setFieldValue ( const fieldNum SmallInt, const value AnyType ) Logical
```

Description

setFieldValue sets the value of a field (*fieldName* or *fieldNum*) to *value*. This method returns True if successful; otherwise, it returns False.

You can achieve the same results using dot notation. For example, this statement uses dot notation to change the value in the Last Bid field:

```
tcVar."Last Bid" = 32.25
```

The following statement uses **setFieldValue** to achieve the same results:

```
tcVar.setFieldValue("Last Bid", 32.25)
```

setFieldValue example

In the following example, the **pushButton** method for *correctName* locates a misspelled name in the Name field, then uses **setFieldValue** to replace the original name.

```
; correctName::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  badName, goodName String
endVar

badName = "Usco"
goodName = "Unisco"
tc.open("Customer.db")
if tc.locate("Name", badName) then
  tc.edit()
  tc.setFieldValue("Name", goodName)    ; correct misspelled name
  tc.postRecord()                       ; post record to the table
  tc.endEdit()                          ; end Edit mode
  message("Usco replaced with Unisco.")
else                                    ; can't find "Usco" in Name field
  message("Can't find " + badName)
endIf
endMethod
```

setFlyAwayControl method

See also Example TCursor Type

Specifies whether flyaway information is made available to the **didFlyAway** method.

Syntax

```
setFlyAwayControl ( [ const yesNo Logical ] )
```

Description

setFlyAwayControl specifies in *yesNo* whether flyaway information is made available to the **didFlyAway** method.

When you're working with indexed tables, the **didFlyAway**, **setFlyAwayControl**, and **unlockRecord** methods are closely related. When you call **unlockRecord**, the record is posted (if no key violation exists) to the table and moved to its sorted position. Depending on whether the record moved to a new position, the TCursor may not continue to point to the posted record. This behavior is referred to as record flyaway.

You can use **didFlyAway** to test whether the record did, in fact, fly away.

When **setFlyAwayControl** is set to Yes, Paradox performs extra record-level checking for many operations. This extra work can slow an application down, so you should set **setFlyAwayControl** to Yes only when the application needs flyaway information, and set it to No (the default) otherwise.

▪

setFlyAwayControl example

See the example for **didFlyAway**.

▪

setGenFilter method

See also   Example   TCursor Type

Specifies conditions for including records in a TCursor.

Syntax

```
1. setGenFilter ( [ idxName String, [ tagName String, ] ] criteria
DynArray[ ] AnyType ) Logical
2. setGenFilter ( [ idxName String, [ tagName String, ] ] criteria Array[ ]
AnyType [ , fieldId Array[ ] AnyType ] ) Logical
```

Description

setGenFilter specifies conditions for including records in a TCursor. Records that meet all the specified conditions are included, records that don't are filtered out. Unlike **setRange**, this method does not require an indexed table.

In syntax 1, the DynArray *criteria* specifies fields and filtering conditions as follows: the tag is the field name or field number, and the item is the filter expression. For example, the following code specifies criteria based on the values of three fields.

```
criteriaDA[1]      = "Widget"              ; The value of the first field
                                           ; in the table is Widget.
criteriaDA["Size"] = "> 4"                 ; The value of the field named Size
                                           ; is greater than 4.
criteriaDA["Cost"] = ">= 10.95, < 22.50"  ; The value of the field named Cost
                                           ; is greater than or equal to 10.95
                                           ;and less than 22.50.
```

If the DynArray is empty, any existing filter criteria are removed.

In syntax 2, the Array *criteria* specifies filtering conditions, and the optional Array *fieldId* specifies field names and/or field numbers. If you omit *fieldID*, conditions are applied to fields in the order they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to the second field, and so on). The following example fills arrays for syntax 2 to specify the same criteria as the example for syntax 1.

```
criteriaAR[1] = "Widget"
criteriaAR[2] = "> 4"
criteriaAR[3] = ">= 10.95, < 22.50"
fieldAR[1] = 1
fieldAR[2] = "Size"
fieldAR[3] = "Cost"
```

If the Array is empty, any existing filter criteria are removed.

For both syntaxes, *idxName* specifies an index name (Paradox and dBASE tables) and *tagName* specifies a tag name (dBASE tables only). If you use these optional items, the index (and tag) are applied to the TCursor before the filtering criteria.

This method fails if the current record cannot be committed.

▪

setGenFilter example

In this example, the built-in **run** method for a script opens a TCursor onto the *Customer* table, then sets filter criteria on the *State/Prov* field to equal "CA". Then a **scan** loop is used to fill the DynArray *dynView* with the customer name and phone number. Finally, a **view** dialog box displays the data.

```
;Script :: run
method run(var eventInfo Event)
   var
      tc  TCursor
      dyn,
      dynView  DynArray[] AnyType
   endVar

   dyn["State/Prov"] = "CA"

   tc.open("CUSTOMER.DB")
   tc.setGenFilter(dyn)

   scan tc:
      dynView[tc."Name"] = tc."Phone"
   endScan

   dynView.view()
endMethod
```

setRange method

See also Example1 Example2 TCursor Type

Specifies a range of records to include.

Syntax

1. setRange ([const ***exactMatchVal*** AnyType] * [, const ***minVal*** AnyType, const ***maxVal*** AnyType] **)** Logical

2. setRange (***rangeVals*** Array[ ] AnyType **)** Logical

Description

setRange specifies conditions for including a range of records. Records that meet the conditions are included; records that don't are excluded. **setRange** compares the criteria you specify with values in the corresponding fields of a table's index; it fails if the current record cannot be committed or if the table is not indexed. Calling **setRange** without any arguments resets the range criteria to include the entire table.

Note: This method replaces **setFilter** included in earlier versions: both functionality and performance are enhanced. Code that calls **setFilter** will continue to execute as before.

In syntax 1, to set a range based on the value of the first field of the index, specify values in *minVal* and *maxVal*. For example, the following statement checks values in the first field of the index of each record;

```
tcVar.setRange(14, 88)
```

If a value is less than 14 or greater than 88, that record is excluded. To specify an exact match on the first field of the index, assign *minVal* and *maxVal* the same value. For example, the following statement excludes all values except 55:

```
tcVar.setRange(55, 55)
```

You can set a range based on the values of more than one field. To do so, specify exact matches *except* for the last one in the list. For example, the following statement looks for exact matches on "Borland" and "Paradox" (assuming they are the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field:

```
tcVar.setRange("Borland", "Paradox", 100, 500)
```

In syntax 2, you can pass an array of values to specify the range criteria, as listed in the following table.

Number of array items	Range specification
No items (empty array)	Resets range criteria to include the entire table.
One item	Specifies a value for an exact match on the first field of the index.
Two items	Specifies a range for the first field of the index.
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size *n*, specify exact matches on the first *n*-2 fields of the index. The last two array items specify a range for the *n*-1 field of the index.

▪

setRange example 1

For the following example, assume that the first field in Lineitem's key is "Order No." and you want to know the total for order number 1005. When you press the *getDetailSum* button, the **pushButton** method opens a TCursor for *Lineitem*, then limits the number of records included in the TCursor to those with 1005 in the first key field. After the call to **setRange**, this example uses **cSum** to display the sum of the Total field. Because the TCursor is pointing only to order number 1005, **cSum** reports summary information only for that order.

```
; getDetailSum::pushButton
method pushButton(var eventInfo Event)
var
  lineTC TCursor
  tblName String
endVar
tblName = "LineItem.db"
if lineTC.open(tblName) then

  ; this limits TCursor's view to records that have
  ; 1005 as their key value (Order No. 1005).
  lineTC.setRange(1005, 1005)

  ; now display the total for Order No. 1005
  msgInfo("Total for Order 1005", lineTC.cSum("Total"))
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

▪

setRange example 2

This example shows how to call **setRange** with a criteria array that contains more than three items. The following code sets a range to include orders from a person with a specific first name, middle initial, and last name, and an order quantity ranging from 100 to 500 items. Then it counts the number of records in this range and displays the value in a dialog box. This example assumes that the *PartsOrd* table is indexed on the FirstName, MiddleInitial, LastName, and Qty fields.

```
; setQtyRange::pushButton
method pushButton(var eventInfo Event)
   var
      tcPartsOrd   TCursor
      arRangeInfo   Array[5] AnyType
      nuCount      Number
   endVar

   arRangeInfo[1] = "Frank"      ; FirstName (exact match)
   arRangeInfo[2] = "P."         ; MiddleInitial (exact match)
   arRangeInfo[3] = "Borland"    ; LastName (exact match)
   arRangeInfo[4] = 100          ; Minimum qty value
   arRangeInfo[5] = 500          ; Maximum qty value

   if tcPartsOrd.open("PartsOrd") then
         tcPartsOrd.setRange(arRangeInfo)
         nuCount = tcPartsOrd.cCount(1)
         nuCount.view("Number of big orders by Frank P. Borland:")
   else
         errorShow("Can't open the table.")
   endIf
endMethod
```

setShowDeleted method

See also Example TCursor Type

Specifies whether to show deleted records (dBASE tables only).

Syntax

```
setShowDeleted ( [ yesNo ] ) Logical
```

Description

setShowDeleted specifies whether to show deleted records. You can use *yesNo* to specify Yes, to show deleted records, or No, if you don't want to show them. If you omit the argument, the value is True by default.

Note: setShowDeleted is valid only for dBASE tables.

setShowDeleted example

```
var
   dbfTable TCursor
endVar
if dbfTable.open("orders.dbf") then
   dbfTable.setShowDeleted(Yes)
endIf
```

▪

showDeleted method

See also Example TCursor Type

Specifies whether to show deleted records in a dBASE table.

Syntax

```
showDeleted ( [ yesNo ] ) Logical
```

Description

showDeleted specifies whether to show deleted records in a dBASE table. You can use *yesNo* to specify Yes to show deleted records, or No if you don't want to show them. If omitted, *yesNo* is Yes by default. **showDeleted** is valid only for dBASE tables because deleted records in a Paradox table cannot be shown.

▪

showDeleted example

In the following example, the **pushButton** method attached to *showDeletedRecs* calls **showDeleted** to show deleted records in ORDERS.DBF.

```
; showDeletedRecs::pushButton
method pushButton(var eventInfo Event)
var
  dbfTC TCursor
endVar
if dbfTC.open("Orders.dbf") then
  dbfTC.showDeleted(Yes)
else
  msgStop("Sorry", "Can't open Orders.dbf table.")
endIf
endMethod
```

▪

skip method

See also Example TCursor Type

Moves forward or backward a specified number of records in a table.

Syntax

```
skip ( [ const nRecords LongInt ] ) Logical
```

Description

skip sets the current record (and the record buffer) to the record *nRecords* from the current record. If **skip** tries to move beyond the limits of the table, you'll get an error, and the current record will be the first or last record of the table, as appropriate. This operation fails if the current record cannot be committed (for example, because of a key violation).

Positive values for *nRecords* move forward through the table (**skip**(1) is the same as **nextRecord**), negative values move backward (**skip**(-1) is the same as **priorRecord**), and a value of 0 doesn't move (**skip**(0) is the same as **currRecord**). If omitted, *nRecords* is 1 by default.

skip example

The following example demonstrates how **skip** affects a TCursor's record position in a table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.db")

tc.skip(5)      ; ahead 5 records. tc.recNo() = 6
tc.skip(-3)     ; back 3 records. tc.recNo() = 3
tc.skip(-5)     ; fails--attempted to move beyond the
                ; beginning of the table.
                ; tc.recNo() = 1
                ; tc.bot() = True

endMethod
```

▪

sortTo method

See also Example TCursor Type

Sorts a table.

Syntax

```
1. sortTo ( const destTable String, const numFields SmallInt, const
sortFields Array[ ] String, const sortOrder Array[ ] SmallInt ) Logical
2. sortTo ( const destTable Table, const numFields SmallInt, const sortFields
Array[ ] String, const sortOrder Array ) Logical
```

Description

sortTo sorts a table based on values of fields, and puts the results into *destTable*.

sortFields is an array of strings or integers specifying the fields on which to sort. The size of the *sortFields* array is specified in *numFields*. *sortOrder* is an array of integers, where a value of 0 specifies a sort in ascending order, and a value of 1 specifies descending order. The two arrays must be the same size, specified in *numFields*. Element 1 of *sortOrder* specifies how to sort the field named in element 1 of *sortFields*, and so on.

This method requires at least a read only lock on the source table, and a full lock on the destination table. If *destTable* exists, it will be overwritten without asking for confirmation. If *destTable* is open, this method fails. You cannot use **sortTo** to sort a table onto itself; use a **sort** structure for that.

sortTo example

The following example sorts the *Customer* table to the CUSTSORT.DB table, then opens the sorted table. If the *Customer* table cannot be write-locked, this example informs the user of the error and aborts the operation. If the *CustSort* destination table exists, the user is given an opportunity to continue or abort.

The following code goes in the Const window for the *sortCustButton* button:

```
; sortCustButton::Const
const
   kAscending = 0
   kDescending = 1
endConst
```

The following code goes in the Var window for the *sortCustButton* button:

```
; sortCustButton::var
var
  sortFlds Array[2]  String
  sortOrder Array[2] SmallInt
  tc                 TCursor
  srcTbl, destTbl    String
  noSort             Logical
  sortTbl            TableView
endVar
```

The following code is attached to the button's **open** method. This code assigns **open** a TCursor for the *Customer* table and initializes the array elements. These assignments determine the sort criteria for **sortTo**.

```
; sortCustButton::pushButton
method open(var eventInfo Event)
srcTbl = "Customer.db"
destTbl = "CustSort.db"
if tc.open(srcTbl) then
  noSort = False                ; flag for pushButton method
  sortFlds[1] = "First Contact"  ; sort by First Contact
  sortOrder[1] = kAscending      ; in ascending order

  sortFlds[2] = "Country"        ; then by Country
  sortOrder[2] = kDescending     ; in descending order
else
  noSort = True
endIf

endMethod
```

The following code is attached to the **pushButton** method for the *sortCustButton* button. When the button is pressed, the code attempts to place a write lock on the source table (CUSTOMER.DB), prompts the user if the destination table exists (CUSTSORT.DB), then sorts *Customer* to *CustSort* based on the values in the *sortFlds* and *sortOrder* arrays. After CUSTSORT.DB is created (or overwritten), this example opens it as a TableView.

```
; sortCustButton::pushButton
method pushButton(var eventInfo Event)
if noSort = False then
  if tc.lock("Write") then
    if isTable(destTbl) then
      if msgQuestion("Overwrite?",
           "Replace " + destTbl + " ?") <> "Yes" then
           msgInfo("Canceled", "Operation canceled.")
           return
      endIf
    endIf
    tc.sortTo(destTbl, 2, sortFlds, sortOrder)
    sortTbl.open(destTbl)
  else
    msgStop("Stop!", "Can't write-lock " + srcTbl + " table.")
  endIf
else
  msgStop("Sorry", "Can't open " + srcTbl + " table.")
endIf
endMethod
```

subtract method

See also Example TCursor Type

Subtracts the records in one table from another table.

Syntax

```
1. subtract ( const destTable String ) Logical
2. subtract ( const destTable Table ) Logical
3. subtract ( const destTable TCursor ) Logical
```

Description

subtract checks whether any records in the source table are also in *destTable*. If so, **subtract** deletes them from *destTable* without asking for confirmation.

If *destTable* is indexed, **subtract** deletes all records with indexes that exactly match values in corresponding index fields in the source table. If *destTable* is not indexed, **subtract** deletes all records that exactly match any record in the source table. Whether tables are indexed or not, this method considers only fields that *could* be keyed (based on data type, not position). For example, numeric fields are considered, but formatted memos are not. This method requires read/write access to both tables.

Note: If the destination table is not indexed, this operation can be time-consuming.

This method tries, for the duration of the retry period, to place a full lock on both tables. If locks cannot be placed, an error results.

▪

subtract example

In the following example, the **pushButton** method for *subtractCust* deletes records from the *Customer* table that exactly match those in the *Answer* table.

```
; subtractCust::pushButton
method pushButton(var eventInfo Event)
var
  ansTC, custTC TCursor
endVar

if ansTC.open(":PRIV:Answer.db") and
   custTC.open("Customer.db") then

  ansTC.subtract(custTC)              ; subtract Answer records from Customer

else
  msgStop("Stop!", "Can't open tables.")
endIf

endMethod
```

▪

switchIndex method

See also Example TCursor Type
Beginner

Specifies another index to use to view the records in a table.

Syntax

```
1. switchIndex ( [ const indexName String ] [ , const stayOnRecord Logical ]
) Logical
2. switchIndex ( [ const indexFileName String [ , const tagName String ] ]
[ , const stayOnRecord Logical ] ) Logical
```

Description

switchIndex specifies in *indexName* an index file to use with a table. In syntax 1, *indexName* specifies an index to use with a Paradox table. If you omit *indexName*, the table's primary index is used.

Syntax 2 is for dBASE tables, where *indexFileName* can specify a .NDX file or a .MDX file, and optional argument *tagName* specifies an index tag in a production index (.MDX) file.

In both syntaxes, if optional argument *stayOnRecord* is Yes, this method maintains the current record after the index switch; if it is No, the first record in the table becomes the current record. If omitted, *stayOnRecord* is No by default.

switchIndex example

In the following example, assume that *Customer* is a keyed Paradox table that has a secondary index named "NameAndState". This example opens a TCursor for *Customer*, calls **switchIndex** to switch from the primary index to the "NameAndState" index, then displays the first value in the Name field. Since the TCursor is sorted on Name and State fields in ascending order, the field value displayed is the first name in ascending sort order.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
   var
      tc TCursor
   endvar

   tc.open("Customer.db")              ; open TCursor for Customer
   tc.switchIndex("NameAndState")      ; switch to index NameAndState
   tc.home()                           ; move to the first record
   msgInfo("First Record", tc.Name)    ; display value in Name field
   tc.switchIndex( )                   ; to restore primary index
   { tc.switchIndex (" ", True) to stay on the same record. }
   msgInfo("First Record", tc.Name)    ; display value in Name field
endMethod
```

▪

tableName method

See also Example TCursor Type

Returns the name of the table associated with a TCursor.

Syntax

```
tableName ( ) String
```

Description

tableName returns the name of the table associated with a TCursor. This method is useful when you're passing variables to the TCursor open method.

▪

tableName example

In the following example, the **pushButton** method for *thisButton* uses **findFirst** and **findNext** methods from the FileSystem type to locate Paradox tables in the current working directory. This example searches each table for a value in the Name field of the current table. This example opens all of the tables in the current directory that have "Unisco" in the "Name" field.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  tc TCursor
  tb TableView
endVar
if fs.findFirst("*.db") then
  while fs.findNext()
    tc.open(fs.Name())                  ; open TCursor for a .db file
    if tc.locate("Name", "Unisco") then ; if we find Unisco in Name field
      tb.open(tc.tableName())           ; open table associated with TCursor
    endIf
    tc.close()
  endWhile
endIf

endMethod
```

■

tableRights method

See also Example TCursor Type

Reports about the operations you can perform on a table.

Syntax

`tableRights ( const rights String ) Logical`

Description

tableRights reports about a user's rights to a table, where *rights* is one of the following:

Value	Description
"ReadOnly"	Read from the table, but not change it.
"Modify"	Enter or change data.
"Insert"	Add new records.
"InsDel"	Add and delete records.
"Full" or "All"	Perform all of the above operations.

This method returns True if the user has the specified rights; otherwise, it returns False.

tableRights example

The following example reports whether the user has "InsDel" rights to the *Orders* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   myRights Logical
   ordersTC TCursor
endVar
ordersTC.open("orders.db")
ordersTC.edit()
myRights = ordersTC.tableRights("InsDel")

  ; this displays True if you have InsDel rights to Orders.db
msgInfo("Rights to Enter?", myRights)

endMethod
```

▪

type method

See also Example TCursor Type

Returns the type of a table.

Syntax

```
type ( ) String
```

Description

type returns a string describing the type of a table, either Paradox or dBASE.

type example

The following example compacts (removes deleted records) from the *Orders* table if **type** returns dBASE; otherwise a message indicates that *Orders* is a Paradox table.

```
; compact::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

tc.open("Orders.db")

; if Orders.db is a dBASE table
if tc.type() = "dBASE" then
  ; remove deleted records
  tc.compact()
else
  ; otherwise, display the type of table
  msgStop("Stop!", "Orders.db is a " + tc.type() + " table.")
endIf

endMethod
```

▪

unDeleteRecord method

See also Example TCursor Type
Beginner

Undeletes the current record from a dBASE table.

Syntax

```
unDeleteRecord ( ) Logical
```

Description

unDeleteRecord undeletes the current record of a dBASE table. This operation can be successful only if **showDeleted** has been set to True, the current record is a deleted record, and the TCursor is in Edit mode.

▪

unDeleteRecord example

The following example opens a TCursor for SCORES.DBF (a dBASE table), then uses **showDeleted** to display all deleted records. Then, the code attempts to locate a specific record in the table. This example uses **isRecordDeleted** to determine whether the record has been deleted; if it has, it is undeleted with **unDeleteRecord**. The following code is attached to the **pushButton** method for *thisButton*:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Scores.dbf")                  ; open TCursor on a dBASE table
tc.showDeleted()                       ; show deleted records
if tc.locate("Name", "Jones") then     ; if locate finds Jones in Name field
  if tc.isRecordDeleted() then         ; if the record has been deleted
    tc.edit()                          ; begin Edit mode
    tc.undeleteRecord()                ; undelete the record
    message("Jones record undeleted")
  endIf
else
  msgStop("Error", "Can't find Jones.")
endIf
endMethod
```

unlock method

See also Example TCursor Type
Beginner

Removes specified locks from a TCursor.

Syntax

`unlock ( const lockType String ) Logical`

Description

unlock attempts to remove locks explicitly placed on the table pointed to by a TCursor, where *lockType* is one of the following String values, listed in order of decreasing strength and increasing concurrency.

String value	Description
Full	The current session has exclusive access to the table. No other session can open the table. Cannot be used with dBASE tables.
Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

unlock removes locks explicitly placed by a particular user or application using **lock**; it has no effect on locks placed automatically by Paradox. Each time you lock a table explicitly, be sure to unlock it as soon as you no longer need the explicit lock. This ensures maximum concurrent availability of tables. Also, when you lock a table twice, you must unlock it twice. You can use **lockStatus** (defined for the TCursor and UIObject types) to determine how many explicit locks you have placed on a table. **unlock** returns False if you try to unlock a table that isn't locked or cannot be unlocked.

If successful, this method returns True; otherwise, it returns False.

▪

unlock example

The following example opens a TCursor for *Customer* (a Paradox table), places a full lock on the table, then uses **reIndex** to rebuild the *Phone_Zip* index. Once the index is rebuilt, this code unlocks *Customer* so other users on a network can gain access to the table.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

if tc.open(pdoxTbl) then
  if tc.lock("Full") then    ; attempt to gain exclusive access
    tc.reIndex("Phone_Zip")  ; rebuild Phone_Zip index
    tc.unLock("Full")        ; unlock the table
  else
    msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
  endIf
else
  msgStop("Sorry", "Can't open " + pdoxTbl + " table.")
endIf
endMethod
```

unLockRecord method

See also Example TCursor Type
Beginner

Unlocks the current record.

Syntax
`unLockRecord ( ) Logical`

Description
unLockRecord unlocks the current record if it is locked. If you try to unlock a record that isn't locked, you'll get an error. This operation fails if the current record cannot be committed (for example, because of a key violation).

If the table is indexed, the record is posted to the table and moved to its sorted position. Depending on whether the record moved to a new position, the TCursor may not continue to point to the posted record. This behavior is referred to as record *flyaway*.

If a key value changes when the record is unlocked, the record may fly away to a new position in the table; however, the TCursor will not fly with it. You can also use **didFlyAway** to test whether the record, did, in fact, fly away.

unLockRecord example

In the following example, the **pushButton** method for *thisButton* attempts to locate a misspelled value in the *Name* field of the *Customer* table. If the value is found, this code locks the record, corrects the value in the field, then unlocks the record with **unLockRecord**.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
if tc.open("Customer.db") then
  if tc.locate("Name", "Usco") then
    tc.edit()
    tc.lockRecord()           ; lock current record
    tc.Name = "Unisco"        ; change field value
    tc.unlockRecord()         ; unlock current record
    message("Name changed to \"Unisco\"")
  else
    msgStop("Sorry", "Can't find \"Usco\" in \"Name\" field.")
  endIf
else
  msgStop("Sorry", "Can't open Customer.db table.")
endIf

endMethod
```

update method

See also Example TCursor Type

Assigns values to fields in the current record of a TCursor.

Syntax

```
1. update ( const fieldName String, const fieldValue AnyType
     [ , const fieldName String, const fieldValue AnyType ] * ) Logical

2. update ( const fieldNum SmallInt, const fieldValue AnyType
     [ , const fieldNum SmallInt, const fieldValue AnyType ] * ) Logical
```

Description

update assigns values to one or more fields in the current record of a TCursor. update improves performance by enabling you to update an entire record with a single statement instead of assigning field values one at a time. Use *fieldName* (syntax 1) or *fieldNum* (syntax 2) to specify fields, and use *fieldValue* to specify the new field value.

You can combine field names and field numbers in the same update statement. Performance improves when you use field numbers instead of field names.

▪

update example

The following example shows how to use **update** to set the values of three fields with one statement. First, the following code shows how to assign values to the *PartNum*, *PartName*, and *Cost* fields of the *Parts* table without using *update*:

```
var
   partsTC    TCursor
   partNumID   SmallInt
endVar

partsTC.open("parts")
partNumID = partsTC.fieldNo("PartNum")

if partsTC.locate("PartName", "Widget") then
   partsTC.edit()

   partsTC.(partNumID) = "G01"
   partsTC.PartName    = "Gadget"
   partsTC.Cost        = 2.50

   partsTC.endEdit()
endIf
```

The following code calls **update** to accomplish the same thing:

```
var
   partsTC    TCursor
   partNumID   SmallInt
endVar

partsTC.open("parts")
partNumID = partsTC.fieldNo("PartNum")

if partsTC.locate("PartName", "Widget") then
   partsTC.edit()

   partsTC.update(partNumID, "G01", "PartName", "Gadget", "Cost", 2.50)

   partsTC.endEdit()
endIf
```

updateRecord method

See also Example TCursor Type
Beginner

Updates the existing record with data from the new record when a key violation exists.

Syntax

`updateRecord ( [ const moveTo Logical ] ) Logical`

Description

updateRecord overwrites the existing record with values from the unposted new record when a key violation exists. The record is posted to the table and does not remain locked. If optional argument *moveTo* is True, the TCursor will point to the record after it is posted to the table; if False, the TCursor points to the record following the position of the original record.

If no key violation exists, this method behaves like **unlockRecord**.

▪

updateRecord example

See the example for **attachToKeyViol**.

TextStream type

A TextStream is a sequence of characters read from (or written to) a text file. TextStreams contain only ANSI characters; formatting information such as font, alignment, and margins is not included. However, nonprinting characters, such as carriage returns and line feeds (CR/LF) are included.

Paradox maintains a file position pointer that behaves like an insertion point cursor in a word processor. The pointer tells you how far (how many characters) you are from the beginning of the file. Counting begins with 1 (not with 0, as in some other languages).

Methods for the TextStream type

advMatch method

See also Example TextStream Type

Searches for a pattern of characters in a text file.

Syntax

```
advMatch ( var startIndex LongInt, var endIndex LongInt, const pattern String
) Logical
```

Description

advMatch searches a text file for a pattern of characters represented by the variable *pattern*. If *startIndex* is assigned a value, the search starts at the *startIndex* position; otherwise, the search starts at the beginning of the file. The position in *endIndex* does not indicate the end of the range to search. If the pattern is found, the position of the first matching character is stored in *startIndex*, and the position of the last matching character is stored in *endIndex.*

advMatch returns True if *pattern* is found in the file; otherwise, it returns False. This method is case sensitive by default, but you can use the String procedure **ignoreCaseInStringCompares** to change the case behavior.

If you supply *pattern* from within a method, you need to use two backslashes when you want to tell **advMatch** to treat a special character as a literal; for example, \\(tells **advMatch** to treat the parenthesis as a literal character. Two backslashes are required in this situation because of the ambiguity between the compiler's interpretation of a backslash (used in escape sequences such as \t for a tab) and **advMatch**'s understanding of a backslash. When the compiler sees a string with an embedded escape sequence, such as a "\tstart", it interprets the "\t" as a tab, followed by the word "start." The backslash character has a special meaning to the compiler, but it also has a special meaning to **advMatch**. (See the entry for **advMatch** in the String type.)

If you supply *pattern* from a field in a table or a TextStream, special **advMatch** symbols are recognized without a preceding backslash, and one backslash and plus symbol (\+) yields a literal character.

To specify *pattern*, use a string with the optional symbols listed in the table below.

Symbol	Matches
\	Use backslash to include special characters (for example, \t for Tab) as regular characters. (Remember to use two backslashes in quoted strings.)
[ ]	Match the enclosed set. For instance, [aeiou0-9] match a, e, i, o, u, and 0 through 9.
[^]	Do not match the enclosed set. For instance, [^aeiou0-9] matches anything except a, e, i, o, u, and 0 through 9.
()	Grouping.
^	Beginning of string (do not confuse this with [^], where the ^ acts as a logical NOT).
$	End of string.
..	Match anything.
@	Match any single character.
*	Zero or more of the preceding character or expression.
+	One or more of the preceding character or expression.
?	None or one of the preceding character or expression.
|	OR operation.

For examples, see Sample search strings with wildcards in the User's Guide help.

▪

advMatch example

The following example assumes that a file named PDXQUOTE.TXT exists in the current working directory. The file contains the following text:

```
How wonderful that we have met with paradox.
Now we have some hope of making progress.
Niels Bohr
```

The call to **advMatch** specifies "@o@e" as the pattern to search. This pattern matches any character followed by an **o** followed by any character followed by an **e**. When this pattern is found, the variables *firstChar* and *lastChar* store the positions of the first and last matching characters. The calls to **setPosition** and **readChars** read the matching characters and store them in the variable *theMatch*.

```
; findSome::pushButton
method pushButton(var eventInfo Event)
var
  pdq                  TextStream
  firstChar, lastChar LongInt
  theMatch             String
endvar
if pdq.open("pdxquote.txt", "R") then
  if pdq.advMatch(firstChar, lastChar, "@o@e") then
    msgInfo("The position found", firstChar)
    pdq.setPosition(firstChar)
    pdq.readChars(theMatch, lastChar - firstChar)
    message(theMatch)               ; displays "some"
  else
    msgInfo("Sorry", "Match not found.")
  endIf
  pdq.close()
else
  msgInfo("Sorry", "Couldn't open the requested text file.")
endIf
endMethod
```

▪

close method

See also Example TextStream Type

Closes a text file.

Syntax

```
close ( ) Logical
```

Description

close closes a text file and writes the contents of all text buffers to disk. It also ends the association between a TextStream variable and the underlying text file.

▪

close example

The following example declares one TextStream variable, *ts*, and calls **open** to associate *ts* with the text file PDXQUOTE.TXT, then calls **close** to end the association.

```
; quoteALine::pushButton
method pushButton(var eventInfo Event)
var
  ts        TextStream
  firstLine String
endvar
ts.open("pdxQuote.txt", "R")
ts.readLine(firstLine)
firstLine.view("Line 1 of PDXQUOTE.TXT")
ts.close()
endMethod
```

■

commit method

See also Example TextStream Type

Writes the contents of the text buffer to disk.

Syntax

`commit ( )`

Description

commit empties the text buffer and writes the contents to disk. The file stays open and the position of the file pointer does not change.

commit example

In the following example, the *createText* button creates a new file called MYTEXT.TXT, writes a line to it, commits the current version of the TextStream, then closes the file.

```
; createText::pushButton
method pushButton(var eventInfo Event)
var
  ts TextStream
endVar

ts.create("myText.txt")
msgInfo("TextStream position is now", ts.position()) ; displays 1

ts.writeLine("This is some text.")
msgInfo("TextStream position is now", ts.position()) ; displays 21

ts.commit()
msgInfo("TextStream position is now", ts.position()) ; still 21
ts.close()

endMethod
```

▪

create method

See also Example TextStream Type

Creates a text file for reading and writing.

Syntax

```
create ( const fileName String ) Logical
```

Description

create creates the text file *fileName* and opens it for reading and writing. If *fileName* exists, **create** overwrites it without prompting for confirmation. You can specify a directory in which to create the file using a full DOS path or an alias. If you don't specify a path or an alias, Paradox creates the file in the working directory (:WORK:).

This method returns True if successful; otherwise, it returns False. If the file is successfully created, it is opened for reading and writing.

Note: The following statements are equivalent:

```
ts.create("newText.txt")
ts.open("newText.txt, "NW")
```

▪

create example

The following code is attached to a button's **pushButton** method. It consists of a variable declaration block, a procedure declaration, and the body of the method. In the body of the method, the call to the FileSystem method **findFirst** checks for the existence of a file named RICK.TXT. If it doesn't exist, the custom procedure **addLine** creates it and adds a line to it. If the file does exist, a dialog box confirms the decision to overwrite the file.

```
; createFile::pushButton
var
  ts                TextStream
  firstLine         String
  allLines Array[] String
  fs                FileSystem
endvar

proc addLine()
; Create a file, open for writing and reading
  ts.create(":PRIV:rick.txt")
  ts.writeLine("Here's looking at you, kid.")
  ts.home()
  ts.readLine(allLines)
  allLines.view("Rick says:")
endProc

method pushButton(var eventInfo Event)
if not fs.findFirst(":PRIV:rick.txt") then
  addLine()
else
  if msgYesNoCancel(":PRIV:RICK.TXT",
                    "Overwrite this file?") = "Yes" then
    addLine()
  endIf
endIf
endMethod
```

end method

See also Example TextStream Type

Sets the pointer to the end of a text file.

Syntax

```
end ( )
```

Description

end sets the pointer to the last character of a text file.

▪

end example

The following example assumes that a file named PDXQUOTE.TXT exists in the user's private directory. The file contains the following text:

```
How wonderful that we have met with paradox.
Now we have some hope of making progress.
Niels Bohr
```

The code in this example is attached to the built-in **newValue** method of a field object displayed as two radio buttons. The values of the radio buttons are "Overwrite" and "Append." Choose one to specify whether to insert text at the beginning of the file (which overwrites existing text) or append it to the end of the file. If you choose "Overwrite," the call to **home** moves the pointer to position 1. If you choose "Append," the call to **end** moves the pointer to position 103 (the end of this particular file).

```
; insertAppendField::changeValue
method newValue(var eventInfo Event)
var
  ts TextStream
  allLines Array[] String
endVar
if eventInfo.reason() = EditValue then
  ts.open(":PRIV:pdxquote.txt", "W")
  switch
    case self.value = "Overwrite" :
      ts.home()
      ts.writeLine(DateTime())   ; time stamp the file at beginning
      ; file will read:
      ; DateTimeStamp (depends on date/time)
      ; have met with Paradox.
      ; Now we have some hope of making progress.
      ; Niels Bohr
    case self.value = "Append" :
      ts.end()
      ts.writeLine(DateTime())   ; time stamp the file at end
      ; file will read:
      ; How wonderful that we have met with Paradox.
      ; Now we have some hope of making progress.
      ; Niels Bohr
      ; DateTimeStamp (depends on date/time)
  endSwitch
  ts.home()
  ts.readLine(allLines)
  allLines.view()
  ts.close()
endIf
endMethod
```

eof method

See also Example TextStream Type

Tests for a move past the end of a text file.

Syntax

`eof ( ) Logical`

Description

eof returns True if an operation tries to move the file pointer past the end of a text file; otherwise, it returns False.

▪

eof example

The following example assumes that a file named PDXQUOTE.TXT exists in the user's private directory. The file contains the following text:

```
How wonderful that we have met with paradox.
Now we have some hope of making progress.
Niels Bohr
```

The **while** loop reads each of the three lines from the file and displays it in a dialog box. Then **eof** returns True, and a dialog box tells the user that there's no more text in the file.

```
; lineAtATime::pushButton
method pushButton(var eventInfo Event)
var
  pdq      TextStream
  textLine String
endVar

pdq.open(":PRIV:pdxquote.txt", "r")
while not pdq.eof()        ; quit loop when you hit the end of the file
  pdq.readLine(textLine)   ; read the next line
  msgInfo("Position " + String(pdq.position()), textLine)
endWhile
msgInfo("Finished", "No more text")
endMethod
```

home method

See also Example TextStream Type

Sets the pointer to the beginning of a text file.

Syntax

`home ( )`

Description

home sets the file pointer to the first character of a text file.

▪

home example

See the example for **end**.

▪

open method

See also Example1 Example2 TextStream Type

Opens a text file in a specified mode.

Syntax

```
open ( const fileName String, const mode String ) Logical
```

Description

open opens *fileName* in the mode specified in *mode*, and associates a FileSystem variable with the underlying file. The modes are listed in the following table (the case of a mode specification doesn't matter).

Mode specification	Description
"a"	Append and read.
"r"	Read only.
"w"	Write and read.
"nw"	New file, write and read.

If the file exists, the "nw" mode overwrites the file without asking for confirmation.

Note: The following statements are equivalent:

```
ts.open("new.txt", "NW")
ts.create("new.txt")
```

Opening a file in any mode except "a" (append) sets the pointer to the beginning of the file.

You can specify a directory from which to open the file using a full DOS path or an alias. If you don't specify a path or an alias, Paradox looks for the file in the working directory.

This method returns True if successful; otherwise, it returns False.

open example 1

The following example uses an alias with **open** to create a text file in the private directory and write a line of text to it:

```
var
   ts TextStream
endVar
if ts.open(":PRIV:memo14.txt", "NW") then
   ts.writeLine("This is private!")
endIf
```

▪

open example 2

You can associate more than one TextStream variable with the same file. Both variables have equal rights to the file, and Paradox maintains separate pointers for each variable. The following example declares two TextStream variables, *ts1* and *ts2*, and calls **open** to associate each of them with the text file NEWTEXT.TXT. As statements are written to the file, messages display the pointer position for each variable.

```
; openStreams::pushButton
method pushButton(var eventInfo Event)
var
  ts1, ts2  TextStream
  firstLine String
  allLines  Array[] String
endvar
ts1.open("newText.txt", "nw")       ; open a new file read/write
ts1.writeLine("Written by ts1.")
ts1.writeLine("This is line 2.")
msgInfo("Text stream one", ts1.position()) ; displays 35
ts1.commit()                        ; write it out to disk, so that
                                    ; ts2 will get most current version

ts2.open("newText.txt", "w")        ; open existing file read/write
msgInfo("Text stream one", ts1.position()) ; displays 35
msgInfo("Text stream two", ts2.position()) ; displays 1

ts2.writeLine("Written by ts2.")
msgInfo("Text stream one", ts1.position()) ; displays 35
msgInfo("Text stream two", ts2.position()) ; displays 18

ts1.home()
ts1.readLine(allLines)      ; reads all lines into an array
allLines.view("ts1")        ; displays:
                            ; Written by ts1.
                            ; This is line 2.
; ts1 does not reflect changes made by ts2
; unless ts1 is closed and reopened.
endMethod
```

▪

position method

[See also](#) [Example](#) [TextStream Type](#)

Returns the pointer's position in a text file.

Syntax

`position ( )` `LongInt`

Description

position returns an integer representing the pointer's position in a text file. **position** counts both printing and nonprinting characters. Counting begins with 1 (not with 0).

▪

position example

It may be helpful to think of **position** as returning the number of the next character in the file. As the following example shows, when you create a new text file and call **position**, it returns 1. The call to **writeLine** adds 14 characters to the file: 12 printing characters and the carriage return and line feed (CR/LF) pair. The next character will be 15, so **position** returns 15.

```
var newFile TextStream endVar
newFile.open("newmemo.txt", "nw")
message(newFile.position()) ; displays 1
sleep(1000)
newFile.writeLine("Don't panic.")
message(newFile.position()) ; displays 15
                            ; 12 printing characters + CR/LF = 14
                            ; next character will be 15
sleep(1000)
```

▪

readChars method

See also Example TextStream Type

Reads a specified number of characters from a text file.

Syntax

`readChars ( var string String, const nChars SmallInt ) Logical`

Description

readChars reads the number of characters specified in *nChars* and stores them in *string*. **readChars** starts reading from the current pointer position. This method returns True if successful; otherwise, it returns False.

readChars example

The following example assumes that a file named PDXQUOTE.TXT exists in the current working directory. The file contains the following text:

```
How wonderful that we have met with paradox.
Now we have some hope of making progress.
Niels Bohr
```

The call to **readChars** reads the first 100 characters from the file:

```
; getLetters::pushButton
method pushButton(var eventInfo Event)
var
  letter  TextStream
  myChars String
endVar
letter.open("pdxquote.txt", "r")
if letter.readChars(myChars, 100) then

  msgInfo("The first 100 characters are:", myChars)
endIf
endMethod
```

▪

readLine method

[See also](#) [Example1](#) [Example2](#) [TextStream Type](#)

Reads a line from a text file.

Syntax

```
1. readLine ( var value String ) Logical
2. readLine ( var stringArray Array[ ] String ) Logical
```

Description

readLine reads characters from a line of text from a file until a CR/LF pair is encountered (or 1,023 characters have been read), and moves the file pointer to the first position after the CR/LF pair (or after the 1,023rd character). **readLine** begins reading from the current pointer position. This method returns True if successful; otherwise, it returns False.

Syntax 1 stores a single line in *value*. The CR/LF pair is not stored.

Syntax 2 stores the entire file in *stringArray,* where *stringArray* is a resizable array of strings and each array item stores one line from the file. The CR/LF pair is not stored.

▪

readLine example 1

The following example creates a 2-line text file, then calls **readLine** to read the first line into a String variable. **readLine** reads the four characters before the CR/LF in the first line, then skips over the CR/LF characters, and sets the pointer.

```
method pushButton(var eventInfo Event)
var
   ts TextStream
   oneLine String
endvar

ts.create("newtext.txt")
ts.writeLine("1234")
ts.writeLine("5678")
ts.home()

ts.readLine(oneLine)
message(oneLine.size()) ; displays 4 (doesn't include CR/LF)
sleep(1000)
message(ts.position()) ; displays 7 (skips over CR/LF)
sleep(1000)
endMethod
```

▪

readLine example 2

The following example creates a 3-line text file, then calls **readLine** to read the entire file into an array, then displays the array in a dialog box.

```
method pushButton(var eventInfo Event)
var
   letter TextStream
   allLines Array[] String
endVar

letter.open("letter.txt", "nw")
letter.writeLine("Dear Customer,")
letter.writeLine("Thank you for your interest in our new product.")
letter.writeLine("A representative will call you next week.")

letter.home() ; move the pointer to the beginning of the file

letter.readLine(allLines)
allLines.view("Entire letter")     ; displays the entire letter
letter.close()
endMethod
```

▪

setPosition method

See also Example1 Example2 TextStream Type

Positions the pointer in a text file.

Syntax

setPosition (const ***offset*** LongInt **)**

Description

setPosition positions the file pointer *offset* characters from the beginning of a text file. (CR/LF) characters are considered part of the file, and can be overwritten. Specifying a position before the beginning or after the end of file moves the pointer to the corresponding position.

setPosition example 1

In the following example, the *showPositions* button first writes a line to a new text file, MEMO.TXT. The method then moves back to the fourth character, overwrites that character with "4", then rereads and displays the line.

```
; showPositions::pushButton
method pushButton(var eventInfo Event)
var
  myFile  TextStream
  lineOne String
endVar
myFile.open(":PRIV:memo.txt", "nw")         ; open new file as read/write
myFile.writeLine("1235")                    ; 4 characters plus CR/LF
msgInfo("Where am I?", myFile.position())   ; displays 7

myFile.setPosition(4)                ; move to character 4
myFile.writeString("4")              ; now, line is "1234"
myFile.home()                        ; same as setPosition(1)
myFile.readLine(lineOne)
msgInfo("This is line one", lineOne) ; displays  "1234"
endMethod
```

▪

setPosition example 2

The following example shows what happens when you attempt to move the pointer beyond the end of a file or before the beginning of a file.

```
; showPositions::pushButton
method pushButton(var eventInfo Event)
var
  myFile  TextStream
endVar

myFile.open(":PRIV:memo.txt", "r")  ; open existing file for read
myFile.setPosition(100)             ; beyond end of file
msgInfo("End", myFile.position())   ; displays 7▪the real end
myFile.setPosition(-100)            ; before beginning of file
msgInfo("Home", myFile.position())  ; displays 1
▪the beginning
endMethod
```

▪

size method

See also Example TextStream Type

Returns the number of characters in a text file.

Syntax

size ( ) `LongInt`

Description

size returns the number of characters in a text file, including nonprinting characters such as carriage returns and line feeds (CR/LF).

size example

The following example creates a TextStream, writes a line to it, then shows the size of the file.

```
; showSize::pushButton
method pushButton(var eventInfo Event)
var
  myText TextStream
endVar
myText.create("short.txt")
myText.writeLine("1234")
msgInfo("What size am I?", myText.size()) ; displays 6
; 4 printing characters "1234", and 2 nonprinting characters CR/LF
myText.close()
endMethod
```

▪

writeLine method

See also Example TextStream Type

Writes a string to a text file.

Syntax

`writeLine ( const value AnyType [ , const value AnyType ] * ) Logical`

Description

writeLine writes a comma-separated list of *value*s to a text file, and appends a CR/LF character pair. Compare this method to **writeString**, which doesn't append a CR/LF pair.

▪

writeLine example

See the example for **create**.

■

writeString method

[See also](#) [Example](#) [TextStream Type](#)

Writes a character string to a text file.

Syntax

`writeString ( const value AnyType, [ , const value AnyType ] * ) Logical`

Description

writeString writes a comma-separated list of *values* to a text file, but does not append a CR/LF pair. Compare this method to **writeLine**, which does append a CR/LF pair.

▪

writeString example

The following example assigns strings to the variables *lo* and *hi*, then uses **writeString** to write them to an open TextStream.

```
; goodAdvice::pushButton
method pushButton(var eventInfo Event)
var
   myText TextStream
   lo, hi String
endVar
lo = "Buy low. "
hi = "Sell high."
myText.open(":PRIV:advice.txt", "nw")          ; open a new file
myText.writeString(lo, hi)
msgInfo("File size:", string(myText.size()))  ; displays 19
; Buy low.  = 9 characters, Sell High. = 10 characters. 10 + 9 = 19.
myText.close()
endMethod
```

Time type

Time variables store times in the form hour-minute-second-millisecond. You can use the following characters as separators: blank, tab, space, comma (,), hyphen (-), slash (/), period (.), colon (:), and semicolon (;).

Time values must be cast (explicitly declared). For example, the following statements assign to the Time variable *ti* a time of 10 minutes and 40 seconds past eleven o'clock in the morning:

```
var ti Time endVar
ti = Time("11:10:40 am")
```

The quotes around the value are required. Whether a time is valid depends on the current Paradox time format. For example, if the current time format is set to 12-hour format (such as hh:mm:ss), methods in the Time type consider hh:mm:ss a valid time format. Use **formatSetTimeDefault** procedure defined for the System type to set Paradox time formats with ObjectPAL.

The Time type includes several derived methods from the DateTime and AnyType types.

Methods for the Time type

AnyType	DateTime	**Time**
blank	hour	**time**
dataType	milliSec	
isAssigned	minute	
isBlank	second	
isFixedType		
view		

time procedure

See also   Example1   Example2   Time Type
Beginner

Casts a value as a time, or returns the current time.

Syntax

```
time ( [ const value AnyType ] ) Time
```

Description

time casts (converts) *value* as a time, or returns the current time according to the system clock. *value*, if given, must match the current Paradox time format. For more information, refer to the System type procedure **formatSetTimeDefault.**

▪

time example 1

The following example calls **time** to convert a string value to a time value:

```
var
   st String
   ti Time
endVar

st = "12:21:33 am"
ti = time(st)
```

time example 2

The following example displays the current time in a dialog box. The display format varies according to the user's current time format. This code is attached to a button's **pushButton** method.

```
; timeButton::pushButton
method pushButton(var eventInfo Event)

  ; displays the current time in a dialog box
   msgInfo("Current Time", time())

endMethod
```

UIObject type

Changes

UIObjects (the UI stands for user interface) create the user interface for an application: anything you can place in a form or report is a UIObject. Only UIObjects in forms have built-in event methods. The different UIObjects are band, bitmap, box, button, cell, chart, crosstab, ellipse, field object, form, group, line, list, multi-record object, OLE object, page, record object, table frame, and text box.

Note: The form behaves like a UIObject: a form has built-in event methods, you can attach code to those built-in event methods, and a form responds to events. There is also a separate type, Form, for methods and procedures that work only with forms.

You can also use built-in object variables to refer to UIObjects. This technique can be useful for creating generalized code.

Many UIObject methods duplicate TCursor methods. The UIObject methods that work with tables work on the underlying table through the visible object. Actions directed to the UIObject that affect the table are immediately visible in the object the table is bound to. TCursor methods, by contrast, work with a table behind the scenes; actions that affect the table are not necessarily visible in any object, even if the TCursor is acting on the same table to which a visible object is bound.

Note: Some table operations require Paradox to create temporary tables. Paradox creates these tables in the private directory.

Some table operations are considerably faster with TCursors than with UIObjects. For instance, if you need to perform a table-oriented operation that will cause a high volume of screen refreshes which are time-consuming

you can use this technique: declare a TCursor, attach it to the object the table is already bound to (such as a table frame), do the operation with the TCursor, then resynchronize the display object to the TCursor. When you attach a TCursor to an object bound to a table, the TCursor's record pointer is set to the current record for the object. After you perform a TCursor operation, such as a **locate**, the TCursor might point to a different record than the object. To make the object point to the same record as the TCursor, use the **resync** method; to make the TCursor point to the same record as the object, use the **attach** method. See the example for **insertRecord.**

Methods for the UIObject type

UIObject
action
atFirst
atLast
attach
bringToFront
broadcastAction
cancelEdit
convertPointWithRespectTo
copyFromArray
copyToArray
copyToToolbar
create
currRecord
delete

deleteRecord
dropGenFilter
edit
empty
end
endEdit
enumFieldNames
enumLocks
enumObjectNames
enumSource
enumSourceToFile
enumUIClasses
enumUIObjectNames
enumUIObjectProperties
execMethod
forceRefresh
getBoundingBox
getGenFilter
getPosition
getProperty
getPropertyAsString
getRange
getRGB
hasMouse
home
insertAfterRecord
insertBeforeRecord
insertRecord
isContainerValid
isEdit
isEmpty
isLastMouseClickedValid
isLastMouseRightClickedValid
isRecordDeleted
keyChar
keyPhysical
killTimer
locate
locateNext
locateNextPattern

locatePattern
locatePrior
locatePriorPattern
lockRecord
lockStatus
menuAction
mouseClick
mouseDouble
mouseDown
mouseEnter
mouseExit
mouseMove
mouseRightDouble
mouseRightDown
mouseRightUp
mouseUp
moveTo
moveToRecNo
moveToRecord
nextRecord
nFields
nKeyFields
nRecords
pixelsToTwips
postAction
postRecord
priorRecord
pushButton
recordStatus
resync
rgb
sendToBack
setGenFilter
setPosition
setProperty
setRange
setTimer
skip
switchIndex
twipsToPixels

unDeleteRecord
unlockRecord
view
wasLastClicked
wasLastRightClicked

Changes to UIObject type methods

The following table lists new methods and methods that were changed for version 7.

New	Changed
None	create

The following table lists new methods and methods that were changed for version 5.0.

New	Changed
bringToFront	enumUIObjectProperties
copyToToolbar	**setFilter** was replaced by **setRange** which offers enhanced functionality and performance. Code that calls **setFilter** will continue to execute as before.
dropGenFilter	
getGenFilter	
getRange	
sendToBack	
setGenFilter	
setRange	

■

action method

See also Example UIObject Type
Beginner

Performs a specified action.

Syntax

```
action ( const actionId SmallInt ) Logical
```

Description

action specifies an *actionId* to perform in response to an event, where *actionId* is a constant in one of the following action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use **action** to send a user-defined action constant to a built-in **action** method. User-defined action constants are simply integers that don't interfere with any of ObjectPAL's constants. You can use them to signal other parts of an application.

This **action** method is distinct from the built-in **action** method for a form or for any other UIObject. The built-in **action** method for an object responds to an action event; this method causes an ActionEvent.

action example

The code in the following example is attached to a button's **mouseUp** method and does the following: if you press and hold Shift and click the button, the pointer moves to the next set of records. If you click the button without pressing Shift, the pointer moves to the next record.

The action constants DataFastForward and DataNextRecord behave like the Fast Forward and Next Record Toolbar buttons. Assume that *CUSTOMER* refers to a table frame on the form and that *nextRecordOrFast* is a button on the same form. The *nextRecordOrFast* button is not in the same containership hierarchy as *CUSTOMER*, so the action won't bubble up to *CUSTOMER* automatically. Thus, the action must be sent to the *CUSTOMER* object explicitly.

```
; nextRecordOrFast::mouseUp
method mouseUp(var eventInfo MouseEvent)
; if the tableFrame isn't active, then move to it
if NOT CUSTOMER.focus then
  CUSTOMER.Name.moveTo()
endIf
; if Shift key is down, go to next set of records,
;  otherwise go to next record
if eventInfo.isShiftKeyDown() then
   CUSTOMER.action(DataFastForward)
else
   CUSTOMER.action(DataNextRecord)
endIf
endMethod
```

■

atFirst method

See also		Example		UIObject Type

Reports if the pointer is at the first record of a table.

Syntax

```
atFirst ( ) Logical
```

Description

atFirst returns True if the pointer is at the first record of a table; otherwise, it returns False. **atFirst** respects the limits of restricted views displayed in a linked table frame or multi-record object.

▪

atFirst example

In the following example, assume that *CUSTOMER* refers to a table frame on the form and *goToFirstButton* is a button on the same form. The method checks the pointer position. If the pointer is not on the first record of *CUSTOMER*, the method moves it to that record.

```
; goToFirstButton::pushButton
method pushButton(var eventInfo Event)
if NOT CUSTOMER.atFirst() then
  CUSTOMER.home()
  ; this has the same effect as:  CUSTOMER.action(DataBegin)
endIf
endMethod
```

▪

atLast method

See also Example UIObject Type

Reports if the pointer is at the last record in a table.

Syntax

```
atLast ( ) Logical
```

Description

atLast returns True if the pointer is at the last record of a table; otherwise, it returns False. **atLast** respects the limits of restricted views displayed in a linked table frame or multi-record object.

▪

atLast example

In the following example, assume that *CUSTOMER* refers to a table frame on the form and *goToLastButton* is a button on the same form. The method checks the pointer position. If the pointer is not on the last record of *CUSTOMER*, the method moves it to that record.

```
; goToLastButton::pushButton
method pushButton(var eventInfo Event)
if NOT CUSTOMER.atLast() then
  CUSTOMER.end()
  ;this has the same effect as:  CUSTOMER.action(DataEnd)
endIf
endMethod
```

▪

attach method

See also Example1 Example2 UIObject Type

Binds a UIObject variable to a specified design object.

Syntax

```
1. attach ( ) Logical
2. attach ( const objectVar UIObject ) Logical
3. attach ( const objectName String ) Logical
4. attach ( const form Form [ , objectName String ] ) Logical
5. attach ( const report Report [ , objectName String ] ) Logical
```

Description

attach binds a UIObject variable to a specified design object. You can also use **attach** to assign a UIObject to an item in an Array.

Syntax 1 binds the variable to the object that called **attach**. In other words, it binds the variable to *self*.

Syntax 2 binds the variable to another UIObject specified by a UIObject variable *objectVar* in one of the following ways:

Specification	Example

UIObject variable	`var`
	`  u1, u2 UIObject`
	`endVar`
	`u1.attach()    ; Attach to self.`
	`u2.attach(u1) ; Attach to a UIObject variable.`

UIObject name	`var`
	`  u1 UIObject`
	`endVar`
	`; Attach to an object named nameFld.`
	`u1.attach(nameFld)`

Containership path	`var`
	`  u1       UIObject`
	`  aForm Form`
	`endVar`
	`aForm.open("aform.fsl")`
	`; Attach to an object named aField.`
	`u1.attach(aForm.aPage.aField)`

Syntax 3 binds the variable to another UIObject specified by name in *objectName*. For example, if a form contains a box named *theFrame*, the following statement binds the UIObject variable *ui* to the box:

```
ui.attach("theFrame")
```

Syntax 4 binds the variable to the form specified by the Form variable *form*, or to a UIObject in that form specified by *objectName*.

Syntax 5 binds the variable to the report specified by the Report variable *report*, or to a UIObject specified by *objectName*.

Note: Some of the methods in the UIObject class can be used for forms, but only if you attach a UIObject variable to the form. Syntax 4 of the **attach** method lets you attach a UIObject variable to a form so that you can access those methods. For instance, to send a mouseUp event to another form's form-level **mouseUp** built-in event method, you need to attach a UIObject variable (a Form variable won't work) to an open form.

▪

attach example 1

The following example shows various forms of the syntax. First, the method attaches the variable *objBox* to the current object (self), then changes its color. Next, the method attaches *objBox* to another object, then changes that object's color via *objBox*. A second example for of the same syntax opens another form, attaches *objBox* to a box on the second form, and changes the color of the other form's object via *objBox*.

Notice that you can attach to an object name on another form by including the form handle (previously obtained) in the object name. You can supply the handle to the form in the first argument; the second argument supplies the object name on the specified form as a string.

In this example, assume the current form contains two boxes, *thisBox* and *thatBox*. The method is attached to *thisBox*. The secondary form contains one box, called *otherBox*.

```
; thisBox::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  objBox,
  objForm    UIObject
  otherForm  Form
endVar

objBox.attach()           ; binds objBox to thisBox
objBox.color = DarkMagenta

objBox.attach(thatBox)    ; binds objBox to thatBox
objBox.color = Magenta

; assume the form uiattch2.fsl exists and it has
; one object called otherBox
if otherForm.open("uiattch2.fsl") then
  objBox.attach(otherForm.otherBox)
  objBox.color = DarkBlue
  sleep(2000)
  otherForm.close()
endIf

if otherForm.open("uiattch2.fsl") then
  ; notice that the object name is given as a string
  objBox.attach(otherForm, "otherBox")
  objBox.color = LightBlue
  sleep(2000)
  otherForm.close()
endIf

endMethod
```

attach example 2

The following example shows how to use **attach** to assign a UIObject to an item in an array.

```
method pushButton(var eventInfo Event)
   const
      kOneInch = 1440 ; One inch = 1,440 twips.
      kShowHandles = Yes
   endConst

   var
      foForm      Form
      uiTempObj   UIObject
      arObjects   Array[2] UIObject
   endVar

   foForm.create()

   uiTempObj.create(BoxTool, 700, 700, kOneInch, kOneInch, foForm)
   arObjects[1].attach(uiTempObj)

   uiTempObj.create(BoxTool, 700, 2500, kOneInch, kOneInch, foForm)
   arObjects[2].attach(uiTempObj)

   foForm.setSelectedObjects(arObjects, kShowHandles)

endMethod
```

▪

bringToFront method

See also Example UIObject Type

Displays an object in front of other objects.

Syntax

```
bringToFront ( )
```

Description

bringToFront moves a UIObject to the front drawing layer of a window, displaying it on top of other objects. (If you're using a form as a UIObject, this method displays the form window in front of other windows.)

This method works in design mode and run mode; you do not have to select the object. Paradox moves the object in front, so it appears to be on top of other objects. This might not be noticeable unless the objects partially overlap each other. You might want to bring an object to the front of the stack of objects if

- You have objects that overlap each other
- You want to rearrange the tab order

Note: When you change the front-to-back positions of objects, you change their tab order, because objects always tab from back to front.

bringToFront example

In the following example, the **pushButton** method for a button displays a sequence of twelve bitmaps to create animation. It uses two **for** loops along with **bringToFront** to first cycle through the bitmaps; then to cycle through the bitmaps again in reverse order.

```
;btn1 :: pushButton
method pushButton(var eventInfo Event)
   var
      siCounter SmallInt
   endVar

   ;Cycle through bitmaps.
   for siCounter from 1 to 12
      ; Assume the bitmap objects have names like bmp1, bmp2, etc.
      pge1.("bmp" + string(siCounter)).bringToFront()
      sleep(100)
   endFor

   ;Cycle through bitmaps in reverse order.
   for siCounter from 11 to 1 step -1
      pge1.("bmp" + string(siCounter)).bringToFront()
      sleep(100)
   endFor
endMethod
```

▪

broadcastAction method

See also Example UIObject Type

Broadcasts an action to an object and the objects it contains.

Syntax

```
broadcastAction (const actionID SmallInt)
```

Description

broadcastAction sends the ActionEvent specified in *actionID* to an object, and then sequentially to each object it contains, with each contained object in turn as the target. The action is sent depth-first through the containership hierarchy, not breadth-first. By default, contained objects bubble the action up through the hierarchy.

For example, suppose a page named *thePage* contains two boxes, *boxOne* and *boxTwo*, and *boxOne* contains a button *btnOne*. A call to **thePage.broadcastAction(actionID)** would send the action represented by *actionID* to the objects in the following order:

1. thePage (specified by dot notation)
2. boxOne (contained by thePage)
3. btnOne (contained by boxOne, at a lower level in the hierarchy)
4. boxTwo (also contained by thePage, at the same level as boxOne in the hierarchy)

The value of *actionID* can be a user-defined action constant or a constant from one of the following Action classes:

ActionDataCommands

ActionEditCommands

ActionFieldCommands

ActionMoveCommands

ActionSelectCommands

broadcastAction example

In the following example, the form's built-in **action** method uses **broadcastAction** to send all the objects in the page *pge1* a user-defined action. When the form goes into edit mode, it sends *UserAction + 1*. When the form ends edit mode, it sends *UserAction + 2*. Every field's label then uses the user-defined action to turn every label Red or Black.

The following code is attached to the form's built-in **action** method.

```
;frm1 :: action
method action(var eventInfo ActionEvent)

   if eventInfo.isPreFilter() then
      ;// This code executes for each object on the form:

   else
      ;// This code executes only for the form:

      switch
         case eventInfo.id() = DataBeginEdit :
               pge1.broadCastAction(UserAction + 1)

         case eventInfo.id() = DataEndEdit :
               pge1.broadCastAction(UserAction + 2)
      endSwitch

   endIf

endmethod
```

The following code is attached to each label's built-in **action** method.

```
;label :: action
method action(var eventInfo ActionEvent)
   ;Duplicate this code on every object (or create a prototype object)
   ;you wish toggle the font color from black to red when
   ;the form goes in and out of edit mode.

switch
   case eventInfo.id() = UserAction + 1 : self.font.color = Red
   case eventInfo.id() = UserAction + 2 : self.font.color = Black
endSwitch
endmethod
```

cancelEdit method

See also   Example   UIObject Type
Beginner

Cancels record changes without ending Edit mode.

Syntax

```
cancelEdit ( ) Logical
```

Description

cancelEdit leaves a table in Edit mode but cancels changes to the current record. It returns True if successful; otherwise, it returns False. To abort changes to the current record, you must use **cancelEdit** before moving the pointer from the current record; once you move the pointer, changes to the record are committed.

cancelEdit has the same effect as the action constant DataCancelEdit, so the following statements are equivalent:

```
obj.cancelEdit()
obj.action(DataCancelEdit)
```

▪

cancelEdit example

The following method attaches a UIObject variable, *noChange*, to a table frame, *CUSTOMER*. (From then on *noChange* is used as a handle to the table frame.) The method searches for a value in the *Customer* table, and, if found, changes the value. Before leaving the record, the change is canceled with the **cancelEdit** method. In this example, assume that you have one page on the form, called *pageOne*; a table frame attached to the *Customer* table; and a button named *CancelEditButton*.

```
; CancelEditButton::pushButton
method pushButton(var eventInfo Event)
var
  noChange UIObject
endVar

noChange.attach()
noChange.attach(pageOne.CUSTOMER)
noChange.edit()
if noChange.locate("Name", "Unisco") then
  noChange."Name" = "Jones"   ; prepare to change the record
  msgInfo("noChange.'Name'", noChange."Name".value)
  noChange.cancelEdit()        ; belay that order!
                           ; record not changed,
endIf
noChange.endEdit()         ; exit Edit mode

endMethod
```

▪

convertPointWithRespectTo method

See also   Example   UIObject Type

Changes the frame of reference for calculating the coordinates of a point.

Syntax

`convertPointWithRespectTo ( ``const `***`otherUIObject`***` UIObject, const `***`oldPoint`***` Point, var `***`convertedPoint`***` Point `**`)`**

Description

convertPointWithRespectTo changes the frame of reference for calculating the position of a point. Normally, coordinates are calculated relative to the upper left corner of the object's container (or the container's frame, in the case of an ellipse). This method instead calculates a point's position relative to the upper left corner of the object specified in *otherUIObject*.

convertPointWithRespectTo example

The following example gets and shows the position of an object called *innerBox*. *innerBox* is contained by *outerBox*, and is on a page called *pageOne*. First, the position of *outerBox* relative to the upper left corner of the page is obtained and displayed. Next, the position of *innerBox* is taken, relative to the upper left corner of *outerBox*. Finally, the position of *innerBox* is converted with respect to the page, so you can see how far *innerBox* is from the top and left edges of the page.

```
; alignInnerBox::pushButton
method pushButton(var eventInfo Event)
var
  innerPos,
  outerPos,
  convertedPos  Point
  x, y, w, h    LongInt
endVar

outerBox.getPosition(x, y, w, h)
outerPos = point(x, y) ; convert x and y from
outerPos.view("Outer box position") ; outerBox to a point
innerBox.getPosition(x, y, w, h)
innerPos = point(x, y)
innerPos.view("Inner box position unconverted")
; how far is innerPos from the upper left corner of the page?
outerBox.convertPointWithRespectTo(pageOne, innerPos, convertedPos)
convertedPos.view("Inner box position converted")
endMethod
```

▪

copyFromArray method

See also Example UIObject Type

Copies data from an array to a record of a table.

Syntax

`copyFromArray ( const ar Array[ ] AnyType) Logical`

Description

copyFromArray copies data from an array *ar* to a UIObject (typically a table frame or multi-record object). The first element of the array is copied to the first field of the table, the second element to the second field, and so on until the array is exhausted or the record is full.

The method fails if an attempt is made to copy an unassigned array element or if the structures do not match. (This can never happen if the array was created by **copyToArray**, which assigns a blank value if a field is blank.) In addition, the method fails if the form is not in Edit mode. If there are more elements in the array than fields in the record, the extra elements are ignored.

▪

copyFromArray example

In the following example, suppose a form contains a table frame named *CUSTNAME*. The *CUSTNAME* table has three fields: Last name, A20; First name, A20; and Middle Initial, A1. This method starts editing *CUSTNAME*, creates an array with three elements, creates a new record in *CUSTNAME*, then copies data from the array to the record.

```
; createRecord::pushButton
method pushButton(var eventInfo Event)
var
  nameArray Array[3] String
endvar
CUSTNAME.edit()             ; start Edit mode
nameArray[1] = "Hall"       ; fill the array with the record to insert
nameArray[2] = "Robert"
nameArray[3] = "A"
CUSTNAME.action(DataInsertRecord) ; insert a blank record first
CUSTNAME.copyFromArray(nameArray) ; copy the array to the new record
CUSTNAME.endEdit()
endMethod
```

▪

copyToArray method

See also Example UIObject Type

Copies data from a record to an array.

Syntax

`copyToArray ( var ar Array [ ] AnyType ) Logical`

Description

copyToArray copies the fields of the current record of a UIObject (typically a table frame or a multi-record object) to the elements of an array specified in *ar*. You must declare the array to be of type AnyType, or of a type that matches every field in the table. If the array is resizeable, it grows automatically to hold the number of fields in the record. If the array is not resizeable, it holds as many fields as it can, and the rest are discarded.

The value of the first field is copied to the first element of the array, the value of the second field to the second element, and so on. The size of the array is equal to the number of fields in the record. The record number field and any display-only or calculated fields are not copied to the array.

copyToArray example

The following example assumes that there are two table frames on a form, *CUSTOMER* and *CUSTARC*, and one button, named *archiveButton*. The form itself is renamed *thisForm*. When *archiveButton* is pushed, the current record in *CUSTOMER* is moved to *CUSTARC*.

First, the method looks at the Editing property of the form; if it's False, the method starts Edit mode. The method then copies the current record in *CUSTOMER* to the *arcRecord* array and attempts to delete the current record. If the current record can't be locked and deleted, the record is not copied to the target table *CUSTARC*. If the record delete is successful, the method adds a new blank record to the target table, and writes the contents of the array to the record.

```
; archiveButton::pushButton
method pushButton(var eventInfo Event)
var
  arcRecord Array[] String
endVar

; check to see if form is in edit mode
if thisForm.Editing = False then ; if not, then start
  CUSTOMER.action(DataBeginEdit)
endIf

; move the current record from CUSTOMER to archive in CUSTARC
CUSTOMER.copyToArray(arcRecord)
arcRecord.view()           ; take a look at the array
; if the record can't be locked, it won't be deleted
if CUSTOMER.deleteRecord() = True then
  ; if it is deleted, then copy it to the archive table
  CUSTARC.insertRecord()            ; insert blank record
  CUSTARC.copyFromArray(arcRecord)  ; copy array to blank record
endIf

endMethod
```

■

copyToToolbar method

See also Example UIObject Type

Copies an object to the Toolbar where it can be used as a prototype object.

Syntax

```
copyToToolbar ( ) Logical
```

Description

copyToToolbar copies an object (including all its properties and methods) to the Toolbar. Objects subsequently created with the corresponding Toolbar tool will have the new properties. Objects that existed in the form previously do not change.

For example, suppose you create a box (interactively or using ObjectPAL), then set its color to red and add code to its built-in **mouseClick** method. When you copy this box to the Toolbar, all subsequent boxes you create will be red and have the same code attached to the **mouseClick** method.

copyToToolbar copies all component objects in a compound object. For example, when you copy a labeled field object, you copy the field object, the label, and the edit region. Tables come with headers, labels, records, and fields. Multirecord objects come with records (but not fields). Crosstabs come with cells and fields, and can distinguish the three different cell types, so you can have three different types of fields in them, different colors in them, and so on.

You can also copy the component objects separately. For example, if you only want to copy an edit region, you can do that. If an object contains objects, but is not a compound object, the contained objects are not copied.

Changes you make using **copyToToolbar** last only for the current Paradox session. To save the tool's new properties to the next session, call **saveStyleSheet.**

If an object does not have a corresponding tool on the Toolbar, Paradox copies its properties and methods to a "hidden" tool, and subsequent objects of that type will have those properties and methods. For example, the Toolbar has no tool for creating a page. However, you can set a page's properties and methods, and then call **copyToToolbar**. Pages created subsequently will have those same properties and methods.

▪

copyToToolbar example

In this example, a button named *btnCreateStyleSheet* uses **enumObjectNames** to fill an array *arObjNames*. A **for** loop cycles through the array and copies the object to the Toolbar using **copyToToolbar**. Finally, a call to **saveStyleSheet** creates (or overwrites) a style sheet with the name in the String variable *stSheet*. Paste the following code into the **pushButton** method for a button on any form you want to use as a style sheet.

```
;btnCreateStyleSheet :: pushButton
method pushButton(var eventInfo Event)
   var
      f                   Form
      stSheet       String
      arObjNames    Array[] Anytype
      siCounter    SmallInt
   endVar

   f.attach()               ; Attach to this form.
   f.enumObjectNames(arObjNames)   ; Fill array with object names.

   ; Prompt user for name of new style sheet.
   stSheet = "Style sheet name"
   stSheet.view("Enter name of style sheet")

   if stSheet = "Style sheet name" then   ; If variable was not changed,
      return                  ; quit the operation.
   endIf

   for siCounter from 1 to arObjNames.size()   ; Cycle through array
      copyToToolbar(f.(arObjNames[siCounter]))   ; and copy objects to
   endFor                            ; the Toolbar.

   if not f.saveStyleSheet(stSheet, True) then
      errorShow("Error saving style sheet", "Check path & filename")
   endIf
endMethod
```

▪

create method

See also   Example   UIObject Type

Creates an object.

Syntax

1. create (const **objectType** SmallInt, const **x** LongInt, const **y** LongInt, const **w** LongInt, const **h** LongInt [, const **container** UIObject] **)**

2. create (const **nativeObject** Binary, const **container** UIObject **)** Logical

Description

create creates the object specified in **objectType** (use one of the UIObjectTypes constants) at a position specified in **x** and **y**, with a width specified in **w**, and a height specified in **h** (**x**, **y**, **w**, and **h** are assumed to be in twips). The optional argument container specifies a container object for the object you are creating.

Using syntax 2, **create** creates the object specified by **nativeObject**. **nativeObject** is a binary object that can be generated by pasting a UIObject ("Borland Form Object") from the Clipboard. **container** specifies the container object for the object you are creating. **create** works only in form design mode. **create** returns True if successful and False if unsuccessful.

Note: When you use **create** to create an object, the object is invisible. To make it visible, set its Visible property to True. Objects can also be deleted at run time with the **delete** method.

▪

create example

The following code is attached to the **mouseUp** method for *pageOne* on a form. This example creates a box, names it *Fred*, colors it blue, and sets it visible. An ellipse is then created with a size position in *Fred*, and its container is set to *Fred*.

```
; pageOne::mouseUp
const
   kOneInch = 1440 ; One inch = 1,440 twips.
endConst
method mouseUp(var eventInfo MouseEvent)
var
  ui UIObject
endvar

; create a Blue box, called Fred, and make it visible
ui.create(BoxTool, 144, 144, 2 * kOneInch, 2 * kOneInch)
ui.Name = "Fred"
ui.Color = Blue
ui.Visible = True
; create a Green ellipse inside Fred, called Bill
ui.create (EllipseTool, 288, 288, kOneInch, kOneInch, self.Fred)
ui.Name = "Bill"
ui.Color = Green
ui.Visible = True
endMethod
```

currRecord method

See also Example UIObject Type

Reads the current record into the record buffer.

Syntax

```
currRecord ( ) Logical
```

Description

currRecord cancels changes to the current record, refreshing the current record from saved data. Any changes to the record are not committed. **currRecord** leaves a locked record locked. It returns True if successful; otherwise, it returns False.

currRecord has the same effect as the action constant DataRefresh, so the following statements are equivalent:

```
obj.currRecord()
obj.action(DataRefresh)
```

currRecord example

In the following example, assume that a form contains a table frame bound to *Orders*.

```
;refreshRecord::pushButton
method pushButton(var eventInfo Event)
ORDERS.edit()             ; start edit
ORDERS.Amount_Paid = 321.45   ; make a change
message("Watch closely now.")
sleep(2000)
ORDERS.currRecord()       ; refreshes record from disk,
                          ; any changes are lost, record
                          ; is not locked
if ORDERS.recordStatus("Locked") then
  msgInfo("FYI", "The record is still locked.")
endIf
endMethod
```

▪

delete method

See also Example UIObject Type

Deletes an object from a form.

Syntax

```
delete ( )
```

Description

delete deletes an object from a form at run time.

▪

delete example

In the following example, assume that a form contains a method that creates a box named *Fred* and a ellipse inside *Fred* called *Bill*. These objects are created at run time and thus can't be referenced directly by this method, because they don't exist yet. The technique used here attaches to the object using a string evaluated at run time. See the example for **create** for details about the **mouseUp** method (on the same form) that creates the objects to be deleted.

```
; pageOne::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  ui   UIObject
endVar

;  Fred and Bill are objects created by the mouseUp method
;  for pageOne of this form. Because they are created at
;  run time, you can't directly refer to them as objects in
;  code. Consequently, attach is used to attach the ui var
;  to the string "Fred.Bill", which is evaluated at run time.
;  As long as mouseUp is called before mouseRightUp, those
;  objects will exist.
if ui.attach("Fred.Bill") then
  ui.delete()
  ui.attach("Fred")
  ui.delete()
  {This would do the same thing as previous four lines,
    because Fred contains Bill at run time:
    ui.attach("Fred")
    ui.delete()
  }
endIf
endMethod
```

▪

deleteRecord method

See also Example UIObject Type
Beginner

Deletes the current record from the table.

Syntax
```
deleteRecord ( ) Logical
```

Description

deleteRecord deletes the current record of a table without prompting for confirmation. It returns True if successful; otherwise, it returns False. This operation cannot be undone, for Paradox tables, but it can be undone for dBASE tables.

deleteRecord has the same effect as the action constant DataDeleteRecord, so the following statements are equivalent:

```
obj.deleteRecord()
obj.action(DataDeleteRecord)
```

▪

deleteRecord example

The following example assumes that there are two table frames on a form, *CUSTOMER* and *CUSTARC*, and one button, named *archiveButton*. The form is renamed *thisForm*. When *archiveButton* is pushed, the current record in *CUSTOMER* is moved to *CUSTARC*.

First, the method looks at the Editing property of the form; if Editing is False, the method starts Edit mode. The method then copies the current record in *CUSTOMER* to the *arcRecord* array and attempts to delete the current record. If the current record can't be locked and deleted, the record is not copied to the target table *CUSTARC*. If the record delete is successful, the method adds a new blank record to the target table, and writes the contents of the array to the record.

```
; archiveButton::pushButton
method pushButton(var eventInfo Event)
var
  arcRecord Array[] String
endVar

; check to see if form is in edit mode
if thisForm.editing = False then ; if not, then start
  CUSTOMER.action(DataBeginEdit)
endIf

; move the current record from CUSTOMER to archive in CUSTARC
CUSTOMER.copyToArray(arcRecord)
arcRecord.view()           ; take a look at the array
; if the record can't be locked, it won't be deleted
if CUSTOMER.deleteRecord() = True then
  ; if it is deleted, then copy it to the archive table
  CUSTARC.insertRecord()
  CUSTARC.copyFromArray(arcRecord)
endIf

endMethod
```

▪

dropGenFilter method

See also Example UIObject Type

Drops (removes) the filter criteria associated with a field, multirecord object, or table frame.

Syntax

`dropGenFilter ( ) Logical`

Description

dropGenFilter drops (removes) the filter criteria associated with a UIObject, leaving it unfiltered. Indexes and ranges (if any) remain in effect.

dropGenFilter example

In a complex form, it is sometimes convenient to include a button that removes all the filter criteria from the form. In the following example, a form's data model contains the *Orders* and *Lineitem* tables linked 1:M. The form also contains a button called *btnDropFilters*. The **pushButton** method for *btnDropFilters* uses **dropGenFilter** on one UIObject connected to each table in the data model.

```
;btnDropFilters :: pushButton
method pushButton(var eventInfo Event)
   ; Order_No is a field object bound to
   ; the Order No field in the Orders table.
   Order_No.dropGenFilter()

   ; LINEITEM is a table frame bound to the Lineitem table.
   LINEITEM.dropGenFilter()
endMethod
```

▪

edit method

See also		Example		UIObject Type
Beginner

Puts a table into Edit mode.

Syntax
edit ( ) Logical

Description

edit puts all tables on a form into Edit mode so changes can be made. If a form is already in Edit mode, an unnecessary edit does not cause an error, and does not toggle out of Edit mode.

In Edit mode, record changes are posted when the focus moves off the record, when the table receives a DataPostRecord or DataUnlockRecord action, or when **endEdit** is executed. Use **cancelEdit** to cancel changes to the record before departing from the record.

edit has the same effect as the action constant DataBeginEdit, so the following statements are equivalent:

```
obj.edit()
obj.action(DataBeginEdit)
```

edit example

In the following example, assume that a form contains a table frame bound to the *Orders* table, and one button, named *changeDate*. The **pushButton** method for *changeDate* checks the Sale Date and Ship Date fields of the current record, and updates Sale Date if Ship Date is less than Sale Date. Once the transaction is complete, **endEdit** posts the record and ends Edit mode.

```
; changeDate::pushButton
method pushButton(var eventInfo Event)

; first, see if you want to change Ship Date
if ORDERS."Sale Date".value > ORDERS."Ship Date".value then
  ; start Edit mode for the form
  ORDERS.edit()
  ; if Sale Date is later than Ship Date, change Ship Date
  ORDERS."Ship Date".value = ORDERS."Sale Date".value + 5
  ORDERS.endEdit()       ; end editing▪changes to the record
                         ; can't be canceled
endIf

endMethod
```

▪

empty method

See also Example UIObject Type

Deletes all records from a table.

Syntax

`empty ( ) Logical`

Description

empty deletes all records from a table without prompting for confirmation. The table does not have to be in Edit mode, but a write lock (at least) is required. This operation cannot be undone for Paradox tables.

empty removes information from the table, but does not delete the table itself. Compare this method to Table::**delete**, which does delete the table.

empty tries to place a write lock on the table. **empty** must delete each record one at a time. This can take a long time for large tables. For dBASE tables, this method flags all records as deleted, but does not remove them from the table. Records can be undeleted from a dBASE table using the **unDeleteRecord** method (unless they have been removed with the TCursor::**compact** method).

empty example

The following example assumes a form with three buttons: *createTable*, *emptyTable*, and *deleteTable*. *createTable* creates a copy of the *Orders* table called *TmpOrder*, then places a table frame on the form and binds *TmpOrder* to it. *emptyTable* deletes all the records from *TmpOrder*. *deleteTable* removes the table frame, removes the table from the data model of the form, and deletes the temporary table.

Following is the code for the *createTable* button:

```
; createTable::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  ui  UIObject
endVar

tbl.attach("Orders.db")
tbl.copy("TmpOrder.db")  ; Copy Orders to TmpOrder.

ui.create(TableFrameTool, 720, 720, 4320, 1440) ; Create a TableFrame.
ui.TableName = "TmpOrder.db" ; This also adds table to data model.
ui.Visible = True

endMethod
```

Following is the code for the *emptyTable* button:

```
; emptyTable::pushButton
method pushButton(var eventInfo Event)
var
  ui  UIObject
endVar

if ui.attach("TMPORDER") then
  if msgYesNoCancel("Empty",
    "Delete all records from this table?") = "Yes" then
    ui.empty()  ; Deletes all records from the TMPORDERS table.

  endIf
endIf

endMethod
```

Following is the code for the *deleteTable* button:

```
; deleteTable::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  ui  UIObject
endVar

; Clean up.
if ui.attach("TMPORDER") then
  ui.delete()                    ; Delete table frame.
  DMRemoveTable("TmpOrder.db")   ; Remove table from data model.
  tbl.attach("TmpOrder.db")
  tbl.delete()                   ; Delete table.
endIf
endMethod
```

▪

end method

See also Example UIObject Type
Beginner

Moves to the last record in a table.

Syntax

```
end ( ) Logical
```

Description

end sets the current record to the last record in a table.

end has the same effect as the action constant DataEnd, so the following statements are equivalent:

```
obj.end()
obj.action(DataEnd)
```

▪

end example

The following example moves to the last record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form; *moveToEnd* is a button on the same form.

```
; moveToEnd::pushButton
method pushButton(var eventInfo Event)
CUSTOMER.end()  ; move to the last record
                ; same as:  CUSTOMER.action(DataEnd)
msgInfo("At the last record?", CUSTOMER.atLast())
endMethod
```

▪

endEdit method

See also   Example   UIObject Type
Beginner

Leaves Edit mode, and accepts changes to the current record.

Syntax

`endEdit ( )` `Logical`

Description

endEdit takes a table out of Edit mode and posts changes to the current record.

endEdit has the same effect as the action constant DataEndEdit, so the following statements are equivalent:

```
obj.endEdit()
obj.action(DataEndEdit)
```

▪

endEdit example

See the example for **edit**.

▪

enumFieldNames method

See also Example UIObject Type

Fills an array with the names of fields in a table.

Syntax

`enumFieldNames ( var fieldArray Array[ ] String ) Logical`

Description

enumFieldNames fills *fieldArray* with the names of the fields in a table, where *fieldArray* is a resizeable array that you must declare and pass as an argument. If *fieldArray* already exists, this method overwrites it without asking for confirmation. This method returns True if it succeeds; otherwise, it returns False.

enumFieldNames example

The following example uses **enumFieldNames** to write the field names from the *Orders* table to an array named *fieldNames*. Assume that a form has a table frame bound to *Orders* and a button called *getFieldNames*.

```
; getFieldNames::pushButton
method pushButton(var eventInfo Event)
var
  fieldNames Array[] String
endVar
ORDERS.enumFieldNames(fieldNames)
fieldNames.view()
endMethod
```

▪

enumLocks method

See also Example UIObject Type

Creates a Paradox table listing the locks currently applied to a UIObject; returns number of locks.

Syntax

```
enumLocks ( const tableName String ) LongInt
```

Description

enumLocks creates the Paradox table specified in *tableName*. *tableName* lists the locks currently applied to the table object. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. For dBASE tables, this method lists only the lock you've placed (not all locks currently on the table).

You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of *tableName* is listed below:

Field name	Field type	Description
UserName	A15	User name.
LockType	A32	String describing type of lock, for example, Table Write Lock.
NetSession	N	Net level session number.
Session	N	BDE session number, if the lock was placed by BDE.
RecordNumber	N	Record number, if the lock is a record lock or an image lock; otherwise 0.

enumLocks example

In the following example, the built-in **pushButton** method for the *showLocks* button creates a table listing the locks currently applied to the *Customer* table.

```
; showLocks::pushButton
method pushButton(var eventInfo Event)
var
  obj       UIObject
  howMany   LongInt
  enumTable TableView
endVar
obj.attach(CUSTOMER)            ; table frame on form
lock("Customer", "Write")       ; put a write lock on Customer
howMany = obj.enumLocks("lockenum.db")  ; enumerate locks
message("There are ", howMany, " locks on Customer table.")
enumTable.open("lockenum.db")   ; show the resulting table
enumTable.wait()
enumTable.close()
endMethod
```

▪

enumObjectNames method/procedure

See also Example UIObject Type

Fills an array with the names of the objects in a form.

Syntax

`enumObjectNames ( var objectNames Array[ ] String )`

Description

enumObjectNames fills an array with object names, where *arrayName* is a resizeable array that you declare and pass as an argument. If *arrayName* already exists, this method overwrites it without asking for confirmation.

This method returns the names of bound objects and unbound objects, beginning with the object that called this method, and including paths to objects that object contains. So, to enumerate all objects in a form, make **enumObjectNames** start with the form. To enumerate all objects in a page, make it start with the page. To enumerate all objects in a box, make it start with the box.

To list object names to a table (instead of an array) use **enumUIObjectNames**.

enumObjectNames example

This example demonstrates the difference between **enumObjectNames** (which lists object names to an array) and **enumUIObjectNames** (which lists object names to a table). In this example, the **pushButton** method for *getObjectNames* writes all object names on the form to an array, then to a table.

```
; getObjectNames::pushButton
method pushButton(var eventInfo Event)
   var
      foThisForm   Form
      arObjNames   Array[] String
      stTbName   String
      tvObjNames   TableView
   endVar

   stTbName = "objTable.db"
   foThisForm.attach() ; Get a handle to the current form.

   foThisForm.enumObjectNames(arObjNames)
   arObjNames.view("Objects in this form:")

   foThisForm.enumUIObjectNames(stTbName)
   tvObjNames.open(stTbName)
endMethod
```

enumSource method

See also Example UIObject Type

Fills a table with the source code of the methods on a form.

Syntax

```
enumSource ( const tableName String, [ const recurse Logical ] ) Logical
```

Description

enumSource fills a table with the source code of the methods on a form. If *tableName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of the table is

Field name	Type	Description
Object	A128	Object name.
MethodName	A128	Method name.
Source	M64	ObjectPAL code.

If *recurse* is False, this method returns the method definitions for overridden methods on the current object only. To include the source code for overridden methods on objects contained by the current object, *recurse* should be True.

When *recurse* is True, **enumSource** returns the method definitions for overridden methods, beginning with the object that called this method, and including paths to objects that object contains. So, to enumerate the source for all objects in a form, make **enumSource** start with the form. To enumerate the source for all objects in a page, make it start with the page. To enumerate the source for all objects in a box, make it start with the box.

enumSource example

This example uses **enumSource** twice: first to get the source code for the entire form, then to get only the source code for a button named *btnCancel*.

```
; getObjectNames::pushButton
const
   kRecurse   = Yes
   kNoRecurse = No
endConst

method pushButton(var eventInfo Event)
   var
      foThisForm   Form
      stTbName   String
      tvSource   TableView
   endVar

   stTbName = "objSrc.db"
   foThisForm.attach() ; Get a handle to the current form.

   foThisForm.enumSource(stTbName, kRecurse)
   tvSource.open(stTbName)

   ; Suspend execution until you close the table view.
   tvSource.wait()

   btnCancel.enumSource(stTbName, kNoRecurse)
   tvSource.open(stTbName)
endMethod
```

▪

enumSourceToFile method

See also Example UIObject Type

Writes the source code for a form or an object to a text file.

Syntax

```
enumSourceToFile ( const fileName String [ , const recurse Logical ] )
Logical
```

Description

enumSourceToFile writes the source code of the methods on a form to a text file. If *fileName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *fileName*; if no alias or path is specified, Paradox creates *fileName* in the working directory (:WORK:).

If *recurse* is False, this method writes the method definitions for overridden methods on the current object only. To include the source code for overridden methods on objects contained by the current object, *recurse* should be True.

When *recurse* is True, **enumSourceToFile** writes the method definitions for overridden methods, beginning with the object that called this method, and including paths to objects that object contains. So, to enumerate the source for all objects in a form, make **enumSourceToFile** start with the form. To enumerate the source for all objects in a page, make it start with the page. To enumerate the source for all objects in a box, make it start with the box.

enumSourceToFile example

This example uses **enumSourceToFile** twice: first to get the source code for the entire form, then to get only the source code for a button named *btnCancel*.

```
; getObjectNames::pushButton
const
   kRecurse   = Yes
   kNoRecurse = No
endConst

method pushButton(var eventInfo Event)
   var
      foThisForm   Form
   endVar

   foThisForm.attach() ; Get a handle to the current form.
   foThisForm.enumSource("formSrc.txt", kRecurse)

   btnCancel.enumSource("btnSrc.txt", kNoRecurse)
endMethod
```

▪

enumUIClasses procedure

See also　　Example　　UIObject Type

Writes a list of UIObject classes to a table.

Syntax

```
enumUIClasses ( const tableName String ) Logical
```

Description

enumUIClasses creates a table *tableName* containing a list of all UIObject classes (such as bitmap, box, and field) and the names of associated properties. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The table structure is

Field name	Type	Description
ClassName	A32	Name of object class.
PropertyName	A64	Name of property.

enumUIClasses example

The following example writes the types and properties to a table named *Tmpclass*.

```
; writeClasses::pushButton
method pushButton(var eventInfo Event)
enumUIClasses("TmpClass.db")
endMethod
```

■

enumUIObjectNames method/procedure

See also Example UIObject Type

Gets the names of each object in a form and writes them to a table.

Syntax

```
enumUIObjectNames ( const tableName String ) Logical
```

Description

enumUIObjectNames fills a table with object names. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of the table is

Field name	Type	Description
ObjectName	A128	Name of object.
ObjectClass	A32	Type of object (for example, button).

This method returns the names of bound objects and unbound objects, beginning with the object that called this method, and including paths to any objects that object contains. So, to enumerate all objects in a form, make **enumUIObjectNames** start with the form. To enumerate all objects in a page, make it start with the page. To enumerate all objects in a box, make it start with the box.

To list object names to an array (instead of a table), use **enumObjectNames.**

▪

enumUIObjectNames example

See the example for **enumObjectNames**.

▪

enumUIObjectProperties method/procedure

See also Example1 Example2 UIObject Type

Lists the properties of an object.

Syntax

```
1. enumUIObjectProperties ( const tableName String ) Logical
2. enumUIObjectProperties ( const properties DynArray[ ] String ) Logical
```

Description

enumUIObjectProperties lists the properties of an object to a table or a DynArray.

Syntax 1 writes the data to the Paradox table specified in *tableName*. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:). If the table already exists, Paradox overwrites it without asking for confirmation.

The table lists the properties of the specified object, and the properties of the objects it contains. The structure of the table is

Field name	Type	Description
ObjectName	A128	Name of the object
PropertyName	A64	Name of the property
PropertyType	A48	Data type of the corresponding property
PropertyValue	A255	Value of the corresponding property

In syntax 2 (added in version 5.0), the properties of the object (but *not* the properties of objects it contains) are written to a DynArray *properties*. The DynArray keys are the property names, and the items are the corresponding property values. You must declare the DynArray before calling this method.

enumUIObjectProperties example 1

For the following example, assume that *getProperties* is a button on a form designed to show fields from the *Customer* table. The **pushButton** method for *getProperties* uses **enumUIObjectProperties** to write all of the property values for each object on the form to the table *CstProps*.

```
; getProperties::pushButton
method pushButton(var eventInfo Event)
   enumUIObjectProperties("CstProps.db")
endMethod
```

enumUIObjectProperties example 2

For the following example, assume that *getProperties* is a button on a form. The **pushButton** method for *btnProperties* uses **enumUIObjectProperties** to write all of its property values to the dynamic array *dyn* and then display it.

```
; btnProperties::pushButton
method pushButton(var eventInfo Event)
   var
      dyn    DynArray[] String
   endVar

   self.enumUIObjectProperties(dyn)
   dyn.view("Properties of this button:")
endMethod
```

■

execMethod method/procedure

See also Example UIObject Type

Calls a custom method that takes no arguments.

Syntax

execMethod (const ***methodName*** String **)**

Description

execMethod calls the custom method indicated by the string *methodName*. The method named in *methodName* can take no arguments. **execMethod** allows you to call a method based on the contents of a variable, which means the compiler does not know the method to call until run time.

execMethod example

In the following example, assume that a form contains several fields, *fieldOne*, *fieldTwo*, and *fieldThree*. The form's Var window declares a dynamic array called *objPreProc*. The form's one custom method is called *fieldOnePreProc*. The form's **open** method (in the **isPreFilter=False** clause) creates elements in the *objPreProc* array: an element is created for each object on the form for which there is a preprocessing custom method.

In this example, *fieldOne* is assumed to require some preprocessing. An array element is created with an index of the object name "pageOne.fieldOne"; the value of the custom method name is "fieldOnePreProc". The **isPreFilter=True** clause of the form's open method called for each object on the form

sorts out if an array element in *objPreProc* corresponds to the current object; if so, the custom method for that object is called.

Following is the code for the custom method **fieldOnePreProc**:

```
; form design::fieldOnePreProc  (custom method)
; This method is called during the form's preFilter clause,
; when the current object is fieldOne.
method fieldOnePreProc()
fieldOne.color = "Red"    ; change the color of the field
fieldOne.Value = "Initialized by the form's open method"
endMethod
```

The following code goes in the form's Var window:

```
; Var window for the form
Var
  ObjPreProc DynArray[] String  ; indexed by object name, will
                                ; hold names of methods to execute
                                ; when isPreFilter is true
endVar
```

Following is the code for the form's **open** method:

```
method open(var eventInfo Event)
var
  targObj   UIObject   ; holds the target object
  targName  String     ; target object's name
  element   AnyType    ; index to dynamic array objPreProcs
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    eventInfo.getTarget(targObj)  ; identify the current target
    targName = targObj.name       ; get the name of the target
    forEach element in objPreProc ; iterate through array
      if element = targName then  ; is the target name there?
                                  ; if so, execute the corresponding
                                  ; custom method
        execMethod(objPreProc[targName])
      endIf
    endForEach
  else
    ; code here executes just for form itself

    ; assign elements to the objPreProc array to indicate
    ; objects for which there is a preprocess custom method
    objPreProc["fieldOne"] = "fieldOnePreProc"
endIf
endMethod
```

forceRefresh method

See also Example UIObject Type

Makes an object display the current data in the underlying table, and makes a calculated field recalculate.

Syntax

```
forceRefresh ( ) Logical
```

Description

forceRefresh initiates an action that says, in effect, "Throw out any buffers and regenerate your data from the table information. And do this recursively for all objects you contain." **forceRefresh** will also make a calculated field recalculate its value, and make a crosstab or chart re-evaluate its components.

Calling **active.forceRefresh( )** is exactly the same as calling **active.action(DataRecalc)** or pressing Shift + F9. It is a UIObject counterpart to the **forceRefresh** method defined for the TCursor type.

A call to **forceRefresh** affects the target object, objects contained by the target object, and objects bound to the same table as the target object. It does not affect objects in other windows. For example, calling **forceRefresh** in a form would not refresh data displayed in a table window. You refresh every object in a form by declaring a UIObject variable and calling attach to assign it a value; you can't use a variable declared as a Form variable.

forceRefresh behaves as follows:

- If a table frame or MRO is active when you call **forceRefresh**, only the underlying table refreshes. Child tables repaint, but they will not necessarily discard cached data.
- If a field object is active when you call **forceRefresh**, the table associated with that field refreshes, thereby repainting all fields dependent on it.
- You will not lose your current record position, provided the record still exists in the table.
- On a SQL server, a call to **forceRefresh** forces a read from the server. This is the only way to get a refresh from the server. **forceRefresh** only works on an SQL table if the table has a unique index.

forceRefresh example

The following example uses **forceRefresh** in code attached to a button's built-in **pushButton** method to let the user control when data is refreshed. This example assumes you have interactively chosen Database page from the Edit|Preferences tabbed dialog box and entered a large value (at least 3,600 seconds) in the Refresh rate dialog box. The code uses **forceRefresh** to refresh the Parts table each time the user clicks the button. Other tables bound to this form are refreshed only once in 3,600 seconds (one hour).

```
method pushButton(var eventInfo Event)
   Parts.forceRefresh()
endMethod
```

▪

getBoundingBox method

See also Example UIObject Type

Returns the coordinates of the frame that bounds an object.

Syntax

`getBoundingBox ( var topLeft Point, var bottomRight Point )`

Description

getBoundingBox returns the coordinates of the top left corner (*topLeft*) and the bottom right corner (*bottomRight*) of the invisible box (frame) that bounds an object, relative to the form. The bounding box is only visible in a design window. When you select an object in a design window, you can see its bounding box.

▪

getBoundingBox example

The following example draws a box around an ellipse based on the ellipse's bounding box. Assume that a form contains an ellipse called *redCircle*.

```
; redCircle::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  TopLeft,
  BotRight Point      ; to hold the points returned by getBoundingBox
  ui       UIObject   ; to create a new object
endVar

self.getBoundingBox(TopLeft, BotRight)
ui.create(BoxTool, TopLeft.x(),
                   TopLeft.y(),
                   BotRight.x() - TopLeft.x(),
                   BotRight.y() - TopLeft.y())
ui.Color = Green
ui.Translucent = Yes
ui.Visible = Yes

endMethod
```

▪

getGenFilter method

See also   Example1   Example2   UIObject Type

Retrieves the filter criteria associated with a field, table frame, or multirecord object.

Syntax

```
1. getGenFilter ( criteria DynArray[ ] AnyType ) Logical
2. getGenFilter ( criteria Array[ ] AnyType [ , fieldName Array[ ] AnyType ]
) Logical
3. getGenFilter ( criteria String ) Logical
```

Description

getGenFilter retrieves the filter criteria associated with a field, table frame, or multirecord object. This method does not return values directly; instead, it assigns them to a DynArray variable (syntax 1) or to two Array variables (syntax 2) that you declare and include as arguments.

In syntax 1, the DynArray *criteria* lists fields and filtering conditions as follows: the index is the field name, and the item is the corresponding filter expression.

In syntax 2, the Array *criteria* lists filtering conditions, and the optional Array *fieldName* lists corresponding field names. If you omit *fieldName*, conditions apply to fields in the order they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to the second field, and so on).

If the arrays used in syntax 2 are resizeable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

In syntax 3, the filter criteria is assigned to a String variable *criteria* that you must declare and pass as an argument.

▪

getGenFilter example 1

In this example, the **pushButton** method for a button named *btnSetFilter* uses **getGenFilter** to populate a DynArray *dyn* with a table frame's filter criteria. Then it checks the DynArray to see if the current criteria filters the Balance Due field for values greater than 10,000 and the Total Invoice field for values less that 65,000 and resets the filter if necessary.

```
;btnSetFilter :: pushButton
method pushButton(var eventInfo Event)
   var
      currentDyn,
      filterDyn   DynArray[] AnyType
      keysAr      Array[] AnyType
   endVar

   filterDyn["Balance Due"]   = "> 10000"
   filterDyn["Total Invoice"] = "< 65000"

   ORDERS.getGenFilter(currentDyn)   ; ORDERS is a table frame on a form.

   if currentDyn = filterDyn then
         return                        ; Filter is OK.
   else
         ORDERS.setGenFilter(filterDyn) ; Reset filter.
   endIf
endMethod
```

getGenFilter example 2

In this example, the **pushButton** method for a button named *btnShowFilter* uses **getGenFilter** to populate a DynArray *dyn* with the current filter criteria. Finally, the DynArray is displayed in a **view** dialog box. This technique is an alternative to setting flags to track the current filter criteria.

```
;btnShowFilter :: pushButton
method pushButton(var eventInfo Event)
   var
      dyn   DynArray[] AnyType
   endVar

   ORDERS.getGenFilter(dyn)   ; ORDERS is a table frame on a form.
   dyn.view("Current filter criteria")
endMethod
```

▪

getPosition method

See also Example UIObject Type

Reports the position of an object.

Syntax

`getPosition ( const x LongInt, const y LongInt, const w LongInt, const h LongInt)`

Description

getPosition gets the position of an object on the screen, relative to its container. Variables *x* and *y* specify the coordinates (in twips) of the upper left corner of the object. Variables *w* and *h* specify the width and height (in twips) of the object. If the object is not specified, self is implied.

To ObjectPAL, the screen is a two-dimensional grid, with the origin (0, 0) at the upper left corner of an object's container, positive x-values extending to the right, and positive y-values extending down.

For dialog boxes and for the Paradox Desktop application, the position is given relative to the entire screen; for forms, reports, and table windows, the position is given relative to the Paradox Desktop.

■

getPosition example

The following example moves a circle across the screen in response to timer events. The **pushButton** method for *toggleButton* uses **setTimer** and **killTimer** to start or stop a timer, depending on the condition of the button. When the timer starts, it issues a timer event every 100 milliseconds. Each **timer** event causes *toggleButton's* timer method to execute. The timer method locates the current position of the ellipse with **getPosition**, then moves it 100 twips to the right with **setPosition**.

Following is the code for *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
if buttonLabel = "Start Timer" then  ; if stopped, then start
  buttonLabel = "Stop Timer"         ; change label
  self.setTimer(100)          ; tell timer to issue a timer
                              ; event every 100 milliseconds
else
  buttonLabel = "Start Timer"         ; change label
  self.killTimer()                    ; stop the timer
endIf

endMethod
```

Following is the code for *toggleButton*'s **timer** method:

```
; toggleButton::timer
;   this method is called once for every timer event
method timer(var eventInfo TimerEvent)
var
  ui          UIObject
  x, y, w, h  SmallInt
endVar

ui.attach(floatCircle)          ; attach to the circle
ui.getPosition(x, y, w, h)      ; assign coordinates to vars
if x < 4320 then                ; if not at right edge of area
  ui.setPosition(x + 100, y, w, h)  ; move to the right
else
  ui.setPosition(1440, y, w, h)     ; return to the left
endIf

endMethod
```

getProperty method

See also Example UIObject Type

Returns the value of a specified property.

Syntax

getProperty **(** const ***propertyName*** String **)** AnyType

Description

getProperty returns the value of the property specified in *propertyName*. Not all properties take strings as values. For example, if a property value is a number, this method returns a number. To return a string in every case, use **getPropertyAsString**.

getProperty is an alternative to getting a property directly; it's useful when *propertyName* is a variable. Otherwise, access the property directly, as in

```
thisColor = myBox.Color
```

getProperty example

The following example creates a dynamic array, indexed by property names, to contain property values. The array is filled by using the array's index as the argument to the **getProperty** command.

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  propNames DynArray[] AnyType    ; to hold property names & values
  arrayIndex           String     ; index to dynamic array
endVar

propNames["Color"] = ""
propNames["Visible"] = ""
propNames["Name"] = ""

foreach arrayIndex in propNames   ; assign the properties to the array
  propNames[arrayIndex] = self.getProperty(arrayIndex)
endforeach

propNames["Color"] = "DarkBlue"

foreach arrayIndex in propNames   ; set properties from the array
  self.setProperty(arrayIndex, propNames[arrayIndex])
endforeach

endMethod
```

▪

getPropertyAsString method

See also Example UIObject Type

Returns the value of a specified property as a string.

Syntax

`getPropertyAsString ( const propertyName String ) String`

Description

getPropertyAsString returns a string containing the value of the property specified in *propertyName*.

▪

getPropertyAsString example

The following example assigns the value of the Color property to an AnyType variable using the **getProperty** method. The value returned is a LongInt, because colors are long integer constants. Next, the Color property is obtained using **getPropertyAsString**. The value returned is a String type, such as "Blue".

```
; boxOne::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  myColor  AnyType
endVar

myColor = self.getProperty("Color")
myColor.view()                              ; shows as LongInt
myColor = self.getPropertyAsString("Color")
myColor.view()                              ; shows as String
endMethod
```

▪

getRange method

See also Example UIObject Type

Retrieves the values that specify a range for a field, table frame, or multirecord object.

Syntax

```
getRange ( var rangeVals Array[ ] String ) Logical
```

Description

getRange retrieves the values that specify a range for a field, table frame, or multirecord object. This method does not return values directly; instead, it assigns them to an Array variable that you declare and include as an argument. The array values describe the range criteria, as listed in the following table.

Number of array items	Range specification
No items (empty array)	No range criteria associated with the UIObject.
One item	Specifies a value for an exact match on the first field of the index.
Two items	Specifies a range for the first field of the index.
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size *n*, specify exact matches on the first *n*-2 fields of the index. The last two array items specify a range for the *n*-1 field of the index.

If the array is resizeable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

▪

getRange example

The following example demonstrates using two unlinked tables in the data model and using ObjectPAL to link them. Assume a form has the *Orders* and *Lineitem* tables in its data model and they are not linked. **getRange** is used on a table frame bound to the *Lineitem* table to retrieve the values that specify the current range (if any).

The following code is attached to the **arrive** method of the record object of a table frame bound to the *Orders* table.

```
;Record :: arrive
method arrive(var eventInfo MoveEvent)
   var
      arSet   Array[] AnyType
      arGet   Array[] AnyType
   endVar

   LINEITEM.getRange(arGet)      ;Retrieve values of range.

   arSet.setSize(2)          ;Specify size of array.
   arSet[1] = string(Order_No.value)
   arSet[2] = string(Order_No.value)

   if (arSet.size() = arGet.size()) and (arSet <> arGet) then
      LINEITEM.setRange(arSet)      ;Specify range of records.
   endIf
endMethod
```

■

getRGB procedure

See also Example UIObject Type

Finds the red, green, and blue components of a color.

Syntax

`getRGB ( const rgb LongInt, var red SmallInt, var green SmallInt, var blue SmallInt )`

Description

getRGB returns the component red, green and blue values of the color specified in *rgb*, where *rgb* is one of the Colors constants. **getRGB** assigns the component values to the variables *red*, *green*, and *blue*, which you must declare and pass as arguments.

▪

getRGB example

The following example determines the red, green, and blue components of the constant Brown.

```
; decompBrown::pushButton
method pushButton(var eventInfo Event)
var
  thisRed, thisBlue, thisGreen SmallInt
endVar
getRGB(Brown, thisRed, thisGreen, thisBlue)
msgInfo("Brown is really",
        String("Red ", thisRed, "  Green ", thisGreen,
               "  Blue ", thisBlue))
endMethod
```

■

hasMouse method

See also Example UIObject Type

Tells if the mouse is positioned over an object.

Syntax

`hasMouse ( )` `Logical`

Description

hasMouse returns True if the pointer is positioned inside the boundaries of an object; otherwise, it returns False.

▪

hasMouse example

The following example assumes that a form has a bitmap object called *cat*. The **open** method for *cat* sets the timer interval to 250 milliseconds. The **timer** method uses **hasMouse** to determine if *cat* has the mouse; if not, it moves *cat* to the mouse's position.

Following is the code for *cat*'s **open** method:

```
; cat::open
method open(var eventInfo Event)
; set the timer interval to 250 milliseconds
self.setTimer(250)
endMethod
```

Following is the code for *cat*'s **timer** method:

```
; cat::timer
method timer(var eventInfo TimerEvent)
var
  mousePt  Point                     ; to get mouse position
endVar
if NOT cat.hasMouse() then           ; am I on the mouse?
  mousePt = getMouseScreenPosition() ; find the mouse
  cat.setPosition(mousePt.x() - 350,
                  mousePt.y() - 2880,
                  4320, 1750)        ; chase the mouse
  ; moves cat above and slightly to the left of mouse
  ; assumes cat is a bitmap with width 4320, height 1750
  ; since getMouseScreenPosition returns position of mouse
  ; on desktop, these numbers assume form is maximized
  ; offset (2880-1750) allows for height of menu and Toolbar
endIf
endMethod
```

home method

See also Example UIObject Type
Beginner

Moves to the first record in a table.

Syntax

```
home ( ) Logical
```

Description

home sets the current record to the first record of a table. **home** respects the limits of restricted views displayed in a linked table frame or multi-record object; **home** moves to the first record in a restricted view.

home has the same effect as the action constant DataBegin, so the following statements are equivalent:

```
obj.home()
obj.action(DataBegin)
```

▪

home example

The following example moves to the first record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form; *moveToHome* is a button on the same form.

```
; moveToHome::pushButton
method pushButton(var eventInfo Event)
CUSTOMER.home()  ; move to the first record
                 ; same as:  CUSTOMER.action(DataBegin)
msgInfo("At the first record?", CUSTOMER.atFirst())
endMethod
```

▪

insertAfterRecord method

See also Example UIObject Type

Inserts a record into a table after the current record.

Syntax

```
insertAfterRecord ( ) Logical
```

Description

insertAfterRecord inserts a record into a table after the current record. The table must be in Edit mode.

insertAfterRecord example

In the following example, suppose that *CustSort* is a copy of the *Customer* table, sorted by the Name field. The form contains a table frame named *CUSTSORT* bound to the *CustSort* table, an undefined field called *newField*, and a button called *insRecButton*. To add a new name to the table, type the name in *newField* and press *insRecButton*.

The following code is attached to the **pushButton** method for *insRecButton*. This method checks for a value in *newField*, then checks if the form is in Edit mode. If so, the method attaches the TCursor *custTC* to *CUSTSORT*, and scans *custTC* for a value greater than the string given in *newField*. If it finds a name greater than the new name, the method uses **insertRecord** to insert a new blank record before the name found; otherwise, it uses **insertAfterRecord** to insert a new blank record at the end of the table.

```
; insRecButton::pushButton
method pushButton(var eventInfo Event)
var
  custTC  TCursor
  nameStr String
endvar

if newField.Value = "" then       ; Quit if the field is blank.
  RETURN
endIf

nameStr = newField.Value          ; Get the name to add.
CUSTSORT."Name".moveTo()

if thisForm.Editing then          ; Check for edit mode first.
  custTC.attach(CUSTSORT)

  scan custTC for custTC."Name"  nameStr:
    quitloop                      ; Stop when you find the name.
  endscan

  msgInfo("Current record no", custTC.recno())
  CUSTSORT.resync(custTC)         ; Resync CUSTSORT to custTC.

  if NOT CUSTSORT.atLast() then
    CUSTSORT.insertBeforeRecord()
  else
    CUSTSORT.insertAfterRecord()  ; Add blank record.
  endIf

  ; ... fill the record with the rest of the customer information

else
  msgInfo("Sorry", "Form must be in Edit mode.")
endIf
endMethod
```

▪

insertBeforeRecord method

See also Example UIObject Type

Inserts a record into a table before the current record.

Syntax

`insertBeforeRecord ( ) Logical`

Description

insertBeforeRecord inserts a record into a table before the current record. The table must be in Edit mode.

insertBeforeRecord has the same effect as the action constant DataInsertRecord, so the following statements are equivalent:

```
obj.insertBeforeRecord()
obj.action(DataInsertRecord)
```

▪

insertBeforeRecord example

In the following example, suppose that *CustSort* is a copy of the *Customer* table, sorted by the Name field. The form contains a table frame named *CUSTSORT* bound to *CustSort*, an undefined field called *newField*, and a button called *insRecButton*. To add a new name to the table, type the name in *newField* and press *insRecButton*.

The following method overrides the **pushButton** method for *insRecButton*. This method checks for a value in *newField*, then checks if the form is in Edit mode. If so, the method attaches a TCursor named *custTC* to *CUSTSORT*, and scans *custTC* for a value greater than the string given in *newField*. If it finds a name greater than the new name, the method uses **insertBeforeRecord** to insert a new blank record before the name found; otherwise, it uses **insertAfterRecord** to insert a new blank record at the end of the table.

```
; insRecButton::pushButton
method pushButton(var eventInfo Event)
var
  custTC  TCursor
  nameStr String
endvar

if newField.Value = "" then       ; Quit if the field is blank.
  RETURN
endIf

nameStr = newField.Value          ; Get the name to add.
CUSTSORT."Name".moveTo()

if thisForm.Editing then          ; Check for edit mode first.
  custTC.attach(CUSTSORT)

  scan custTC for custTC."Name"  nameStr:
    quitloop                      ; Stop when you find the name.
  endscan

  msgInfo("Current record no", custTC.recno())
  CUSTSORT.resync(custTC)         ; Resync CUSTSORT to custTC.

  if NOT CUSTSORT.atLast() then
    CUSTSORT.insertBeforeRecord()
  else
    CUSTSORT.insertAfterRecord()
  endIf

  ; ... fill the record with the rest of the customer information
else
  msgInfo("Sorry", "Form must be in Edit mode.")
endIf
endMethod
```

▪

insertRecord method

See also   Example   UIObject Type
Beginner

Inserts a record into a table.

Syntax

insertRecord ( ) Logical

Description

insertRecord inserts a record before the current record into a table.

insertRecord has the same effect as **insertBeforeRecord** and the action constant DataInsertRecord, so the following three statements are equivalent:

```
obj.insertRecord()
obj.insertBeforeRecord()
obj.action(DataInsertRecord)
```

insertRecord example

See the example for **insertBeforeRecord**.

isContainerValid method

See also Example UIObject Type

Reports whether an object's container is valid.

Syntax

`isContainerValid ( )` `Logical`

Description

isContainerValid reports if the current object's container is valid. For instance, a form has no container, so the ContainerName property for a form is not valid.

▪

isContainerValid example

In the following example, the **arrive** built-in event method for a form uses **isContainerValid** to check for a container:

```
; thisForm::arrive
method arrive(var eventInfo MoveEvent)
  if eventInfo.isPreFilter() then

     ;Code here executes before each object
  else
     ;Code here executes afterwards (or for form)
     if NOT isContainerValid() then
       msgInfo("Form",
               "This object does not have a valid container.")
     endIf
  endIf
endMethod
```

▪

isEdit method

See also   Example   UIObject Type
Beginner

Reports whether an object is in Edit mode.

Syntax

```
isEdit ( ) Logical
```

Description

isEdit reports whether an object is in Edit mode.

▪

isEdit example

See the example for **lockRecord**.

isEmpty method

See also Example UIObject Type

Reports whether a table contains any records.

Syntax

```
isEmpty ( ) Logical
```

Description

isEmpty returns True if no records in the table are associated with the table frame. **isEmpty** respects the limits of restricted views displayed in a linked table frame or multi-record object.

You can also find out if a table is empty by checking the value returned by the **nRecords** method, or by checking the value of the NRecords property of the object.

■

isEmpty example

The *cascadeDelete* button in the following example deletes an order and all the linked detail records for that order. Assume that a form contains a single-record object bound to the *Orders* tables and a linked table frame bound to the *Lineitem* table. *Orders* has a one-to-many link to *Lineitem*.

```
; cascadeDelete::pushButton
method pushButton(var eventInfo Event)
var
  ui         UIObject
endVar

if thisForm.Editing then
  if msgQuestion("Confirm", "Delete this order?") = "Yes" then
    ui.attach(LINEITEM)
    while NOT ui.isEmpty()       ; check to see if linked table is
                                 ; empty■respects restricted view
      ui.deleteRecord()          ; delete the detail records
    endwhile
    ORDERS.action(DataDeleteRecord)   ; delete the master record
  endIf
else
  msgInfo("Status", "You must be editing to delete a record.")
endIf
endMethod
```

▪

isLastMouseClickedValid method

See also Example UIObject Type

Reports if the last object to receive a mouse click is valid.

Syntax

`isLastMouseClickedValid ( ) Logical`

Description

isLastMouseClickedValid reports if the current form has received a mouse click since it opened.

isLastMouseClickedValid example

This method checks to see if a form has been clicked yet.

```
; thisForm::arrive
method arrive(var eventInfo MoveEvent)
   if eventInfo.isPreFilter() then
     ;Code here executes before each object
  else
     ;Code here executes afterwards (or for form)
     if NOT isLastMouseClickedValid() then
       msgInfo("FYI", "This form has not been clicked yet.")
     endIf
  endIf
endMethod
```

▪

isLastMouseRightClickedValid method

See also Example UIObject Type

Reports if the last object to receive a right mouse click is valid.

Syntax

`isLastMouseRightClickedValid ( ) Logical`

Description

isLastMouseRightClickedValid reports if the current form has received a right mouse click since it opened.

isLastMouseRightClickedValid example

This method checks to see if a form has been right-clicked yet.

```
; thisForm::arrive
method arrive(var eventInfo MoveEvent)
  if eventInfo.isPreFilter() then

     ;Code here executes before each object
  else
     ;Code here executes afterwards (or for form)
     if NOT isLastMouseRightClickedValid() then
       msgInfo("FYI", "This form has not been right-clicked yet.")
     endIf
  endIf
endMethod
```

isRecordDeleted method

See also Example UIObject Type

Reports whether the current record has been deleted (dBASE tables only).

Syntax

```
isRecordDeleted ( ) Logical
```

Description

isRecordDeleted reports whether the current record has been deleted. **isRecordDeleted** works only for dBASE tables because deleted Paradox records can't be displayed. This method returns True if the current record has been deleted; otherwise, it returns False.

Deleted records in a dBASE table are not shown by default. For **isRecordDeleted** to work correctly, you must call TCursor::**showDeleted** to show deleted records in the table; otherwise, deleted records are not visible to **isRecordDeleted**.

isRecordDeleted example

See the example for TCursor::**isRecordDeleted**.

▪

keyChar method

See also Example UIObject Type
Beginner

Sends an event to an object's keyChar method.

Syntax

```
1. keyChar ( const characters String [ , const state SmallInt ] ) Logical
2. keyChar ( const ansiKeyValue SmallInt ) Logical
3. keyChar ( const ansiKeyValue SmallInt, const vChar SmallInt, const state SmallInt ) Logical
```

Description

keyChar constructs an event and calls the built-in **keyChar** event method of an object with that event. Specify one or more characters in *characters* (syntax 1), *ansiKeyValue* (syntax 2), or *ansiKeyValue* and *vChar* (syntax 3). Specify the keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as Alt+Ctrl.

▪

keyChar example

The following example overrides the **pushButton** method of a button named *sendKeyChar*. This method sends keystrokes to a field, called *fieldOne,* on the same form.

```
; sendKeyChar::pushButton
method pushButton(var eventInfo Event)
var
  x  SmallInt
endVar
fieldOne.keyChar("Send me an ")     ; send a string
fieldOne.keyChar(65, 65, Shift)     ; send ANSI char, decimal
                                    ; equivalent of VK_Char,
                                    ; and keyboardstate
fieldOne.keyChar(" and a ", Shift) ; send a string with the keyboardstate
x = 98                              ; set the code
fieldOne.keyChar(x)                 ; send ANSI char code
endMethod
```

▪

keyPhysical method

See also   Example   UIObject Type

Sends an event to an object's **keyPhysical** method.

Syntax

`keyPhysical (` `const` ***`aChar`*** `SmallInt, const` ***`vChar`*** `SmallInt, const` ***`state`*** `SmallInt` **`)`**

Description

keyPhysical sends an event to an object's built-in **keyPhysical** method. You must specify the ANSI character code in *aChar*, the virtual key code in *vChar*, and the keyboard state in *state* (using KeyboardStates constants).

▪

keyPhysical example

The following code is attached to the **pushButton** method of a button named *sendKeyPhys*. This method sends the character "a" to the field *fieldOne*.

```
; sendKeyPhys::pushButton
method pushButton(var eventInfo Event)
   fieldOne.keyPhysical(97, 97, Shift)    ; send an "a"
endMethod
```

killTimer method

See also Example UIObject Type

Stops the timer associated with an object.

Syntax

```
killTimer ( )
```

Description

killTimer stops a timer associated with an object.

▪

killTimer example

The following example moves a circle across the screen in response to TimerEvents. The **pushButton** method for *toggleButton* uses **setTimer** and **killTimer** to start or stop a timer, depending on the condition of the button. When the timer starts, it issues a TimerEvent every 100 milliseconds. Each TimerEvent causes *toggleButton's* **timer** method to execute. The **timer** method gets the current position of the ellipse with **getPosition**, then moves it 100 twips to the right with **setPosition**.

Following is the code for *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
if buttonLabel = "Start Timer" then  ; if stopped, then start
  buttonLabel = "Stop Timer"         ; change label
  self.setTimer(100)           ; tell timer to issue a timer
                               ; event every 100 milliseconds
else
  buttonLabel = "Start Timer"         ; change label
  self.killTimer()                    ; stop the timer
endIf

endMethod
```

Following is the code for *toggleButton*'s **timer** method:

```
; toggleButton::timer
;   this method is called once for every timer event
method timer(var eventInfo TimerEvent)
var
  ui           UIObject
  x, y, w, h   SmallInt
endVar

ui.attach(floatCircle)          ; attach to the circle
ui.getPosition(x, y, w, h)      ; assign coordinates to vars
if x < 4320 then                ; if not at right edge of area
  ui.setPosition(x + 100, y, w, h)  ; move to the right
else
  ui.setPosition(1440, y, w, h)     ; return to the left
endIf

endMethod
```

▪

locate method

See also Example UIObject Type
Beginner

Searches for a specified field value.

Syntax

1. locate (const ***fieldName*** String, const ***exactMatch*** AnyType [,const ***fieldName*** String, const ***exactMatch*** AnyType] * **)** Logical

2. locate (const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType [,const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType] * **)** Logical

Description

locate searches a table frame, multi-record object, record object, or field object for records whose values exactly match the criteria specified in one or more field/value pairs. Specify the value to search for in *exactMatch* and the field to search in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). This method uses active indexes when it can to speed the search. It respects the limits of restricted views in linked detail tables.

The search always starts from the beginning of the table, but if no match is found, the insertion point returns to the current record. If a match is found, the insertion point moves to that record. This operation fails if the current record cannot be posted and unlocked (for example, because of a key violation).

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is on.

▪

locate example

In the following example, assume that a form contains a table frame bound to the *Customer* table, and a button named *locateButton*. The **pushButton** method for *locateButton* attempts to find the customer named "Sight Diver" in the city "Kato Paphos". If found, the customer's name is changed to "Right Diver".

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
  Cust UIObject
endVar
Cust.attach(CUSTOMER)
; find customer named "Sight Diver" in Kato Paphos
if Cust.locate("Name", "Sight Diver", "City", "Kato Paphos") then
   Cust.edit()
   Cust."Name" = "Right Diver"
   Cust.endEdit()
endIf
endMethod
```

▪

locateNext method

See also Example UIObject Type
Beginner

Searches forward from the current record for a specified field value.

Syntax

1. locateNext (const ***fieldName*** String, const ***exactMatch*** AnyType [, const ***fieldName*** String, const ***exactMatch*** AnyType] * **)** Logical
2. locateNext (const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType [, const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType] * **)** Logical

Description

locateNext searches a table for records whose values exactly match the criteria specified in one or more field/value pairs. Specify the value to search for in *exactMatch* and the field to search in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). This method uses active indexes when it can to speed the search. It respects the limits of restricted views in linked detail tables.

The search begins with the record after the current record. If a match is found, the insertion point moves to that record. If no match is found, the insertion point returns to the current record. To start a search from the beginning of a table, use **locate**.

This operation fails if the current record cannot be committed (for example, because of a key violation).

Note: The search is case-sensitive unless Session::ignoreCaseInLocate is on.

▪

locateNext example

For the following example, suppose a form contains a table frame bound to the *Customer* table, and one button named *locateButton*. The **pushButton** method for *locateButton* searches for customers in the city of Freeport. If the first **locate** is successful, the method uses **locateNext** to find successive records.

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject
   searchFor String
   numFound  SmallInt
endVar
Cust.attach(CUSTOMER)
searchFor = "Freeport"
if Cust.locate("City", searchFor) then
   numFound = 1
   message("")
   while Cust.locateNext("City", searchFor)
      numFound = numFound + 1
   endWhile
   msgInfo("Found " + searchFor, strval(numFound) + " times.")
endIf
```

▪

locateNextPattern method

See also Example1 Example2 UIObject Type

Locates the next record containing a field that has a specified pattern of characters.

Syntax

```
1. locateNextPattern ( [ const fieldName String,const exactMatch AnyType, ] *
const fieldName String, const pattern String ) Logical
2. locateNextPattern ( [ const fieldNum SmallInt, const exactMatch AnyType, ]
* const fieldNum SmallInt, const pattern String ) Logical
```

Description

locateNextPattern finds substrings (for example, "comp" in "computer"). The search begins with the record after the current record. If a match is found, the insertion point moves to that record. If no match is found, the insertion point returns to the current record. This method uses active indexes when it can to speed the search. It respects the limits of restricted views in linked detail tables.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search at the beginning of a table, use **locatePattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance), and specify a pattern of characters in *pattern*.

You can include the standard pattern operators **@** and **..** in the *pattern* argument. The **..** operator stands for any string of characters (including none at all); **@** stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildCardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** for more information about advanced match pattern operators.

To search records based on the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the next statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, "database").

```
tc.locateNextPattern("Name", "Borland"  "Product", "Paradox" "Keywords",
"data..")
```

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is on.

For examples, see Sample search strings with wildcards in the User's Guide help.

▪

locateNextPattern example 1

The following example searches for multiple occurrences of the letter "C" in the Name field of the *Customer* table, and writes the matching names to an array. Suppose that the *CUSTOMER* table frame is bound to *Customer*, and *locateButton* is a button on the same form.

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject           ; to attach to CUSTOMER table frame
   searchFor String             ; the pattern string to search for
   numFound  SmallInt           ; the number of matches located
   custNames Array[] String     ; the matches found
endVar

cust.attach(CUSTOMER)
searchFor = "C.."               ; find customers whose name
                                ; begins with C
if cust.locatePattern("Name", searchFor) then  ; if you can find one
  numFound = 1                                 ; post it to the array
  custNames.grow(1)                            ; then keep looking
  custNames[numFound] = cust."Name"
  while cust.locateNextPattern("Name", searchFor)
    numFound = numFound + 1
    custNames.grow(1)
    custNames[numFound] = cust."Name"
  endWhile
endIf
if custNames.size() > 0 then     ; if there's anything in the array
  custNames.view()               ; show the array
endIf
endMethod
```

■

locateNextPattern example 2

The following example is similar to Example 1, except that it searches for records based on the value of the City field and a pattern in the Name field:

```
; locateButtonTwo::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject          ; to attach to CUSTOMER TableFrame
   searchFor String            ; the pattern string to search for
   numFound  SmallInt          ; the number of matches located
   custNames Array[] String    ; the matches found
endVar

cust.attach(CUSTOMER)
searchFor = "..C.."            ; find customers whose name
                               ; includes a C
if cust.locatePattern("City", "Marathon", "Name", searchFor) then  ; if you
can find one
  numFound = 1                                    ; post it to the array
  custNames.grow(1)                               ; then keep looking
  custNames[numFound] = cust."Name"
  while cust.locateNextPattern("City", "Marathon", "Name", searchFor)
    numFound = numFound + 1
    custNames.grow(1)
    custNames[numFound] = cust."Name"
  endWhile
endIf
if custNames.size()  0 then     ; if there's anything in the array
  custNames.view()               ; show the array
endIf
endMethod
```

▪

locatePattern method

See also Example UIObject Type

Searches for a record containing a field that has a specified pattern of characters.

Syntax

```
1. locatePattern ( [ const fieldName String, const exactMatch AnyType, ] *
const fieldName String, const pattern String ) Logical
2. locatePattern ( [ const fieldNum SmallInt, const exactMatch AnyType, ] *
const fieldName SmallInt, const pattern String ) Logical
```

Description

locatePattern finds substrings (for example, "comp" in "computer"). This method uses active indexes when it can to speed the search. This method respects the limits of restricted views in linked detail tables.

The search always starts at the beginning of the table, but if no match is found, the insertion point returns to the current record. If a match is found, the insertion point moves to that record. This operation fails if the current record cannot be committed (for example, because of a key violation).

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance), and specify a pattern of characters in *pattern*.

You can include the standard pattern operators **@** and **..** in the *pattern* argument. The **..** operator stands for any string of characters (including none at all); **@** stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildCardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** for more information about advanced match pattern operators.

To search records based on the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the next statement searches the Name field for exact matches on "Borland", the Product field for "Paradox," and the Keywords field for words beginning with "data" (for example, database).

To start a search after the current record, use **locateNextPattern**.

```
tc.locatePattern("Name", "Borland"  "Product", "Paradox" "Keywords",
"data..")
```

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is on.

For examples, see Sample search strings with wildcards in the User's Guide help.

▪

locatePattern example

See the example for **locateNextPattern**.

▪

locatePrior method

See also Example UIObject Type

Searches backward for a specified field value.

Syntax

```
1. locatePrior ( const fieldName String, const exactMatch AnyType [ , const
fieldName String, const exactMatch AnyType ] * ) Logical
2. locatePrior ( const fieldNum SmallInt, const exactMatch AnyType [ , const
fieldNum SmallInt, const exactMatch AnyType ] * ) Logical
```

Description

locatePrior searches backwards from the current record in a table for records whose values exactly match the criteria specified in one or more field/value pairs. Specify the value to search for in *exactMatch* and the field to search in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). This method uses active indexes when it can to speed the search. It respects the limits of restricted views in linked detail tables.

The search begins with the record before the current record and moves up through the table. If a match is found, the insertion point moves to that record. If no match is found, the insertion point returns to the current record. To start a search from the beginning of a table, use **locate**.

This operation fails if the current record cannot be committed (for example, because of a key violation).

Note: The search is case-sensitive unless Session::ignoreCaseInLocate is on.

▪

locatePrior example

The method shown in the following example locates the last occurrence of a value in a table by moving to the end of the table and using **locatePrior** to search up for a match. Assume that the form contains a table frame bound to the *Customer* table, and one button named *locateButton*.

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject      ; to attach to CUSTOMER table frame
   searchFor String        ; the string to search for
endVar
Cust.attach(CUSTOMER)      ; attach to table frame
Cust.end()                 ; move to the end of the table
searchFor = "Freeport"
if Cust.locatePrior("City", searchFor) then   ; find record
   msgInfo("Status", "The last record with a City of " +
           searchFor + " is record " + Cust.recno + ".")
endIf

endMethod
```

▪

locatePriorPattern method

See also Example UIObject Type

Locates the prior record containing a field that has a specified pattern of characters.

Syntax

```
1. locatePriorPattern ( [ const fieldName String, const exactMatch AnyType, ] * const fieldName String, const pattern String ) Logical
2. locatePriorPattern ( [ const fieldNum SmallInt, const exactMatch AnyType, ] * const fieldNum SmallInt, const pattern String ) Logical
```

Description

locatePriorPattern finds substrings (for example, "comp" in "computer"). This method uses active indexes when it can to speed the search. This method respects the limits of restricted views in linked detail tables.

The search begins with the record before the current record and moves up through the table. If a match is found, the insertion point moves to that record. If no match is found, the insertion point returns to the current record. This method uses active indexes when it can to speed the search.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search at the beginning of a table, use **locatePattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance), and specify a pattern of characters in *pattern*.

You can include the standard pattern operators **@** and **..** in the *pattern* argument. The **..** operator stands for any string of characters (including none at all); **@** stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildCardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** for more information about advanced match pattern operators.

To search records based on the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the next statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, "database").

```
obj.locatePriorPattern("Name", "Borland"  "Product", "Paradox" "Keywords", "data*")
```

Note: The search is case-sensitive unless Session::**ignoreCaseInLocate** is on.

For examples, see Sample search strings with wildcards in the User's Guide help.

▪

locatePriorPattern example

The method shown in the following example locates the last occurrence of a value in a table by moving to the end of the table and using **locatePriorPattern**. Assume that the form contains a table frame bound to the *Customer* table, and one button named *locateButton*.

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject      ; to attach to CUSTOMER table frame
   searchFor String        ; the string to search for
endVar
Cust.attach(CUSTOMER)      ; attach to table frame
Cust.end()                 ; move to the end of the table
searchFor = "Freeport"
if Cust.locatePrior("City", searchFor, "Name", "..C..") then   ; find record
   msgInfo("Status", "The last record with a City of " + searchFor +
           "and a name with C is record " + Cust.recno + ".")
endIf

endMethod
```

▪

lockRecord method

See also Example1 Example2 UIObject Type
Beginner

Puts a write lock on the current record.

Syntax
`lockRecord ( )` `Logical`

Description
lockRecord returns True if it successfully places an explicit write lock on the current record; otherwise, it returns False. If the record already exists, it is locked and becomes the current record.

Note: The Locked property is a read-only property. You can examine the property to find out whether an object is locked, but you can't change the property to lock or unlock an object.

▪

lockRecord example 1

The following example first checks to see if the *Customer* table is in Edit mode. If so, the method locates a record, attempts to lock it with **lockRecord**, then checks the status of the lock with **recordStatus**. Assume that a form contains a table frame bound to the *Customer* table, and a button named *lockButton*. Assume also that the record inside the *CUSTOMER* table frame is named *custRec*.

```
; lockButton::pushButton
method pushButton(var eventInfo Event)
var
  obj  UIObject
endVar
obj.attach(CUSTOMER)
obj.locate("Name", "Sight Diver")
if thisForm.editing then
if CUSTOMER.isEdit() then
  if NOT obj.lockRecord() then
     msgStop("Lock failed", "recordStatus(\"Locked\") is " +
             String(obj.recordStatus("Locked")))
  else
     msgStop("Lock succeeded", "recordStatus(\"Locked\") is " +
             String(obj.recordStatus("Locked")))
     obj.custRec."Name" = "Right Diver"  ; quotes on Name indicate
                                         ; field name instead of property
     obj.unlockRecord()
  endIf
else
  msgInfo("Status", "You must be in edit mode to lock and change records.")
endIf
endMethod
```

▪

lockRecord example 2

The following example shows how you can examine the Locked property for a record object to determine if the record is locked. This example behaves roughly the same as Example 1.

```
; lockButtonTwo::pushButton
method pushButton(var eventInfo Event)
var
  obj,
  recObj  UIObject
endVar

obj.attach(CUSTOMER)
obj.locate("Name", "Sight Diver")

if thisForm.editing then
  obj.lockRecord()                   ; no write access to Locked property
                                     ; so use method to lock record
  recObj.attach(CUSTOMER.custRec)
  if NOT recObj.Locked then          ; check the property to see
                                     ; if the record is locked
     msgStop("Lock failed", "recObj.Locked is " +
             String(recObj.Locked))
  else
     msgStop("Lock succeeded", "recObj.Locked is " +
             String(recObj.Locked))
     recObj."Name" = "Right Diver" ; name is in quotes to indicate Name
                                     ; field instead of obj's Name property
     obj.unlockRecord()
  endIf
else
  msgInfo("Status", "You must be in edit mode to lock and change records.")
endIf
endMethod
```

▪

lockStatus method

See also Example UIObject Type

Returns the number of locks on a table.

Syntax

```
lockStatus (const lockType String ) SmallInt
```

Description

lockStatus returns the number of times you've placed a lock of type *lockType* on a table, where *lockType* is one of "Write", "Read", or "Any".

If you haven't placed any locks of a given type, **lockStatus** returns 0.

If you specify "Any" for *lockType*, **lockStatus** returns the total number of locks you've placed on the table. **lockStatus** only reports on locks you've placed explicitly, not on locks placed by Paradox or by other users or applications.

▪

lockStatus example

The following example assumes that a form has a table frame named *CUSTOMER* bound to the *Customer* table, and a button named *lockButton*. The **pushButton** method for *lockButton* removes all locks from *CUSTOMER*, checks for locks with **lockStatus**, places a lock, then reports on the locks with **lockStatus** again.

```
; lockButton::pushButton
method pushButton(var eventInfo Event)
var
  CustTC TCursor     ; to place a lock on the table
  Cust UIObject
  l    Logical
endVar
CustTC.attach(CUSTOMER)      ; attach the TCursor to CUSTOMER
l = unlock(CustTC, "ALL")    ; remove any locks
l.view("Unlock successful:")
Cust.attach(CUSTOMER)        ; attach the UIObject to CUSTOMER
if Cust.lockStatus("ANY") = 0 then    ; check for locks
   l = lock(CustTC, "WL")             ; place a write lock
   l.view("Lock successful:")         ; check up on it
endIf
msgInfo("Status", "Table " + Cust.Name + " has " +
         String(Cust.lockStatus("WL")) + " write lock(s).")
unlock(CustTC, "ALL")                 ; remove any locks
endMethod
```

menuAction method/procedure

See also Example UIObject Type

Sends an event to an object's **menuAction** method.

Syntax

`menuAction ( const action SmallInt )`

Description

menuAction constructs a MenuEvent and sends it to a specified UIObject's **menuAction** method.

action is one of the MenuCommands constants, or a user-defined menu constant.

Note: You can't use **menuAction** to send a menu command constant that is equivalent to a command on the File menu. To simulate a File menu command, use one of the regular Action constants, manipulate a property, or use one of the many System type methods that emulate File menu commands.

menuAction example

In the following example, the *sendATile* button on the current form opens sends the form (*thisForm*) a MenuWindowTile action.

```
; sendATile::pushButton
method pushButton(var eventInfo Event)
thisForm.menuAction(MenuWindowTile)
endMethod
```

■

methodDelete method

See also Example UIObject Type

Deletes a specified method.

Syntax

```
methodDelete ( const methodName String ) Logical
```

Description

methodDelete deletes the method specified by *methodName*. The form that contains the object must be in Form Design window.

▪

methodDelete example

The following example uses **methodGet**, **methodSet**, and **methodDelete** to copy methods from one object to another. The method shown here overrides the **pushButton** method for a button named *copyMethods*. Four other objects are on the same form: the *targetForm* field lets you specify the name of the form containing the objects to copy; the *sourceObject* field holds the name of the object containing the methods to copy; the *destinationObject* field contains the name of the object to copy the methods to; and a radio button field, named *copyOrMove*, lets you specify whether methods in the source should be copied, or copied then deleted.

```
; copyMethods::pushButton
method pushButton(var eventInfo Event)
var
  otherForm            Form       ; a handle to a form
  sourceObj,                      ; object to copy from
  destObj              UIObject   ; object to copy to
  methodStr            String     ; stores the method definition
  methodArray Array[]  String     ; holds method names to copy
  i                    SmallInt   ; array index
endvar

; open the form and attach to the objects
if targetForm = "" OR sourceObject = "" OR destinationObject = "" then
  msgStop("Error", "Please fill in form, source, and destination.")
  return
endIf
if NOT otherForm.load(targetForm.value) then
  msgStop("Error", "Couldn't open named form.")
  return
endIf
if NOT sourceObj.attach(otherForm, sourceObject.value) then
  otherForm.close()
  msgStop("Error", "Couldn't find source object. Please specify entire
path.")
  return
endIf
if NOT destObj.attach(otherForm, destinationObject.value) then
  otherForm.close()
  msgStop("Error", "Couldn't find destination object. Specify entire path.")
  return
endIf

; set up the array of method names to copy
methodArray.addLast("mouseUp")
methodArray.addLast("mouseDown")
methodArray.addLast("mouseDouble")
methodArray.addLast("mouseEnter")
methodArray.addLast("mouseExit")
methodArray.addLast("mouseRightUp")
methodArray.addLast("mouseRightDown")
methodArray.addLast("mouseRightDouble")
methodArray.addLast("mouseMove")
methodArray.addLast("open")
methodArray.addLast("close")
methodArray.addLast("canArrive")
methodArray.addLast("arrive")
methodArray.addLast("setFocus")
methodArray.addLast("canDepart")
methodArray.addLast("depart")
methodArray.addLast("removeFocus")
methodArray.addLast("depart")
methodArray.addLast("timer")
methodArray.addLast("keyPhysical")
methodArray.addLast("keyChar")
methodArray.addLast("action")
methodArray.addLast("menuAction")
methodArray.addLast("error")
```

```
methodArray.addLast("status")

; add the method names specific to fields and buttons
if sourceObj.class = "Field" AND destObj.class = "Field" then
  methodArray.addLast("changeValue")
  methodArray.addLast("newValue")
else
if sourceObj.class = "Button" AND destObj.class = "Button" then
  methodArray.addLast("pushButton")
endIf
if sourceObj.class <> "Button" AND destObj.class <> "Button" then
  methodArray.addLast("mouseClick")
endIf

; copy methods from sourceObj to destObj on form otherForm
for i from 1 to methodArray.size()
  ; write the method named in methodArray to the string
;  msgInfo("methodArray is", methodArray[i])
  try
    methodStr = sourceObj.methodGet(methodArray[i])
    msgInfo("FYI", "Retrieved " + methodArray[i] + " method.")
    ; write the string to the method named in methodArray
    destObj.methodSet(methodArray[i], methodStr)
    if copyOrMove.Value = "Move" then
      sourceObj.methodDelete(methodArray[i])
    endIf
  onfail
  ;  loop
  endTry
endfor

endMethod
```

▪

methodGet method

See also Example UIObject Type

Returns the text of a specified method.

Syntax

`methodGet ( const methodName String ) String`

Description

methodGet returns a string containing the text of the method specified in *methodName*.

▪

methodGet example

See the example for **methodDelete**.

▪

methodSet method

See also Example UIObject Type

Sets the text of a specified method.

Syntax

```
methodSet ( const methodName String, const methodText String ) Logical
```

Description

methodSet specifies in *methodText* the source code for the method named in *methodName*. The form that contains the object should be in Design mode.

▪

methodSet example

See the example for **methodDelete**.

mouseClick method

See also Example UIObject Type

Sends an event to an object's **mouseClick** method.

Syntax

`mouseClick ( )` `Logical`

Description

mouseClick constructs a **mouseClick** MouseEvent and calls the built-in **mouseClick** event method of an object with that event.

▪

mouseClick example

The following example sends a **mouseClick** MouseEvent to *fieldTwo* on the same form:

```
; sendMouseClick::pushButton
method pushButton(var eventInfo Event)
; send a mouseClick to fieldTwo
fieldTwo.mouseClick()
endMethod
```

▪

mouseDouble method

See also Example UIObject Type

Sends an event to an object's **mouseDouble** method.

Syntax

`mouseDouble ( const x LongInt, const y LongInt, const state SmallInt ) Logical`

Description

mouseDouble constructs a double-click event and calls the built-in **mouseDouble** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as Ctrl+LeftButton.

mouseDouble example

The following example sends a double-click to *fieldTwo* on the same form:

```
; sendMouseDouble::pushButton
method pushButton(var eventInfo Event)
; send a mouseDouble to fieldTwo
fieldTwo.mouseDouble(100, 100, LeftButton)
endMethod
```

▪

mouseDown method

See also Example UIObject Type

Sends an event to an object's **mouseDown** method.

Syntax

```
mouseDown ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseDown constructs an event and calls the built-in **mouseDown** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as LeftButton+Ctrl.

mouseDown example

The following example sends a **mouseDown** and a **mouseUp** MouseEvent to the object *fieldOne* on the same form:

```
method pushButton(var eventInfo Event)
var
  fPt  Point
endVar
fPt = fieldOne.Position
fieldOne.mouseDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)
fieldOne.mouseUp(fPt.x(), fPt.y(), LeftButton)
endMethod
```

▪

mouseEnter method

See also Example UIObject Type

Sends an event to an object's **mouseEnter** method.

Syntax

```
mouseEnter ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseEnter constructs an event and calls the built-in **mouseEnter** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as LeftButton+Ctrl.

▪

mouseEnter example

The following example sends a **mouseEnter** MouseEvent to a field named *fieldSix* on the same form:

```
; sendMouseEnter::pushButton
method pushButton(var eventInfo Event)
; send a mouseEnter to fieldSix
fieldSix.mouseEnter(100,100,LeftButton)
endMethod
```

▪

mouseExit method

See also Example UIObject Type

Sends an event to an object's **mouseExit** method.

Syntax

```
mouseExit ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseExit constructs an event and calls the built-in **mouseExit** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as LeftButton+Ctrl.

mouseExit example

The following example sends a **mouseExit** MouseEvent to *fieldSeven* on the same form:

```
; sendMouseExit::pushButton
method pushButton(var eventInfo Event)
; send a mouseExit to fieldSeven
fieldSeven.mouseExit(100, 100, LeftButton)
endMethod
```

▪

mouseMove method

See also Example UIObject Type

Sends an event to an object's **mouseMove** method.

Syntax

`mouseMove ( const x LongInt, const y LongInt, const state SmallInt ) Logical`

Description

mouseMove constructs an event and calls the built-in **mouseMove** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as LeftButton+Ctrl.

▪

mouseMove example

The following example sends a **mouseDown**, a **mouseUp**, and a **mouseMove** MouseEvent to a field named *fieldFive* on the same form:

```
; sendMouseMove::pushButton
method pushButton(var eventInfo Event)
fieldFive.mouseDown(100, 100, LeftButton)
fieldFive.mouseUp(100, 100, LeftButton)
; send a mouseMove to fieldFive
fieldFive.mouseMove(100, 100, LeftButton)
endMethod
```

mouseRightDouble method

See also Example UIObject Type

Sends an event to an object's **mouseRightDouble** method.

Syntax

```
mouseRightDouble ( const x LongInt, const y LongInt, const state SmallInt )
Logical
```

Description

mouseRightDouble constructs an event and calls the built-in **mouseRightDouble** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as LeftButton+Ctrl.

▪

mouseRightDouble example

The following example sends a **mouseRightDouble** MouseEvent to a field named *fieldTwo* on the same form:

```
; sendMouseDouble::pushButton
method pushButton(var eventInfo Event)
; send a mouseDouble to fieldTwo
fieldTwo.mouseDouble(100, 100, LeftButton)
endMethod
```

mouseRightDown method

See also Example UIObject Type

Sends an event to an object's **mouseRightDown** method.

Syntax

mouseRightDown (** const ***x* LongInt, const ***y*** LongInt, const ***state*** SmallInt **)** Logical

Description

mouseRightDown constructs an event and calls the built-in **mouseRightDown** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as LeftButton+Ctrl.

mouseRightDown example

The following example sends a **mouseRightDown** and a **mouseRightUp** MouseEvent to a field named *fieldThree* on the same form:

```
; sendMouseRightUp::pushButton
method pushButton(var eventInfo Event)
var
  fPt Point
endVar
fP = fieldThree.position     ; get the position, send a mouseRightDown
fieldThree.mouseRightDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)                   ; pause, then send a mouseRightUp
fieldThree.mouseRightUp(fPt.x(), fPt.y(), LeftButton)
endMethod
```

■

mouseRightUp method

See also Example UIObject Type

Sends an event to an object's **mouseRightUp** method.

Syntax

```
mouseRightUp ( const x LongInt, const y LongInt, const state SmallInt )
Logical
```

Description

mouseRightUp constructs an event and calls the built-in **mouseRightUp** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as LeftButton+Ctrl.

▪

mouseRightUp example

The following example sends a **mouseRightDown** and a **mouseRightUp** MouseEvent to a field named *fieldThree* on the same form:

```
; sendMouseRightUp::pushButton
method pushButton(var eventInfo Event)
var
  fPt Point
endVar
fP = fieldThree.position     ; get the position, send a mouseRightDown
fieldThree.mouseRightDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)                   ; pause, then send a mouseRightUp
fieldThree.mouseRightUp(fPt.x(), fPt.y(), LeftButton)
endMethod
```

■

mouseUp method

See also Example UIObject Type

Sends an event to an object's **mouseUp** method.

Syntax

```
mouseUp ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseUp constructs an event and calls the built-in **mouseUp** event method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. These constants can be added together to create combined key states, such as LeftButton+Ctrl.

mouseUp example

The following example sends a **mouseDown** and a **mouseUp** MouseEvent o the object *fieldOne* on the same form:

```
method pushButton(var eventInfo Event)
var
  fPt  Point
endVar
fPt = fieldOne.Position
fieldOne.mouseDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)
fieldOne.mouseUp(fPt.x(), fPt.y(), LeftButton)
endMethod
```

▪

moveTo method

See also Example UIObject Type
Beginner

Sets the focus to a specified object.

Syntax

```
1. (Method) moveTo ( ) Logical
2. (Procedure) moveTo ( const objectName String ) Logical
```

Description

moveTo moves the focus to a specified object. When you call **moveTo** as a procedure (syntax 2), *objectName* specifies the name of the object to move to (the destination object).

▪

moveTo example

In the following example, assume a form contains a table frame bound to *Orders*, and another table frame bound to *LineItem*. *Orders* has a one-to-many link to *LineItem*. A button named *findDetails* is also on the form. Suppose you want to be able to search through the entire *LineItem* table▪not just through those records linked to the current order. In this case, the **pushButton** method for *findDetails* searches for orders that include the current part number.

This code is attached to the Var window for *findDetails*:

```
; findDetails::Var
Var
  lineTC TCursor  ; instance of LINEITEM for searching
endVar

; findDetails::open
method open(var eventInfo Event)
lineTC.open("LineItem.db")
endMethod
```

Following is the code for *findDetails*' **pushButton** method:

```
; findDetails::pushButton
method pushButton(var eventInfo Event)
var
  stockNum  Number
  orderTC    TCursor
  OrderNum    Number
endVar

; get Stock No from current LineItem
stockNum = LINEITEM.lineRecord."Stock No"
; lineTC was declared in Var window and opened by open method
if NOT lineTC.locateNext("Stock No", stockNum) then
  lineTC.locate("Stock No", stockNum)
endIf
orderTC.attach(ORDERS)
orderTC.locate("Order No", lineTC."Order No")
ORDERS.moveToRecord(orderTC)       ; move to CUSTOMER and
                             ; resynchronize with TCursor
LINEITEM.lineRecord."Stock No".moveTo()    ; move insertion point to LINEITEM
detail
; move insertion point to matching record
LINEITEM.locate("Stock No", stockNum)
endMethod
```

Following is the code for *findDetails*' **close** method:

```
; findDetails::close
method close(var eventInfo Event)
lineTC.close()   ; close the TCursor to LineItem
endMethod
```

■

moveToRecNo method

See also Example UIObject Type

Moves to a specific record in a dBASE table.

Syntax

```
moveToRecNo ( const recordNum LongInt ) Logical
```

Description

moveToRecNo sets the current record to the record *recordNum*. It returns an error if *recordNum* is not in the table. Use the method **nRecords** or examine the NRecords property to find out how many records a table contains. This method is recommended only for dBASE tables; **moveToRecord** is recommended for Paradox tables.

moveToRecNo example

The following example moves to the midpoint of a table. Assume that a form contains a table frame bound to the *LineItem* table, and a button called *MidWay*.

```
; MidWay::pushButton
method pushButton(var eventInfo Event)
var
  halfWay  LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecNo(halfWay)

endMethod
```

▪

moveToRecord method

See also Example UIObject Type

Moves to a specific record in a table.

Syntax

```
1. moveToRecord ( const recordNum LongInt ) Logical
2. moveToRecord ( const tc TCursor ) Logical
```

Description

moveToRecord sets the current record.

Syntax 1 moves to the record number specified in *recordNum*. It returns an error if *recordNum* is greater than the number of records in the table. Use the method **nRecords** or examine the NRecords property to find out how many records a table contains.

Syntax 2 moves to the record pointed to by the TCursor *tc*. This method can be very slow for dBASE tables; use **moveToRecNo** instead.

▪

moveToRecord example

For an example of how to use **moveToRecord** with a TCursor, see the example for **moveTo**.

The following example moves to the midpoint of a table. Assume that a form contains a table frame bound to the *LineItem* table, and a button called *MidWay*.

```
; MidWay::pushButton
method pushButton(var eventInfo Event)
var
  halfWay  LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecord(halfWay)

endMethod
```

▪

nextRecord method

See also   Example   UIObject Type
Beginner

Moves to the next record in a table.

Syntax

```
nextRecord ( ) Logical
```

Description

nextRecord sets the current record to the next record in a table. It returns an error if the insertion point is already at the last record.

nextRecord has the same effect as the action constant DataNextRecord, so the following statements are equivalent:

```
obj.nextRecord()
obj.action(DataNextRecord)
```

▪

nextRecord example

The following example moves to the next record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form; *moveToNext* is a button on the same form.

```
; moveToNext::pushButton
method pushButton(var eventInfo Event)
if NOT CUSTOMER.atLast() then
  CUSTOMER.nextRecord()  ; move to the next record
  ; same as:  CUSTOMER.action(DataNextRecord)
  msgInfo("What record?", CUSTOMER.recno)
else
  msgInfo("Status", "Already at the last record.")
endIf
endMethod
```

nFields method

See also Example UIObject Type

Returns the number of fields in a table.

Syntax

`nFields ( ) LongInt`

Description

nFields returns the number of fields in a table.

Note: To find the number of columns displayed in an object bound to a table, examine the value of the NCols property for that object.

▪

nFields example

The following example reports on the number of fields and key fields in the *LineItem* table. Assume that a form has a table frame named *LINEITEM* bound to the *LineItem* table, and a button named *tableStats*.

```
; tableStats::pushButton
method pushButton(var eventInfo Event)
msgInfo("Status", "The LineItem table has " +
        String(LINEITEM.nFields()) + " fields and " +
        String(LINEITEM.nKeyFields()) + " key fields." +
        "\nThere are " + String(LINEITEM.NCols) +
        " columns in the table frame.")
endMethod
```

▪

nKeyFields method

See also Example UIObject Type

Returns the fields in the current index.

Syntax

`nKeyFields ( ) LongInt`

Description

nKeyFields returns the number of fields in the current index associated with a UIObject.

▪

nKeyFields example

See the example for **nFields**.

▪

nRecords method

See also Example UIObject Type
Beginner

Returns the number of records in a table.

Syntax

```
nRecords ( ) LongInt
```

Description

nRecords returns the number of records in the table bound to a table frame, multirecord object, or field object. You can also examine the NRecords property for an object to find the number of records in the table bound to that object. Either operation can take a long time for dBASE tables and large Paradox tables.

The **nRecords** method and the NRecords property respect the limits of restricted views. If a table-based object is the detail table in a one-to-many relationship, **nRecords** reports the number of linked detail records, not the total number of records in the entire table.

Note: When you call **nRecords** after setting a filter, the returned value does not represent the number of records in the filtered set. To get that information, attach a TCursor to the UIObject and call **cCount.** When you call **nRecords** after setting a range, the returned value represents the number of records in the set defined by the range.

When working with a Paradox table, **nRecords** returns the number of records in the underlying table, not the number of records displayed in the object. For example, suppose the *Customer* table contains 100 records, but a table frame bound to the *Customer* table displays 5 records. This method would return 100, not 5.

When working with a dBASE table, **nRecords** counts deleted records if the form displays them. If the form does not display them, deleted records are not counted. To make a form display deleted records, choose Form|Show Deleted, or call **action(DataShowDeleted)** or **action(DataToggleDeleted)**.

nRecords example

The following example moves to the midpoint of a table. Assume that a form contains a table frame named *LINEITEM* bound to the *LineItem* table, and a button called *MidWay*.

```
; MidWay::pushButton
method pushButton(var eventInfo Event)
var
  halfWay  LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecord(halfWay)

endMethod
```

■

pixelsToTwips method

See also Example UIObject Type

Converts screen coordinates from pixels to twips.

Syntax

```
pixelsToTwips ( const pixels Point ) Point
```

Description

pixelsToTwips converts the screen coordinates specified in *pixels* from pixels to twips. A pixel (the name comes from picture element) is a dot on the screen, and a twip is a device-independent unit equal to 1/1440 of a logical inch (1/20 of a printer's point).

pixelsToTwips example

The next example assumes that a form contains a two-inch square box named *twoSquare*. The *twoSquare* box contains two text boxes: *pixNum* to display the width of the box in pixels and *twipNum* to display the width in twips.

```
; twoSquare::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  twTopLeft,             ; top left point in twips
  twBottomRight,         ; bottom right point in twips
  pxTopLeft,             ; top left in pixels
  pxBottomRight,         ; bottom right in pixels
  selfPos      Point     ; current position property
endvar
self.getBoundingBox(twTopLeft, twBottomRight)    ; returns points in twips
twipNum.Text = twBottomRight.x() - twTopLeft.x() ; get the width in twips
pxTopLeft = TwipsToPixels(twTopLeft)             ; convert to pixels
pxBottomRight = TwipsToPixels(twBottomRight)
pixNum.Text = pxBottomRight.x() - pxTopLeft.x()  ; get the width in pixels
; cross check
twTopLeft = PixelsToTwips(pxTopLeft)      ; convert from pixels back to twips
twTopLeft.view("Top left in twips")       ; twTopLeft should match selfPos
selfPos = self.Position                   ; get selfPos, twips by default
selfPos.view("Position of box in twips")  ; show the result
endMethod
```

▪

postAction method

See also Example UIObject Type

Posts an action to an action queue for delayed execution.

Syntax

```
postAction ( const actionId SmallInt )
```

Description

postAction works like **action**, except that the action is not executed immediately. Instead, the action specified by *actionID* is posted to an action queue at the time of the method call; Paradox waits until a yield occurs (for example, by the current method completing execution or by a call to **sleep**).

The value of *actionID* can be a user-defined action constant or a constant from one of the following Action classes:

ActionDataCommands

ActionEditCommands

ActionFieldCommands

ActionMoveCommands

ActionSelectCommands

▪

postAction example

The following example demonstrates how to store a value from a calculated field into a table. In this example, an unbound calculated field object named *fldLineTotal* calculates the line total. Whenever the calculation occurs, **postAction** is used to send a custom user action that posts the value to a table frame bound to the *Lineitem* table.

The following code defines the calculation for the calculated field.

```
;fldLineTotal :: Calculation
[LINEITEM.SELLING PRICE]*[LINEITEM.QTY]   ;Calculated field.
```

The following code is attached to the field object's built-in **newValue** method.

```
;fldLineTotal :: newValue
method newValue(var eventInfo Event)
   if Qty.isEdit() then            ;If edit mode,
      Qty.postAction(UserAction + 1)      ;send a custom user
   endIf                         ;action to QTY.
endmethod
```

The following code is attached to the table frame's built-in **action** method.

```
;recTFrame :: action
method action(var eventInfo ActionEvent)
   if eventInfo.id() = UserAction + 1 then   ;If ID is user
      dmPut("LINEITEM", "Total", Total.value)   ;action, then
      Qty.postRecord()               ;post changes.
   endIf
endmethod
```

▪

postRecord method

See also Example UIObject Type
Beginner

Posts a pending record to a table.

Syntax

```
postRecord ( ) Logical
```

Description

postRecord returns True if the current record is successfully posted to the underlying table; otherwise, it returns False. **postRecord** does not unlock a locked record.

postRecord has the same effect as the action constant DataPostRecord, so the following statements are equivalent:

```
obj.postRecord()
obj.action(DataPostRecord)
```

▪

postRecord example

The following example locates a record, attempts to lock it with **lockRecord**, then checks the status of the lock with **recordStatus**. The method changes the record and posts it with **postRecord**. Assume that a form contains a table frame bound to the *Customer* table, and a button named *lockButton*.

```
; lockButton::pushButton
method pushButton(var eventInfo Event)
var
  obj  UIObject
endVar
obj.attach(CUSTOMER)
obj.locate("Name", "Sight Diver")
if thisForm.Editing then
  if NOT obj.lockRecord() then
     msgStop("Lock failed", "recordStatus(\"Locked\") is " +
             String(obj.recordStatus("Locked")))
  else
     msgStop("Lock succeeded", "recordStatus(\"Locked\") is " +
             String(obj.recordStatus("Locked")))
     obj.custRec."Name" = "Right Diver"  ; quotes on Name indicates
                                         ; field name instead of property
     obj.postRecord()
     message("Record is locked: ", obj.custRec.locked)
  endIf
else
  msgInfo("Status", "You must be in edit mode to lock and change records.")
endIf
endMethod
```

▪

priorRecord method

See also Example UIObject Type
Beginner

Moves to the previous record in a table.

Syntax

`priorRecord ( ) Logical`

Description

priorRecord sets the current record to the previous record in a table. It returns an error if the insertion point is already at the first record.

priorRecord has the same effect as the action constant DataPriorRecord, so the following statements are equivalent:

```
obj.priorRecord()
obj.action(DataPriorRecord)
```

priorRecord example

The following example moves to the prior record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form; *moveToPrior* is a button on the same form.

```
; moveToPrior::pushButton
method pushButton(var eventInfo Event)
if NOT CUSTOMER.atFirst() then
  CUSTOMER.priorRecord()  ; move to the previous record
  ; same as CUSTOMER.action(DataPriorRecord)
  msgInfo("What record?", CUSTOMER.recno)
else
  msgInfo("Status", "Already at the first record.")
endIf
endMethod
```

pushButton method

See also Example UIObject Type

Generates a pushButton Event and sends it to an object.

Syntax

```
pushButton ( ) Logical
```

Description

pushButton constructs a **pushButton** Event and calls the built-in **pushButton** method of an object with that event.

pushButton example

The following example sends a **pushButton** event to *buttonTwo* on the same form:

```
; sendPushButton::pushButton
method pushButton(var eventInfo Event)
; send a pushButton to buttonTwo
buttonTwo.pushButton()
endMethod
```

▪

recordStatus method

See also Example UIObject Type

Reports about the status of a record.

Syntax

`recordStatus ( const statusType String ) Logical`

Description

recordStatus returns True or False to a question about the status of a record. Use the argument *statusType* to specify the status in question, where *statusType* is "New", "Locked", or "Modified".

"New" means the record has just been inserted into the table. "Locked" means a lock (implicit or explicit) has been placed on the record. "Modified" means at least one of the field values has been changed. You can obtain similar information about the current record by examining the Inserting, Locked, Focus, and Touched properties for the record.

▪

recordStatus example

The following example locates a record, attempts to lock it with **lockRecord**, then checks the status of the lock with **recordStatus**. The method changes the record and unlocks it with **unlockRecord**. Assume that a form contains a table frame bound to the *Customer* table, and a button named *lockButton*.

```
; lockButton::pushButton
method pushButton(var eventInfo Event)
var
  Cust   UIObject                    ; to attach to table frame
  newKey Number
endVar

Cust.attach(CUSTOMER)                ; attach to CUSTOMER table frame
Cust.locate("Name", "Sight Diver")   ; find the record
if NOT thisForm.editing then         ; check if form is in Edit mode
  msgInfo("Status", "You must be in Edit mode for this operation.")
else
  if NOT Cust.lockRecord() then      ; try to lock the record
     msgStop("Status", "Lock Failed. recordStatus(\"Locked\") is " +
             String(Cust.recordStatus("Locked")))
  else
     msgInfo("Record locked?", Cust.recordStatus("Locked"))
     newKey = 1384
     Cust.custRec."Customer No" = newKey   ; change the key value
     msgInfo("Record modified?", Cust.recordStatus("Modified"))
     Cust.unlockRecord()             ; try to unlock the record▪if it
                                     ; causes a keyviol, Paradox
                                     ; leaves record locked
     if Cust.recordStatus("Locked") then

       msgInfo("Status", "Record was a key violation. Changing key.")
       newKey = 1451
       Cust.custRec."Customer No" = newKey ; change to a new key
       Cust.postRecord()                   ; post it
       ; record will "fly away" to a new position based on key
     endIf
     Cust.locate("Customer No", newKey)    ; find the "fly away"
  endIf
endIf
endMethod
```

▪

resync method

See also Example UIObject Type

Resynchronizes an object to a TCursor.

Syntax

```
resync ( const tc TCursor ) Logical
```

Description

resync changes the current record pointer of a UIObject to the current record of the TCursor *tc*. When you resynchronize a table object to a TCursor, the table's filters and indexes will be changed to those of the TCursor. (For dBASE tables, the table will also take the Show Deleted setting of the TCursor.)

Note: **resync** only works when the UIObject and the TCursor are associated with the same table.

-

resync example

See the example for **insertBeforeRecord**.

▪

rgb method

See also Example UIObject Type

Defines a color.

Syntax

```
rgb ( const red SmallInt, const green SmallInt, const blue SmallInt ) LongInt
```

Description

rgb defines a color based on the values of *red*, *green*, and *blue*, which can be integers ranging from 0 to 255, or Colors constants.

■

rgb example

The following example uses **rgb** to set the color of boxes as they're created. The method creates a color palette. Assume that the titles exist on the form in the appropriate locations. The form has one button, named *showPalette*.

```
; drawPalette::pushButton
method pushButton(var eventInfo Event)
var
  palAr Array[5] SmallInt   ; array to hold rgb values
  setBaseX       LongInt    ; base position
  setBaseY       LongInt    ; base position
  ui             UIObject   ; handle to create boxes
endVar
const
  horizInc = 1440           ; amount to move horizontally (twips)
  vertInc =  1080           ; amount to move vertically
endConst

palAr[1] = 0
palAr[2] = 64
palAr[3] = 128
palAr[4] = 192
palAr[5] = 255

for i from 1 to palAR.size()        ; reds(diagonal position)
  setBaseX = 720 + ((i - 1) * 150)  ; change base as i increases
  setBaseY = 720 + ((i - 1) * 150)
  for j from 1 to palAR.size()      ; greens (vertical positioning)
    for k from 1 to palAR.size()    ; blue   (horizontal positioning)
      ui.create(boxTool, setBaseX + (horizInc * (k - 1)),
                setBaseY + (vertInc * (j - 1)), 250, 250)
      ; set the color using rgb and values from array
      ui.Color = rgb(palAr[i], palAr[j], palAr[k])
      ui.Visible = Yes
    endfor                          ; k (blue, horizontal)
  endfor                            ; j (green, vertical)
endfor                              ; i (red, diagonal)

endMethod
```

▪

sendToBack method

See also Example UIObject Type

Displays an object behind other objects.

Syntax

```
sendToBack ( )
```

Description

sendToBack moves a UIObject to the back drawing layer of a window, displaying it behind other objects. (If you're using a form as a UIObject, this method displays the form window behind other windows.)

This method works in design mode and run mode; you do not have to select the object. Paradox moves the object to the back, so it appears to be behind (below) other objects. This might not be noticeable unless the objects partially overlap each other. You might want to bring an object to the back of the stack of objects if

- You have objects that overlap each other
- You want to rearrange the tab order

Note: When you change the front-to-back positions of objects, you change their tab order, because objects always tab from back to front.

▪

sendToBack example

In the following example, assume a form contains two multirecord objects that occupy the same location and size on a form. Two buttons toggle between each multirecord object: *btnShowVendors* uses **sendToBack** to send the *STOCK* multirecord object to the background; the *VENDORS* multirecord object is in front. *btnShowStock* uses **sendToBack** to send the *VENDORS* multirecord object to the background; the *STOCK* multirecord object is in front.

The following code is attached to *btnShowVendors*.

```
;btnShowVendors :: pushButton
method pushButton(var eventInfo Event)
   STOCK.sendToBack()    ; Send the VENDORS MRO to the back
   Vendor_No.moveTo()    ; so the STOCK MRO may be seen.
endmethod
```

The following code is attached to *btnShowStock*.

```
;btnShowStock :: pushButton
method pushButton(var eventInfo Event)
   VENDORS.sendToBack()    ; Send the STOCK MRO to the back
   Stock_No.moveTo()    ; so the VENDORS MRO may be seen.
endmethod
```

▪

setGenFilter method

See also Example UIObject Type

Specifies conditions for including records in a field, table frame, or multirecord object.

Syntax

```
1. setGenFilter ( [ idxName String, [ tagName String, ] ] criteria DynArray [
] AnyType ) Logical
2. setGenFilter ( [ idxName String, [ tagName String, ] ] criteria Array[ ]
AnyType [ , fieldId Array[ ] AnyType ] ) Logical
```

Description

setGenFilter specifies conditions for including records in a table frame or multirecord object. Records that meet all the specified conditions are included; records that don't are filtered out. Unlike **setRange**, this method does not require an indexed table.

In syntax 1, the DynArray *criteria* specifies fields and conditions as follows: the index is the field name or field number, and the item is the criteria expression. For example, the following code specifies criteria based on the values of three fields.

```
criteriaDA[1]      = "Widget" ; The value of the first field in the table is
Widget.
criteriaDA["Size"] = "> 4"    ; The value of the field named Size is greater
than 4.
criteriaDA["Cost"] = ">= 10.95, < 22.50" ; The value of the field named Cost
is greater than or equal to 10.95 and less than 22.50.
```

If the DynArray is empty or contains at least one empty item, any existing filter criteria are removed.

In syntax 2, the Array *criteria* specifies conditions, and the optional Array *fieldId* specifies field names and/or field numbers. If you omit *fieldID*, conditions are applied to fields in the order they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to the second field, and so on). The following example fills arrays for syntax 2 to specify the same criteria as the example for syntax 1.

```
criteriaAR[1] = "Widget"
criteriaAR[2] = "> 4"
criteriaAR[3] = ">= 10.95, < 22.50"
fieldAR[1] = 1
fieldAR[2] = "Size"
fieldAR[3] = "Cost"
```

If the Array is empty or contains at least one empty item, any existing filter criteria are removed.

For both syntaxes, *idxName* specifies an index name (Paradox and dBASE tables) and *tagName* specifies a tag name (dBASE tables only). If you use these optional items, the index (and tag) are applied to the underlying table before the filtering criteria.

This method fails if the current record cannot be committed.

Note: If you use **setGenFilter** on a UIObject in a running report, the filter does not take effect until the next time you run the report. For example, the following code runs a report, then sets a filter, but the filter has no effect until the code switches the report into design mode, then switches it back into run mode.

```
method pushButton(var eventInfo Event)
   var
      reOrders   Report
      daCriteria   DynArray[] AnyType
   endVar

   reOrders.open("orders")

   daCriteria["OrderNo"] = "> 1234"

; Assume the report contains a table frame bound to the Orders table.
; This statement has no effect because the report is in run mode.
   reOrders.ORDERS.setGenFilter(daCriteria)

   reOrders.design()
   reOrders.run() ; Now the filter takes effect.
endMethod
```

▪

setGenFilter example

In this example, the **pushButton** method for a button named *balanceDueBtn* uses **setGenFilter** to filter a table frame on a form. It filters the *ORDERS* table frame to show only those orders with a positive balance due.

```
;balanceDueBtn :: pushButton
method pushButton(var eventInfo Event)
   var
      dyn               DynArray[] String
      stField, stData   String
   endVar

   stField = "Balance Due"
   stData = "> 0"
   dyn[stField] = sData

   ORDERS.setGenFilter(dyn)   ; ORDERS is a detail table frame.
endmethod
```

setPosition method

See also Example UIObject Type

Sets the position of an object.

Syntax

```
setPosition ( const x LongInt, const y LongInt, const w LongInt, const h
LongInt)
```

Description

setPosition sets the position of an object on the screen. Variables *x* and *y* specify the coordinates (in twips) of the upper left corner of the object. Variables *w* and *h* specify the width and height (in twips) of the object. If the object is not specified, self is implied.

Note: This method does not work with forms as UIObjects. To set the position of a form, use Form::**setPosition.**

You can also set and examine an object's position and size with the Position and Size properties. For instance,

```
self.Position = Point(100, 150)
self.Size = Point(2000, 2500)
```

is the same as

```
self.setPosition(100, 150, 2000, 2500)
```

To ObjectPAL, the screen is a two-dimensional grid, with the origin (0, 0) at the upper left corner of an object's container, positive x-values extending to the right, and positive y-values extending down.

For dialog boxes and for the Paradox Desktop application, the position is given relative to the entire screen; for forms, reports, and table windows, the position is given relative to the Paradox Desktop.

setPosition example

The following example moves a circle across the screen in response to timer events. The **pushButton** method for *toggleButton* uses **setTimer** and **killTimer** to start or stop a timer, depending on the condition of the button. When the timer starts, it issues a timer event every 100 milliseconds. Each timer event causes *toggleButton's* **timer** method to execute. The **timer** method gets the current position of the ellipse with **getPosition**, then moves it 100 twips to the right with **setPosition**.

Following is the code for *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
; label for button was renamed to buttonLabel
if buttonLabel = "Start Timer" then    ; if stopped, then start
  buttonLabel = "Stop Timer"           ; change label
  self.setTimer(10)                    ; start the timer
else                                   ; if started, then stop
  buttonLabel = "Start Timer"          ; change label
  self.killTimer()                     ; stop the timer
endIf

endMethod
```

Following is the code for *toggleButton*'s **timer** method:

```
; toggleButton::timer
method timer(var eventInfo TimerEvent)
var
  ui          UIObject
  x, y, w, h  SmallInt
endVar
ui.attach(floatCircle)          ; attach to the circle
ui.getPosition(x, y, w, h)      ; assign coordinates to vars
if x  4320 then                 ; if not at left edge of area
  ui.setPosition(x + 100, y, w, h)  ; move to the left
else
  ui.setPosition(1440, y, w, h)     ; return to the right
endIf
endMethod
```

▪

setProperty method

See also Example UIObject Type

Sets a property to a specified value.

Syntax

`setProperty ( ``const `***`propertyName`***` String, const `***`propertyValue`***` AnyType `**`)`**

Description

setProperty sets the *propertyName* property of an object to *propertyValue*. If the object does not have a property *propertyName*, or if *propertyValue* is invalid, an error results.

setProperty is an alternative to setting a property directly; it's useful when *propertyName* is a variable. Otherwise, access the property directly, for example,

```
aBox.Color = Red
```

instead of

```
aBox.setProperty("Color", Red)
```

setProperty example

The following example creates a dynamic array, indexed by property names, to contain property values. The array is filled by using the array's index as the argument to the **getProperty** command. The method changes one of the values of the properties and resets the object's properties from the dynamic array with the **setProperty** method.

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  propNames DynArray[] AnyType    ; to hold property names & values
  arrayIndex           String     ; index to dynamic array
endVar

propNames["Color"] = ""
propNames["Visible"] = ""
propNames["Name"] = ""

foreach arrayIndex in propNames
  propNames[arrayIndex] = self.getProperty(arrayIndex)
endforeach

propNames["Color"] = "DarkBlue"

foreach arrayIndex in propNames
  self.setProperty(arrayIndex, propNames[arrayIndex])
endforeach

endMethod
```

■

setRange method

See also Example1 Example2 UIObject Type

Specifies a range of records to include in a field, table frame, or multirecord object.

Syntax

```
1. setRange ( [ const exactMatchVal AnyType] * [ , const minVal AnyType,
const maxVal AnyType ] ) Logical
2. setRange ( rangeVals Array[ ] AnyType ) Logical
```

Description

setRange specifies conditions for including a range of records. Records that meet the conditions are included; records that don't are excluded. **setRange** compares the criteria you specify with values in the corresponding fields of a table's index; it fails if the current record cannot be committed or if the table is not indexed. Calling **setRange** without any arguments resets the range criteria to include the entire table.

Note This method replaces **setFilter** included in earlier versions: both functionality and performance are enhanced. Code that calls **setFilter** will continue to execute as before.

In syntax 1, to set a range based on the value of the first field of the index, specify values in *minVal* and *maxVal*. For example, the following statement checks values in the first field of the index of each record:

```
tblObj.setRange(14, 88)
```

If a value is less than 14 or greater than 88, that record is excluded. To specify an exact match on the first field of the index, assign *minVal* and *maxVal* the same value. For example, the following statement excludes all values except 55:

```
tblObj.setRange(55, 55)
```

You can set a range based on the values of more than one field. To do so, specify exact matches *except* for the last one in the list. For example, the following statement looks for exact matches on "Borland" and "Paradox" (assuming they are the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field:

```
tblObj.setRange("Borland", "Paradox", 100, 500)
```

In syntax 2, you can pass an array of values to specify the range criteria, as listed in the following table.

Number of array items	Range specification
No items (empty array)	Resets range criteria to include the entire table.
One item	Specifies a value for an exact match on the first field of the index.
Two items	Specifies a range for the first field of the index.
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size *n*, specify exact matches on the first *n*-2 fields of the index. The last two array items specify a range for the *n*-1 field of the index.

▪

setRange example 1

For the following example, assume that the first field in *Lineitem*'s key is "Order No." and you want to know the total for order number 1005. When you press the *getDetailSum* button, the **pushButton** method limits the number of records included in the LINEITEM object to those with 1005 in the first key field.

```
; getDetails::pushButton
method pushButton(var eventInfo Event)
var
  tblObj  UIObject
endVar
if tblObj.attach(LINEITEM) then

  ; this limits tblObj's view to records that have
  ; 1005 as their key value (Order No. 1005).
  tblObj.setRange(1005, 1005)
  ; now display the number of records for Order No. 1005
  msgInfo("Total records for order 1005", tblObj.nRecords())
else
  msgStop("Sorry", "Can't attach to table.")
endIf
endMethod
```

▪

setRange example 2

This example shows how to call **setRange** with a criteria array that contains more than three items. The following code makes a table frame display orders from a person with a specific first name, middle initial, and last name, and an order quantity ranging from 100 to 500 items. This example assumes that the *PartsOrd* table is indexed on the FirstName, MiddleInitial, LastName, and Qty fields.

```
; setQtyRange::pushButton
method pushButton(var eventInfo Event)
   var
      arRangeInfo   Array[5] AnyType
   endVar

   arRangeInfo[1] = "Frank"       ; FirstName (exact match)
   arRangeInfo[2] = "P."          ; MiddleInitial (exact match)
   arRangeInfo[3] = "Borland"     ; LastName (exact match)
   arRangeInfo[4] = 100           ; Minimum qty value
   arRangeInfo[5] = 500           ; Maximum qty value

   PartsOrd.setRange(arRangeInfo) ; PartsOrd is a table frame
endMethod
```

▪

setTimer method

See also   Example   UIObject Type

Starts the timer for an object.

Syntax

`setTimer ( const milliSeconds LongInt [ , const repeat Logical ] )`

Description

setTimer starts a timer for an object. The timer interval (in milliseconds) is specified using *milliSeconds*. The optional argument *repeat* specifies if the timer automatically repeats. If *repeat* is True or omitted, the timer repeats; otherwise, the timer event is sent once. Usually, **setTimer** is attached to an object's **open** method, and the object's response is defined in its **timer** method.

Note: Windows allows a maximum of 16 timers for all applications. However, Paradox has no limit. System resources may limit the number of timers you can set, and you may run out of Windows timers, but Paradox is not restricted by the 16-timer limit.

setTimer example

The following example moves a circle across the screen in response to timer events. The **pushButton** method for *toggleButton* uses **setTimer** and **killTimer** to start or stop a timer, depending on the condition of the button. When the timer starts, it issues a timer event every 100 milliseconds. Each timer event causes *toggleButton's* **timer** method to execute. The **timer** method gets the current position of the ellipse with **getPosition**, then moves it 100 twips to the right with **setPosition**.

The following code is for *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
if buttonLabel = "Start Timer" then   ; if stopped, then start
  buttonLabel = "Stop Timer"          ; change label
  self.setTimer(10)                   ; start the timer
else
  buttonLabel = "Start Timer"         ; change label
  self.killTimer()                    ; stop the timer
endIf

endMethod
```

The following code is for *toggleButton*'s **timer** method:

```
; toggleButton::timer
method timer(var eventInfo TimerEvent)
var
  ui           UIObject
  x, y, w, h   SmallInt
endVar
ui.attach(floatCircle)           ; attach to the circle
ui.getPosition(x, y, w, h)       ; assign coordinates to vars
if x  4320 then                  ; if not at left edge of area
  ui.setPosition(x + 100, y, w, h)   ; move to the left
else
  ui.setPosition(1440, y, w, h)      ; return to the right
endIf
endMethod
```

▪

skip method

See also Example UIObject Type

Moves forward or backward a specified number of records in a table.

Syntax

```
skip ( const nRecords LongInt ) Logical
```

Description

skip sets the current record to the record *nRecords* from the current record. You'll get an error if **skip** tries to move beyond the limits of the table.

Positive values for *nRecords* move forward through the table (*nRecords* = 1 is the same as **nextRecord**), negative values move backward (*nRecords* = -1 is the same as **priorRecord**), and setting *nRecords* to 0 doesn't move (*nRecords* = 0 is the same as **currRecord**).

▪

skip example

The following example fills a table with a sampling of records from the *Orders* table. Assume that the table *SampOrd* already exists with the same structure as *Orders*. The *createSampling* button, whose **pushButton** method is shown below, exists on a form along with a table frame bound to *Orders*. The method moves the insertion point through the *Orders* table, skips a random number of records, and copies the record it lands on to the sampling table.

```
; createSampling::pushButton
method pushButton(var eventInfo Event)
var
  ordSampleTC         TCursor       ; handle to sampling table
  copyRec Array[]     String        ; holds record copied from Orders
  randInt             SmallInt      ; random number to skip
  OrdObj              UIObject      ; handle to Orders
endVar

ordObj.attach(ORDERS)               ; attach to ORDERS table frame
ordObj.home()                       ; move to the first record
if ordSampleTC.open("OrdSamp.db")  then
  ordSampleTC.empty()               ; clear out sampling table
  ordSampleTC.edit()                ; start editing
  while NOT OrdObj.atLast()
    randInt = int(rand() * 20) + 1  ; create an integer between 1 and 20
    randInt.view()                  ; show the number
    OrdObj.skip(randInt)            ; skip a random number of records
    OrdObj.copyToArray(copyRec)     ; get the record
    ordSampleTC.insertRecord()      ; make a space for it
    ordSampleTC.copyFromArray(copyRec)  ; insert the record
  endwhile
  ordSampleTC.endEdit()             ; end editing
  msgInfo("Status", "OrdSamp table now has " +
           String(ordSampleTC.nRecords()) + " records.")
  ordSampleTC.close()               ; close it out
else
  msgStop("Oops", "Sorry. Couldn't find OrdSamp table.")
endIf
endMethod
```

switchIndex method

See also Example UIObject Type

Specifies another index to use to view the records in a table.

Syntax

```
1. switchIndex ( [ const indexName String ] [ , const stayOnRecord Logical ] ) Logical
2. switchIndex ( [const indexFileName String ] [ , const tagName String [ , const stayOnRecord Logical ] ] ) Logical
```

Description

switchIndex specifies in *indexName* an index file to use with a table. In syntax 1, *indexName* specifies an index to use with a Paradox table. If you omit *indexName*, the table's primary index is used.

Syntax 2 is for dBASE tables, where *indexFileName* can specify a .NDX file or a .MDX file, and optional argument *tagName* specifies an index tag in a production index (.MDX) file.

In both syntaxes, if optional argument *stayOnRecord* is Yes, this method maintains the current record after the index switch; if it is No, the first record in the table becomes the current record. If omitted, *stayOnRecord* is No by default.

For information on indexes, see About keys and indexes in tables in the User's Guide help.

switchIndex example

For the following example, assume that *Customer* is a keyed Paradox table that has a secondary index named "NameAndState". This example attaches to a table frame bound to *Customer*, and calls **switchIndex** to switch from the primary index to the "NameAndState" index.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblObj UIObject
endvar

tblObj.attach(CUSTOMER)               ; attach to Customer
tblObj.switchindex("NameAndState")    ; switch to index NameAndState
tblObj.home()                         ; make sure we're on the first record
msgInfo("First Record", tblObj."Name")  ; display value in Name field
; quotes around "Name" distinguish field name from property name
endMethod
```

▪

twipsToPixels method

See also Example UIObject Type

Converts screen coordinates from twips to pixels.

Syntax

```
twipsToPixels ( const twips Point ) Point
```

Description

twipsToPixels converts the screen coordinates specified in *twips* from twips to pixels. A pixel (the name comes from picture element) is a dot on the screen, and a twip is a device-independent unit equal to 1/1440 of a logical inch (1/20 of a printer's point).

twipsToPixels example

See the example for **pixelsToTwips**.

▪

unDeleteRecord method

See also Example UIObject Type

Undeletes the current record from a dBASE table.

Syntax

unDeleteRecord ( ) Logical

Description

unDeleteRecord undeletes the current record of a dBASE table. This operation can only be successful if **showDeleted** has been set True, the current record is a deleted record, and the table object is in Edit mode.

▪

unDeleteRecord example

See the example for TCursor::**unDeleteRecord**.

▪

unlockRecord method

See also Example UIObject Type
Beginner

Removes a write lock from the current record.

Syntax
`unlockRecord ( ) Logical`

Description
unlockRecord returns True if it successfully removes an explicit write lock on the current record; otherwise, it returns False.

Note: The Locked property is a read-only property. You can examine the property to find out whether an object is locked, but you can't change the property to lock or unlock an object.

▪

unlockRecord example

See the example for **recordStatus**.

▪

view method

See also Example UIObject Type
Beginner

Displays the value of an object in a dialog box.

Syntax

```
view ( [ const title String ] )
```

Description

view displays the value of an object in a dialog box. Paradox suspends method execution until you close the dialog box. You have the option to specify, in *title*, a title for the dialog box. If you omit *title*, the title is the data type of the value.

This method works only with the following UIObjects:

- Buttons as checkboxes or radio buttons.
- Unbound fields only as lists or radio buttons.
- Fields bound to a table; the field's data type can be any data type except Memo and Graphic.

Calling **view** with any other UIObject causes a run-time error.

▪

view example

For the following example, assume that a form contains a table frame, named *CUSTOMER* bound to the *Customer* table, and a button. The following code is attached to the button's **pushButton** method. It creates an array of seven UIObjects, then tries to view each item in the array.

```
;  page::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
   obj          UIObject
   arr Array[7] UIObject
   i            SmallInt
endVar
arr[1].attach(CUSTOMER.Phone) ; the Phone field (A15) in the table frame
                              ; shows the phone number
arr[2].attach(aGraphic)       ; a bitmap (invalid)
arr[3].attach(someText)       ; a text object (invalid)
arr[4].attach(someList)       ; an unbound list field
                              ; shows the list item selected
arr[5].attach(someUnField)    ; an unbound field (invalid)
arr[6].attach(someRadio)      ; an unbound field as a radio button
                              ; shows the value of the active radio button
arr[7].attach(someButton)     ; an unbound field as a checkbox
                              ; True if checked, otherwise False
for i from 1 to arr.size()
  arr[i].view(arr[1].Class + ": Item " + String(i))
endFor
endMethod
```

wasLastClicked method

See also Example UIObject Type

Tells if an object was the last object to receive a mouse click.

Syntax

`wasLastClicked ( ) Logical`

Description

wasLastClicked returns True if an object was the last object to receive a mouse click; otherwise, it returns False. This method can be used only with objects in the current form.

wasLastClicked example

The following code is attached to the **mouseUp** method for an object called *boxOne*. If *boxOne* received the click, the message appears; if *boxOne* was sent a **mouseUp** event from another object, the method beeps instead.

Following is the code for *boxOne*'s **mouseUp** method:

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
if self.wasLastClicked() then
  msgInfo("Hey!", "Quit clicking me.")  ; method invoked by clicking
else
  beep()                                ; method invoked indirectly
endIf
endMethod
```

Following is the code for *sendAClick*'s **mouseUp** method:

```
; sendAClick::mouseUp
method mouseUp(var eventInfo MouseEvent)
boxOne.mouseUp(eventInfo)  ; when boxOne's mouseUp gets this,
                           ; it will beep
endMethod
```

▪

wasLastRightClicked method

See also   Example   UIObject Type

Tells if an object was the last object to receive a right mouse click.

Syntax

`wasLastRightClicked ( )` `Logical`

Description

wasLastRightClicked returns True if an object was the last object to receive a right mouse click; otherwise, it returns False. This method can be used only with objects in the current form.

▪

wasLastRightClicked example

The following is attached to the **mouseRightUp** method for an object called *circleOne*. If the ellipse received the right click, the message displays; if the ellipse was sent a **mouseRightUp** event from another object, the method beeps instead.

Following is the code for *circleOne*'s **mouseUp** method:

```
; circleOne::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
if self.wasLastRightClicked() then
  ; method invoked by right-click
  msgInfo("Right-click", "Go click on someone your own size.")
else
  beep()       ; method invoked indirectly
endIf
endMethod
```

Following is the code for *sendARightClick*'s **mouseUp** method:

```
; sendARightClick::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
circleOne.mouseRightUp(eventInfo)  ; when circleOne gets this,
                                   ; it will beep
endMethod
```

ValueEvent type

ValueEvent methods control field value changes. The **changeValue** built-in event method is the only method triggered by a ValueEvent. The built-in **newValue** method is not called with a ValueEvent; **newValue** takes an Event instead.

Do not confuse **changeValue** with **newValue**. The built-in **changeValue** method is called when the value of a field is about to change. **changeValue** gives you a chance to check the value and decide whether you want to post it. The built-in **newValue** method reports when a field has received a new value; **newValue** is usually called after the fact (fields defined as buttons and lists behave differently). Also note that the built-in **newValue** method is *not* the same as the **newValue** method for the ValueEvent type.

The ValueEvent type includes several derived methods from the Event type.

Methods for the ValueEvent type

Event	ValueEvent
errorCode	**newValue**
getTarget	**setNewValue**
isFirstTime	
isPreFilter	
isTargetSelf	
reason	
setErrorCode	
setReason	

▪

newValue method

See also Example ValueEvent Type
Beginner

Returns the unposted new value of a ValueEvent.

Syntax

```
newValue ( ) AnyType
```

Description

newValue returns the new value to be assigned to a field for a ValueEvent. The new value is not yet assigned to the field so the following two statements may return different values:

```
field.Value
eventInfo.newValue()
```

Note: This method is not the same as the built-in **newValue** method.

▪

newValue example

In the following example, the **changeValue** method for the creditLimit field checks the old value and the new value to see if there is more than a 25% change. If the difference between the old and new values is too large, the method blocks the change. Assume that *creditLimit* is an unbound field on a form, and that there is at least one other field to move to.

```
; creditLimit::changeValue
method changeValue(var eventInfo ValueEvent)
var
  oldVal,
  newVal   Number
endVar
oldVal = self.Value              ; the property may be different
newVal = eventInfo.newValue()    ; than the new value
if (newVal > oldVal) AND (oldVal <> 0) then
  if (newVal - oldVal)/oldVal > 0.25 then
    msgStop("Stop", "You are not allowed to increase the " +
                    "credit limit more than 25%.")
    self.action(EditUndoField)  ; ▪use this to restore old value
    eventInfo.setErrorCode(CanNotDepart)   ; block departure
  endIf
endIf
endMethod
```

▪

setNewValue method

See also Example ValueEvent Type

Specifies a value to set for a ValueEvent.

Syntax

setNewValue (const ***newValue*** AnyType **)**

Description

setNewValue specifies in *newValue* a value to set for a ValueEvent. The data type of the value supplied in *newValue* should be consistent with the field's type.

▪

setNewValue example

In the following example, assume a form contains the field *authorAbbrToName*, as well as at least one other field. When the user enters an author abbreviation, then moves off the field, the **changeValue** method fills in the full author name.

```
; authorAbbrToName::changeValue
method changeValue(var eventInfo ValueEvent)
var
  abbrValue,
  fullValue String
endVar

abbrValue = upper(eventInfo.newValue())  ; get the value and convert
                                         ; to uppercase
; user enters an abbreviation--change to full name
switch
  case abbrValue = "AC" : fullValue = "Agatha Christie"
  case abbrValue = "SP" : fullValue = "Sara Paretsky"
  case abbrValue = "MHC": fullValue = "Mary Higgins Clark"
  case abbrValue = "FK" : fullValue = "Faye Kellerman"
  case abbrValue = "SG" : fullValue = "Susan Grafton"
  case abbrValue = "AF" : fullValue = "Antonia Fraser"
  otherwise : fullValue = "Author Unknown"
endswitch

eventInfo.setNewValue(fullValue)
endMethod
```

▪

Alphabetical list of ObjectPAL methods

{button A,JI(`',`gloss_a')} {button B,JI(`',`gloss_b')} {button C,JI(`',`gloss_c')} {button D,JI(`',`gloss_d')} {button E,JI(`',`gloss_e')} {button F,JI(`',`gloss_f')} {button G,JI(`',`gloss_g')} {button H,JI(`',`gloss_h')} {button I,JI(`',`gloss_i')} {button J,JI(`',`gloss_k')} {button K,JI(`',`gloss_k')} {button L,JI(`',`gloss_l')} {button M,JI(`',`gloss_m')} {button N,JI(`',`gloss_n')} {button O,JI(`',`gloss_o')} {button P,JI(`',`gloss_p')} {button Q,JI(`',`gloss_q')} {button R,JI(`',`gloss_r')} {button S,JI(`',`gloss_s')} {button T,JI(`',`gloss_t')} {button U,JI(`',`gloss_u')} {button V,JI(`',`gloss_v')} {button W,JI(`',`gloss_w')} {button X,JI(`',`gloss_x')} {button Y,JI(`',`gloss_x')} {button Z,JI(`',`gloss_x')}

See also ▪

A

abs
accessRights
acos
actionClass
action (Form Type)
action (TableView Type)
action (UIObject Type)
addAddress
addAlias
addArray (Menu Type)
addArray (PopUpMenu Type)
AddAttachment
addBar
addBreak (Menu Type)
addBreak (PopUpMenu Type)
addButton
addLast
add (Table Type)
add (TCursor Type)
addPassword
addPopUp (Menu Type)
addPopUp (PopUpMenu Type)
addProjectAlias
addressBook
addressBookTo
addSeparator
addStaticText (Menu Type)
addStaticText (PopUpMenu Type)
addText (Menu Type)
addText (PopUpMenu Type)
advancedWildcardsInLocate
advMatch (String Type)
advMatch (TextStream Type)
aliasName
ansiCode
appendASCIIFix
appendASCIIVar

cCount (TCursor Type)
ceil
charAnsiCode
char
checkField
checkRow
chrOEM
chr
chrToKeyName
clearCheck
clipboardErase
clipboardHasFormat
close (Database Type)
close (DDE Type)
close (Form Type)
close (Library Type)
close (OleAuto Type)
close (Report Type)
close (Session Type)
close (System Type)
close (TCursor Type)
close (TextStream Type)
close (TableView Type)
cMax (Table Type)
cMax (TCursor Type)
cMin (Table Type)
cMin (TCursor Type)
cNpv (Table Type)
cNpv (TCursor Type)
commit
commitTransaction
compact (Table Type)
compact (TCursor Type)
compileInformation
constantNameToValue
constantValueToName
contains (Array Type)
contains (DynArray Type)
contains (Menu Type)
convertPointWithRespectTo
copyFromArray (TCursor Type)
copyFromArray (UIObject Type)
copy (FileSystem Type)
copy (Table Type)
copy (TCursor Type)

deleteDir
delete (Database Type)
delete (FileSystem Type)
delete (Table Type)
delete (UIObject Type)
deleteRecord (TCursor Type)
deleteRecord (UIObject Type)
deleteRegistryKey
deliver (Form Type)
design (Form Type)
design (Report Type)
desktopMenu
didFlyAway
disableBreakMessage
disablePreviousError
distance
dlgAdd
dlgCopy
dlgCreate
dlgDelete
dlgEmpty
dlgExport
dlgImportAsciiFix
dlgImportAsciiVar
dlgImport
dlgImportSpreadSheet
dlgImportTable
dlgNetDrivers
dlgNetLocks
dlgNetRefresh
dlgNetRetry
dlgNetSetLocks
dlgNetSystem
dlgNetUserName
dlgNetWho
dlgRename
dlgRestructure
dlgSort
dlgSubtract
dlgTableInfo
dmAddTable
dmAttach (Form Type)
dmAttach (TCursor Type)
dmBuildQueryString
dmEnumLinkFields

enumControls
enumDataBaseTables
enumDataModel
enumDesktopWindowHandles
enumDesktopWindowNames
enumDriverCapabilities
enumDriverInfo
enumDriverNames
enumDriverTopics
enumEngineInfo
enumEnvironmentStrings
enumEvents
enumExperts
enumFamily
enumFieldNamesInIndex (Table Type)
enumFieldNamesInIndex (TCursor Type)
enumFieldNames (Table Type)
enumFieldNames (TCursor Type)
enumFieldNames (UIObject Type)
enumFieldStruct (Query Type)
enumFieldStruct (Table Type)
enumFieldStruct (TCursor Type)
enumFileList
enumFolder
enumFonts
enumFormats
enumFormNames
enumIndexStruct (Table Type)
enumIndexStruct (TCursor Type)
enumLocks (TCursor Type)
enumLocks (UIObject Type)
enumMethods
enumObjectNames
enumObjects
enumOpenDatabases
enumPrinters
enumProperties
enumRefIntStruct (Table Type)
enumRefIntStruct (TCursor Type)
enumRegistryKeys
enumRegistryValueNames
enumReportNames
enumRTLClassNames
enumRTLConstants
enumRTLErrors

G

H

isAltKeyDown
isAssigned (AnyType Type)
isAssigned (Database Type)
isAssigned (Query Type)
isAssigned (SQL Type)
isAssigned (Session Type)
isAssigned (Table Type)
isAssigned (TCursor Type)
isBelow
isBlank
isBlankZero
isCompileWithDebug (Form Type)
isContainerValid
isControlKeyDown (KeyEvent Type)
isControlKeyDown (MouseEvent Type)
isCreateAuxTables
isDesign
isDir
isEdit (TCursor Type)
isEdit (UIObject Type)
isEmpty (Query Type)
isEmpty (Table Type)
isEmpty (TCursor Type)
isEmpty (UIObject Type)
isEncrypted (Table Type)
isEncrypted (TCursor Type)
isErrorTrapOnWarnings
isExecuteQBELocal
isFile
isFirstTime
isFixed
isFixedType
isFromUI (KeyEvent Type)
isFromUI (MouseEvent Type)
isFromUI (MenuEvent Type)
isIgnoreCaseInLocate
isIgnoreCaseInStringCompares
isInside
isLastMouseClickedValid
isLastMouseRightClickedValid
isLeapYear
isLeftDown
isLeft
isLinked
isMaximized

O

P

S

U

V

W

X

Y-Z

■

ObjectPAL glossary

{button A,JI(`',`pglos_a')} {button B,JI(`',`pglos_b')} {button C,JI(`',`pglos_c')} {button D,JI(`',`pglos_d')} {button E,JI(`',`pglos_e')} {button F,JI(`',`pglos_f')} {button G,JI(`',`pglos_g')} {button H,JI(`',`pglos_h')} {button I,JI(`',`pglos_i')} {button J,JI(`',`pglos_k')} {button K,JI(`',`pglos_k')} {button L,JI(`',`pglos_l')} {button M,JI(`',`pglos_m')} {button N,JI(`',`pglos_n')} {button O,JI(`',`pglos_o')} {button P,JI(`',`pglos_p')} {button Q,JI(`',`pglos_q')} {button R,JI(`',`pglos_r')} {button S,JI(`',`pglos_s')} {button T,JI(`',`pglos_t')} {button U,JI(`',`pglos_t')} {button V,JI(`',`pglos_t')} {button W,JI(`',`pglos_t')} {button X,JI(`',`pglos_t')} {button Y,JI(`',`pglos_t')} {button Z,JI(`',`pglos_t')}

A
active
ANSI
application
argument
array
array element
ASCII

B
blank
braces
branching commands
breakpoint
bubbling
built-in event method

C
column
compound object
constant
container
containership
control structure
Ctrl+Break

D
data
Database
data type
DDE
deadlock
Debugger
Desktop
Display manager
DLL
dynamic array

E
Editor
encrypt
event

Active

A built-in object variable that represents the currently active object▪the last object to receive focus from a **moveTo** method. Typically, the active object is highlighted. Even when focus is removed from an object (for example, to activate another form), Active still refers to that object. Only when someone moves off that object is Active reset to the new object.

Don't underestimate the importance of Active, since general routines can be written to operate on the active object without really knowing which one it is. For example, suppose a form contains two table frames, each bound to a different table. The following statement automatically operates on the active table frame:

```
active.action(DataNextRecord)
```

alias

A name you assign to a full path name that makes it easier to access. For instance, you might assign the alias MYDIR to C:\DATA\DEMOAPP\. Then you refer to MYDIR instead of the full path name.

alpha operator (+)
Used to concatenate two alpha or memo fields.

ANSI

American National Standards Institute; a sequence of 8-bit codes that defines 256 standard characters, letters, numbers, and symbols. The ASCII character set (defined separately) consists of the first 128 ANSI characters.

application

1. An ObjectPAL type you use to get a handle (a unique variable identifier) for a Paradox application.
2. A group of forms, methods, queries, and procedures forming a single unit, where users can enter, view, maintain, and report their data.

argument
A variable, constant, or expression that you pass to a method or procedure (also called a formal parameter).

array

An ordered set of data elements of the same data type. Array elements (sometimes called items) are designated by a subscript (sometimes called an index in other languages) enclosed in square brackets, such that ar[1] and ar[2] are the first two elements of an array named ar.

Note: Array subscripts begin with 1 in ObjectPAL, and not with 0 as with other languages.

Array is an ObjectPAL data type.

array element

One item in an array, specified by the array name and a subscript enclosed in brackets. For example, the array Ar created with the following declaration has seven string elements, ar[1] through ar[7]:

```
ar Array [7] String
```

The following reference

```
Ar[3]
```

refers to the third element in Ar.

Note: Array subscripts begin with 1 in ObjectPAL, and not with 0 as with other languages.

ASCII

American Standard Code for Information Interchange; a sequence of 7-bit codes that define 128 standard characters, letters, numbers, and symbols. ASCII codes have been extended to 8-bit ANSI codes (used by Windows products) that include special graphic characters.

blank
A field or variable that has no value.

braces

The symbols { and } . Braces mark comments in ObjectPAL code.

branching commands

Commands that determine which ObjectPAL statements are executed (or whether they are executed at all), depending on whether conditions that they specify are met. Examples: if, iif, switch.

breakpoint

A flag you set in source code that suspends execution. Used in debugging.

When code that is executing reaches a breakpoint, code execution is suspended; you then can inspect the values of selected variables, and trace the code statements that have executed thus far.

bubbling

A process by which events pass from the target object up through the containership hierarchy.

An external event, directed at a target object that does not have a method to process it, passes up through the containership hierarchy until it reaches an object that can process it; or until it reaches the form level. If the form is unable to process the event, it dies.

See also: container

built-in event method

Pre-defined code that comes with every object you place in a form. Built-in event methods define an object's default response to events.

See also: Sequence of Execution

column

In a Paradox table, a vertical component that contains one field. In a Paradox report, a vertical area containing one or more fields.

comparison operator

Compares the values of two fields of the same data type; returns a logical value, True or False. The six comparison operators are: = (equal to), <> (not equal to), < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to).

Note: = (equal to) is a comparison operator only in expressions; otherwise it is an assignment operator.

compound object

An object made up of two or more other objects. For example, a table frame is a compound object made of field objects and record objects.

constant

A constant represents a value that cannot be changed. Paradox also contains many pre-defined constants. For example, DataNextRecord is an ObjectPAL constant that specifies a move to the next record in a table. You can also create constants that are used much like variables in ObjectPAL code in a **const...endConst** block.

See also: the topic Types of Constants for a complete list of ObjectPAL constant types; and the example for the topic enumRTLConstants to create a complete list of ObjectPAL constants.

container

1. A container is an object that completely surrounds other objects on a form. A container can itself be contained by another object. All objects on a form coexist in such a hierarchy of containers. For an object to be a container, its Contain Objects property must be checked in its Design menu. See also: bubbling; event model; containership.
2. *Container* is a built-in object variable that represents the object that contains *Self*. For example, suppose a box contains a button, and the button's **pushButton** method is as follows:

```
container.color = Red
```

When this code executes, the box turns red, because the box contains the button, and the button is executing the code.

containership

One object contains another object if the other object is completely within its borders. Containership affects the availability of variables, methods, and procedures.

See also: scope; container

control structure

One of three types of structures that control the execution of ObjectPAL code:

- Branching control structure, such as **if...then...endIf**
- Looping structure, such as **while ...endWhile**
- Terminating structure, such as **quitLoop**

ObjectPAL uses the following methods as control structures:

for	**return**
forEach	**scan**
if	**switch**
iif	**try**
loop	**while**
quitLoop	

Ctrl+Break

A key sequence that halts program execution. You can configure Paradox to respond to Ctrl+Break by choosing Properties|ObjectPAL and checking Enable Crtl + Break.

data
The information that Paradox stores in a table.

data type

The type of data that a field, variable, or array element can contain. Sometimes called *classes* in other languages. ObjectPAL has the following data types:

AnyType	Graphic	OLE
Array	Logical	Point
Binary	LongInt	Record
Date	Memo	SmallInt
DateTime	Money	String
DynArray	Number	Time

Database

1. Data and objects about a related topic that are organized logically into Paradox tables. See also: normalized database structure
2. An ObjectPAL variable that contains information about relationships between tables or access to the tables.

Date, Time and DateTime operator
Used to add (+) or subtract (-) values from Date, Time, and DateTime fields.

DDE
Dynamic Data Exchange; provides a way for Windows applications to share data.

deadlock
A situation created in a multiuser environment when two incompatible lock commands are issued concurrently.

Debugger

Part of the ObjectPAL Integrated Development Environment (IDE), the Debugger lets you interactively find and correct errors in your code by testing and tracing execution of commands.

Typically, you place breakpoints, which suspend execution, at important places in your code . You can then inspect the values of variables at those points, and trace the execution of code statements up to the breakpoint.

Desktop
The main window in Paradox.

dialog box

A special type of form that provides you with information and prompts you to select from a set of options. A dialog box stays on top of other windows, and can be moved on top of the menu bar. The form's Form Window Properties specify that the form is a dialog box.

See also: modal

Display manager
A category of object types that includes Application, Form, Report, and Table View.

DLL

Dynamic Link Library; a library of external routines that perform common tasks that Windows programs can share. DLL routines (DLLs) are loaded and linked to ObjectPAL methods and procedures at run-time. DLLs perform such common tasks as handling user input and managing memory. DLLs also allow you to create custom routines to perform tasks over and above ObjectPAL's functionality.

dynamic array

A special kind of array where each item has a string for an index. For example, ["Product"], [" Paradox"], [" Type"] [" Relational database"], ["]Version"] [1.0]. Called dynamic because its size is not fixed, but changes as items are added to and removed from it. Its size is limited only by system memory. Compact and flexible.

The DynArray data type in ObjectPAL lets you look up values in a dynamic array, even a very large one, quickly and easily.

Editor

The component of the ObjectPAL Integrated Development Environment (IDE) used to create and edit ObjectPAL methods.

encrypt
To translate a table or script into code that cannot be read without the proper password.

error stack

An ObjectPAL mechanism that stores information about the most recently detected run-time error. When a method or procedure executes successfully, the error stack is cleared.

When a run-time error occurs, error information records, which contain an error code and an error message, are pushed onto the stack in a last-in-first-out arrangement. The following methods allow control of and access to records on the error stack: **errorCode**, **errorMessage**, **errorPop**, **errorClear**, **errorHasErrorCode**, **errorHasNativeErrorCode**, **errorLog**, and **errorShow**.

event

1. An action or condition that triggers the execution of a method. Internal events are triggered by Paradox. External events are triggered by the user or by an ObjectPAL method that simulates a user action.
2. An ObjectPAL type that contains information about an event.

event model

The rules that specify how events are processed by objects in a form.

See also: bubbling; container

event packet

An ObjectPAL structure that contains detailed information about each event, such as its target object and the reason it occurred. The event packet (passed into built-in event methods through the variable *eventInfo*) accompanies the event as it moves up the containership hierarchy. You use RTL methods to examine the contents of the event packet.

event-driven application

An application whose code executes in response to events; compare with a procedural application, whose code executes in a linear sequence.

example element

In a query statement, an arbitrary sequence of characters that represents any value in a field. In Paradox you indicate an example element by pressing Example and typing the characters in the query image. In methods, you indicate example elements by preceding the characters with an underscore.

expression

A group of characters, which can include data values, variables, arrays, operators, and functions, that evaluate to a single a quantity or value. An expression can evaluate to a specific data type or, in certain cases, first be converted to string values before it is evaluated.

field
One item in a table that contains a specific category of information, such as phone number. Represented in a table as a vertical column. A horizontal row of fields in a table comprises a record.

field assignment

Referes to the assignment linking a variable with a field.   When the variable changes, the value in the field changes.

field object
A UIObject. When bound to a field in a table, a field object is used to display or change its data. An unbound field object is typically used to get input from the user.

field type

The representation of data in a table that is specific to the table's driver. For example, alphanumeric is a field type in Paradox; character is the corresponding field type in dBase.

See also: data type

field value

The data contained in one field of a record. If no data is present, the field is considered blank. Field objects have a Value property.

Feld View

Lets you move the insertion point through a field, character by character. It is used to view field values that are too large to be displayed in the current field width, or to edit a field value.

file
A collection of information stored under one name on a disk. For example, Paradox tables are stored in files.

FileSystem Type

An ObjectPAL variable that contains information about disk files, drives and directories. A FileSystem variable provides a handle to a file or directory that you can work with in ObjectPAL statements.

focus
An object that has focus (the active object) is ready to handle keyboard or mouse input; it is usually highlighted. Only one object can have focus at a time.

form

1. A window for displaying data and objects.
2. An ObjectPAL type (Form). The form is the highest-level container object.

format specification

The way in which a field value is displayed onscreen or output to a printer, such as alignment (left, right or center), font style and size, upper- or lower-case, and date and time formats. Also, the way in which input data is read, or validated with the edit format specification.

function keys
The 12 keys across the top of the keyboard labeled F1 through F12. (Some keyboards have 10 keys at the far left of the keyboard labeled F1 through F10).

global variable
A variable available to all objects in a form.

See also: local variable

handle

A variable that uniquely identifies an object you want to manipulate in ObjectPAL.

Help

The Paradox online Help system. You can press F1 at any point in Paradox to display information about the current operation.

hierarchy

The relationship of objects in a form, derived from their visual, spatial relationship.

See also: containership

IBM extended codes

Keys or combinations of keys on the keyboard that do not correspond to any of the standard ASCII character codes and are given special extended code numbers between -1 and -132.

identifier
A label that refers to an object, variable, property, or value (such as that returned by a function).

incremental development

A process of application development in which small parts of the application, or its overall structure, are designed and tested interactively.

index
A file that determines the order in which Paradox can access the records in a table. The key field of a Paradox table establishes its primary index.

Note: Some languages refer to an array subscript as an index; not ObjectPAL.

See also: key; secondary index

insertion point

The place where text is inserted when you type. The insertion point is usually represented by a flashing vertical bar.

inspect
To right-click an object to see its menu.

key

A field (or group of fields) that uniquely identifies each record in a Paradox table. A key provides three benefits: the table is prevented from containing duplicate records; its records are maintained in sorted order based on the key; and a primary index based on the key is created for the table. A composite key is comprised of more than one field. Sometimes called a primary key.

See also: index

key field
The field that is used as the index key for organizing and sorting a table.

keycode

A code that represents a keyboard character in ObjectPAL methods. May be an ANSI number or a string representing a key name known to Paradox.

keyword

A word reserved by ObjectPAL. A keyword must not be used as the name of a variable, array, method, or procedure.

lastMouseClicked

A built-in object variable that represents the last object to receive a **mouseDown.** It is reset when the mouse button is released, but only after the object has been given a chance to do its **mouseUp**.

lastMouseRightClicked

A built-in object variable that represents the last object to receive a **mouseRightDown.** It is reset when the mouse button is released, but only after the object has been given a chance to do its **mouseRightUp**.

library

1. A collection of ObjectPAL code that can be used by objects in one or more forms.
2. An ObjectPAL data type that stores custom methods and procedures, variables, constants, and user-defined data types. Libraries are useful for storing and maintaining frequently used routines, and for sharing custom methods and variables among several forms.

lifetime
The length of time during which a local variable, proc, or method is active or available.

link key
In a linked multi-table form, the part of the subordinate table's key that is linked or matched to fields in the master table.

local variable

A variable that is available only to the method or procedure in which it is declared.

See also: global variable

logical operator

One of three operators (AND, OR, or NOT) that can be used on logical data. For example, an AND between two logical values results in a logical value of True if both the original values are also True. Also known as Boolean operators.

logical type
An ObjectPAL data type that can contain one of two values: True and False. Also known as Boolean type.

logical value
A value (True or False) assigned to an expression when it is evaluated. Also known as a Boolean value.

looping commands

Commands that repeat a series of commands while or until a certain condition is met. Examples: for, loop, while.

See also: control structures

menu

1. A display of the choices or options available. Using ObjectPAL, you can create and edit both application menus and pop-up menus.

   The menu bar is a special menu at the top of the Desktop, below the title bar. Click an item on the menu bar to see a list of available commands.

2. An ObjectPAL type that stores information about an application's menu bar.

menu choice
Refers to the selection one of the items from the lists that appear when the cursor is placed on the Windows menu bar at the top of the screen (in most applications).

message
A string expression displayed in the status bar.

method

ObjectPAL code attached to an object that defines the object's response to an event.

There are three types of methods:

- Built-in event methods, which are attached to UIObjects, and respond to events;
- RTL (Run-time Library) methods, which are part of the ObjectPAL language; and
- Custom methods, which you create when a built-in event method or RTL method does not provide the functionality you want.

The syntax for a method is: ***object.method***(***arguments***)

See also: procedure

modal

A type of dialog box that retains focus until you close it. A modal dialog box cannot be resized.

normalized database structure

A logical arrangement of database information in small tables that minimizes redundancy; allows efficient access to data; provides a logical and coherent view of data categories; promotes data integrity; and allows easy insertion and deletion of data. Normalization involves decomposing a single large table into smaller tables that are linked by key fields.

numeric operator

A numeric operator performs arithmetic operations on the operands that surround it. There are four numeric operators: + (addition), - (subtraction), * (multiplication), and / (division). Valid with date, time and number fields only.

object

Objects include forms, reports, tables, queries, scripts, SQL files, and libraries.

Form and report objects can contain UI objects such as boxes, lines, ellipses, text, graphics, OLE objects, buttons, fields, table frames, and multi-record objects.

ObjectPAL recognizes the following Object Type Categories:

Data Model Objects	Display Managers
Data Types	Events
Design Objects	System Data Objects

Object Tree
A diagram that shows how objects in a form are related in terms of containership.

OEM
Original Equipment Manufacturer; your computer's manufacturer.

OLE

1. Object Linking and Embedding (OLE); provides a means by which an object, such as a graphic image, a spreadsheet, or a word processing document, can be link or embedded in a Paradox table or form.
2. An ObjectPAL data type that is used with OLE objects.

parameter

The variable, defined in a procedure declaration, through which an argument is passed at run-time. Also called a formal parameter.

See also: argument

picture
A pattern of characters that defines what a user can type into a field during editing or data entry, or in response to a prompt.

pixel
A single point on the screen; the smallest display unit on the screen. The name comes from picture element.

point

1. An ordered pair of numbers that represents a location on screen.
2. An ObjectPAL data type that contains information about a point on a screen.

pointer
A visual marker that indicates the mouse location on screen.

post
Accept changes to a record and update them in the table. Also called commit.

primary index

An index based on the key field(s) of a Paradox table. A primary index determines the location of records; lets you use the table as the detail in a link; keeps records in sorted order; and speeds up operations on the table.

See also: key; secondary index

procedure

Code bracketed by the keywords **proc** and **endProc**. Unlike a method, it is global, that is, not bound to an object that gives it context.

There are two types of procedures:

- RTL (Run-time Library) Procedures, which are part of the ObjectPAL language; and
- Custom Procedures, which you create when an RTL procedure does not provide the functionality you want.

A procedure's syntax is: ***procedure***(***arguments***)

See also: method

prompt
Instructions displayed on the screen, usually in the status bar. Prompts ask for information or guide you through an operation.

property
The named attribute of an object that determines one aspect of its behavior or characteristics, such as its visibility, color, or font. You right-click an object to view or change its properties.

QBE

Query by example. A way of creating a query by selecting its fields and selection criteria visually, without the need for formal SQL statements.

query

1. An inquiry about the data in a table; or an instruction to update the data, through the INSERT, DELETE or CHANGETO operators.
2. An ObjectPAL variable that represents a QBE query.

query by example (QBE)
A way of creating a query by selecting its fields and selection criteria visually, without the need for formal SQL statements.

quoted string
Text enclosed in double quotation marks.

raster operation
An operation that specifies how colors are blended on the screen.

record

1. A horizontal row in a Paradox table that contains a group of related fields of data.
2. An ObjectPAL data type: a programmatic, user-defined collection of information, similar to a **record** in Pascal or a **struct** in C. Separate and distinct from records associated with a table.

record number
A unique number that identifies each record in a table.

relational database

A database designed in accordance with a set of principles called the relational model. Data in a relational database must be organized into tables.

See also: normalized database structure

reserved words

Part of either Paradox or the ObjectPAL language, reserved words are the names of commands, keywords, functions, system variables, and operators. They may not be used as ObjectPAL variables or array names.

See also: keywords

restricted view

A detail table on a multi-table form, linked to the master table on a one-to-one or one-to-many basis, limited to showing only those records that match the current master record.

row
A horizontal component of a Paradox table that contains a record.

run-time error
An error that occurs when a syntactically valid statement cannot be carried out in the current context.

run-time library (RTL)
A collection of pre-defined methods and procedures that operate on specific types of objects .

scope

The availability of a variable, method, or procedure to other objects, methods or procedures.

Variables, methods, and procedures are attached to objects. When that object completes (or goes inactive for any other reason), the attached variables, methods, and procedures also go inactive. A variable declared inside a method is said to go “out of scope” when that method completes. The variable has no existence outside the operation of the method.

script

A collection of ObjectPAL statements, usually attached to an object on a form, that you use to perform operations automatically. Sometimes called a macro in other products.

See also: standalone script

secondary index
An index used for linking, querying, and changing the view order of tables.

Self

A built-in object variable that represents the UIObject to which the currently executing code is attached. For example, when the following statement executes in the **mouseEnter** method attached to *theBox*, *Self* refers to *theBox*.

```
self.color = Red
```

But, suppose a method attached to *theBox* calls a custom method named **changeColor** attached to the page. Suppose the code for **changeColor** is

```
method changeColor()
self.color = Blue
endMethod
```

When the method attached to *theBox* calls **changeColor**, the page turns blue, not *theBox*. Why? Because *Self* refers to the object to which the code is attached▪regardless of which object actually called the code

▪and in this case, the code is attached to the page. The single exception to this rule is when *Self* appears in a statement in a library. In this case, *Self* refers to the object that called the library routine.

When an event occurs, *Self* and **eventInfo.getTarget** may refer to the same object, but as events bubble up the containership chain, the target remains fixed while *Self* changes to refer to the object executing the method.

Self always refers to a UIObject, not to the object's value or the object's name.

session

1. A channel to the database engine. A session occurs whenever you open Paradox, either interactively or through ObjectPAL. You can have multiple sessions running simultaneously.
2. An ObjectPAL object type you use to open additional sessions beyond the session that Paradox opens by default.

slash sequence

A backslash followed by one or more characters, to represent an ASCII character. Examples are \" or \018. Slash sequences are used for placing quotation marks within strings and including other characters that have special meaning to Paradox.

standalone script

A script that is not attached to an object in a form. You run standalone scripts directly from the desktop, not after you trigger an event on an object in a form.

status bar

A row of four windows across the bottom of the Desktop. You use the **reason** and **setReason** methods, and the StatusReasons constants, to find out and specify where a message will be displayed.

string

1. An alphanumeric value, or an expression consisting of alphanumeric characters.
2. An ObjectPAL data type (String).

structure
The arrangement of fields in a table.

subject

A built-in object variable that specifies which object a custom method should operate on. For example, suppose a page in a form has a custom method **setColor**, and this is the code for **setColor**:

```
method setColor()
   subject.color = red
endMethod
```

Any object on that page can make the following call, and the object named *someObject* will turn red. When **setColor** executes, it replaces *Subject* with *someObject*.

```
someObject.setColor()
```

substring
Any part of a string.

syntax error
An error that occurs due to an incorrectly expressed statement.

table
A structure made up of horizontal rows (records) and vertical columns (fields) that contains your stored information.

table alias
Alternate name for a table in a data model.

TableView
An ObjectPAL object type. Use it to get a handle to a table view, the representation of a table in rows and columns.

target
The object for which an event is intended. For example, when you click a button, the button is the target.

TCursor

An ObjectPAL type that points to the data in a table. Using TCursors, you can manipulate the data without displaying the actual table.

tilde variable
A variable used in a query form, which must be preceded by a tilde(~).

Toolbar
A collection of buttons and design tools that appears in a row below the menu. The buttons available in the Toolbar depend on the type of object that is active on the Desktop.

transaction
A group of related changes to a database.

twip

A unit of measurement equal to 1/1440 of a logical inch (or 1/20 of a printer's point). There are 567 twips in one centimeter.

type

A way of classifying objects that have similar attributes. For example, all tables have attributes in common, and all forms have attributes in common, but the attributes of tables and forms are different. Therefore, tables and forms belong to different types.

validity check
A constraint on the values you can enter in a field. Sometimes called a val check.

variable

A named placeholder in memory where data is temporarily stored and manipulated while ObjectPAL code is running. Each variable must have a name that is unique within its scope.

Variables that are declared by a **var** statement run much faster (and often use less space) than undeclared variables. Declaring variables also enables compiler type checking, which helps detect bugs before run-time.

ObjectPAL OleAuto type reference

Changes

OLE Automation is a way to manipulate an application's objects from outside that application. OLE Automation uses OLE's component object model, but can be implemented independently from the rest of OLE. Using OLE Automation, you can create and manipulate objects from an application that exposes objects to OLE.

Methods for the OleAuto type

OleAuto

attach

close

enumAutomationServers

enumConstants

enumConstantValues

enumControls

enumEvents

enumMethods

enumObjects

enumProperties

enumServerInfo

first

invoke

next

open

openObjectTypeInfo

openTypeInfo

registerControl

unregisterControl

version

Changes to OleAuto type methods

The entire OleAuto type is new for version 7. All of the methods and procedures are new.

▪

attach method

See also Example OleAuto Type

Attaches an OLE Automation variable to a UIObject.

Syntax

`attach ( ``const `***`object`***` UIObject `**`)`**` Logical`

Description

attach attaches an OLE Automation variable to the UIObject specified with *object*. The attach will succeed only if the UIObject denotes an OCX control. After a successful attach, the methods and properties are accessible from the OLE Automation variable.

attach example

The following example attaches to an OLE custom control called MyCntl embedded on the form.

```
method pushButton ( var eventInfo Event )
var
   oa   oleauto
endvar
   oa.attach(MyCntl)
endMethod
```

close method

See also Example OleAuto Type

Closes the OLE Automation variable.

Syntax

```
close ( ) Logical
```

Description

close releases the reference from an OLE Automation variable to the automation server. Some servers will remain open when all references to it are gone. **close** is most useful for global variables, because it is called automatically when an OLE Automation variable goes out of scope.

close example

The following example closes the OLE Automation server application opened elswhere.

```
var
   pdx oleauto
endvar
method pushButton ( var eventInfo Event )
   pdx.close()
endMethod
```

▪

enumAutomationServers procedure

See also Example OleAuto Type

Reads the registry on the current machine and gathers all the available OLE Automation servers.

Syntax

enumAutomationServers (** `var` ***servers* `DynArray[ ] String` **)** `Logical`

Description

enumAutomationServers lists all the OLE Automation servers and OLE custom controls registered in the registry.

The information is assigned to *servers*, a DynArray that you must declare and pass as an argument. The indexes of the DynArray are the end user OLE Automation server names (for example, Paradox 7), and the corresponding values are the names used internally by OLE (for example, Paradox.Application).

enumAutomationServers returns True if successful.

Use **enumAutomationservers** to get the internal server name to pass to **open** and **openTypeInfo**.

▪

enumAutomationServers example

The following example

```
method pushButton ( var eventInfo Event )
var
   da DynArray[] String
endVar
enumautomationservers(da)
da.view()
endMethod
```

enumConstants method

See also   Example   OleAuto Type

Enumerates the constants defined by an OLE Automation server.

Syntax

```
enumConstants ( var types DynArray[ ] String ) Logical
```

Description

enumConstants enumerates the constant type names in a type library of an OLE Automation server. The information is assigned to the DynArray *types*. The indexes hold the OLE type name and the corresponding items are the equivalent ObjectPAL type. The constant type name can be used as input to the **enumConstantValues** to get the constant values of this type. These constants are only available through these methods.

enumConstants example

The following example enumerates the constants from Excel.

```
method pushButton
   var
      oa oleauto
      da DynArray[] String
   endvar
   oa.open("Excel.application.5")
   oa.enumConstants(da)
   da.view("Excel constant types")
endmethod
```

enumConstantValues method

See also Example OleAuto Type

Enumerates the constants accessible from an OLE Automation server.

Syntax

```
enumConstantValues ( const constantType String, var values DynArray[ ]
AnyType ) Logical
```

Description

enumConstantValues enumerates all the existing constants in a type library of an OLE Automation object. *constantType* is the type returned by **enumConstants**.

The information is assigned to the DynArray *values*. The indexes are the OLE constant names and the corresponding items are the constant's values.

■

enumConstantValues example

The following example enumerates the values of ‘Constants’ in Excel.

```
method pushButton ( var eventInfo Event )
var
   oa oleauto
   da DynArray[] AnyType
endvar
   oa.open("Excel.Application.5")
   oa.enumConstantValues("Constants",da)
   da.view()
endmethod
```

■

enumControls procedure

See also Example OleAuto Type

enumControls enumerates all the OLE custom controls in the registry.

Syntax

`enumControls ( var controls DynArray[ ] String ) Logical`

Description

enumControls enumerates all the OLE custom controls in the registry. The information is assigned to the DynArray *controls*. The indexes of the DynArray are the end user OLE Automation control names (for example, "My Own Control"), and the corresponding values are the names used internally by OLE (for example, MyCtrl.Ctrl1).

Use **enumControls** to get internal OLE control names, as input for the **open** and **openTypeInfo** methods and for the "progid" property for the OLE object.

enumControls example

The following example builds the controls DynArray and displays it.

```
method pushbutton ( var eventInfo Event )
var
   da DynArray[] String
endvar
   enumControls(da)
   da.view()
endmethod
```

The following example creates a form with an OCX object. "MyCtrl.Ctrl1" is an internal OLE control name listed by **enumControls** in the example above.

```
method pushButton ( var eventInfo Event )
var
   f form
   o uiobject
endvar
   f.create()
   o.create(OLETool, 200, 300, 1000, 500, f)
   o.ProgId = "MyCtrl.Ctrl1"
endMethod
```

enumEvents method

See also Example OleAuto Type

Enumerates the events accessible from an OLE Automation server.

Syntax

enumEvents (**`var` ***events* `DynArray[ ] String` **)** `Logical`

Description

enumEvents enumerates the events of controls. The form of the output is similar to the output of **enumMethods**. The information is assigned to the DynArray *events*. The DynArray will be empty if the OLE Automation variable is bound to an object that is not an OLE Automation control.

enumEvents example

The following example opens the type library of MyCtrl.Ctrl1, builds the DynArray of the enumerated events and displays the DynArray.

```
method pushButton ( var eventInfo Event )
var
   oa oleauto
   dy DynArray[] String
endvar
   oa.openTypeInfo("MyCtrl.Ctrl1")
   oa.enumEvents(dy)
   dy.view()
endMethod
```

■

enumMethods method

See also Example OleAuto Type

Enumerates the methods accessible from an OLE Automation server.

Syntax

`enumMethods ( var methods DynArray[ ] String ) Logical`

Description

enumMethods enumerates all the existing methods that can be accessed from an OLE Automation server. The information is assigned to the DynArray *methods*. The index of the DynArray is the method name, and the value is the ObjectPAL prototype. Some of these methods might NOT be accessible by ObjectPAL because their types are not supported, in which case the prototype will show a asterisk character (*).

The argument types might be specified with commentary information. For example, MoveCursorToPos(x LongInt {OLE_XPOS_PIXELS}, y LongInt {OLE_YPOS_PIXELS}), where OLE_XPOS_PIXELS is the OLE type of the argument. The OLE type name will often give an idea of the nature of the argument.

enumMethods example

The following example builds and displays the DynArray of the enumerated methods.

```
method viewMethods(var oa oleauto)
var
   dy DynArray[] String
endvar
   oa.enumMethods(dy)
   dy.view()
endMethod
```

■

enumObjects method

See also Example OleAuto Type

Enumerates the events accessible from an OLE Automation server.

Syntax

`enumObjects ( var objects DynArray[ ] String ) Logical`

Description

enumObjects lists the names of objects in a type library of a server. The object names are sub-objects in that particular OLE server. The sub-objects are often retrieved through methods and properties of the "Application" server object retrieved with the **open** method. This method lists the object names, which can be passed into **openObjectTypeInfo**, from which the methods and properties of the sub-object can be enumerated.

enumObjects example

The following example builds and displays the DynArray of the enumerated objects.

```
method viewObjects ( oa oleauto )
var
   dy DynArray[] String
endvar
   oa.enumObjects(dy)
   dy.view()
endmethod
```

▪

enumProperties method

See also Example OleAuto Type

Enumerates the properties accessible from an OLE Automation server.

Syntax

`enumProperties ( var properties DynArray[ ] String ) Logical`

Description

enumProperties enumerates all the existing properties that can be accessed from an OLE Automation server. The information is assigned to the DynArray *properties*. The index of the DynArray is the property name, and the item is the ObjectPAL type. Some of these properties might NOT be accessible by ObjectPAL because their types are not supported. Unsupported ObjectPAL types will show an asterisk (*).

The property types might be specified with commentary information. For example, ForeColor LongInt {OLE_COLOR}, BackColor LongInt {OLE_COLOR}, where OLE_COLOR is the OLE type of the argument. The OLE type name will often give an idea of the nature of the argument.

enumProperties example

The following example builds and displays the DynArray of the enumerated properties.

```
method viewProperties(oa oleauto)
var
   dy DynArray[] String
endvar
   oa.enumProperties(dy)
   dy.view()
endMethod
```

▪

enumServerInfo procedure

See also Example OleAuto Type

Enumerates information about the OLE Automation server.

Syntax

enumServerInfo (const ***serverName*** String, var ***info*** DynArray[ ] AnyType **)**
Logical

Description

enumServerInfo enumerates information about the server from the registry. The *serverName* is one of the internal OLE server names returned from either **enumAutomationServers** or **enumControls**.

The information provided is:

Key	Type	Comment
CLSID	String	The classID used internally by OLE. If this key is present the server is an OLE control.
ProgID	String	The internal OLE server name (for example, Paradox.Application).
TypeLib	String	The ClassID of the type library. If this item is present, openTypeInfo can be used with this server.
ToolboxBitmap32	Graphic	Toolbar bitmap for the control.
Version	String	The internal version of this server.

Because the *info* DynArray only holds information retrieved from the registry, the actual list depends on how a specific vendor has registered the server.

enumServerInfo example

The following example builds and displays the DynArray of the server information.

```
method pushButton ( var eventInfo Event )
var
   da DynArray[] anytype
endvar
   enumServerInfo("MyCtrl.Ctrl1", da)
   da.view()
endMethod
```

▪

first method

See also Example OleAuto Type

Returns the first object in a collection.

Syntax

```
first ( var AnyType )
```

Description

When an OLE Automation variable denotes a sub-object in a server that is itself a collection of other sub-objects, then **first** returns the first item in the given collection. The items in a collection will mostly be of type OleAuto, that is, a reference to another OLE automation object. If the collection is empty, the result will be a blank value. The collection can be checked with the **isBlank** method. If the current object is not a collection object, this method fails. Some servers have not implemented this method.

A collection object will behave as any other "OleAuto" object. It will always have a "Count" property and an "Item" method, and most of the time an "Add" and a "Remove" method. Specific implementations often have other methods and properties available.

▪

first example

The following example returns the first page of the object.

```
method getFirstPage ( var oa oleauto ) oleauto
var
   pages oleauto
   page oleauto
endvar
   pages = oa.pages()
   page = oa.first()
   return page
endMethod
```

■

invoke procedure

[See also](#) [Example](#) [OleAuto Type](#)

Invokes a method or property in an OLE Automation server.

Syntax

`invoke ( const methodName String [, var arg]* ) AnyType`

Description

invoke represents an alternate way of accessing methods and properties in a OLE Automation server. The argument *methodName* specifies the OLE Automation server's internal method. The optional *arg* arguments are the parameters of the method specified with *methodName*.

This method is useful in cases where the OLE Automation server has a method or property name that conflicts with an ObjectPAL keyword.

▪

invoke example

The following example shows three ways to call the **msgbox** method of the passed automation server.

```
method callMsgBox (oa oleauto)
var
   ret LongInt
endvar
   ret = oa.msgbox("Hello", 5)
   ret = oa^msgbox("Hello", 5)

   ret = oa.invoke("msgbox", "Hello", 5)
endMethod
```

▪

next method

See also Example OleAuto Type

Returns the next object in a collection.

Syntax

`next ( var AnyType )`

Description

When an OLE Automation variable denotes a sub-object in a server that is itself a collection of other sub-objects, then **next** returns the next item in the given collection. When there are no more items in the collection, the result will be a blank value. The items in a collection will mostly be of type OleAuto, that is, a reference to another OLE Automation object. If the collection is empty, the result will be a blank value. The collection can be checked with the **isBlank** method. If the current object is not a collection object, this method fails. Some servers have not implemented this method.

A collection object will behave as any other "OleAuto" object. It will always have a "Count" property and an "Item" method, and most of the time an "Add" and a "Remove" method. Specific implementations often have other methods and properties available.

▪

next example

The following example returns the first three pages of the object.

```
method getPages ( oa oleauto )
var
   pages oleauto
   page1 oleauto
   page2 oleauto
   page3 oleauto
endvar
   pages = oa.pages()
   page1 = pages.first()
   page2 = pages.next()
   page3 = pages.next()
endMethod
```

■

open method

See also Example OleAuto Type

Opens a server.

Syntax

open (const serverName String) Logical

Description

open opens the server specified with *serverName*. This operation will fail if the specified server doesn't denote an automation server.

open example

The following example opens Paradox as an OLE Automation server.

```
var
   pdx oleauto
endvar
method pushbutton ( var eventInfo Event )
   pdx.open("Paradox.Application")
endMethod
```

▪

openObjectTypeInfo method

See also Example OleAuto Type

Enumerates the events accessible from an OLE Automation server.

Syntax

`openObjectTypeInfo ( const server OleAuto, const objectName String ) Logical`

Description

openObjectTypeInfo connects to the type library of the specified sub-object in a server. This method is similar to **openTypeInfo**, except that by using **openObjectTypeInfo** the user can use **enumMethods** and **enumProperties** to retrieve the methods and properties of the sub-object specified in *objectName*. The object names can be enumerated by **enumObjects**.

openObjectTypeInfo example

The following example connects to the type library of the sub-object ‘chart’ in Excel then builds and displays the DynArray of the chart’s properties.

```
method pushButton ( var eventInfo Event )
var
   oa oleauto
   excel oleauto
   chart oleauto
   da DynArray[] String
endvar
   excel.openTypeInfo("Excel.application.5")
   chart.openObjectTypeInfo(excel, "chart")
   chart.enumProperties(da)
   da.view()
endMethod
```

▪

openTypeInfo method

See also Example OleAuto Type

Opens the type library of an OLE Automation server.

Syntax

openTypeInfo (**var ***serverName* String **)** Logical

Description

openTypeInfo connects to the type library of the specified *serverName*. After connecting, it is possible to call the type enumeration methods to retrieve information about the server. This is in contrast to the **open** method, which creates an instance of the server. After an **open**, the server methods and properties are accessible. If a server doesn't provide a type library, this method will return False.

This method is designed for type browsing only.

openTypeInfo example

The following example connects to the Paradox type library then builds and displays the DynArray of Paradox's properties.

```
method pushButton ( var eventInfo Event )
var
   oa oleauto
   dy DynArray[] String
endvar
   oa.openTypeInfo("Paradox.application")
   oa.enumProperties(dy)
   dy.view()
endMethod
```

registerControl procedure

See also Example OleAuto Type

Registers an OLE Automation control.

Syntax

`registerControl ( const fileName String ) String`

Description

registerControl attempts to auto-register the OLE Automation control specified in *fileName*.

▪

registerControl example

The following example registers the control MyCntl.cntl1. Notice that the registered name is actually the complete pathname of the file containing the control.

```
method pushButton ( var eventInfo Event )
registerControl("C:\\OCXLIB\\MYCNTL1.OCX")
endMethod
```

▪

unregisterControl method

See also Example OleAuto Type

Unregisters an OCX control.

Syntax

`unregisterControl ( const fileName String ) Logical`

Description

unregisterControl is used to unregister a control. The argument *fileName* is the filename of the OCX control to unregister. The OCX control must support the ability to unregister itself. This method returns True if the file is a valid OCX control.

▪

unregisterControl example

See the example for **registerControl**.

version method

See also Example OleAuto Type

Returns the version number of the current OLE2 server.

Syntax

```
version ( ) String
```

Description

version returns a string containing the version number of the currently attached OLE2 server (for example, "2.0").

▪

version example

The following example opens the OLE Automation server Paradox and retrieves its version.

```
method pushButton ( var eventInfo Event )
var
   oa oleauto
   v string
endvar
   oa.open("Paradox.Application")
   v = oa.version()
endMethod
```

ObjectPAL DataTransfer type reference

Changes

The DataTransfer type contains methods and procedures that create, delete, import, and export data.

Methods and procedures for the DataTransfer type

DataTransfer

appendASCIIFix

appendASCIIVar

dlgExport

dlgImport

dlgImportASCIIFix

dlgImportASCIIVar

dlgImportSpreadsheet

dlgImportTable

empty

enumSourcePageList

enumSourceRangeList

exportASCIIFix

exportASCIIVar

exportParadoxDOS

exportSpreadsheet

getAppend

getDestCharSet

getDestDelimitedFields

getDestDelimiter

getDestFieldNamesFromFirst

getDestName

getDestSeparator

getDestType

getKeyviol

getProblems

getSourceCharSet

getSourceDelimitedFields

getSourceDelimiter

getSourceFieldNamesFromFirst

getSourceName

getSourceRange

getSourceSeparator

getSourceType

importASCIIFix
importASCIIVar
importSpreadsheet
loadDestSpec
loadSourceSpec
setAppend
setDest
setDestCharSet
setDestDelimitedFields
setDestDelimiter
setDestFieldNamesFromFirst
setDestSeparator
setKeyviol
setProblems
setSource
setSourceCharSet
setSourceDelimitedFields
setSourceDelimiter
setSourceFieldNamesFromFirst
setSourceRange
setSourceSeparator
transferData

Changes to DataTransfer type methods

The entire DataTransfer type is new for version 7, but many of the methods were taken directly from the System type. Some methods are entirely new. The following table shows which methods are taken from the System type.

Moved from System type
dlgExport
dlgImportASCIIFix
dlgImportASCIIVar
dlgImportSpreadsheet
exportASCIIFix
exportASCIIVar
exportParadoxDOS
exportSpreadsheet
importASCIIFix
importASCIIVar
importSpreadsheet

▪

appendASCIIFix procedure

See also Example DataTransfer Type

Appends fixed format ASCII data from *fileName* to *tableName*.

Syntax

```
appendASCIIFix ( const fileName String, const tableName String, const
specTableName String [ , const ANSI Logical ] ) Logical
```

Description

Appends data from the fixed format ASCII file specified by *fileName* to the table specified by *tableName* using the layout in *specTableName*.

The argument *specTableName* is the name of a table that specifies the layout for the imported data. The structure of the file specified with *specTableName* is as follows:

Field name	Type & size	Description
Field Name	A 25	Name of a field to import.
Type	A 4	Field type (must be a valid Paradox or dBASE field specification; see Table::**create** for details).
Start	S	Number of the column where you want the field value to begin.
Length	S	Field size.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

appendASCIIFix example

The following example imports ASCII Fixed Text to Paradox (short form):

```
ImportASCIIFix("NewRecords.txt", "TimeCards.db", "ImpSpec.db")
```

appendASCIIVar procedure

See also Example DataTransfer Type

Appends delimited ASCII data from *fileName* to *tableName*.

Syntax

`appendASCIIVar ( ``const `***`fileName`***` String, const `***`tableName`***` String [ , const `***`separator`***` String, const `***`delimiter`***` String, const `***`allFieldsDelimited`***` Logical, const `***`ANSI`***` Logical ] `**`)`**` Logical`

Description

Appends data from the delimited ASCII file specified by *fileName* to the table specified by *tableName* using the options specified by *separator*, *delimiter*, *allFieldsDelimited*, and *ANSI*.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

appendASCIIVar example

The following example imports ASCII Delimited Text to Paradox (short form):

```
ImportASCIIVar("NewRecords.txt", "TimeCards.db")
```

▪

dlgExport procedure

See also	Example	DataTransfer Type

Displays the Export <*tableName*> as: dialog box.

Syntax

```
dlgExport ( const tableName String [ , const fileName String ] ) Logical
```

Description

dlgExport displays the Export <*tableName*> as: dialog box with the specified *tableName* already filled in. *tableName* specifies the name of the table to export and *fileName* is the name of the file created by the export. The type of export file is determined by the extension of *fileName*.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it is up to the user to close the dialog box.

dlgExport example

This example displays the Export As dialog box for the ORDERS.DB table.

```
method pushButton ( var eventInfo Event )
   var
      tableName String
   endVar

   tableName = "orders.db"
   ; invoke the Export As dialog box
   dlgExport ( tableName )
endMethod
```

▪

dlgImport procedure

See also Example DataTransfer Type

Displays the Import Data dialog box.

Syntax

`dlgImport ( const fileName String [ , const tableName String ] ) Logical`

Description

dlgImport displays the Import Data dialog box with the specified file and table names already filled in. Import file type defaults by extension. Text files (*.TXT or any file with an unknown extension) will be opened and the first few lines read to determin whether the file should be read as delimited or fixed length text.

dlgImport example

The following example displays the Import Data dialog box. The target table name will default to the same name as the source file with a .DB extension. The target file type will be Paradox 7, unless the table already exists of another type.

```
method pushButton(var eventInfo Event)
   ;the following line displays the Import Data dialog box
   dlgImport("Customer.txt")
endmethod
```

■

dlgImportASCIIFix procedure

See also Example DataTransfer Type

Displays the Import Data dialog box.

Syntax

```
dlgImportASCIIFix ( const fileName String ) Logical
```

Description

dlbImportASCIIFix displays the Import Data dialog box with the specified *fileName* already filled in and the import file type set to fixed length ASCII. *fileName* specifies the name of both the source file and the destination table for the imported data. If you specify a filename extension, Paradox uses it to find the source file.

The destination table's extension depends on its table type. The default type is .DB (for Paradox); for dBASE tables it is .DBF. Dates and numbers are formatted as specified in the Control Panel.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

dlgImportASCIIFix example

This example displays the Import Data dialog box to import data from the text file ORDERS.TXT to the Paradox table ORDERS.DB.

```
method pushButton ( var eventInfo Event )
   var
      fileName String
   endVar

   fileName = "orders.txt"

   ; invoke the Import Data dialog box
   ; by default, Paradox will use ORDERS.TXT as the source file
   ; and ORDERS.DB as the destination table
   dlgImportASCIIFix ( fileName )
endMethod
```

▪

dlgImportASCIIVar procedure

See also Example DataTransfer Type

Displays the Import Data dialog box.

Syntax

`dlgImportASCIIVar ( const fileName String ) Logical`

Description

dlgImportASCIIVar displays the Import Data dialog box with the specified *fileName* already filled in and the import file type set to delimited ASCII text. *fileName* specifies the name of both the source file and the destination table for the imported data. If you specify a filename extension, Paradox uses it to find the source file.

The destination table's extension depends on its table type. The default type for Paradox is .DB. The default for dBASE tables is .DBF. Dates and numbers are formatted as specified in the Windows Control Panel.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it is up to the user to close the dialog box.

The default settings include: fields separated by commas; fields delimited by quotes; only text fields delimited; and the OEM character set used.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

dlgImportASCIIVar example

This example displays the Import Data dialog box to import data from the text file ORDERS.TXT to the Paradox table ORDERS.DB.

```
method pushButton ( var eventInfo Event )
   var
      fileName String
   endVar

   fileName = "orders.txt"

   ; invoke the Import Data dialog box.
   ; by default, Paradox will use ORDERS.TXT as the source file
   ; and ORDERS.DB as the destination table.
   dlgImportASCIIVar ( fileName )
endMethod
```

▪

dlgImportSpreadsheet procedure

See also Example DataTransfer Type

Displays the Import Data dialog box.

Syntax

```
dlgImportSpreadsheet ( const fileName String ) Logical
```

Description

dlgImportSpreadsheet displays the Import Data dialog box with the specified *fileName* already filled in. *fileName* specifies the source file, the spreadsheet type of the source file, and the name of the destination table for the imported data. You specify a filename extension that Paradox uses to find and identify the spreadsheet type of the source file, as described below.

The destination table's extension depends on its table type. The default type for Paradox is .DB. The default for dBASE tables is .DBF. Dates and numbers are formatted as specified in the Windows Control Panel.

Note: The filename extensions and their spreadsheet formats:

Extension	Format
WB1, WB2, WB3	Quattro Pro Win
WQ1	Quattro Pro DOS
WKQ	Quattro
WK1	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0/5.0

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it is up to the user to close the dialog box.

The default settings include: the From cell is the first cell of the first page of the spreadsheet; the To cell is the last cell of the last page; and the Get Field Names From First Row check box is checked.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

dlgImportSpreadsheet example

This example instructs the Import Data dialog box to import data from the Quattro Pro for Windows spreadsheet ORDERS.WB1 to the Paradox table ORDERS.DB.

```
method pushButton ( var eventInfo Event )
   var
      fileName String
   endVar

   fileName = "orders.wb1"

   ; invoke the Import Data dialog box
   ; by default, Paradox will use ORDERS.WB1 as the source file
   ; and ORDERS.DB as the destination table
   dlgImportSpreadsheet ( fileName )
endMethod
```

▪

dlgImportTable procedure

See also Example DataTransfer Type

Displays the Import Data dialog box.

Syntax

```
dlgImportTable ( const tableName String ) Logical
```

Description

dlgImportTable displays the Import Data dialog box with the specified *tableName* already filled in as an import source.

dlgImportTable example

This example displays the Import Data dialog box to import data from the dBASE table ORDERS.DBF to the Paradox table ORDERS.DB.

```
method pushButton ( var eventInfo Event )
   var
      tblName String
   endVar

   tblName = "orders.dbf"

   ; invoke the Import Data dialog box
   ; by default, Paradox will use ORDERS.DBF as the source file
   ; and ORDERS.DB as the destination table
   dlgImportTable ( tblName )
EndMethod
```

▪

empty method

See also Example DataTransfer Type

Deletes the data from a structure.

Syntax

```
empty ( )
```

Description

empty re-initializes the structure by deleting the data from the structure leaving the form of the structure intact. **empty** can initialize mail variable structures and tables but cannot initialize forms, databases, or reports.

empty example

The following example specifies a DataTransfer data type. This structure is used with the transferData method.   It is assumed that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method cmTransfer() is within the scope of the variable, dt.

```
method cmTransfer()   ;this example completes a DataTransfer

   dt.setSource("CUSTOMER.TXT", DTASCIIVar)  ; sets the datatransfer source
                                             ; to CUSTOMER.TXT
   dt.setSourceSeparator("/")   ; specifies the forward slash "/" character
                                ; to separate each field
   dt.setSourceDelimiter("'")   ; specifies the single quote to surround
                                ; the fields
   dt.setSourceDelimitedFields(DTDelimJustText)  ; specifies that the single
                                                 ; quote (delimiter)
surrounds
                                                 ; only text fields of the
                                                 ; source file
   dt.setSourceCharSet(DTANSI)   ; specifies that the character set used
                                 ; when creating the source file
                                 ; was the ANSI character set

   dt.setSourceFieldNamesFromFirst(False)   ; specifies to use the first
                                            ; row of the source file as
                                            ; field names
   dt.setDest("NEWCUST.DB")   ; sets the destination file to NEWCUST.DB
   dt.setProblems(True)   ; specifies to create a PROBLEMS.DB if there are
                          ; any problems importing the source file
   dt.transferData()   ; executes the data transfer.  In this case it
                       ; imports the CUSTOMER.TXT file as NEWCUST.DB.
   dt.empty()   ; empties the dt variable structure to set it up for
                ; a new transfer.

endmethod
```

enumSourcePageList method

See also Example DataTransfer Type

Puts the list of spreadsheet pages into a string array.

Syntax

```
enumSourcePageList ( var pages Array[] String )
```

Description

enumSourcePageList gets the list of pages and puts the list into the string array *pages* (requires that file name and type are set to an existing spreadsheet).

enumSourcePageList only applies when the source is a spreadsheet.

enumSourcePageList example

The following example enumerates the pages of the LEDGER.WB3 spreadsheet into an array and displays the list.

```
method pushButton(var eventInfo Event)
   var
      dt        DataTransfer
      arPage    Array[]  String
   endVar
   dt.setSource("ledger.wb3")
   dt.enumSourcePageList(arPage)
   arPage.view()
endMethod
```

▪

enumSourceRangeList method

See also   Example   DataTransfer Type

Gets the list of named ranges into a string array.

Syntax

enumSourceRangeList **(** var ***ranges*** Array[] String **)**

Description

enumSourceRangeList puts the list of named ranges into the string array *ranges* (requires that file name and type are set to an existing spreadsheet).

enumSourceRangeList only applies when the source is a spreadsheet.

enumSourceRangeList example

The following example enumerates the pages of the LEDGER.WB3 spreadsheet into an array and displays the list.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
      arRange  Array[]  String
   endVar
   dt.setSource("ledger.wb3")
   dt.enumSourceRangeList(arRange)
   arRange.view()
endMethod
```

▪

exportASCIIFix procedure

See also Example DataTransfer Type

Exports data from a table to an ASCII (text) file in which the fields of each record are the same length.

Syntax

```
exportASCIIFix ( const tableName String, const fileName String, const
specTableName String [ , const ANSI Logical ] ) Logical
```

Description

exportASCIIFix exports data from a table to an ASCII (text) file in which the fields of each record are the same length. This duplicates the function of the Export Data dialog box. *tableName* specifies the table whose data is exported. *fileName* specifies the file to which the data is exported. If the file does not exist, this procedure creates it using the layout of *specTableName*.

The argument *specTableName* is the name of a table that specifies the layout for the imported data. The structure of the file specified with *specTableName* is as follows:

Field name	Type & size	Description
Field Name	A 25	Name of a field to import.
Type	A 4	Field type (ignored for export, may be left off).
Start	S	Number of the column where you want the field value to begin.
Length	S	Field size.

In version 7, **exportASCIIFix** can use the same *specTableName* as **importASCIIFix**. For an export operation the field type is determined by the table structure, so that field is ignored. Version 7 can accept tables built with version 5.0, but version 5.0 will not recognize a version 7 level table.

For each field you export, *specTableName* contains a Start position (the column where the field value begins) and a Length (how many characters in the field). *specTableName* serves the same purpose as EXPORT.DB, which is created when you use the Export Data dialog box to export a table interactively.

ANSI (optional) specifies whether to use the ANSI or OEM character set: set *ANSI* to True to use the ANSI character set, or False to use the OEM character set.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

exportASCIIFix example

Exports data in the ORDERS.DB table to a text file ORDERS.TXT; reads the export format from the ORDEREXP.DB table; and exports the data using the ANSI character set.

```
method pushButton ( var eventInfo Event )
   exportASCIIFix ( "orders.db", "orders.txt", "orderexp.db", True )
endMethod
```

▪

exportASCIIVar procedure

See also Example DataTransfer Type

Exports data from a table to a delimited (variable field length) ASCII (text) file.

Syntax

exportASCIIVar (const ***tableName*** String, const ***fileName*** String [, const ***separator*** String, const ***delimiter*** String, const ***allFieldsDelimited*** Logical, const ***ANSI*** Logical] **)** Logical

Description

exportASCIIVar exports data from a table to a delimited (variable field length) ASCII (text) file. If the file does not exist, this method creates it. This method duplicates the function of the Export Data dialog box.

tableName specifies the table whose data is exported. *fileName* specifies the target file where the data is moved. *separator* (optional) specifies the character that surrounds field values in the target file; choose a comma or any other single character, including a special character such as a tab. *delimiter* (optional) specifies the character that defines the limits of field values in the target file; use the empty string if you do not want a character to define the limits of the values. *allFieldsDelimited* (optional) specifies whether data of all field types is delimited (True), or only data of text field types, alphanumeric or character (False).

Note: Paradox cannot export fields of type memo (Paradox or dBASE), formatted memo, graphic, OLE, or binary, to delimited text.

ANSI (optional) specifies whether to use the ANSI or OEM character set: set *ANSI* to True to use the ANSI character set, or to False to use the OEM character set.

The default settings for optional arguments:

- *separator* is "," (comma)
- *delimiter* is "\"" (double quote)
- *allFieldsDelimited* is False
- *ANSI* is False

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

exportASCIIVar example

This example exports the data from the ORDERS.DB table to the text file ORDERS.TXT; uses tabs to delimit field values; uses percent signs to enclose each value; delimits only text fields; and uses the ANSI character set.

```
method pushButton ( var eventInfo Event )
   exportASCIIVar ( "orders.db", "orders.txt", "\t", "%", False, True )
endMethod
```

Exports Paradox to ASCII Delimited Text (medium form):

```
var
   dt DataTransfer
endVar

dt.setSource("TimeCards.db")
dt.setDest("Records.txt", DTAsciiVar)
dt.TransferData()
```

Exports Paradox to ASCII Delimited Text (short form):

```
ExportASCIIVar("TimeCards.db", "NewRecords.txt")
```

▪

exportParadoxDOS procedure

See also Example DataTransfer Type

Exports data from a Paradox for Windows or a dBASE table to a level 4 Paradox for DOS table.

Syntax

```
exportParadoxDOS ( const tableName String, const fileName String ) Logical
```

Description

exportParadoxDOS exports data from a Paradox for Windows or a dBASE table to a level 4 Paradox for DOS table. This method duplicates the function of the Table Export dialog box.

Note: **exportParadoxDOS** cannot export fields of type Bytes (type Y) since they are excluded from the destination file. Will not export OLE and Binary fields when you export a dBASE table to Paradox for DOS format.

tableName specifies the table whose data is exported. *fileName* specifies the target file to which the data is moved. *fileName* can include an extension, but the extension must be .DB. For example, in the following list, the first two file names are valid, but the third is not:

File name	Remarks
ORDERS	OK to omit the extension.
ORDERS.DB	Valid.
ORDERS.DBF	Invalid. If provided, extension must be .DB.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

exportParadoxDOS example

This example exports data from the dBASE table ORDERS.DBF to a Paradox for DOS table ORDERS.DB.

```
method pushButton ( var eventInfo Event )
   if not exportParadoxDOS ( "orders.dbf", "orders" ) then
      errorShow ( "Export to Paradox DOS failed." )
   endIf
endMethod
```

▪

exportSpreadsheet procedure

See also Example DataTransfer Type

Exports the data from a table to a spreadsheet file.

Syntax

`exportSpreadsheet ( const tableName String, const fileName String [ , const makeRowHeaders Logical ] ) Logical`

Description

exportSpreadsheet exports the data from a table to a spreadsheet file, duplicating the function of the Export Data dialog box. If the spreadsheet file does not exist, this method creates it. The type of the spreadsheet is determined by the file extension. When you export data to a spreadsheet, Paradox converts each record to a row and each field to a column. If a value is wider than the column display width, the full value is converted but partially hidden.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

If a date in the original table is beyond the range of the allowable dates in the spreadsheet, the date is exported as the value ERROR.

tableName specifies the table whose data is exported. *fileName* specifies the target file where the data is moved. *makeRowHeaders* (optional) specifies whether the table's column headers will label corresponding rows in the spreadsheet file: True (default) uses column headers as labels; False does not.

Note: The file extension in *fileName* specifies the format of the spreadsheet file. The following shows extensions and their spreadsheet formats:

Extension	Format
WB1, WB2, WB3	Quattro Pro Win
WQ1	Quattro Pro DOS
WKQ	Quattro
WK1	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0/5.0

exportSpreadsheet example

This example exports data from the ORDERS.DB table to a file in Quattro Pro for Windows format. It uses field names from the table as labels in the spreadsheet file.

```
method pushButton ( var eventInfo Event )
   exportSpreadsheet ( "orders.db", "orders.wb1", True )
endMethod
```

getAppend method

See also Example DataTransfer Type

Retrieves value set by **setAppend** (True/False).

Syntax

`getAppend ( )` `Logical`

Description

getAppend retrieves value set by the **setAppend** method (True or False). This method applies only when the destination is a table. Leave it unset for other destinations.

getAppend example

The following example specifies a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Text

```
var
   dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIFixed Then
   dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setDest ( "Existing Data.db" )
if dt.getAppend ( False ) then
   dt.setAppend ( True )

dt.transferData ( )
```

▪

getDestCharSet method

See also Example DataTransfer Type

Retrieves value set by **setDestCharSet**.

Syntax

`getDestCharSet ( ) SmallInt`

Description

getDestCharSet retrieves value set by **setDestCharSet**. This method applies only when the source or destination is a fixed or delimited ASCII text file. Leave it unset for other cases.

getDestCharSet example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT.   It uses **setDestCharSet** to specify using the ANSI character set.   To specify using the OEM character set, use the DTOEM constant with **setDestCharSet**.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   dt.setDestDelimiter("'")
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   ;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
if dt.getDestCharSet ( DTOEM ) then
   dt.setDestCharSet(DTAnsi)
else
endif
   ;run Export
   dt.transferData()
endMethod
```

■

getDestDelimitedFields method

See also Example DataTransfer Type

Retrieves value set by **setDestDelimitedFields**.

Syntax

`getDestDelimitedFields ( ) SmallInt`

Description

getDestDelimitedFields retrieves value set by the **setDestDelimitedFields** method.

getDestDelimitedFields only applies when the source or the destination is a delimited ASCII text file.

getDestDelimitedFields example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. It uses **setDestDelimitedFields** to specify surrounding text fields only. To specify all fields to be delimited, use the DTDelimAllFields constant with **setDestDelimitedFields**.

```
method pushButton(var eventInfo Event)
   var
      dt      DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   dt.setDestDelimiter("'")
   dt.view(dt.getDestDelimitedFields)
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   ;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
   dt.setDestCharSet(DTAnsi)

   ;run Export
   dt.transferData()
endMethod
```

getDestDelimiter method

See also Example DataTransfer Type

Retrieves value set by **setDestDelimiter**.

Syntax

`getDestDelimiter ( ) String`

Description

getDestDelimiter retrieves value set by the **setDestDelimiter** method.

getDestDelimiter only applies when the source or the destination is a delimited ASCII text file.

getDestDelimiter example

This example exports the ORDERS.DB table into an ASCII delimited text file. The delimiter specified is the single quote character.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   msgInfo("Info", "current delimiter is "+dt.getDestDelimiter)
   dt.setDestDelimiter("'")
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   ;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
   dt.setDestCharSet(DTAnsi)

   ;run Export
   dt.transferData()
endMethod
```

getDestFieldNamesFromFirst method

See also Example DataTransfer Type

Retrieves the value set by **setDestFieldNamesFromFirst** (True/False).

Syntax

```
getDestFieldNamesFromFirst ( ) Logical
```

Description

getDestFieldNamesFromFirst retrieves the value set by **setDestFieldNamesFromFirst** (True/False).

getDestFieldNamesFromFirst only applies when the source is a spreadsheet.

getDestFieldNamesFromFirst example

This example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. The **setDestFieldNamesFromFirst** is used to create the first row of the text file with field names.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   dt.setDestDelimiter("'")
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   If dt.getDestFieldNamesFromFirst = True Then
      msgInfo("Info", "SetDestFieldNamesFromFirst is On")
   else
      msgInfo("Info", "Setting DestFieldNamesFrom First to On")
   endIf
;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
   dt.setDestCharSet(DTAnsi)

   ;run Export
   dt.transferData()
endMethod
```

getDestName method

See also Example DataTransfer Type

Retrieves the destination file name.

Syntax

getDestName ( ) String

Description

getDestName retrieves the destination file name.

getDestName example

The following example specifies a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Export to Text

```
var
   dt DataTransfer
endVar
msgInfo("Info", "The current source is " + dt.getSourceName)
dt.setSource ( "ANSWER.db" )
msgInfo("Info", "The current destination is " + dt.getDestName)
dt.SetDest ( "NEWFILE.TXT" )
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

▪

getDestSeparator method

See also Example DataTransfer Type

Retrieves value set by **setDestSeparator**.

Syntax

`getDestSeparator ( )` `String`

Description

getDestSeparator retrieves value set by the **setDestSeparator** method.

getDestSeparator only applies when the source or the destination is a delimited ASCII text file.

getDestSeparator example

The following examples specify a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Spreadsheet

```
var
   dt DataTransfer
endVar
msgInfo("Info", "The current source separator is " + dt.getSourceSeparator)
dt.setSource ( "ANSWER.db" )
msgInfo("Info", "The current destination separator is " +
dt.getDestSeparator)
dt.SetDest ( "NEWFILE.TXT" )
dt.setProblems ( True )
dt.transferData ( )
```

Import from Text

```
var
   dt DataTransfer
endVar
msgInfo("Info", "The current source separator is " + dt.getSourceSeparator)
dt.setSource ( "ANSWER.db" )
msgInfo("Info", "The current destination separator is " +
dt.getDestSeparator)
dt.SetDest ( "NEWFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIFixed Then
   dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

Export to Text

```
var
   dt DataTransfer
endVar
msgInfo("Info", "The current source separator is " + dt.getSourceSeparator)
dt.setSource ( "ANSWER.db" )
msgInfo("Info", "The current destination separator is " +
dt.getDestSeparator)
dt.SetDest ( "NEWFILE.TXT" )
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

▪

getDestType method

See also Example DataTransfer Type

Retrieves the destination file type (constant).

Syntax

```
getDestType ( ) SmallInt
```

Description

getDestType retrieves the destination file type (constant).

getDestType example

The following example specifies a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Spreadsheet

```
var
   dt DataTransfer
endVar
   msgInfo("Info", "the current dest type is " + dt.getDestType())
dt.setSource ( "MYFILE.WKS" )
dt.setDest ( "New Data.db" )
dt.setProblems ( True )
dt.transferData ( )
```

Import from Text

```
var
   dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIFixed Then
   dt.loadDestSpec ( "SpecTable" )
EndIf
   msgInfo("Info", "the current dest type is " + dt.getDestType())
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

Export to Text

```
var
   dt DataTransfer
endVar
dt.setSource ( "ANSWER.db" )
   msgInfo("Info", "the current dest type is " + dt.getDestType())
dt.SetDest ( "NEWFILE.TXT" )
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

getKeyviol method

See also Example DataTransfer Type

Retrieves value set by **setKeyviol** (True/False).

Syntax

```
getKeyviol ( [ const tableName String, var count LongInt ] ) Logical
```

Description

getKeyviol retrieves value set by the **setKeyviol** method (True or False). The argument *tableName* is the name of the Key Violations table. The argument *count* is the number of key violations in the table. This method applies only when the destination is a table. Leave it unset for other destinations.

getKeyviol example

This example retrieves the key violations from the file kvTbl.

```
method pushButton(var eventInfo Event)
   var
      dt                   DataTransfer
      kvTbl, probTbl       String
      kvNum, probNum       Longint
   endVar
   dt.setSource("MYFILE.TXT")
   dt.LoadSourceSpec("SPECFILE.DB")
   dt.setDest("MYFILE.DB")
   if isTable("MYFILE.DB ") then
      dt.setAppend(True)
   endIf
   if msgQuestion("Import Option",
                 "Would you like to produce auxilliary tables?") = "Yes" then
      dt.setKeyviol(True)
      dt.setProblems(True)
   endIF
   dt.transferData()
   if dt.getKeyviol(kvTbl, kvNum) then
      msgInfo("Import Status",
            "# Key violations = "+string(kvNum)+
            "\nKeyviol table name = " + kvTbl)
   endIf
   if dt.getProblems(probTbl, probNum) then
         msgInfo("Import Status",
               "# Record errors = "+string(probNum) +
               "\nProblem table name = " + probTbl)
   endIf
endMethod
```

getProblems method

See also Example DataTransfer Type

Retrieves value set by **setProblems** (True/False).

Syntax

getProblems ([var ***tableName*** String, var ***count*** LongInt] **)** Logical

Description

getProblems retrieves the value set by the **setProblems** method (True or False). The optional *tableName* and *count* arguments specify the name of the problems table and the number of problems, respectively.

This method applies only when the destination is a table. Leave it unset for other destinations.

getProblems example

This example retrieves the problems from the file probTbl.

```
method pushButton(var eventInfo Event)
   var
      dt                    DataTransfer
      kvTbl, probTbl        String
      kvNum, probNum        Longint
   endVar
   dt.setSource("MYFILE.TXT")
   dt.LoadSourceSpec("SPECFILE.DB")
   dt.setDest("MYFILE.DB")
   if isTable("MYFILE.DB ") then
      dt.setAppend(True)
   endIf
   if msgQuestion("Import Option",
                 "Would you like to produce auxilliary tables?") = "Yes" then
      dt.setKeyviol(True)
      dt.setProblems(True)
   endIF
   dt.transferData()
   if dt.getKeyviol(kvTbl, kvNum) then
      msgInfo("Import Status",
            "# Key violations = "+string(kvNum)+
            "\nKeyviol table name = " + kvTbl)
   endIf
   if dt.getProblems(probTbl, probNum) then
         msgInfo("Import Status",
               "# Record errors = "+string(probNum) +
               "\nProblem table name = " + probTbl)
   endIf
endMethod
```

getSourceCharSet method

See also Example DataTransfer Type

Retrieves value set by **setSourceCharSet**.

Syntax

`getSourceCharSet ( )` `SmallInt`

Description

getSourceCharSet retrieves value set by **setSourceCharSet**. This method applies only when the source or destination is a fixed or delimited ASCII text file. Leave it unset for other cases.

getSourceCharSet example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. It uses **setDestCharSet** to specify using the ANSI character set. To specify using the OEM character set, use the DTOEM constant with **setDestCharSet**.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   dt.setDestDelimiter("'")
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   ;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
   msgInfo("Info", "the source char set is " + dt.getSourceCharSet())
if dt.getDestCharSet ( DTOEM ) then
   dt.setDestCharSet(DTAnsi)
else
endif
   ;run Export
   dt.transferData()
endMethod
```

▪

getSourceDelimitedFields method

See also Example DataTransfer Type

Retrieves value set by **setSourceDelimitedFields**.

Syntax

```
getSourceDelimitedFields ( ) SmallInt
```

Description

getSourceDelimitedFields retrieves value set by the **setSourceDelimitedFields** method.

getSourceDelimitedFields only applies when the source or the destination is a delimited ASCII text file.

getSourceDelimitedFields example

This example uses **getSourceDelimitedFields** to determine whether all fields or just text fields are delimited.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setSource("iesimpld.txt")

;The following lines check to see what Paradox 7 determined as the type of
;of the source file.  If it is delimited, Paradox 7 determines the separator,
;delimiter and which fields are delimited.
   switch
      case dt.getSourceType() = DTASCIIVar :
         fldType = "Delimited"
         fldDelimiter = dt.getSourceDelimiter()
         if dt.getSourceDelimitedFields() = DTDelimAllFields then
            fldDelimitedFields = "All"
         else
            fldDelimitedFields = "Text"
         endIF
         fldSeparator = dt.getSourceSeparator()
      case dt.getSourceType() = DTASCIIFixed :
         fldType = "Fixed"
      otherwise :
         msgInfo("Hello","File missing or not text.")
   endSwitch
endMethod
```

▪

getSourceDelimiter method

See also Example DataTransfer Type

Retrieves value set by **setSourceDelimiter**.

Syntax

```
getSourceDelimiter ( ) String
```

Description

getSourceDelimiter retrieves value set by the **setSourceDelimiter** method.

getSourceDelimiter only applies when the source or the destination is a delimited ASCII text file.

getSourceDelimiter example

This example uses **getSourceDelimiter** to display the delimiter used in the source.   This is useful for the user to confirm if the delimiter is set correctly and specify a new delimiter if necessary.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setSource("iesimpld.txt")

;The following lines check to see what Paradox 7 determined as the type of
;of the source file.  If it is delimited, Paradox 7 determines the separator,
;delimiter and which fields are delimited.
   switch
      case dt.getSourceType() = DTASCIIVar :
         fldType = "Delimited"
         fldDelimiter = dt.getSourceDelimiter()
         if dt.getSourceDelimitedFields() = DTDelimAllFields then
            fldDelimitedFields = "All"
         else
            fldDelimitedFields = "Text"
         endIF
         fldSeparator = dt.getSourceSeparator()
      case dt.getSourceType() = DTASCIIFixed :
         fldType = "Fixed"
      otherwise :
         msgInfo("Hello","File missing or not text.")
   endSwitch
endMethod
```

■

getSourceFieldNamesFromFirst method

See also		Example		DataTransfer Type

Retrieves the value set by **setSourceFieldNamesFromFirst**.

Syntax

```
getSourceFieldNamesFromFirst ( ) Logical
```

Description

getSourceFieldNamesFromFirst retrieves the value set by **setSourceFieldNamesFromFirst** (True/False).

getSourceFieldNamesFromFirst only applies when the source is a spreadsheet.

getSourceFieldNamesFromFirst example

The following example specifies a DataTransfer data type. This structure is used with the transferData method. It is assumed that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method cmTransfer() is within the scope of the variable, dt.

```
method cmTransfer()   ;this example completes a DataTransfer

   dt.setSource("CUSTOMER.TXT", DTASCIIVar)  ; sets the datatransfer source
                                             ; to CUSTOMER.TXT
   dt.setSourceSeparator("/")   ; specifies the forward slash "/" character
                                ; to separate each field
   dt.setSourceDelimiter("'")   ; specifies the single quote to surround
                                ; the fields
   dt.setSourceDelimitedFields(DTDelimJustText)  ; specifies that the single
                                                 ; quote (delimiter) surrounds
                                                 ; only text fields of the
                                                 ; source file
   dt.setSourceCharSet(DTANSI)   ; specifies that the character set used
                                 ; when creating the source file
                                 ; was the ANSI character set
   msgInfo("Info", "the current setting is " + dt.getSourceFieldNamesFrom
First")
   dt.setSourceFieldNamesFromFirst(False)   ; specifies to use the first
                                            ; row of the source file as
                                            ; field names
   dt.setDest("NEWCUST.DB")   ; sets the destination file to NEWCUST.DB
   dt.setProblems(True)   ; specifies to create a PROBLEMS.DB if there are
                          ; any problems importing the source file
   dt.transferData()   ; executes the data transfer.  In this case it
                       ; imports the CUSTOMER.TXT file as NEWCUST.DB.
   dt.empty()   ; empties the dt variable structure to set it up for
                ; a new transfer.

endmethod
```

getSourceName method

See also   Example   DataTransfer Type

Retrieves the source file name.

Syntax

getSourceName ( ) `String`

Description

getSourceName retrieves the source file name.

getSourceName example

The following example checks to see if the user has attempted to import data from the SYSTEM.INI file.

```
var
   dt DataTransfer
   importSourceFile String
endVar

importSourceFile = "Your sourcename here"
importsourcefile.view("Import what file?")

dt.setSource(importSourceFile, dtAuto)  ;// allow Paradox to determine
filetype
if dt.getSourceName() = "system.ini" then
   msgStop("No!", "This source file won't create useable data.")
   return
else
   dt.setDest("importSample". dtParadox7)  ;// import into Paradox 7 table

dt.transferData ( )
endit
endMethod
```

▪

getSourceRange method

See also Example DataTransfer Type

Retrieves the range set by **setSourceRange**.

Syntax

`getSourceRange ( )` `String`

Description

getSourceRange retrieves the range set with the **setSourceRange** method.

getSourceRange only applies when the source is a spreadsheet.

getSourceRange example

The following illustrates using the **setSourceRange** method. Use this method to specify the range in a spreadsheet to import. Named ranges as well as standard ranges will work.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setSource("092595.wb2")

   ;Set the range to import from the spreadsheet.
   ;Either named range or specified range (ie. Page1:A1..Page3:AB10)
   msgInfo("Info", "The Current range is " + dt.getSourceRange)
   dt.setSourceRange("myRange")
   dt.setSourceFieldNamesFromFirst(True)
   dt.setDest("delme09.db")

   ;Prompt the user to verify range to import.  getSourceRange returns the
   ;actual range notation.
   view(dt.getSourceRange(),"Import Range")
   dt.transferData()
endMethod
```

getSourceSeparator method

See also Example DataTransfer Type

Retrieves the value set by **setSourceSeparator**.

Syntax

getSourceSeparator ( ) String

Description

getSourceSeparator retrieves value set by the **setSourceSeparator** method.

getSourceSeparator only applies when the source or the destination is a delimited ASCII text file.

getSourceSeparator example

This example uses **getSourceSeparator** to display the separator used in a field on the form.   This is useful for the user to confirm if the separator is set correctly and specify a new separator if necessary.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setSource("iesimpld.txt")

;The following lines check to see what Paradox 7 determined as the type of
;of the source file.  If it is delimited, Paradox 7 determines the separator,
;delimiter and which fields are delimited.
   switch
      case dt.getSourceType() = DTASCIIVar :
         fldType = "Delimited"
         fldDelimiter = dt.getSourceDelimiter()
         if dt.getSourceDelimitedFields() = DTDelimAllFields then
            fldDelimitedFields = "All"
         else
            fldDelimitedFields = "Text"
         endIF
         fldSeparator = dt.getSourceSeparator()
      case dt.getSourceType() = DTASCIIFixed :
         fldType = "Fixed"
      otherwise :
         msgInfo("Hello","File missing or not text.")
   endSwitch
endMethod
```

▪

getSourceType method

See also Example DataTransfer Type

Retrieves the source file type (constant).

Syntax

`getSourceType ( ) SmallInt`

Description

getSourceType retrieves the source file type (constant). Note that the version part of the file type is typically unimportant for the source and should be ignored.

getSourceType example

This example uses **getSourceType** to determine the file type of the source file.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setSource("iesimpld.txt")

;The following lines check to see what Paradox 7 determined as the type of
;of the source file.  If it is delimited, Paradox 7 determines the separator,
;delimiter and which fields are delimited.
   switch
      case dt.getSourceType() = DTASCIIVar :
         fldType = "Delimited"
         fldDelimiter = dt.getSourceDelimiter()
         if dt.getSourceDelimitedFields() = DTDelimAllFields then
            fldDelimitedFields = "All"
         else
            fldDelimitedFields = "Text"
         endIF
         fldSeparator = dt.getSourceSeparator()
      case dt.getSourceType() = DTASCIIFixed :
         fldType = "Fixed"
      otherwise :
         msgInfo("Hello","File missing or not text.")
   endSwitch
endMethod
```

▪

importASCIIFix procedure

See also Example DataTransfer Type

Imports data from a fixed record length ASCII text file to a table.

Syntax

```
importASCIIFix ( const fileName String, const tableName String, const
specTableName String [ , const ANSI Logical ] ) Logical
```

Description

importASCIIFix imports data to a table from an ASCII (text) file in which the fields in each record are the same length. If the destination table exists, its contents are replaced with the imported data. If the table does not exist, this method creates it. This method duplicates the function of the Import Data dialog box.

The argument *fileName* specifies the source file from which to import data. The argument *tableName* specifies the destination table to which data is imported. Dates and numbers are formatted as specified in the Windows Control Panel.

Note: The extension in *tableName* specifies the table type. Use .DB to specify a Paradox table, or .DBF to specify a dBASE table. If you omit the extension, the data is imported to a Paradox table by default.

The argument *specTableName* is the name of a table that specifies the layout for the imported data. The structure of the file specified with *specTableName* is as follows:

Field name	Type & size	Description
Field Name	A 25	Name of a field to import.
Type	A 4	Field type (must be a valid Paradox or dBASE field specification; see Table::**create** for details).
Start	S	Number of the column where you want the field value to begin.
Length	S	Field size.

This table serves the same purpose as IMPORT.DB, which is created automatically in your private directory when you import a table interactively with the Import Data dialog box. It defines the structure of the table that will receive the imported data. For each field you import, enter the field's name; its type (must be a valid Paradox or dBASE field specification; see Table::**create** for details); its Start position (the column where you want the field value to begin); and its Length (size).

ANSI specifies whether to use the ANSI or OEM character set: True specifies ANSI, and False specifies OEM (default).

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

importASCIIFix example

This example imports data from a text file ORDERS.TXT to the ORDERS.DB table; reads the ORDERS.DB table's structure from the ORDERIMP.DB table; and specifies the OEM character set.

```
method pushButton ( var eventInfo Event )
   importASCIIFix ( "orders.txt", "orders.db", "orderimp.db", False )
endMethod
```

▪

importASCIIVar procedure

See also Example DataTransfer Type

Imports data from a variable record length ASCII text file to a table.

Syntax

```
importASCIIVar ( const fileName String, const tableName String [ , const
separator String, const delimiter String, const allFieldsDelimited Logical,
const ANSI Logical ] ) Logical
```

Description

importASCIIVar imports data from an ASCII file in which the (variable length) field values in each record may be delimited by an optionally specified character. If the destination table exists, its contents are replaced with the imported data. If the table does not exist, this method creates it. This method duplicates the function of the Import Data dialog box.

The argument *fileName* specifies the source file from which to import data. The argument *tableName* specifies the destination table to which data is imported. Dates and numbers are formatted as specified in the Windows Control Panel.

Note: The extension in *tableName* specifies the table type: .DB specifies a Paradox table (default), and .DBF a dBASE table.

separator (optional) specifies the character that surrounds field values in the target file; choose a comma or any other single character, including special characters such as tabs. *delimiter* (optional) specifies the character that defines the limits of field values in the target file; use the empty string if you do not want any character to delimit the fields. *allFieldsDelimited* specifies which fields are delimited: True specifies all field types; False specifies only text field types, alphanumeric or character (default setting).

Note: Paradox truncates strings longer than 255 characters when it imports them.

ANSI specifies whether to use the ANSI or OEM character set: True specifies ANSI, and False specifies OEM (default setting).

The default settings: fields separated by commas; fields delimited by quotes, but only text fields; and the OEM character set is used.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

importASCIIVar example

This example imports data from a text file ORDERS.TXT to the ORDERS.DB table; uses commas to delimit field values; does not enclose each value; delimits all fields; and uses the ANSI character set.

```
method pushButton ( var eventInfo Event )
   importASCIIVar ( "orders.txt", "orders.db", ",", "", True, True )
endMethod
```

Imports ASCII Delimited Text to Paradox (long form):

```
var
   dt DataTransfer
endVar

dt.setSource("orders.txt", DTAsciiVar)
dt.setDest("orders.db")
dt.setSourceDelimiter("")
dt.setSourceSeparator(",")
dt.setSourceCharSet(dtANSI)
dt.setSourceDelimitedFields(dtDelimAllFields)

dt.TransferData()
endMethod
```

Imports ASCII Delimited Text to Paradox (medium form):

```
var
   dt DataTransfer
endVar

dt.setSource("NewRecords.txt", DTAsciiVar)
dt.setDest("TimeCards.db")
dt.TransferData()
```

Imports ASCII Delimited Text to Paradox (short form):

```
ImportASCIIVar("NewRecords.txt", "TimeCards.db")
```

▪

importSpreadsheet procedure

See also Example DataTransfer Type

Imports the data from a spreadsheet file to a table.

Syntax

```
importSpreadsheet ( const fileName String, const tableName String, const
fromCell String, const toCell String [ , const getFieldNames Logical ] )
Logical
```

Description

importSpreadsheet imports the data from a spreadsheet file to a table. If the table does not exist, this method creates it. This method duplicates the function of the Import Data dialog box. Paradox converts each row to a record and each column to a field.

fileName specifies the spreadsheet file to import from, and *tableName* the table to import to. *fromCell* specifies the upper left cell, and *toCell* the lower right cell, of the block to import. *getFieldNames* specifies whether to use the spreadsheet's top row cells as column headers for the table: True creates column headers (default); False does not.

Note: The extension in *fileName* specifies the format of the spreadsheet file. The extensions and their formats:

Extension	Format
WB1, WB2, WB3	Quattro Pro Win
WQ1	Quattro Pro DOS
WKQ	Quattro
WK1	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0/5.0

Note: The extension in *tableName* specifies the table type. .DB specifies a Paradox table (default), and .DBF a dBASE table.

Paradox automatically assigns a field type to each column of data. The following table shows how Paradox determines a field's type:

Spreadsheet value	Paradox field type	dBASE field type
Label	Alpha	Character
Integer	Short	Float number (5,0)
Number	Number	Float number (20,4)
Currency	Money	Float number (20,4)
Date	Date	Date

The following rules determine which category a column falls into:

- A column containing a label (text) is converted to an alpha field (or character field for a dBASE table).
- A column containing both dates and numbers is converted to an alpha field (or character field for a dBASE table).
- A column containing only values formatted as currency is converted to a money field in a Paradox table.
- A column containing both currency and number (or integer) values is converted to a number field.

As a result of these conversion rules, Paradox often imports dates and numbers from unedited spreadsheets as alpha fields. For example, spreadsheets often have rows of hyphens separating sections of numbers. Since only an alphanumeric field can have both numbers and hyphens, each spreadsheet column is converted to an alpha field even though it contains mostly numbers.

To avoid conversion problems, edit the spreadsheet before importing it. Follow these steps:

1. Remove extraneous entries (such as hyphens, asterisks, and exclamation points).
2. Make sure each column contains only one kind of data and uses only one formatting option.
3. Place the titles you want to become table column headings in the top row of the selected range, because Paradox uses the first row that contains text to generate field names. (If there are no column titles in the spreadsheet, make sure to include the optional parameter *getFieldNames* with a False value.)

If the table does not have the format you want after you import it, restructure it in Paradox.

This method was previously included in the System type and has been moved to the DataTransfer type in version 7.

importSpreadsheet example

This example imports data from a Quattro Pro for Windows file to the ORDERS.DB table. Uses the first row of the spreadsheet file as column headers for the table.

```
method pushButton ( var eventInfo Event )
   importSpreadsheet ( "orders.wb1", "orders.db","A:A1", "A:H25", True )
endMethod
```

■

loadDestSpec method

See also   Example   DataTransfer Type

Loads a fixed length import file specification.

Syntax

```
loadDestSpec ( const tableName String )
```

Description

loadDestSpec loads a fixed length import file spec. The argument *tableName* specifies the table to use as the pattern for the destination specification. This method applies only when the destination is a fixed length ASCII text file. Leave it unset for other cases.

loadDestSpec example

The following examples specify a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Text

```
var
   dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIFixed Then
   dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

▪

loadSourceSpec method

See also   Example   DataTransfer Type

Loads a fixed length import file specification.

Syntax

loadSourceSpec (`const` *tableName* `String`)

Description

loadSourceSpec loads a fixed length import file spec. The argument *tableName* specifies the table to use as the pattern for the source specification. This method applies only when the source is a fixed length ASCII text file. Leave it unset for other cases.

loadSourceSpec example

The following examples specify a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Text

```
var
   dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIFixed Then
   dt.loadSourceSpec ( "SpecTable" )
EndIf
dt.setSource ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

▪

setAppend method

See also Example DataTransfer Type

Appends data to the existing table.

Syntax

setAppend (const ***AppendToTable*** Logical **)**

Description

setAppend appends data to the existing table when set to True. Overwrites the table when set to False. Ignored for new tables. This method applies only when the destination is a table. Leave it unset for other destinations.

setAppend example

The following example specifies a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Text

```
var
   dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIFixed Then
   dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

▪

setDest method

See also Example DataTransfer Type

Specifies the file or table to receive data.

Syntax

`setDest ( const destName String, [ const destType SmallInt ] )`

Description

setDest specifies the file or table to receive data (*destName*) and the file's type (*destType*). If no *destType* specified, the file extension will determine its type.

The following file types are recognized by Paradox 7:

FileType	Description
DT123V1	Lotus 123 (.WKS)
DT123V2	Lotus 123 (.WK1)
DTASCIIFixed	ASCII Fixed (BDE)
DTASCIIVar	ASCII Delimited
DTAuto	Automatically determine file type based on file extension
DTdBase3	Export to dBASE III+ compatible
DTdBase4	Export to dBASE IV compatible
DTdBase5	Export to dBASE 5 compatible, Import any dBASE
DTdBaseAny	Import (or Export) any dBASE version
DTExcel4	Excel Version 3,4 (.XLS)
DTExcel5	Excel Version 5 (.XLS)
DTParadox3	Export to Paradox 3 compatible
DTParadox4	Export to Paradox 4 compatible
DTParadox5	Export to Paradox 5 compatible
DTParadox7	Export to Paradox 7 compatible
DTParadoxAny	Import (or Export) any Paradox version
DTQPW1	Quattro Pro Windows 1,5 (.WB1)
DTQPW6	Quattro Pro Windows 6 (.WB2)
DTQPW95	Quattro Pro Windows 95 (.WB3)
DTQuattro	Quattro DOS (.WKQ)
DTQuattroPro	Quattro Pro DOS (.WQ1)

setDest example

The following example specifies a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Spreadsheet

```
var
   dt DataTransfer
endVar
dt.setSource ( "MYFILE.WKS" )
dt.setDest ( "New Data.db" )
dt.setProblems ( True )
dt.transferData ( )
```

setDestCharSet method

See also Example DataTransfer Type

Sets the file character set to dtOEM or dtANSI.

Syntax

`setDestCharSet ( const CharSetCode SmallInt )`

Description

setDestCharSet sets the file character set to dtOEM or dtANSI. This method applies only when the destination is a fixed or delimited ASCII text file. Leave it unset for other cases.

setDestCharSet example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. It uses **setDestCharSet** to specify using the ANSI character set. To specify using the OEM character set, use the DTOEM constant with **setDestCharSet**.

```
method pushButton(var eventInfo Event)
   var
      dt      DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   dt.setDestDelimiter("'")
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   ;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
   dt.setDestCharSet(DTAnsi)

   ;run Export
   dt.transferData()
endMethod
```

■

setDestDelimitedFields method

See also Example DataTransfer Type

Sets the delimited fields setting to DtDelimAllFields or DtDelimJustText.

Syntax

`setDestDelimitedFields ( const` ***`delimitCode`*** **`SmallInt )`**

Description

setDestDelimitedFields sets the delimited fields setting. The argument *delimitCode* specifies one of two possible delimiter codes: DtDelimAllFields or DtDelimJustText.

setDestDelimitedFields only applies when the destination is a delimited ASCII text file.

setDestDelimitedFields example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. It uses **setDestDelimitedFields** to specify surrounding text fields only. To specify all fields to be delimited, use the DTDelimAllFields constant with **setDestDelimitedFields**.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   dt.setDestDelimiter("'")
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   ;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
   dt.setDestCharSet(DTAnsi)

   ;run Export
   dt.transferData()
endMethod
```

setDestDelimiter method

See also Example DataTransfer Type

Sets the delimiter to the specified character.

Syntax

`setDestDelimiter ( const delimiterChar String )`

Description

setDestDelimiter sets the delimiter to the character specified by *delimiterChar*. The default delimiter is a comma.

setDestDelimiter only applies when the destination is a delimited ASCII text file.

setDestDelimiter example

This example exports the ORDERS.DB table into an ASCII delimited text file. The delimiter specified is the single quote character.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   dt.setDestDelimiter("'")
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   ;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
   dt.setDestCharSet(DTAnsi)

   ;run Export
   dt.transferData()
endMethod
```

setDestFieldNamesFromFirst method

See also Example DataTransfer Type

Sets the field names using the data in the first row of input.

Syntax

```
setDestFieldNamesFromFirst ( const namesFirst Logical )
```

Description

setDestFieldNamesFromFirst sets the the first row of the destination file to be the field names of the table. Setting *namesFirst* to True creates the first row as field names and data will begin on the second row.

setDestFieldNamesFromFirst applies to both Spreadsheets and delimited text files.

setDestFieldNamesFromFirst example

This example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. The **setDestFieldNamesFromFirst** is used to create the first row of the text file with field names.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setDest("ordinfo.txt", DTASCIIVar)
   dt.setSource("orders.db")

   ;Specify the single quote (') to surround the fields.
   ;The delimited fields will be text fields only.
   dt.setDestDelimiter("'")
   dt.setDestDelimitedFields(DTDelimJustText)

   ;Specify the tab character to separate the fields.
   dt.setDestSeparator("\t")

   ;Set the first row of the ORDINFO.TXT to be the field names
   dt.setDestFieldNamesFromFirst(True)

   ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
   ;character set.
   dt.setDestCharSet(DTAnsi)

   ;run Export
   dt.transferData()
endMethod
```

setDestSeparator method

See also Example DataTransfer Type

Set the separator character of delimited ASCII text.

Syntax

```
setDestSeparator ( const separatorChar String )
```

Description

setDestSeparator sets the separator to the character specified by *separatorChar*. The default separator is the comma character.

setDestSeparator only applies when the destination is a delimited ASCII text file.

setDestSeparator example

The following examples specify a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Export to Text

```
var
   dt DataTransfer
endVar
msgInfo("Info", "The current source separator is " + dt.getSourceSeparator)
dt.setSource ( "ANSWER.DB" )
msgInfo("Info", "The current destination separator is " +
dt.getDestSeparator)
dt.SetDest ( "NEWFILE.TXT" )
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

setKeyviol method

See also Example DataTransfer Type

Writes violations to the Keyviol table.

Syntax

setKeyviol (const ***GenerateKeyviol*** Logical **)**

Description

setKeyviol writes violations to the Keyviol table. The argument *generateKeyviol* is a logical that is set to True to write violations to the Keyviol table. *generateKeyviol* is ignored for unkeyed tables. This method applies only when the destination is a table. Leave it unset for other destinations.

setKeyviol example

The following example specifies a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Imports ASCII Delimited Text to Paradox (long form):

```
var
   dt DataTransfer
endVar

; Fields Quoted even if numeric
dt.setAppend(True)        ; Append to an existing Table
dt.setProblems(True)      ; Generate a Problems Table (if Any)
dt.setKeyviol(True)       ; Generate a Keyviol Table (if any)

dt.setSource("NewRecords.txt", DTAsciiVar)
dt.setDest("TimeCards.db")
dt.TransferData()
```

▪

setProblems method

See also Example DataTransfer Type

Writes problems to the Problems table.

Syntax

`setProblems ( const generateProblems Logical )`

Description

setProblems writes problems to the Problems table. This method applies only when the destination is a table. Leave it unset for other destinations.

setProblems example

The following example specifies a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Imports ASCII Delimited Text to Paradox (long form):

```
var
   dt DataTransfer
endVar

; Fields Quoted even if numeric
dt.setAppend(True)       ; Append to an existing Table
dt.setProblems(True)     ; Generate a Problems Table (if Any)
dt.setKeyviol(True)      ; Generate a Keyviol Table (if any)

dt.setSource("NewRecords.txt", DTAsciiVar)
dt.setDest("TimeCards.DB")
dt.TransferData()
```

▪

setSource method

See also   Example   DataTransfer Type

Specifies the file or table to act as a data source and its type.

Syntax

setSource **(** const ***sourceName*** String, [const ***sourceType*** SmallInt] **)**

Description

setSource specifies the file or table to use as the source of data. The type of the file (the application that generated the file) is specified with *sourceType*. If no type is specified, the extension of *sourceName* is used to determine the file's type.

The following file types are recognized by Paradox 7:

FileType	Description
DT123V1	Lotus 123 (.WKS)
DT123V2	Lotus 123 (.WK1)
DTASCIIFixed	ASCII Fixed (BDE)
DTASCIIVar	ASCII Delimited
DTAuto	Automatically determine file type based on file extension
DTdBase3	Export to dBASE III+ compatible
DTdBase4	Export to dBASE IV compatible
DTdBase5	Export to dBASE 5 compatible, Import any dBASE
DTdBaseAny	Import (or Export) any dBASE version
DTExcel4	Excel Version 3,4 (.XLS)
DTExcel5	Excel Version 5 (.XLS)
DTParadox3	Export to Paradox 3 compatible
DTParadox4	Export to Paradox 4 compatible
DTParadox5	Export to Paradox 5 compatible
DTParadox7	Export to Paradox 7 compatible
DTParadoxAny	Import (or Export) any Paradox version
DTQPW1	Quattro Pro Windows 1,5 (.WB1)
DTQPW6	Quattro Pro Windows 6 (.WB2)
DTQPW95	Quattro Pro Windows 95 (.WB3)
DTQuattro	Quattro DOS (.WKQ)
DTQuattroPro	Quattro Pro DOS (.WQ1)

setSource example

The following example specifies a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Spreadsheet

```
var
   dt DataTransfer
endVar
dt.setSource ( "MYFILE.WKS" )
dt.setDest ( "New Data.db" )
dt.setProblems ( True )
dt.transferData ( )
```

▪

setSourceCharSet method

See also Example DataTransfer Type

Sets the file character set to dtOEM or dtANSI.

Syntax

setSourceCharSet (const *charSetCode* SmallInt)

Description

setSourceCharSet sets the file character set. The argument *charSetCode* specifies one of two character sets: dtOEM or dtANSI. This method applies only when the source is a fixed or delimited ASCII text file. Leave it unset for other cases.

setSourceCharSet example

The following example specifies a DataTransfer data type. This structure is used with the transferData method. It is assumed that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method cmTransfer() is within the scope of the variable, dt.

```
method cmTransfer()   ;this example completes a DataTransfer

   dt.setSource("CUSTOMER.TXT", DTASCIIVar)  ; sets the datatransfer source
                                             ; to CUSTOMER.TXT
   dt.setSourceSeparator("/")   ; specifies the forward slash "/" character
                                ; to separate each field
   dt.setSourceDelimiter("'")   ; specifies the single quote to surround
                                ; the fields
   dt.setSourceDelimitedFields(DTDelimJustText)  ; specifies that the single
                                              ; quote (delimiter) surrounds
                                              ; only text fields of the
                                              ; source file
   dt.setSourceCharSet(DTANSI)   ; specifies that the character set used
                                 ; when creating the source file
                                 ; was the ANSI character set

   dt.setSourceFieldNamesFromFirst(False)   ; specifies to use the first
                                            ; row of the source file as
                                            ; field names
   dt.setDest("NEWCUST.DB")   ; sets the destination file to NEWCUST.DB
   dt.setProblems(True)   ; specifies to create a PROBLEMS.DB if there are
                          ; any problems importing the source file
   dt.transferData()   ; executes the data transfer.  In this case it
                       ; imports the CUSTOMER.TXT file as NEWCUST.DB.
   dt.empty()   ; empties the dt variable structure to set it up for
                ; a new transfer.

endmethod
```

setSourceDelimitedFields method

See also Example DataTransfer Type

Sets the delimited fields setting to DtDelimAllFields or DtDelimJustText.

Syntax

setSourceDelimitedFields (**`const` ***delimitCode* `SmallInt` **)**

Description

setSourceDelimitedFields sets the delimited fields setting to DtDelimAllFields or DtDelimJustText.

setSourceDelimitedFields only applies when the source or the destination is a delimited ASCII text file.

setSourceDelimitedFields example

The following example specifies a DataTransfer data type. This structure is used with the transferData method.   It is assumed that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method cmTransfer() is within the scope of the variable, dt.

```
method cmTransfer()   ;this example completes a DataTransfer

   dt.setSource("CUSTOMER.TXT", DTASCIIVar)  ; sets the datatransfer source
                                             ; to CUSTOMER.TXT
   dt.setSourceSeparator("/")   ; specifies the forward slash "/" character
                                ; to separate each field
   dt.setSourceDelimiter("'")   ; specifies the single quote to surround
                                ; the fields
   dt.setSourceDelimitedFields(DTDelimJustText)  ; specifies that the single
                                              ; quote (delimiter) surrounds
                                              ; only text fields of the
                                              ; source file
   dt.setSourceCharSet(DTANSI)   ; specifies that the character set used
                                 ; when creating the source file
                                 ; was the ANSI character set

   dt.setSourceFieldNamesFromFirst(False)   ; specifies to use the first
                                            ; row of the source file as
                                            ; field names
   dt.setDest("NEWCUST.DB")   ; sets the destination file to NEWCUST.DB
   dt.setProblems(True)   ; specifies to create a PROBLEMS.DB if there are
                          ; any problems importing the source file
   dt.transferData()   ; executes the data transfer.  In this case it
                       ; imports the CUSTOMER.TXT file as NEWCUST.DB.
   dt.empty()   ; empties the dt variable structure to set it up for
                ; a new transfer.

endmethod
```

setSourceDelimiter method

See also Example DataTransfer Type

Sets the delimiter to the specified character.

Syntax

```
setSourceDelimiter ( const delimiterChar String )
```

Description

setSourceDelimiter sets the delimiter to the character specifed by *delimiterChar*. The default delimiter is a comma.

setSourceDelimiter only applies when the source or the destination is a delimited ASCII text file.

setSourceDelimiter example

The following example specifies a DataTransfer data type. This structure is used with the transferData method.   It is assumed that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method cmTransfer() is within the scope of the variable, dt.

```
method cmTransfer()   ;this example completes a DataTransfer

   dt.setSource("CUSTOMER.TXT", DTASCIIVar)  ; sets the datatransfer source
                                             ; to CUSTOMER.TXT
   dt.setSourceSeparator("/")   ; specifies the forward slash "/" character
                                ; to separate each field
   dt.setSourceDelimiter("'")   ; specifies the single quote to surround
                                ; the fields
   dt.setSourceDelimitedFields(DTDelimJustText)  ; specifies that the single
                                              ; quote (delimiter) surrounds
                                              ; only text fields of the
                                              ; source file
   dt.setSourceCharSet(DTANSI)   ; specifies that the character set used
                                 ; when creating the source file
                                 ; was the ANSI character set

   dt.setSourceFieldNamesFromFirst(False)   ; specifies to use the first
                                            ; row of the source file as
                                            ; field names
   dt.setDest("NEWCUST.DB")   ; sets the destination file to NEWCUST.DB
   dt.setProblems(True)   ; specifies to create a PROBLEMS.DB if there are
                          ; any problems importing the source file
   dt.transferData()   ; executes the data transfer.  In this case it
                       ; imports the CUSTOMER.TXT file as NEWCUST.DB.
   dt.empty()   ; empties the dt variable structure to set it up for
                ; a new transfer.

endmethod
```

setSourceFieldNamesFromFirst method

See also Example DataTransfer Type

Sets the field names using the data in the first row of input.

Syntax

```
setSourceFieldNamesFromFirst ( const namesFirst Logical)
```

Description

setSourceFieldNamesFromFirst set the field names to the data that is in the first row of the input data. Setting *namesFirst* to True always skips the first row. However, the field names only apply to newly created tables without explicit field names.

setSourceFieldNamesFromFirst only applies when the source is a spreadsheet.

setSourceFieldNamesFromFirst example

The following example specifies a DataTransfer data type. This structure is used with the transferData method.   It is assumed that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method cmTransfer() is within the scope of the variable, dt.

```
method cmTransfer()   ;this example completes a DataTransfer

   dt.setSource("CUSTOMER.TXT", DTASCIIVar)  ; sets the datatransfer source
                                             ; to CUSTOMER.TXT
   dt.setSourceSeparator("/")   ; specifies the forward slash "/" character
                                ; to separate each field
   dt.setSourceDelimiter("'")   ; specifies the single quote to surround
                                ; the fields
   dt.setSourceDelimitedFields(DTDelimJustText)  ; specifies that the single
                                              ; quote (delimiter) surrounds
                                              ; only text fields of the
                                              ; source file
   dt.setSourceCharSet(DTANSI)   ; specifies that the character set used
                                 ; when creating the source file
                                 ; was the ANSI character set

   dt.setSourceFieldNamesFromFirst(False)   ; specifies to use the first
                                            ; row of the source file as
                                            ; field names
   dt.setDest("NEWCUST.DB")   ; sets the destination file to NEWCUST.DB
   dt.setProblems(True)   ; specifies to create a PROBLEMS.DB if there are
                          ; any problems importing the source file
   dt.transferData()   ; executes the data transfer.  In this case it
                       ; imports the CUSTOMER.TXT file as NEWCUST.DB.
   dt.empty()   ; empties the dt variable structure to set it up for
                ; a new transfer.

endmethod
```

■

setSourceRange method

See also Example DataTransfer Type

Specifies a sub range of the spreadsheet to import.

Syntax

setSourceRange (const ***range*** String **)**

Description

setSourceRange specifies a sub range of the spreadsheet to import. May be a named *range*, a page name, or an explicit range in QPW or Excel format.

setSourceRange only applies when the source is a spreadsheet.

setSourceRange example

The following illustrates using the **setSourceRange** method. Use this method to specify the range in a spreadsheet to import.   Named ranges as well as standard ranges will work.

```
method pushButton(var eventInfo Event)
   var
      dt       DataTransfer
   endVar
   dt.setSource("092595.wb2")

   ;Set the range to import from the spreadsheet.
   ;Either named range or specified range (ie. Page1:A1..Page3:AB10)
   dt.setSourceRange("myRange")
   dt.setSourceFieldNamesFromFirst(True)
   dt.setDest("delme09.db")

   ;Prompt the user to verify range to import.  getSourceRange returns the
   ;actual range notation.
   view(dt.getSourceRange(),"Import Range")
   dt.transferData()
endMethod
```

▪

setSourceSeparator method

See also Example DataTransfer Type

Set the separator character of delimited ASCII text.

Syntax

```
setSourceSeparator ( const separatorChar String )
```

Description

setSourceSeparator sets the separator to the character specified by *separatorChar*. The default separator is the comma character.

setSourceSeparator only applies when the source or the destination is a delimited ASCII text file.

setSourceSeparator example

The following example specifies a DataTransfer data type. This structure is used with the transferData method.   It is assumed that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method cmTransfer() is within the scope of the variable, dt.

```
method cmTransfer()   ;this example completes a DataTransfer

   dt.setSource("CUSTOMER.TXT", DTASCIIVar)  ; sets the datatransfer source
                                             ; to CUSTOMER.TXT
   dt.setSourceSeparator("/")   ; specifies the forward slash "/" character
                                ; to separate each field
   dt.setSourceDelimiter("'")   ; specifies the single quote to surround
                                ; the fields
   dt.setSourceDelimitedFields(DTDelimJustText)  ; specifies that the single
                                              ; quote (delimiter) surrounds
                                              ; only text fields of the
                                              ; source file
   dt.setSourceCharSet(DTANSI)   ; specifies that the character set used
                                 ; when creating the source file
                                 ; was the ANSI character set

   dt.setSourceFieldNamesFromFirst(False)   ; specifies to use the first
                                            ; row of the source file as
                                            ; field names
   dt.setDest("NEWCUST.DB")   ; sets the destination file to NEWCUST.DB
   dt.setProblems(True)   ; specifies to create a PROBLEMS.DB if there are
                          ; any problems importing the source file
   dt.transferData()   ; executes the data transfer.  In this case it
                       ; imports the CUSTOMER.TXT file as NEWCUST.DB.
   dt.empty()   ; empties the dt variable structure to set it up for
                ; a new transfer.

endmethod
```

transferData method

See also Example DataTransfer Type

Copies data from the source to the destination.

Syntax

```
transferData ( )
```

Description

transferData copies data from the source to the destination. Either the source, destination, or both must be a table.

transferData example

The following examples specify a DataTransfer data type. Use this to build up an Import or Export specification, and then call the **transferData** method.

Import from Spreadsheet

```
var
   dt DataTransfer
endVar
dt.setSource ( "MYFILE.WKS" )
dt.setDest ( "New Data.db" )
dt.setProblems ( True )
dt.transferData ( )
```

Import from Text

```
var
   dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIFixed Then
   dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

Export to Text

```
var
   dt DataTransfer
endVar
dt.setSource ( "ANSWER.DB" )
dt.SetDest ( "NEWFILE.TXT" )
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

■

ObjectPAL Toolbar type reference

Changes ■

The Toolbar type contains methods that create, delete, manipulate, and modify Toolbars.

The Toolbar type includes several derived methods from the AnyType type.

Methods for the Toolbar type

AnyType ■	**Toolbar**
blank	**addButton**
dataType	**attach**
isAssigned	**create**
isBlank	**createTabbed**
isFixedType	**empty**
unAssign	**getPosition**
	getState
	hide
	isVisible
	remove
	removeButton
	setPosition
	setState
	show
	unAttach

Changes to Toolbar type methods

The entire Toolbar type is new for version 7.

▪

addButton method

See also Example Toolbar Type

Adds a button to a Toolbar.

Syntax

`1.` **`addButton (`** `const` ***`idCluster`*** `SmallInt, const` ***`buttonType`*** `SmallInt, const` ***`idCommand`*** `SmallInt, const` ***`grBmp`*** `Graphic, const` ***`buttonHelp`*** `String` **`)`** `Logical`

`2.` **`addButton (`** `const` ***`idCluster`*** `SmallInt, const` ***`buttonType`*** `SmallInt, const` ***`idCommand`*** `SmallInt, const` ***`idBmp`*** `SmallInt, const` ***`buttonHelp`*** `String` **`)`** `Logical`

Description

addButton adds a button to a Toolbar. The position of the new button on the Toolbar is specified by *idCluster*. *idCluster* is the Cluster Identifier, which identifies a specific area on the Toolbar. *idCluster* is an integer that ranges from 0 to 12. The type of button added is specified by *buttonType*. There are four button types: pushbutton, radiobutton, togglebutton, and repeatbutton. The menu command that is sent when the button is pressed is specified by *idCommand*. The contents of the small popup window that appears when the cursor is placed on the new button is specified with the string *buttonHelp*.

Syntax 1 is used to add a button to the Toolbar using a graphic bitmap (*grBmp*) to specify the button's image on the Toolbar. This allows the use of a user-defined bitmap file or a bitmap object of a graphic type stored in a table.

Syntax 2 is used to add a button to the Toolbar using a bitmap constant. The bitmap constant specifies the button's image on the Toolbar. This method allows the creation of a button using any of the defined Toolbar button bitmaps in the system resource.

The only item that can be added to a Toolbar is a button.

addButton returns True if the button is successfully created.

▪

addButton example

This example creates a Toolbar called 'Edit' and adds three buttons to the Toolbar using defined Paradox bitmap constants.

```
method pushButton (var eventInfo Event)
var
   tb  Toolbar
endvar

   ;// Create a Toolbar called "Edit" with 3 buttons: Cut, Copy, Paste
if tb.create("Edit") then
   tb.addButton(ToolbarEditCluster, ToolbarButtonPush,
                MenuEditCut, BitmapEditCut, "Cut")

   tb.addButton(ToolbarEditCluster, ToolbarButtonPush,
                MenuEditCopy, BitmapEditCopy, "Copy")

   tb.addButton(ToolbarEditCluster, ToolbarButtonPush,
                MenuEditPaste, BitmapEditPaste, "Paste")

endif
endMethod
```

This example creates a Toolbar called 'File', adds three buttons using Paradox constants and adds a fourth button using a custom graphic object.

```
method pushButton (var eventInfo Event)
var
   tb  Toolbar
   gr  graphic
endvar

if tb.create("File") then
   tb.addButton(ToolbarFileCluster, ToolbarButtonPush,
                MenuTableOpen, BitmapOpenTable, "Open Table")

   tb.addButton(ToolbarFileCluster, ToolbarButtonPush,
                MenuFormOpen, BitmapOpenForm, "Open Form")

   tb.addButton(ToolbarFileCluster, ToolbarButtonPush,
                MenuReportOpen, BitmapOpenReport, "Open Report")

   ;// Add a button with a custom bitmap (pick a valid name)
   gr.readFromFile("Alias.bmp")
   tb.addButton(ToolbarModeCluster, ToolbarButtonPush,
                MenuFileAliases, gr, "Alias")
endif
endMethod
```

attach method

See also Example Toolbar Type

Binds a Toolbar type to an existing Toolbar.

Syntax

```
attach ( const toolbarName String ) Logical
```

Description

attach binds a Toolbar type to an existing Toolbar using the name specified in *toolbarName*. The reserved name 'Standard' can be used to attach to the Paradox Toolbar.

You can access a Toolbar by attaching to an existing one or creating a new one.

▪

attach example

This example attaches the Toolbar named “MyToolbar”.

```
method pushbutton (var eventInfo Event)
var
   tbar      Toolbar
endvar

   if tbar.attach("MyToolbar") then
      msginfo("Attach", "Successful")
   else
      msginfo("Attach", "Failed")
   endif
endMethod
```

▪

create method

See also Example Toolbar Type

Creates a Toolbar.

Syntax

`create ( const toolbarName String [, const parentToolbarName String ]) Logical`

Description

create creates a Toolbar identified by *toolbarName*. The name cannot be 'Standard', which is reserved for the Paradox Toolbar. *toolbarName* identifies the Toolbar and is used in the caption when the Toolbar is floating. *parentToolbarName* is the name of the parent Toolbar of the new Toolbar.

You can access a Toolbar by attaching to an existing one or creating a new one.

create example

This example uses **createTabbed** to create a tabbed Toolbar (named ‘Test’) using two Toolbars created with the **create** method (the ‘Edit’ and ‘File’ Toolbars).

```
method pushButton (var eventInfo Event)
var
   tbTabbed  Toolbar
   tbEdit    Toolbar
   tbFile    Toolbar
endvar

;// Create a tabbed Toolbar called "Test"
;// that will be composed of two Toolbars:
;// "Edit" and "File"
if tbTabbed.createTabbed("Test") then
   ; Create a Toolbar called "Edit" with 3 buttons: Cut, Copy, Paste
   if tbEdit.create("Edit", "Test") then
      tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
                       MenuEditCut, BitmapEditCut, "Cut")

      tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
                       MenuEditCopy, BitmapEditCopy, "Copy")

      tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
                       MenuEditPaste, BitmapEditPaste, "Paste")
   endif

   if tbFile.create("File", "Test") then
      tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
                       MenuTableOpen, BitmapOpenTable, "Open Table")

      tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
                       MenuFormOpen, BitmapOpenForm, "Open Form")

      tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
                       MenuReportOpen, BitmapOpenReport, "Open Report")
   endif
endif
endMethod
```

▪

createTabbed method

See also Example Toolbar Type

Creates a tabbed Toolbar.

Syntax

`createTabbed ( const toolbarName String ) Logical`

Description

createTabbed creates a tabbed Toolbar. *toolbarName* identifies the new Toolbar and is used in the caption when the Toolbar is floating. The name cannot be 'Standard', which is reserved for the Paradox Toolbar.

▪

createTabbed example

This example uses **createTabbed** to create a tabbed Toolbar (named ‘Test’) using two Toolbars created with the **create** method (the ‘Edit’ and ‘File’ Toolbars).

```
method pushButton (var eventInfo Event)
var
   tbTabbed  Toolbar
   tbEdit    Toolbar
   tbFile    Toolbar
endvar

;// Create a tabbed Toolbar called "Test"
;// that will be composed of two Toolbars:
;// "Edit" and "File"
if tbTabbed.createTabbed("Test") then
   ; Create a Toolbar called "Edit" with 3 buttons: Cut, Copy, Paste
   if tbEdit.create("Edit", "Test") then
      tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
                       MenuEditCut,BitmapEditCut, "Cut")

      tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
                       MenuEditCopy,BitmapEditCopy, "Copy")

      tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
                       MenuEditPaste,BitmapEditPaste, "Paste")
   endif

   if tbFile.create("File", "Test") then
      tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
                       MenuTableOpen, BitmapOpenTable, "Open Table")

      tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
                       MenuFormOpen, BitmapOpenForm, "Open Form")

      tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
                       MenuReportOpen, BitmapOpenReport, "Open Report")
   endif
endif
endMethod
```

empty method

See also Example Toolbar Type

Removes all the existing buttons from the Toolbar.

Syntax

```
empty ( ) Logical
```

Description

empty removes all the existing buttons from the attached Toolbar.

empty returns True if the Toolbar is successfully emptied.

▪

empty example

This example attaches the Toolbar named "MyToolbar" and empties it.   If the attach did not succeed, this method prints the message "Unable to attach".

```
method pushbutton (var eventInfo Event)
var
   tbar      Toolbar
endvar

if tbar.attach("MyToolbar") then
   tbar.empty()
else
   msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

■

getPosition method

See also Example Toolbar Type

Returns the position of a floating Toolbar.

Syntax

`getPosition ( var x LongInt, var y LongInt ) Logical`

Description

getPosition returns the position of a floating Toolbar. The coordinates are in pixels and relative to the top/left corner of the screen.

getPosition example

This example displays the X and Y coordinates of the attached Toolbar, if it is named “MyToolbar”.

```
method pushbutton (var eventInfo Event)
var
   liX, liY  LongInt
   tbar      Toolbar
endvar

if tbar.attach("MyToolbar") then
   tbar.getPosition(liX, liY)
   liX.view(“X coordinate”)
   liY.view(“Y coordinate”)
endif
endMethod
```

▪

getState method

See also Example Toolbar Type

Gets the current state of the Toolbar.

Syntax

`getState ( ) Logical`

Description

getState gets the current state of the Toolbar.

getState returns True if the Toolbar state is successfully retrieved.

▪

getState example

This example displays the current state of the Toolbar named "MyToolbar". If this method can not attach to "MyToolbar" it prints an error "Unable to attach".

```
method pushbutton (var eventInfo Event)
var
   tbar      Toolbar
endvar

if tbar.attach("MyToolbar") then
   msgInfo("MyToolbar", "Current State: " + String(tbar.getState())
else
   msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

▪

hide method

See also Example Toolbar Type

Hides a Toolbar. Same as the procedure **hideToolbar**.

Syntax

`hide ( )` `Logical`

Description

hide hides a Toolbar. The function of **hide** is the same as the procedure **hideToolbar**. **hide** returns True if the Toolbar is successfully hidden.

▪

hide example

This example hides the Toolbar named "MyToolbar" if it is visible. If the Toolbar is not visible, this method shows it (changes it to visible).

```
method pushbutton (var eventInfo Event)
var
   tbar  Toolbar
endvar

if tbar.attach("MyToolbar") then
   if tbar.isVisible() then
      tbar.hide()
   else
      tbar.show()
   endif
endif

endMethod
```

isVisible method

See also Example Toolbar Type

Reports the visibility status of the Toolbar.

Syntax

`isVisible ( ) Logical`

Description

isVisible reports the visibility status of the Toolbar. **isVisible** returns True if the Toolbar is visible and False if the Toolbar is not visible. This method performs the same function as the **isToolbarShowing** procedure.

▪

isVisible example

This example prints a message stating whether the Toolbar named "MyToolbar" is visible or not. If this method can not attach to "MyToolbar" it prints an error "Unable to attach".

```
method pushbutton (var eventInfo Event)
var
   tbar  Toolbar
endvar

   if tbar.attach("MyToolbar") then
      if tbar.isVisible() then
         msgInfo("MyToolbar" , “Toolbar is Visible”)
      else
         msgInfo("MyToolbar" , “Toolbar is not Visible”)
      endif
   else
      msgInfo("Toolbar error", "Unable to attach.")
   endif
endMethod
```

▪

remove method

See also Example Toolbar Type

Removes the Toolbar from the screen.

Syntax

`remove ( ) Logical`

Description

remove removes the Toolbar from the screen. **remove** returns True if the Toolbar was successfully removed and False otherwise.

▪

remove example

This example removes the Toolbar named "MyToolbar". If this method can not attach to "MyToolbar" it prints an error "Unable to attach".

```
method pushbutton (var eventInfo Event)
var
   tbar  Toolbar
endvar

if tbar.attach("MyToolbar") then
   tbar.remove()
else
   msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

▪

removeButton method

See also Example Toolbar Type

Removes a button from the Toolbar.

Syntax

`removeButton ( const idCluster SmallInt, const idNum SmallInt ) Logical`

Description

removeButton removes a button from the Toolbar using the host cluster and the position in the cluster. The cluster is specified with *idCluster* and the position of the button in the cluster from left to right starting at 0 is specified with *idNum*. **removeButton** returns True if the button is successfully removed.

▪

removeButton example

This example removes the a button from the Toolbar named "MyToolbar". Both *idCluster* and *idNum* start with zero, so this example removes the third button from the second cluster. If this method can not attach to "MyToolbar" it prints an error "Unable to attach".

```
method pushbutton (var eventInfo Event)
var
   tbar  Toolbar
endvar

if tbar.attach("MyToolbar") then
   tbar.removebutton(1,2)       ;//idcluster=1, the 2nd from left
                                ;//idnum=2, the 3rd from left
else
   msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

setPosition method

See also Example Toolbar Type

Changes the position of a floating Toolbar.

Syntax

```
setPosition ( const x LongInt, const y LongInt ) Logical
```

Description

setPosition sets the position of a floating Toolbar to the coordinates specified in *x* and *y*. The *x* and *y* coordinates are in pixels and relative to the top/left corner of the screen. **setPosition** returns True if the position of the Toolbar is successfully changed.

▪

setPosition example

This example sets the position of the Toolbar names “MyToolbar” from it's current position to 500 pixels to the right and 400 pixels further up.

```
method pushbutton (var eventInfo Event)
var
   liX, liY  LongInt
   tbar      Toolbar
endvar

if tbar.attach("MyToolbar") then
   tbar.getPosition(liX, liY)
   view("From: " + string(liX) + ", "
                 + string(liY) +
          "To: " + string(liX + 2800) + " , "
                 + string(liY + 2800))
   tbar.setPosition(liX + 500, liY + 400)
endif

endMethod
```

▪

setState method

See also   Example   Toolbar Type

Sets the shape of the Toolbar to horizontal.

Syntax

`setState ( const state SmallInt ) Logical`

Description

setState sets the state of the Toolbar to the specified ***state***.

There are six Toolbar states:

ToolbarStateTop: docked at the top of the window
ToolbarStateLeft: docked at the left of the window
ToolbarStateRight: docked on the right side of the window
ToolbarStateBottom: docked at the bottom of the window
ToolbarStateFloatHorizontal: floating horizontally
ToolbarStateFloatVertical: floating vertically

setState returns True if the state of the Toolbar is successfully set.

▪

setState example

This example displays the current state of the Toolbar named "MyToolbar", then displays a dialog that allows the user to set the state of the Toolbar. If this method can not attach to "MyToolbar" it prints an error "Unable to attach".

```
method pushbutton (var eventInfo Event)
var
   siState  SmallInt
   tbar     Toolbar
endvar

if tbar.attach("MyToolbar") then
   siState = tbar.getState()
   siState.view("Enter State: (0-7)")
   tbar.setState(siState)
else
   msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

-

show method

See also Example Toolbar Type

Shows a Toolbar.

Syntax

`show ( ) Logical`

Description

show shows a Toolbar. This function is the same as the **showToolbar** procedure.

▪

show example

This example hides the Toolbar named "MyToolbar" if it is visible. If the Toolbar is not visible, this method shows it (changes it to visible).

```
method pushbutton (var eventInfo Event)
var
   tbar      Toolbar
endvar

if tbar.attach("MyToolbar") then
   if tbar.isVisible() then
      tbar.hide()
   else
      tbar.show()
   endif
endif

endMethod
```

▪

unAttach method

See also Example Toolbar Type

Removes the attachment to the Toolbar.

Syntax

`unAttach ( )` `Logical`

Description

unAttach removes the attachment to the Toolbar.

unAttach example

This example attaches the Toolbar named “MyToolbar”, sets its state and then unattaches.

```
method pushbutton (var eventInfo Event)
var
   tbar      Toolbar
endvar

   if tbar.attach("MyToolbar") then
      tbar.setState(ToolbarStateTop)
   tbar.unattach()
   endif

endMethod
```

▪

ObjectPAL mail type reference

Changes ▪

The Mail type allows you to compose electronic mail messages and transmit them via a MAPI compliant mail system (for example, Microsoft Mail). A variable of type MAIL holds a single mail message. It also holds current mail session status (set by **logon**), so that multiple mail messages can be sent (sequentially) in a single session. Declare variables of type MAIL to facilitate the manipulation of mail messages. Then, use the Mail methods to set (and retrieve) information about the message (such as the message subject, the recipients, etc.).

Methods for the Mail type

Mail

addAddress

addAttachment

addressBook

addressBookTo

empty

emptyAddresses

emptyAttachments

getAddress

getAddressCount

getAttachment

getAttachmentCount

getMessage

getMessageType

getSubject

logoff

logoffDlg

logon

logonDlg

send

sendDlg

setMessage

setMessageType

setSubject

Changes to Mail type methods

The entire Mail type is new for version 7. All of the methods and procedures are new.

▪

addAddress method

See also Example Mail Type

Adds an addressee to a message.

Syntax

```
1. addAddress ( const address String )
2. addAddress ( const address String, const addressType SmallInt )
```

Description

addAddress adds an addressee to the message. Syntax 1 defaults to a "To" type addressee, syntax 2 allows you to specify one of the following types of addressees: MailAddrTo, MailAddrCC, MailAddrBC. Addressees are not checked for validity until the message is sent.

MailAddressTypes Constants

Constant	DataType	Description
MailAddrTo	SmallInt	Specifies this address goes on the “to” line.
MailAddrCC	SmallInt	Specifies this address goes on the “cc” line.
MailAddrBC	SmallInt	Specifies this person gets a copy of the message, without letting anyone else see it. May not be supported by all mail systems.

addAddress example

The following example sends a message (about sales results) to John Doe and copies Susan Smith.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.send()  ; Send the message
endMethod
```

▪

addAttachment method

See also Example Mail Type

Adds an attachment to the message.

Syntax

1. addAttachment (const ***fileName*** String **)**
2. addAttachment (const ***fileName*** String, const ***moniker*** String **)**
3. addAttachment (const ***fileName*** String, const ***moniker*** String, const ***displayPos*** LongInt **)**

Description

addAttachment adds an attachment to the message. Syntax 1 sends the specified *fileName*. Syntax 2 sends the specified *fileName*, but displays the name specified in *moniker*. Some mail systems (for example, Microsoft Mail) allow the attachment icon to be displayed in the message text, in this case, you can use syntax 3 to specify the position in the text that the file should appear (with Microsoft mail, specifying "1" will cause the first character of the message to be displaced by the icon for the specified attachment).

Some mail systems place limits on the number, size, and/or type of attachments you can use (a few mail systems still don't support binary attachments). No attempt is made to verify the existence of the files until the message is sent. Aliases can be used to specify attachment names.

addAttachment example

The following example sends a message (about sales results) to John Doe and copies Susan Smith.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.send()  ; Send the message
endMethod
```

▪

addressBook method

See also		Example		Mail Type

Displays the address book.

Syntax

```
1. addressBook ( )
2. addressBook ( const numberOfLists SmallInt )
```

Description

addressBook displays the address book, and allows the user to modify the list of addressees. Syntax 1 allows all types of addressees (To, CC, BC) to be updated. Syntax 2 allows you to limit the number of address lists to be updated: *numberOfLists* = 1 shows only the "To" addressees, *numberOfLists* = 2 shows the "To" and "CC" addressees, *numberOfLists* = 3 shows the "To," "CC," and "BC" addressees.

If an existing mail session is not active, the user may be prompted with a logon dialog. Use the **logon** method to create a mail session.

-

addressBook example

The following example allows the user to update a distribution list kept in a table

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var m MAIL tc TCURSOR index LONGINT address STRING addrtype SMALLINT
endvar
   tc.open("distribution list.db")
   scan tc:  ; read the address list
      m.addAddress( tc."Addressee" )
   endscan
   m.addressBook( 1 ) ; Display the list for editing
   tc.edit( )
   tc.empty( ) ; clear the old list
   for index from 1 to m.getAddressCount( ) ; write out the new list
       tc.insertRecord( )
       m.getAddress( index, address, addrtype )
       tc."Addressee" = address
       tc.unlockRecord( )
   endfor
   tc.close( )
endMethod
```

▪

addressBookTo method

See also Example Mail Type

Displays the "To" list from the address book.

Syntax

`addressBookTo ( const prompt String )`

Description

addressBookTo displays the "To" list from the address book, and allows the user to modify the list of addressees. **addressBookTo** displays only the "To" list, but allows you to override what the list is called (for example, "Routing").

If an existing mail session is not active, the user may be prompted with a logon dialog. Use the **logon** method to create a mail session.

▪

addressBookTo example

The following example allows the user to update a distribution list kept in a table

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var m MAIL tc TCURSOR index LONGINT address STRING addrtype SMALLINT
endvar
   tc.open("distribution list.db")
   scan tc:  ; read the address list
      m.addAddress( tc."Addressee" )
   endscan
   m.addressBookTo( "Fundraiser Mail List" )
   tc.edit( )
   tc.empty( ) ; clear the old list
   for index from 1 to m.getAddressCount( ) ; write out the new list
       tc.insertRecord( )
       m.getAddress( index, address, addrtype )
       tc."Addressee" = address
       tc.unlockRecord( )
   endfor
   tc.close( )
endMethod
```

■

empty method

[See also](#)   [Example](#)   [Mail Type](#)

Empties the contents of the mail variable

Syntax

`empty ( )`

Description

empty empties the contents of the mail variable (clears the message). The session (which is set by the **logon** method), if any, is unaffected.

empty example

The following example sends a message (about sales results) to John Doe and copies Susan Smith, then sends a different message to Bill Brown.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.send()  ; Send the message

   m.empty()  ; Clear out the old message

   m.addAddress("BBROWN")
   m.setSubject("Final sales numbers sent")
   m.setMessage("Bill, John and Susan have the final sales now")
   m.send()  ; Send the message
endMethod
```

▪

emptyAddresses method

See also Example Mail Type

Deletes all the addresses attached to a message.

Syntax

```
emptyAddresses ( )
```

Description

emptyAddresses sets the number of addresses attached to the message to zero.

emptyAddresses example

The following example sends a message (about sales results) to John Doe and copies Susan Smith, then sends a different message to Bill Brown.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.send()  ; Send the message

   m.emptyAddresses()  ; Clear out the old Addresses

   m.addAddress("BBROWN")
   m.setMessage("Bill, John and Susan have the final sales now")
   m.send()  ; Send with subject & attachment specified earlier
endMethod
```

▪

emptyAttachments method

See also Example Mail Type

Deletes all the attachments to a message.

Syntax

`emptyAttachments ( )`

Description

emptyAttachments sets the number of attachments to the message to zero.

▪

emptyAttachments example

The following example sends a message (about sales results) to John Doe and copies Susan Smith, then sends a different message to Bill Brown.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.send()  ; Send the message

   m.emptyAddresses()  ; Clear out the old Addressee's
   m.emptyAttachment()  ; Clear out the old Attachment

   m.addAddress("BBROWN")
   m.setMessage("Bill, John and Susan have the final sales now")
   m.send()  ; Send with subject specified earlier
endMethod
```

getAddress method

See also Example Mail Type

Retrieves the specified addressee information.

Syntax

getAddress (const ***index*** LongInt, var ***address*** String, var ***addressType*** SmallInt **)**

Description

getAddress retrieves the specified addressee information, where *index* is between 1 and **getAddressCount**, inclusive.

▪

getAddress example

The following example allows the user to update a distribution list kept in a table

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var m MAIL tc TCURSOR index LONGINT address STRING addrtype SMALLINT
endvar
   tc.open("distribution list.db")
   scan tc:  ; read the address list
      m.addAddress( tc."Addressee" )
   endscan
   m.addressBookTo( "Fundraiser Mail List" )
   tc.edit( )
   tc.empty( ) ; clear the old list
   for index from 1 to m.getAddressCount( ) ; write out the new list
       tc.insertRecord( )
       m.getAddress( index, address, addrtype )
       tc."Addressee" = address
       tc.unlockRecord( )
   endfor
   tc.close( )
endMethod
```

getAddressCount method

[See also](#) [Example](#) [Mail Type](#)

Returns the number of addressees attached to the current message.

Syntax

`getAddressCount ( )` `LongInt`

Description

getAddressCount returns the number of addressees attached to the current message.

getAddressCount example

The following example allows the user to update a distribution list kept in a table

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var m MAIL tc TCURSOR index LONGINT address STRING addrtype SMALLINT
endvar
   tc.open("distribution list.db")
   scan tc:  ; read the address list
      m.addAddress( tc."Addressee" )
   endscan
   m.addressBookTo( "Fundraiser Mail List" )
   tc.edit( )
   tc.empty( ) ; clear the old list
   for index from 1 to m.getAddressCount( ) ; write out the new list
       tc.insertRecord( )
       m.getAddress( index, address, addrtype )
       tc."Addressee" = address
       tc.unlockRecord( )
   endfor
   tc.close( )
endMethod
```

■

getAttachment method

See also Example Mail Type

Retrieves specific attachment information.

Syntax

`getAttachment ( const index LongInt, var fileName String, var moniker String, var displayPos LongInt )`

Description

getAttachment retrieves the specified attachment information, where *index* is between 1 and **getAttachmentCount**, inclusive.

getAttachment example

The following example gets the list of attachments from a mail variable (the attachments are presumed to have been added elsewhere):

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var list ARRAY [] STRING indx LONGINT
       filename STRING moniker STRING pos LONGINT
   endvar
   for indx from 1 to getAttachmentCount()
       getAttachment(indx, filename, moniker, pos)
list[indx]=filename
   endfor
endMethod
```

getAttachmentCount method

See also Example Mail Type

Returns the number of attachments to the current message.

Syntax

`getAttachmentCount ( ) LongInt`

Description

getAttachmentCount returns the number of attachments to the current message.

getAttachmentCount example

The following example displays the number of attachments:

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var count longint endvar
   m.addAttachment( "SALES.TXT" )
   count = m.getAttachmentCount( )
   count.view( "Number of attachments" )
endMethod
```

▪

getMessage method

See also		Example		Mail Type

Returns the current text of the message.

Syntax

`getMessage ( )` `String`

Description

getMessage returns the current text of the message.

getMessage example

The following example displays the (previously set) message text:

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var msgtext string endvar
   msgtext = m.getMessage( )
   msgtext.view( "Message text" )
endMethod
```

getMessageType method

[See also](#) [Example](#) [Mail Type](#)

Returns the current message type.

Syntax

getMessageType ( ) String

Description

getMessageType returns the current message type.

■

getMessageType example

The following example displays the (previously set) message type:

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var msgtype string endvar
   msgtype = m.getMessageType( )
   msgtype.view( "Message type" )
endMethod
```

▪

getSubject method

See also Example Mail Type

Returns the current subject of the message.

Syntax

getSubject ( ) `String`

Description

getSubject returns the current subject of the message.

getSubject example

The following example displays the (previously set) subject:

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   var subject string endvar
   subject = m.getsubject( )
   subject.view( "Subject" )
endMethod
```

▪

logoff method

See also Example Mail Type

Attempts to logoff the mail system.

Syntax

`logoff ( )`

Description

logoff attempts to logoff the mail system without user intervention, and terminate the mail session created by **logon**. Any errors will trigger an exception.

logoff example

The following example logs on, displays the send dialog, then logs off:

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.logon("mypassword", "special" )
   m.sendDlg( )
   m.logoff( )
endMethod
```

▪

logoffDlg method

See also Example Mail Type

Attempts to logoff the mail system with user interaction.

Syntax

```
logoffDlg ( ) Logical
```

Description

logoffDlg attempts to logoff the mail system, and terminate the mail session created by **logon**. If supported by the mail system, the user is prompted to enter logoff information, otherwise a straight logoff is done.

logoffDlg returns True if the user logs off, and False if they cancel. Any errors will trigger an exception.

▪

logoffDlg example

The following example logs on, displays the send dialog, then logs off, displaying a logoff dialog if appropriate:

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.logon("mypassword", "special" )
   m.sendDlg( )
   m.logoffDlg( )
endMethod
```

▪

logon method

See also Example Mail Type

Attempts to logon to the mail system.

Syntax

logon (const ***password*** String, const ***profileName*** String **)**

Description

logon attempts to logon to the mail system without user intervention. Any errors will trigger an exception.

The *password* argument is an input parameter that specifies a credential string (maximum 256 characters). If the messaging system does not require password credentials, or if it requires that the user actively enter them, *password* should be blank. When the user must enter credentials, use **logonDlg**.

The argument *profileName* is an input parameter specifying a named profile string (maximum of 256 characters). This is the profile to use when logging on. Some mail providers accept a null *profileName* as specifying the default profile.   If you don't know the *profileName*, use **LogonDlg**.

▪

logon example

The following example sends a message (about sales results) to John Doe and copies Susan Smith, then sends a different message to Bill Brown. It uses logon to specify a special mail session, and group everything together.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.logon("mypassword", "special" )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.send()  ; Send the message

   m.empty()  ; Clear out the old message

   m.addAddress("BBROWN")
   m.setSubject("Final sales numbers sent")
   m.setMessage("Bill, John and Susan have the final sales now")
   m.send()  ; Send the message
   m.logoff()
endMethod
```

▪

logonDlg method

See also Example Mail Type

Attempts to logon to the mail system with user interaction.

Syntax

1. logonDlg ( ) Logical
2. logonDlg (const ***password*** String, const ***profile*** String **)** Logical

Description

logonDlg attempts to logon to the mail system with user interaction. If necessary, the user is prompted to enter logon information. If successful, a mail session is created. The session stays active until the **logoff** method is called, or the mail variable goes out of scope.

logonDlg returns True if the user logs on, and False if they cancel. Any errors will trigger an exception.

logonDlg example

The following example sends a message (about sales results) to John Doe and copies Susan Smith, then sends a different message to Bill Brown. It uses logonDlg so that the user will only have to specify their mail password once (if they have one).

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.logonDlg( )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.send()  ; Send the message

   m.empty()  ; Clear out the old message

   m.addAddress("BBROWN")
   m.setSubject("Final sales numbers sent")
   m.setMessage("Bill, John and Susan have the final sales now")
   m.send()  ; Send the message
   m.logoff()
endMethod
```

send method

See also Example Mail Type

Sends a mail message.

Syntax

`send ( )`

Description

send sends a mail message without user interaction. At least one addressee must have been defined. Most mail systems require that some additional information is defined (for example, the subject).

If an existing mail session is not active, the user may be prompted with a logon dialog. Use the **logon** method to create a mail session. Some mail provider systems may require an explicit logon call before a send, others may not.

send example

The following example sends a message (about sales results) to John Doe and copies Susan Smith.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.send()  ; Send the message
endMethod
```

sendDlg method

See also Example Mail Type

Sends a mail message with user interaction.

Syntax

`sendDlg ( ) Logical`

Description

sendDlg sends a mail message with user interaction. The user will be shown the message as it currently exists (using the user's default MAPI mail system provider). They can then modify it before sending it.

If an existing mail session is not active, the user may be prompted with a logon dialog. Use the **logon** method to create a mail session.

sendDlg returns True if the user sends the message, and False if they cancel. Any errors will trigger an exception.

sendDlg returns True if the user cancels the logon dialog.

sendDlg example

The following example sends a message (about sales results) to John Doe and copies Susan Smith.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.sendDlg()  ; Display the message so the user can edit before sending
endMethod
```

The following example simply displays a mail dialog for the user to enter their own message:

```
method pushButton ( var eventInfo Event )
   m.sendDlg()
endMethod
```

■

setMessage method

See also Example Mail Type

Sets the text of the message.

Syntax

```
setMessage ( const message String )
```

Description

setMessage sets the text of the message to *message*. The maximum length of *message* is limited by the shorter of the mail system and ObjectPAL's maximum string length. This is typically at least 32,000 characters.

setMessage example

The following example sends a message (about sales results) to John Doe and copies Susan Smith.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.sendDlg()  ; Display the message so the user can edit before sending
endMethod
```

setMessageType method

See also Example Mail Type

Sets the type of the message.

Syntax

`setMessageType ( ``const `***`messageType`***` String `**`)`**

Description

setMessageType sets the type of the message. Some mail systems support a *messageType*. Typically, an untyped message is assumed to be an Inter-Personal Message, while typed messages can only be read by a program asking for that particular message type.

Using message types typically requires special support from your mail system. Consult your mail vendor for more information.

setMessageType example

The following example sends a message (about sales results) to John Doe and copies Susan Smith, it uses a special message type "IPM.URGENT" that was previously set up on this mail system.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.setMessageType("IPM.URGENT")
   m.sendDlg()  ; Display the message so the user can edit before sending
endMethod
```

setSubject method

See also Example Mail Type

Sets the subject of the message.

Syntax

`setSubject (` `const` ***`subject`*** `String` **`)`**

Description

setSubject sets the subject of the message to *subject*. The maximum length of *subject* is limited by the mail system. This is typically at least 80 characters.

-

setSubject example

The following example sends a message (about sales results) to John Doe and copies Susan Smith.

```
var
   m  MAIL
endVar
method pushButton ( var eventInfo Event )
   m.addAddress("JDOE")
   m.addAddress("SSMITH", MailAddrCC)
   m.setSubject("Final sales numbers")
   m.setMessage("The final sales numbers are attached")
   m.addAttachment("SALES.TXT")
   m.sendDlg()  ; Display the message so the user can edit before sending
endMethod
```