Alphabetic List of ObjectPAL Methods

broadcastAction
cancelEdit
canReadFromClipboard
cAverage
cCount
ceil
char
charAnsiCode
chr
chrOEM
chrToKeyName
close
cMax
cMin
cNpv
commit
commitTransaction
compact
constantNameToValue
constantValueToName
contains
convertPointWithRespectTo
copy
copyFromArray
copyRecord
copyToArray
cos
cosh
count
countOf
create
cSamStd
cSamVar
cStd
cSum
currRecord
currency
currentPage
cVar
data

dataModelAddTable
dataModelHasTable
dataModelRemoveTable
dataType
date
DateTime
dateVal
day
daysInMonth
debug
delayScreenUpdates
delete
deleteDir
deleteRecord
design
didFlyAway
disableBreakMessage
distance
dlgAdd
dlgCopy
dlgCreate
dlgDelete
dlgEmpty
dlgExport
dlgImportASCIIFix
dlgImportASCIIVar
dlgImportSpreadsheet
dlgNetDrivers
dlgNetLocks
dlgNetRefresh
dlgNetRetry
dlgNetSetLocks
dlgNetSystem
dlgNetUserName
dlgNetWho
dlgRename
dlgRestructure
dlgSort
dlgSubtract
dlgTableInfo

dmAddTable
dmGet
dmHasTable
dmLinkToFields
dmLinkToIndex
dmPut
dmRemoveTable
dmUnlink
dow
dowOrd
doy
drives
dropIndex
edit
empty
end
endEdit
enumAliasLoginInfo
enumAliasNames
enumDataBaseTables
enumDesktopWindowNames
enumDriverCapabilities
enumDriverInfo
enumDriverNames
enumDriverTopics
enumEngineInfo
enumFieldNames
enumFieldNamesInIndex
enumFieldStruct
enumFileList
enumFolder
enumFonts
enumFormNames
enumIndexStruct
enumLocks
enumObjectNames
enumOpenDatabases
enumRefIntStruct
enumReportNames
enumRTLClassNames

enumRTLConstants
enumRTLMethods
enumSecStruct
enumSource
enumSourceToFile
enumTableLinks
enumTableProperties
enumUIClasses
enumUIObjectNames
enumUIObjectProperties
enumUsers
enumVerbs
enumWindowNames
eof
eot
errorClear
errorCode
errorLog
errorMessage
errorPop
errorShow
errorTrapOnWarnings
exchange
execMethod
execute
executeQBE
executeQBEFile
executeQBEString
executeSQL
executeSQLFile
executeSQLString
existDrive
exit
exp
exportASCIIFix
exportASCIIVar
exportSpreadsheet
fail
familyRights
fieldName

fieldNo
fieldRights
fieldSize
fieldType
fieldUnits2
fieldValue
fileBrowser
fill
findFirst
findNext
floor
forceRefresh
formReturn
format
formatAdd
formatDelete
formatExist
formatSetCurrencyDefault
formatSetDateDefault
formatSetDateTimeDefault
formatSetLogicalDefault
formatSetLongIntDefault
formatSetNumberDefault
formatSetSmallIntDefault
formatSetTimeDefault
fraction
freeDiskSpace
fullName
fv
getAliasPath
getAliasProperty
getBoundingBox
getDestination
getDir
getDrive
getFileAccessRights
getKeys
getLanguageDriver
getLanguageDriverDesc
getMenuChoiceAttribute

getMenuChoiceAttributeById
getMousePosition
getMouseScreenPosition
getNetUserName
getObjectHit
getPosition
getProperty
getPropertyAsString
getRGB
getServerName
getTarget
getTitle
getValidFileExtensions
grow
hasMenuChoiceAttribute
hasMouse
helpOnHelp
helpQuit
helpSetIndex
helpShowContext
helpShowIndex
helpShowTopic
helpShowTopicInKeywordTable
hide
hideSpeedBar
home
hour
id
ignoreCaseInLocate
ignoreCaseInStringCompares
importASCIIFix
importASCIIVar
importSpreadsheet
indexOf
initRecord
insert
insertAfter
insertAfterRecord
insertBefore
insertBeforeRecord

insertFirst
insertRecord
int
isAbove
isAdvancedWildcardsInLocate
isAltKeyDown
isAssigned
isBelow
isBlank
isBlankZero
isContainerValid
isControlKeyDown
isDir
isEdit
isEmpty
isEncrypted
isExecuteQBEFileLocal
isExecuteQBELocal
isExecuteQBEStringLocal
isFile
isFirstTime
isFromUI
isFixed
isFixedType
isIgnoreCaseInLocate
isIgnoreCaseInStringCompares
isInside
isLastMouseClickedValid
isLastMouseRightClickedValid
isLeapYear
isLeft
isLeftDown
isMaximized
isMiddleDown
isMinimized
isOnSQLServer
isOpenOnUniqueIndex
isPreFilter
isRecordDeleted
isRemote

isRemovable
isResizeable
isRight
isRightDown
isShared
isShiftKeyDown
isShowDeletedOn
isSpace
isSpeedBarShowing
isTable
isTargetSelf
isValid
isVisible
keyChar
keyNameToChar
keyNameToVKCode
keyPhysical
killTimer
ln
load
locate
locateNext
locateNextPattern
locatePattern
locatePrior
locatePriorPattern
lock
lockRecord
lockStatus
log
logical
longInt
lower
lTrim
makeDir
match
max
maximize
memo
menuAction

menuChoice
message
methodDelete
methodGet
methodSet
milliSec
min
minimize
minute
mod
month
mouseClick
mouseDouble
mouseDown
mouseEnter
mouseExit
mouseMove
mouseRightDouble
mouseRightDown
mouseRightUp
mouseUp
moveTo
moveToPage
movetoRecNo
moveToRecord
moy
msgAbortRetryIgnore
msgInfo
msgQuestion
msgRetryCancel
msgStop
msgYesNoCancel
nFields
nKeyFields
nRecords
name
nextRecord
numVal
number
oemCode

open
openAsDialog
pixelsToTwips
play
pmt
point
position
postAction
postRecord
pow
pow10
pushButton
priorRecord
privDir
protect
pv
qLocate
rTrim
rand
reIndex
reIndexAll
readChars
readEnvironmentString
readFromClipboard
readFromFile
readLine
readProfileString
reason
recNo
recordStatus
remove
removeAlias
removeAllItems
removeAllPasswords
removeItem
removeMenu
removePassword
rename
replaceItem
resync

retryPeriod
rgb
rollBackTransaction
run
save
saveCFG
search
second
seqNo
setAliasPassword
setAliasPath
setAliasProperty
setAltKeyDown
setChar
setControlKeyDown
setData
setDir
setDrive
setErrorCode
setFieldValue
setFilter
setFlyAwayControl
setId
setIndex
setInside
setItem
setLeftDown
setMenuChoiceAttribute
setMenuChoiceAttributeById
setMiddleDown
setMousePosition
setMouseScreenPosition
setMouseShape
setNewValue
setPosition
setProperty
setReason
setRetryPeriod
setRightDown
setShiftKeyDown

setSize
setStatusValue
setTimer
setTitle
setVChar
setVCharCode
setX
setXY
setY
show
showDeleted
showSpeedBar
sin
sinh
size
skip
sleep
smallInt
sort
sortTo
sound
space
splitFullFileName
sqrt
startUpDir
statusValue
strVal
string
substr
subtract
switchIndex
sysInfo
tableName
tableRights
tan
tanh
time
toANSI
toOEM
today

totalDiskSpace
tracerClear
tracerHide
tracerOff
tracerOn
tracerSave
tracerShow
tracerToTop
tracerWrite
transactionActive
twipsToPixels
type
unAssign
unAttach
unDeleteRecord
unLock
unLockRecord
unprotect
updateRecord
upper
usesIndexes
vChar
vCharCode
version
view
vkCodeToKeyName
wait
wasLastClicked
wasLastRightClicked
winGetMessageID
winPostMessage
winSendMessage
windowClientHandle
windowHandle
windowsDir
windowsSystemDir
workingDir
writeEnvironmentString
writeLine
writeProfileString

writeQBE
writeSQL
writeString
writeToClipboard
writeToFile
x
y
year

bitAND

Select the TYPE

- LongInt
- SmallInt

bitOR

Select the TYPE

- LongInt
- SmallInt

bitXOR

Select the TYPE

- LongInt
- SmallInt

pixelsToTwips

Select the TYPE

- System
- UIObject

twipsToPixels

Select the TYPE

System

UIObject

action

Select the TYPE

- Form
- UIObject
- TableView

add

Select the TYPE

- TCursor
- Table

addArray

Select the TYPE

- Menu
- PopupMenu

addBreak

Select the TYPE

- Menu
- PopupMenu

addPopup

Select the TYPE

- Menu
- PopupMenu

addStaticText

Select the TYPE

- Menu
- PopupMenu

addText

Select the TYPE

- Menu
- PopupMenu

advMatch

Select the TYPE

- String
- TextStream

atFirst

Select the TYPE

- TCursor
- UIObject

atLast

Select the TYPE

- TCursor
- UIObject

attach

Select the TYPE

- Form
- TCursor
- Table
- UIObject

bitIsSet

Select the TYPE

- LongInt
- SmallInt

cAverage

Select the TYPE

- Table
- TCursor

cCount

Select the TYPE

- Table
- TCursor

cMax

Select the TYPE

- Table
- TCursor

cMin

Select the TYPE

- Table
- TCursor

cNpv

Select the TYPE

- Table
- TCursor

cSamStd

Select the TYPE

- Table
- TCursor

cSamVar

Select the TYPE

- Table
- TCursor

cStd

Select the TYPE

- Table
- TCursor

cSum

Select the TYPE

- Table
- TCursor

cVar

Select the TYPE

- Table
- TCursor

cancelEdit

Select the TYPE

- UIObject
- TCursor

close

Select the TYPE

- DDE
- DataBase
- Form
- Report
- Session
- TCursor
- TableView
- TextStream
- Library
- System

compact

Select the TYPE

- TCursor
- Table

contains

Select the TYPE

- Array
- DynArray
- Menu

copy

Select the TYPE

- FileSystem
- TCursor
- Table

copyFromArray

Select the TYPE

- TCursor
- UIObject

copyToArray

Select the TYPE

- TCursor
- UIObject

create

Select the TYPE

- Form
- TextStream
- UIObject

currRecord

Select the TYPE

- TCursor
- UIObject

delete

Select the TYPE

- DataBase
- FileSystem
- UIObject

deleteRecord

Select the TYPE

- TCursor
- UIObject

design

Select the TYPE

- Form
- Report

dropIndex

Select the TYPE

- TCursor
- Table

edit

Select the TYPE

- TCursor
- UIObject
- OLE

empty

Select the TYPE

- Array
- DynArray
- Menu
- TCursor
- Table
- UIObject

end

Select the TYPE

- TCursor
- UIObject
- TextStream

endEdit

Select the TYPE

TCursor

UIObject

enumFieldNames

Select the TYPE

- Table
- TCursor
- UIObject

enumFieldNamesInIndex

Select the TYPE

- Table
- TCursor

enumIndexStruct

select the TYPE

- Table
- TCursor

enumLocks

Select the TYPE

- TCursor
- UIObject

enumRefIntStruct

Select the TYPE

- Table
- TCursor

enumSecStruct

Select the TYPE

- Table
- TCursor

enumSource

Select the TYPE

- Library
- UIObject
- Form

enumSourceToFile

Select the TYPE

- Library
- UIObject
- Form

enumUIObjectNames

Select the TYPE

- Form
- Report
- UIObject

enumUIObjectProperties

Select the TYPE

- Form
- Report
- UIObject

errorCode

Select the TYPE

- System
- Event

execMethod

Select the TYPE

- UIObject
- Library

execute

Select the TYPE

DDE

System

executeQBE

Select the TYPE

DataBase

Query

familyRights

Select the TYPE

- Table
- TCursor

fieldNo

Select the TYPE

- Table
- TCursor

fieldType

Select the TYPE

- Table
- TCursor

fill

Select the TYPE

- Array
- String

forceRefresh

Select the TYPE

- UIObject
- TCursor

getPosition

Select the TYPE

- UIObject
- Form

home

Select the TYPE

- TCursor
- UIObject
- TextStream

id

Select the TYPE

- ActionEvent
- MenuEvent

insertAfterRecord

Select the TYPE

- TCursor
- UIObject

insertBeforeRecord

Select the TYPE

- TCursor
- UIObject

insertRecord

Select the TYPE

- TCursor
- UIObject

isAssigned

Select the TYPE

- AnyType
- Session
- DataBase
- Table

isControlKeyDown

Select the TYPE

- KeyEvent
- MouseEvent

isEdit

Select the TYPE

- UIObject
- TCursor

isEmpty

Select the TYPE

- Table
- TCursor
- UIObject

isEncrypted

Select the TYPE

- Table
- TCursor

isFromUI

Select the TYPE

- KeyEvent
- MenuEvent
- MouseEvent

isRecordDeleted

Select the TYPE

- TCursor
- UIObject

isShared

Select the TYPE

- Table
- TCursor

isShiftKeyDown

Select the TYPE

KeyEvent

MouseEvent

isTable

Select the TYPE

- DataBase
- Table

keyChar

Select the TYPE

- Form
- UIObject

keyPhysical

Select the TYPE

- Form
- UIObject

load

Select the TYPE

Form

Report

locate

Select the TYPE

- TCursor
- UIObject

locateNext

Select the TYPE

- TCursor
- UIObject

locateNextPattern

Select the TYPE

- TCursor
- UIObject

locatePattern

Select the TYPE

- TCursor
- UIObject

locatePrior

Select the TYPE

- TCursor
- UIObject

locatePriorPattern

Select the TYPE

- TCursor
- UIObject

lock

Select the TYPE

- Session
- Table
- TCursor

lockRecord

Select the TYPE

- TCursor
- UIObject

lockStatus

Select the TYPE

- TCursor
- UIObject

menuAction

Select the TYPE

- Form
- UIObject

methodDelete

Select the TYPE

- Form
- UIObject

methodGet

Select the TYPE

- Form
- UIObject

methodSet

Select the TYPE

- UIObject
- Form

mouseDouble

Select the TYPE

- Form
- UIObject

mouseDown

Select the TYPE

- Form
- UIObject

mouseEnter

Select the TYPE

- Form
- UIObject

mouseExit

Select the TYPE

- Form
- UIObject

mouseMove

Select the TYPE

- Form
- UIObject

mouseRightDouble

Select the TYPE

- Form
- UIObject

mouseRightDown

Select the TYPE

- Form
- UIObject

mouseRightUp

Select the TYPE

- Form
- UIObject

mouseUp

Select the TYPE

- Form
- UIObject

moveTo

Select the TYPE

- Form
- UIObject

moveToPage

Select the TYPE

- Form
- Report

moveToRecNo

Select the TYPE

- UIObject
- TCursor

moveToRecord

Select the TYPE

- TCursor
- UIObject
- TableView

nFields

Select the TYPE

- Table
- TCursor
- UIObject

nKeyFields

Select the TYPE

- Table
- TCursor
- UIObject

nRecords

Select the TYPE

- Table
- TCursor
- UIObject

nextRecord

Select the TYPE

- TCursor
- UIObject

open

Select the TYPE

- DataBase
- DDE
- Form
- Report
- Session
- TableView
- TextStream
- TCursor
- Library

postAction

Select the TYPE

- Form
- UIObject

postRecord

Select the TYPE

- TCursor
- UIObject

priorRecord

Select the TYPE

- TCursor
- UIObject

reIndex

Select the TYPE

- Table
- TCursor

reIndexAll

Select the TYPE

- Table
- TCursor

readFromClipboard

Select the TYPE

- Graphic
- OLE

readFromFile

Select the TYPE

- Graphic
- Memo
- Binary

reason

Select the TYPE

- Event
- MenuEvent
- StatusEvent
- ErrorEvent
- MoveEvent

recordStatus

Select the TYPE

- TCursor
- UIObject

remove

Select the TYPE

- Array
- Menu

removeItem

Select the TYPE

- Array
- DynArray

rename

Select the TYPE

- FileSystem
- Table

run

Select the TYPE

Form

Report

setControlKeyDown

Select the TYPE

MouseEvent

KeyEvent

setFilter

Select the TYPE

- Table
- TCursor
- UIObject

setId

Select the TYPE

- ActionEvent
- MenuEvent

setPosition

Select the TYPE

- Form
- TextStream
- UIObject

setReason

Select the TYPE

- Event
- MenuEvent
- StatusEvent
- ErrorEvent
- MoveEvent

setX

Select the TYPE

- MouseEvent
- Point

setY

Select the TYPE

- MouseEvent
- Point

show

Select the TYPE

- Form
- Menu
- PopupMenu

showDeleted

Select the TYPE

- Table
- TCursor

size

Select the TYPE

- Array
- DynArray
- FileSystem
- String
- TextStream
- Binary

skip

Select the TYPE

- TCursor
- UIObject

subtract

Select the TYPE

- Table
- TCursor

switchIndex

Select the TYPE

- TCursor
- UIObject

tableRights

Select the TYPE

- Table
- TCursor

time

Select the TYPE

- Time
- FileSystem

type

Select the TYPE

- Table
- TCursor

unDeleteRecord

Select the TYPE

- UIObject
- TCursor

unLock

Select the TYPE

- Session
- Table
- TCursor

unlockRecord

Select the TYPE

- TCursor
- UIObject

view

Select the TYPE

- Array
- UIObject
- Record
- DynArray

wait

Select the TYPE

- Form
- TableView

writeQBE

Select the TYPE

DataBase

Query

writeToClipboard

Select the TYPE

- Graphic
- OLE

writeToFile

Select the TYPE

- Graphic
- Memo
- Binary

x

Select the TYPE

MouseEvent

Point

y

Select the TYPE

MouseEvent

Point

Introduction to ObjectPAL

ObjectPAL is the object-based Paradox application language. It is an event-driven programming language, different from a traditional procedural language in many ways. Using ObjectPAL, you place objects (for example, buttons and fields) in a form or report and attach code modules, called methods, that execute when something happens to the object.

ObjectPAL has two aspects:

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

See Also

ObjectPAL type reference

Programming tasks

Major Language components

Objects

Events

ObjectPAL language components

ObjectPAL language structure

Properties

ObjectPAL IDE

Working with tables

Programming Tasks

Here is a road map if you're looking for a shortcut into the ObjectPAL language. Before you jump to any of these topics, make sure you read about objects 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 Dropdown 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 also includes a built-in file browser dialog.

See Also

Objects

Events

Language Catagories

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
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
Display managers	Let you control how data is presented to the user	Application, Form, Report, TableView
Events	Contain information about actions in Paradox	ActionEvent, ErrorEvent,Event, KeyEvent, MenuEvent, MouseEvent, MoveEvent,StatusEvent, ValueEvent

Using Paradox and ObjectPAL, you can build applications incrementally. That is, you can build some of the tables, create some of the objects, and write some of the methods, test and refine these, then add more tables, objects, and methods, and so on, until the application is complete.

See also
Type Reference

Objects

In Paradox, everything is an object---from the buttons and fields you create using tools in the SpeedBar, to tables and text files store on disk, to menus and pop-up menus create in code. Paradox recognizes two kinds of objects: design objects and data objects. Design objects are objects you place in a form, like a button and data objects are files, data types, and programming structures.

All Paradox objects have

Properties	color, font, line width, and so on
Methods	code that defines how the object responds to an event

You can modify these properties and methods. Any object you can create or modify using Paradox interactively, you can create or modify using ObjectPAL

Objects and methods

Creating Paradox applications is largely a process of placing objects in forms and 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 type reference

ObjectPAL objects are grouped by type. The ObjectPAL Type Reference groups these 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.

Events

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

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. 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
```

ObjectPAL Language Components

There are three components of the ObjectPAL language:

Methods	Code attached to an object which defines that object's behavior
Procedures	Methods bracketed by the commands PROC and ENDPROC
Basic language elements	The fundamental structural elements of ObjectPAL

Procedures

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.

See Also
RTL procedures
Custom procedures

RTL Procedures

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 and msgInfo, 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.

See Also
Custom procedures

Custom Procedures

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 hierarhcy, or to the form itself.

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

Note: ObjectPAL can call a custom procedure faster than it can a call custom method.

The structure is

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

The PROC...ENDPROC block is a basic language element and is discussed in more detail in the *ObjectPAL Reference.*

You can declare procedures in two places:

- within a method
- in an object's Proc window

See Also

RTL Procedures

Basic language elements

Controlling the scope of a library

Containership

Libraries

Procedures Declared in Methods

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 <M>=<D> ", x)
endProc

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

See Also

Procedures declared in an object's Proc window

Custom procedures

Scope

Procedures Declared in an Object's Proc Window

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.

For more detailed information, read the *ObjectPAL Reference.*

See Also

Procedures declared in an object's Proc window

Custom procedures

Language | Object Tree

Attaching methods to a form

Containership

Methods

A method is code that defines the behavior of an object. Methods define how an object responds to events. ObjectPAL methods fall into one of three categories:

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

See Also

Alphabetical list of methods

Editing a method

The Methods dialog box

Built-in methods

Methods in the run-time library

Custom methods

Methods in other objects

Scope

Editing a Method

To edit a method for an object, right-click the object and select Methods from the Properties menu. Select a method from the Method dialog box by double clicking it. The code for the method appears in the Editor window.

To edit the method for a form, select the form then choose Properties | Form | Methods. The Methods dialog box opens.

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.

See Also
The Methods dialog box

The Methods Dialog Box

Use the Methods dialog box to select from the Built-in or Custom methods lists, or define a new custom method.

Built-in Methods	Select the built-in method you want to edit., then click OK. An Editor window opens.
Custom Methods	Choose the custom method you want to edit.
New Custom Method	Create a new custom method by entering its name in the text box and clicking OK. An Editor window opens where you enter your new method.

The Methods dialog box also includes these items:

Use	To declare
Var	variables
Const	constants
Type	types
Procs	procedures
Uses	procedures used by the object's methods

Each of these has its own Editor window, which opens when you choose the item. When you declare a variable, constant, or procedure in one of these windows, it is visible to all methods attached to that object.

For Help on a specific method or procedure, click the Search button and enter the name of the method or procedure in the Search dialog box. You can also use the the Alphabetical list of methods.

To open the Methods dialog box, choose Language | Methods.

See Also

The ObjectPAL Editor

The ObjectPAL Debugger

Alphabetical list of methods

Introduction to ObjectPAL

Scripts

Libraries

Built-In Methods

Every Paradox object comes with built-in methods (for example, **open**, **close**, and **mouseUp**) for each event it can respond to. These built-in methods specify an object's default behavior in response to a given event. You can add your own code to built-in methods using the ObjectPAL Editor. To edit a built-in method for an object, inspect the object and select Methods from the Properties menu. Select one or more methods from the Methods dialog box, then choose OK to open one or more ObjectPAL Editor windows. 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.

Built-in methods are described in the *ObjectPAL Reference*.

See Also
The Methods dialog box

Methods in the Run-Time Library

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. These methods, arranged alphabetically by type, are presented in the *ObjectPAL Reference* as well as online in the ObjectPAL Type Reference section of Help.

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.

See Also

ObjectPAL type reference

The Method dialog box

Containership

Custom Methods

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, inspect an object, choose Methods, then choose the New Custom Method text box. In the edit 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 methods.

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

See Also

The Method dialog box

Containership

Methods in Other Objects

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.

See Also
Scope

Basic Language Elements

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.

The basic language elements are:

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

Method Language Structure and Syntax

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, 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.

See also

Alphabetical list of methods

Containership

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.

Variables

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 (=).

See Also
Scope

The Scope of a Variable

The term "scope" means "accessiblity." The scope of a variable, that is, the range of obects 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 mehtod executes. They are initialized (reset) each time the method executes.

Outside a method

Variables declared in a method window before the word **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 ot the declared data type.

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.

The ObjectPAL language includes many predefined constants.

To see a list of predefined constants, see the Types of constants list.

ObjectPAL IDE

When you work in the ObjectPAL Integrated Development Environment (IDE), you are in either the Editor or the Debugger.

With 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.

See Also

Scripts

Libraries

The ObjectPAL Editor

The ObjectPAL Editor provides the functions of a Windows text editor, along with special functions for editing ObjectPAL methods. Also, the Editor works with the Debugger to provide an integrated environment for creating, testing, and modifying methods.

The ObjectPAL Editor works the same whether you are working with an object, a form, a library, or a script.

See Also

Starting the Editor

Working in the ObjectPAL Editor

ObjectPAL Editor SpeedBar

Leaving the Editor

ObjectPAL Editor menu

The ObjectPAL Debugger

Starting the Editor

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

1. Right-click an object to see its menu.
2. Choose Methods from the object's property list. The Methods dialog box lists the methods you can edit. Scroll the list to see all the entries.
3. Choose a method. An Editor window appears with some default text.

To start the Editor from a Library,

1. Open the Library window and right-click it.
2. When the Methods Dialog box 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 cursor waits on the third line, where you can begin typing the method.

The Editor is always in insert mode, and the usual editing methods are available.

You can have several Editor windows open at once.

Note: Variables, constants, and procedures declared in a method window are visible only to that method. To make a variable, constant, or procedure visible to all of an object's methods, check Uses, Type, Const , Var, and Procs (as many of these as you want) in the Methods dialog box. Paradox opens a separate window for each box you check.

See Also

The Methods dialog box

Working in the ObjectPAL Editor

ObjectPAL Editor menu

Leaving the Editor

The ObjectPAL Debugger

Working in the ObjectPAL Editor

The usual editing methods are available in the Editor window, with two exceptions:

- The Editor is always in insert mode.
- 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.

Using the keyboard:

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 window-full back.
Page down	Moves one window-full 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+Insert	Copies selected text to the clipboard.
Shift+Insert	Pastes text from the clipboard into your method.
Tab	Inserts a Tab character and pushes text to the right.

Selecting text:

To select a word, double-click it.

To select an entire line, click to the left of the line. (The mouse is in position when the I-beam cursor 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.

See Also

ObjectPAL Editor menu

ObjectPAL Editor SpeedBar

Leaving the Editor

ObjectPAL Editor SpeedBar

Click buttons on the SpeedBar to do things quickly in the Editor window:

Cut
Copy
Paste
Run
Print
Check Syntax
Set Breakpoint
Methods Dialog
Save and Exit
Object Tree
Open Folder

For information on each button, click its picture here.

See Also
The ObjectPAL Editor

Cut Button

Clicking the Cut button on the SpeedBar same as choosing Edit | Cut from the menu. The Cut button removes selected text or objects and places them in the Clipboard.

You can then use the Paste button or choose Edit | Paste to paste the contents of the Clipboard into another file, or somewhere else in the same file.

The contents of the Clipboard are not deleted when you paste, so you can paste as many times as you want.

To delete a selection without affecting the Clipboard contents, press Del or choose Edit | Delete.

Shortcut key Shift+Del

Copy Button

Clicking the Copy button on the SpeedBar is the same as choosing Edit | Copy from the menu.

Choose the Copy button to copy the selected text or objects into the Clipboard but not delete anything from your document or query.

To paste the contents of the Clipboard into your document, use either

- Edit | Paste
- Shift+Ins
- Paste button

The contents of the Clipboard are not deleted when you paste, so you can paste as many times as you want.

Shortcut key Ctrl+Ins

Paste Button

Clicking the Paste button on the SpeedBar is the same as choosing Edit | Paste from the menu.

The effects of the Paste button depend on which window is active and whether you are designing or viewing data.

The contents of the Clipboard are not deleted when you paste, so you can paste as many times as you want.

Shortcut key Shift+Ins

Run Button

Click the Run button to run the form or script you are editing. Paradox saves all attached methods, compiles the code, and leaves you in a View window.

Note: If you have set breakpoints, Paradox halts execution at the first breakpoint encountered and a Debugger window opens.

Note: The Run button is not available in a Library window. Methods and procedures in a library are called through a script or a form.

Print Button

Click the print button to print the code you are editing.

If you are	Paradox prints
in a script	all of the code in that script
in a library with no method Editor window open	all of the code in that library
in a library with a method Editor window open	the code for the open method.
in a form with a method Editor window open	the code for the open method

Check Syntax Button

Click the Check Syntax button to compile all methods in the form, script, or library. (not just the current Editor window). If a syntax error is found, a window opens for the corresponding method with the cursor positioned near the error, and an error message appears in the Status line.

Note: If you change the code in more than one Editor window, save the changes before you check the syntax. Otherwise, the syntax checker may report unexpected errors because it's not checking the latest code.

Set Breakpoint Button

Execution will halt at the first breakpoint encountered. You can set as many breakpoints as your system memory allows.

When you click the Set Breakpoint button, the Set Breakpoint dialog box opens.

This is the same as choosing Debug | Set Breakpoint.

Note: You can set breakpoints in a library, but you cannot run the library independently. You must call the method from a form or a script.

Methods Dialog Button

Click the Methods Dialog button to open the Methods dialog box, where you can select from the Built-in or Custom methods lists. You can also define a new custom method or open an Editor window to declare uses, types, constants, variables, or procedures.

This is the same as choosing Language | Methods.

Save and Exit Button

Clicking the Save and Exit button will save the code to memory and close the Editor window. You will still be in the design window.

ObjectPAL Object Tree Button

Click the Object Tree button on the SpeedBar to see the Object Tree. Paradox displays this diagram in a separate window.

A Paradox Object Tree shows you a schematic diagram of the objects in your form and their relations to one another. The diagram shows the object hierarchy, with the currently selected object at the far left, and the tree showing the container hierarchy extending to the right.

Note: The ObjectTree button is not available if you are in a Library window.

Note: A script cannot contain other objects, so the Object Tree for a script shows only that script.

Open Folder Button

Click the Open Folder button to open a folder for the working directory. This is the same as choosing File | Open | Folder.

The folder is a window that shows selected objects in the working directory. Icons represent the objects in the directory.

From the Folder window, you can right-click icons to inspect objects, or double-click to perform the default action (the first item on the menu).

Leaving the ObjectPAL Editor

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

To continue designing your form

- Choose Close from the Editor window's Control menu.
- Double-click the control-menu box.

To view your form and see it in action, click the View Data SpeedBar button.

The ObjectPAL Debugger

The ObjectPAL Debugger lets you interactively test and trace execution of commands in your methods.

Using the Debugger, you can

- Set breakpoints so you can execute instructions up to a certain point, then stop and see what has happened.
- Inspect variables to make sure values are being manipulated as you intended.
- Execute a method one line at a time (called single-stepping).
- Step over procedures and functions that you know are bug-free.

To start the Debugger

1. Display a method in an Editor window.
2. Choose Debug to display a drop-down menu of Debugger functions.

To leave the Debugger

You can leave the ObjectPAL Debugger in three ways:

- Choose Debug | Run. If there are no more breakpoints set, this will close the debug window and let you view data in your form.
- Click the underlying form to get back to your form in its design window.
- Choose Close from the Debug window's control menu or double-click the Control-menu box to close the Debugger window.

See Also

The Debug menu

ObjectPAL Debugger SpeedBar

ObjectPAL Debugger SpeedBar

Click buttons on the SpeedBar to execute commands quickly in the Debugger window:

For information on each button, click its picture here.

See Also
The ObjectPAL Debugger

Run Button

Click the Run button to run the ObjectPAL Debugger. Paradox saves all attached methods, compiles the code, and leaves you in a View window. When Paradox encounters a breakpoint, execution halts, and a Debugger window opens.

If you are currently in a Debugger window, execution resumes from the breakpoint.

Note: You have to set breakpoints in the code in order to run the Debugger.

Note: The Run button is not available in a Library window.

Print Button

Click the print button to print the code you are debugging.

If you are	Paradox prints
in a script	all of the code in that script
In a library with no method Editor window open	all of the code in that library
In a library with a method Editor window open	the code for the open method.
In a form with a method Editor window open	the code for the open method

Set Breakpoint Button

Click the Set Breakpoint button to set breakpoints in a method to halt execution at specified lines. You can set as many breakpoints as your system memory allows.

When you click the Set Breakpoint button, the Set Breakpoint dialog box opens.

This is the same as choosing Debug | Set Breakpoint.

Methods Dialog Button

Click the Methods Dialog button to open the Methods dialog box where you can select from the Built-in or Custom methods lists. You can also define a new custom method, or open an Editor window to declare uses, types, constants, variable, or procedures.

This is the same as choosing Language | Methods.

Step Over Button

Click the Step Over button to single-step through a method, treating procedures and custom methods as single steps. This function is available only when execution is suspended at a breakpoint.

This is the same as choosing Debug | Step Over.

Step Into Button

Click the Step Into button to single-step through every line in a method, and every line in the procedures and custom methods the method calls. This function is available only when execution stops at a breakpoint.

This is the same as choosing Debug | Step Into.

Inspect Button

Click the Inspect button to display and change (optional) the value of a variable. This is available only when execution stops at a breakpoint.

This is the same as choosing Debug | Inspect.

ObjectPAL Object Tree Button

Click the Object Tree button on the SpeedBar to see the Object Tree. Paradox displays this diagram in a separate window.

A Paradox Object Tree shows you a schematic diagram of the objects in your form and their relations to one another. The diagram shows the object hierarchy, with the currently selected object at the far left, and the tree showing the container hierarchy extending to the right.

Note: The ObjectTree button is not available if you are in a Library window.

Note: A script cannot contain other objects, so the Object Tree for a script shows only that script.

Open Folder Button

Click the Open Folder button to open a folder for the working directory. This is the same as choosing File | Open | Folder.

The folder is a window that shows selected objects in the working directory. Icons represent the objects in the directory.

From the Folder window, you can right-click icons to inspect objects, or double-click to perform the default action (the first item on the menu).

Delivering Applications

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, don't forget to deliver copies of all tables in its data model, along with any indexes and referential integrity files.

Choose Language | Deliver to deliver a form, script, or library.

Working with Tables

If you come to ObjectPAL looking for ways to manipulate tables, you should know about the Table, TCursor, TableFrame, and TableView types. This table summarizes the differences.

Type	Description
Table	Tabular data stored in a file.
TCursor	A pointer to a table, stored and manipulated in memory.
TableView	A table displayed in its own window.

Libraries

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 inspect the Library window, the Methods dialog box opens.

In many ways, working with a library is like working with a form. For example, a library has built-in methods. You add code to a library just as you do to a form, using the Methods dialog box 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.

See Also

Library Window Tasks

ObjectPAL Editor commands

Controlling the Scope of a library

Library Window Tasks

Create a library

Add code to a library

- Attach code to built-in methods
- Add custom methods
- Add custom procedures
- Declare variables, constants, data types, & external routines

Edit a library

Deliver a library

Call library methods

Library methods

Control the scope of a library

- Declare a library variable
- Open a library

Use library variables as arguments

Creating a Library

To create a new library, choose File | New | Library. Inspect the Library Design Window to open the Methods dialog box. Then add your code as you would to any other object.

When you're finished adding code, you can

- Save both the source code and the executable code by choosing File | Save.
- Save only the executable code by choosing Language | Deliver.

See Also

Adding code to a library

Calling library methods

Delivering a library

Adding Code to a Library

To add code to a library, inspect the Library Design window to open the Methods dialog box. Then attach your code as you would with any other object. You can add code to a new or existing library.

When you're finished adding code, you can

- Save both the source code and the executable code by choosing File | Save.
- Save only the executable code by choosing Language | Deliver.

Using the Methods dialog box and ObjectPAL Editor windows, you can add code to a library in the following ways:

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

See Also

Attaching code to built-in methods

Adding custom methods

Adding custom procedures

Declaring variables, constants, data types, and external routines

Attaching Code to Built-In Methods

Every library has the following built-in methods: **open**, **close**, and **error**. You can attach code to these built-in 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'll 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 methods and programming using ObjectPAL, see your ObjectPAL documentation.

Adding code to a library

Adding custom methods

Adding custom procedures

Declaring variables, constants, data types, and external routines

Adding Custom Methods

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 display the Methods dialog box, right-click the Library Design window, or press Ctrl+Spacebar. 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.

See Also

Adding code to a library

Attaching code to built-in methods

Adding custom procedures

Declaring variables, constants, data types, and external routines

Adding Custom Procedures

From a library's Methods dialog box, you can choose Procs 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 your ObjectPAL documentation.

See Also

Adding code to a library

Attaching code to built-in methods

Adding custom methods

Declaring variables, constants, data types, and external routines

Declaring Variables, Constants, Data Types, and External Routines

From a library's Methods dialog box, 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 your ObjectPAL documentation.

See Also

Attaching code to a library

Attaching code to built-in methods

Adding custom methods

Adding custom procedures

Editing a Library

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. Inspect the Library window to open the Methods dialog box.
3. Select the method you want to edit then click OK.

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

See Also
ObjectPAL Editor

Delivering a Library

Use Language | Deliver to deliver a library you've created. When you deliver a library, Paradox removes all the source code. Your code isn't lost; it's 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.

When you choose Language | 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 your ObjectPAL documentation.

See Also
Methods

Calling Library Methods

To call a method in a library, you must first declare the method 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 method 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.

Note: You can write code that calls a library method, and the code will compile in Design mode, even if the library doesn't exist. However, the library must be present in order to run the form.

See Also
Library methods

Library Methods

You can use the Library methods in your own code---attached to any object, even another library---to manipulate a library. The run-time methods are:

open	Opens a library and loads it into system memory, making it available to one or more forms and desktops.
close	Removes the library from memory. You don't have to explicitly close a library---it is removed automatically when all referencing forms or libraries are closed---but using this method makes it easier to read the code and understand what's supposed to happen.
enumSource	Writes library code to a Paradox table.
enumSourceToFile	Writes library code to a text file.
execMethod	Executes a specified library method.

See Also
Calling library methods

Controlling the Scope of a Library

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.

See Also

Declaring a library variable

Opening a library

Declaring a Library Variable

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 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.

See Also

Controlling the scope of a library

Opening a library

Opening a Library

The Library method **open** takes arguments that specify the scope. A library can be opened

Private to the form	Only the form that opened the library has access to its code.
Global to the desktop	Every form in the Desktop Paradox session 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. **Note:** 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.

See Also

Controlling the scope of a library

Declaring a library variable

Using Library Variables as Arguments

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.

Scripts

A script consists of code in its own file, not attached to a form. It's an object, and displays on the Desktop as an icon. You can

- Attach one or more methods.
- declare variables, constants, data types, and procedures.
- Call custom DLLs.

A script has no type, doesn't display in a window, and doesn't contain any design objects. A script has the built-in methods **run**, **error**, and **status**. (**Status** is available only in Advanced ObjectPAL.) 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. Use a script when you want to execute code without opening and displaying a form window.

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.

To change the level of ObjectPAL that you are working in, choose Properties | Desktop. In the Desktop Properties dialog box, you can select from Beginner or Advanced ObjectPAL.

See Also
Script window tasks

ObjectPAL Editor menu

Script Window Tasks

Create a script

Add code to a script

- Attach code to built-in methods
- Add custom methods
- Add custom procedures
- Declare variables, constants, data types, & external routines

Edit a script

Debug a script

Play a Script

Deliver a script

Creating a Script

Choose File | New | Script to create a script. An ObjectPAL Editor window opens for the Scripts **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.

From a script, as from any other object, you can open and close other forms, create objects, get and set properties and values, display messages, and trigger methods.

You can display the Methods dialog box by choosing Language | Methods. From there, you can declare variables, constants, data types, procedures, custom methods, and DLLs to use. Keep in mind, though, that whatever you declare is visible only to the script's **run** method.

When you're 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 by choosing File | Save or File | Save As, and it can be delivered by choosing Language | Deliver. Saved scripts can be changed; delivered scripts cannot.

The *ObjectPAL Reference* discusses scripts in detail.

See Also

Adding code to a script

Delivering a script

Adding Code to a Script

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're 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 Language | Deliver.

Using the Methods dialog box and ObjectPAL Editor windows, you can add code to a script in the following ways:

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

See Also

Attach code to the built-in methods

Add custom methods

Add custom procedures

Declare variables, constants, data types, and external routines

Delivering a script

Methods dialog box

Attaching Code to Built-In Methods

Every script has the following built-in methods: **run**, **error**, and **status**. (**Status** is only available in Advanced ObjectPAL.) You can attach code to these built-in methods as you would with any other object. You do this in the Editor window.

You can execute any of these using Paradox interactively, or by calling them from an ObjectPAL method or procedure.

See Also

Adding code to a script

Add custom methods

Add custom procedures

Declare variables, constants, data types, and external routines

ObjectPAL Editor

Adding Custom Methods

You can add custom methods to a script using the Methods dialog box.

To display the Methods dialog box, inspect the Script window, 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.

See Also

Adding code to a script

Attach code to the built-in methods

Add custom methods

Add custom procedures

Declare variables, constants, data types, and external routines

Adding Custom Procedures

From a script's Methods dialog box, you can choose Procs 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.

See Also

Adding code to a script

Attach code to the built-in methods

Add custom methods

Declare variables, constants, data types, and external routines

The Methods dialog box

The ObjectPAL Editor

Declaring Variables, Constants, Data Types, & External Routines

From a script's Methods dialog box, 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 cannot be accessed by other forms or objects. However, other forms and objects can call library routines that access these variables.

Adding code to a script

Attach code to the built-in methods

Add custom methods

Add custom procedures

Editing a Script

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 Design, then click OK. ObjectPAL opens your script in an Editor window with the built-in method **run** displayed.
3. Use the ObjectPAL Editor to edit your script as you would a method.

See Also
The ObjectPAL Editor

Debugging a Script

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 Design, then click OK. The script opens in an Editor window.
3. Choose Debug | Set Breakpoints, or click the Set Breakpoint button, to open the Set Breakpoint dialog box.
4. Enter the line number you want to set the breakpoint on, then click OK.
5. Choose Debug | Run to run the script.

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

See Also
The ObjectPAL Debugger

Playing a Script

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 **run** method.

Using Paradox interactively

1. Choose File | Open | Script. A dialog box lists available scripts.
2. Choose one of the scripts, choose Play, and choose OK. The script executes.

From within a method

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

Use Language | Deliver to deliver a script you've created. When you deliver a script, Paradox removes all the source code. Your code isn't lost; it's 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 Language | Deliver, Paradox saves a copy of the script with an .LDL extension. You can still change your ObjectPAL code using the script with its .LSL extension, but if you want others to use it safely, give them the delivered script.

ObjectPAL Integrated Development Environment Menus

The ObjectPAL Integrated Development Environment (IDE) has some special tools for editing ObjectPAL methods. When you open an Editor window, these menus provide the special tools:

Edit

Language

Debug

Properties

The File, Window, and Help menus do not change. Go to Common menu commands for more information on commands common to all Paradox windows.

See Also
Introduction to ObjectPAL

Edit

Undo	undoes your last edit. Undo is grayed if no changes have been made.
Cut	moves selected text to the clipboard and deletes it from the window. Cut is grayed if nothing is selected.
Copy	copies selected text to the clipboard. Copy is grayed if nothing is selected.
Paste	copies text from the clipboard to the current cursor location. If text is selected, it is replaced with the clipboard contents. Paste is grayed if there is no text on the clipboard.
Delete	deletes selected text. The item is grayed if no text is selected.
Select all	selects all of the text in the window.

See Also

Edit | PasteFromFile
Edit | CopyToFile
Edit | Search
Edit | Search Next
Edit | Replace
Edit | Replace Next
Edit | Go To

Edit | Paste From File

Use Edit | Paste From File to copy a text file into the current method at the cursor position.

When you choose Edit | Paste From File, the Copy From File dialog box opens. Enter the name of the text file and choose OK, or choose Browse to search for the text file you want to copy.

Edit | Copy To File

Use Edit | Copy To File to copy selected text to a text file you specify.

When you choose Edit | Copy To File, the Copy To File dialog box opens. Enter the name of the file you want the selected text copied to.

Note: The text copied to the specified file is the selected text---*not* the contents of the Windows Clipboard.

Edit | Search

Use Edit | Search to find strings of text in your code.

When you choose Edit | Search, Paradox opens the Search dialog box. Specify the text to search for, the direction for the search, and case-sensitivity.

See Also

Search dialog box

Edit | Search Next

Edit | Replace

Search Dialog Box

Use the Search dialog box to find strings of text in your code. To define your search, specify

Search for	the text to look for.
Direction	forward (down) or backward (up) in the code.
Up	from the current cursor position to the beginning of the method.
Down	from the current cursor position to the end of the method.
Case Sensitive	check this to make the search case-sensitive.

To open the Search dialog box, choose Edit | Search.

See Also

Edit | Search

Edit | Search Next

Edit | Replace

Edit | Search Next

Use Edit | Search Next to search for the next occurrence of the text you specified using Search. Search Next is grayed if you have not searched for anything in this session.

See Also

Edit | Search

Edit | Replace

Edit | Replace

Use Edit | Replace to search for text and replace it with a value you specify. Specify the text you are searching for and what you want to replace it with in the Search & Replace dialog box.

See Also

Search and Replace dialog box

Edit | Search

Search & Replace Dialog Box

Use the Search & Replace dialog box to change strings of text in your code.

Search For	Enter the text you want Paradox to look for.
Replace With	Enter the replacement text.
Case Sensitive	Check this to search for the text exactly as you typed it including capitalization.
Replace All	Check this to replace all occurrences of the string in your method with the new text.
Direction	Choose Up or Down to search backward or forward, respectively, in the file.

To open the Search & Replace dialog box, choose Edit | Replace.

See Also
Edit | Replace

Edit | Replace Next

Choose Edit | Replace Next to replace the next occurrence of the text specified in the Search & Replace dialog box. It is grayed until text has been replaced.

See Also

Search & Replace dialog box

Edit | Search

Edit | Replace

Edit | Go To

Use Edit | Go To to move quickly to a line in your code.

When you choose Edit | Go To, Paradox opens the Go To Line dialog box. Enter a line number and click OK to move the cursor to that line. If the line number doesn't exist, the cursor won't move.

Language

Use the commands on the Language menu when you write or editiObjectPAL code.

Check Syntax	compiles all the methods in the form (not just the current window). If syntax errors are found, a window opens for the corresponding method with the cursor positioned near the error, and an error message appears in the Status line.
Next Warning	toggles the display of warning messages from the compiler. When this item is checked, messages in the Status line warn you about undeclared variables and other conditions that may cause errors at run-time.
Methods	provides a quick way to display the Methods menu.
Object Tree	displays a tree diagram showing related objects in the current form.
Keywords	is a cascading menu of frequently used keywords. Use this menu to insert a keyword into a method without typing it.
Types	displays a dialog box listing all object types and their methods. You can choose a type name to display the methods and procedures for that type. Then you can insert a prototype of the method or procedure into your own method at the current cursor position.
Properties	displays a dialog box listing objects and their properties. You can choose an object name to display the properties for that object. Then you can insert the property into your method.
Constants	displays lists of constants. ObjectPAL provides many constants so you can specify colors, mouse shape, menu attributes, and window styles. You can choose a constant and choose Insert to insert it into your method.
Browse Sources	creates and displays a report listing the source code in the current form or report.
Deliver	creates a compiled form or report stripped of its ObjectPAL source code. Users cannot edit the design or the source code.

For more information on each command, select the command you want and press F1.

Language | Check Syntax

Use Language | Check Syntax to compile all the methods in the form (not just the current window). If a syntax error is found, a window opens with the cursor positioned near the error, and an error message appears in the Status line.

Note: If you change the code in more than one Editor window, save the changes before you check the syntax. Otherwise, the syntax checker may report unexpected errors because it's not checking the latest code.

Language | Next Warning

Use Language | Next Warning to move to the next warning message from the compiler. These warnings appear only if Properties | Show Compiler Warnings is toggled on; otherwise they are suppressed.

Language | Methods

Choose Language | Methods to open the Methods dialog box. You can select from the Built-in or Custom methods lists, or define a new custom method. You can also open an Editor window to declare external routines, types, constants, variables, or procedures.

See Also
The Methods dialog box

The Methods Dialog Box

Use the Methods dialog box to select from the Built-in or Custom methods lists, or define a new custom method.

Built-in Methods	Select the built-in method you want to edit., then click OK. An Editor window opens.
Custom Methods	Choose the custom method you want to edit.
New Custom Method	Create a new custom method by entering its name in the text box and clicking OK. An Editor window opens where you enter your new method.

The Methods dialog box also includes these items:

Use	To declare
Var	variables
Const	constants
Type	types
Procs	procedures
Uses	procedures used by the object's methods

Each of these has its own Editor window, which opens when you choose the item. When you declare a variable, constant, or procedure in one of these windows, it is visible to all methods attached to that object.

For Help on a specific method or procedure, click the Search button and enter the name of the method or procedure in the Search dialog box. You can also use the the fal list of methods.

To open the Methods dialog box, choose Language | Methods.

See Also

The ObjectPAL Editor

The ObjectPAL Debugger

Alphabetical list of methods

Introduction to ObjectPAL

Scripts

Libraries

Language | Object Tree

Use Language | Object Tree to display a tree diagram showing related objects in the current form. The diagram shows the object hierarchy, with the currently selected object at the far left, and the tree showing the container hierarchy extending 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 haven't named. If you have written methods for an object, its name is underlined and marked with an asterisk. You can inspect an object and choose Methods to display its Methods dialog box.

Using the Object Tree, you can attach and edit methods for a form, just as you can for a button.

See Also

Attaching methods to a form

Methods

Attaching Methods to a Form

1. In a Design window, press Esc to deselect all other objects (including the page). You may have to press Esc more than once.
2. Choose Form | Object Tree to display a diagram of the objects in the form.
3. The object named #form design1 at the far left of the diagram (the "top" of the Object tree) represents the form. Inspect it and choose Methods to display its Methods dialog box.
4. Choose and edit a method or create a custom method as you would for any other object.

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

See Also

Language | Object Tree

Methods

Editing a method

Custom methods

Language | Keywords

Use Language | Keywords to display a cascading menu of frequently used keywords. Choose a keyword from this menu to insert it into a method without having to type it.

The keywords are basic language elements. See Basic language elements for more information on each keyword.

Language | Types

Use Language | Types to display a dialog box listing all object types and their methods. You can choose a type name to display the methods and procedures for that type. For example, to display a list of methods and procedures for the Array type, choose Array.

Methods are listed first, in alphabetical order, and then the procedures are listed also in alphabetical order.

See Also
Types and Methods dialog box

Types and Methods Dialog Box

Use the Types and Methods dialog box to list all object types and their methods and procedures.

Types	lists all of the ObjectPAL types in alphabetical order. You can select a type name to display its methods and procedures.
Methods	lists all of the methods and procedures for the selected type. Methods are listed first, in alphabetical order, and then the procedures are listed also in alphabetical order.
Insert type	inserts the type name into your own method at the current cursor position. After you select the type, click the Insert type button and click OK.
Insert method	inserts the method or procedure name into your own method at the current cursor position. After you select the method or procedure, click the Insert method button and click OK.

To open the Types and Methods dialog box, choose Language | Types.

Language | Properties

Use Language | Properties to display a dialog box listing objects and their properties.

You can choose an object name to display it's properties. Then you can insert the property into your method. For example, to display a list of Button properties, choose Button. The list of available properties appears in the Properties column. Choose the one you want, then click the Insert Property button. That property is now inserted into your code.

See Also
Display Objects and Properties dialog box

Display Objects and Properties Dialog Box

Use the Display Objects and Properties dialog box to display objects and their properties.

Objects	lists all of the display objects. Choose an object name to display it's properties.
Properties	lists all of the properties for the selected object.
Values	lists the possible values for a property. Select the value you want from the list. If there is only one possible value, the Values box is blank.
Insert object	inserts the object name into your method. After you select the object, click the Insert object button and click OK.
Insert property	inserts the property name into your method. After you select the property, click the Insert property button and click OK.

To open the Display Objects and Properties dialog box, choose Language | Properties.

Language | Constants

Use Language | Constants to display lists of constants. ObjectPAL provides many constants so you can specify colors, mouse shape, menu attributes, and window styles. You can choose a constant and click the Insert button to insert it into your method.

See Also
Constants dialog box

Constants Dialog Box

Use the Constants dialog box to display a list of constants for each type of constant.

Types of constants	lists all of the categories of ObjectPAL constants. Choose a type to display all of its constants.
Constants	lists all of the constants for the selected constant type.
Insert constant	inserts the constant name into your code at the current cursor position. After you select the constant, click the Insert constant button and click OK.

To open the Constants dialog box, choose Language | Constants.

Language | Browse Sources

Use Language | Browse Sources to create and display a report listing the source code in the current form or report.

Language | Deliver

Use Language | Deliver to create a compiled form, script, or library, stripped of its ObjectPAL source code. Users cannot edit the design or the source code.

For more information about delivering applications, see the *ObjectPAL Developer's Guide.*

See Also
Delivering applications

Debug

Use the commands on the Debug menu when you debug ObjectPAL code.

Inspect	Displays and changes (optional) the value of a variable. Available only when execution stops at a breakpoint.
Stack Backtrace	Lists the called methods and procedures since the form started running. The most recently called routine is listed first, followed by its caller; all the way back to the first method or procedure. Available only when execution stops at a breakpoint.
Set Breakpoint	Sets breakpoints in a method to halt execution at specified lines. You can set as many breakpoints as your system memory allows. You cannot set breakpoints in procedures you write, but you can step through them by choosing Debug | Step Into.
List Breakpoints	Lists the line numbers of breakpoints and optionally lets you delete breakpoints.
Trace Execution	Opens a window and lists each line of code as it executes. Your setting for this item is saved with the form, so you don't have to check it every time you want to trace execution.
Trace BuiltIns	Displays a dialog box listing the built-in methods for the selected object. Check a built-in method to display information about the method in the Tracer window as it executes. You must have ObjectPAL code attached to at least one object.
Enable DEBUG Statement	Stops execution whenever the DEBUG statement is encountered. Placing a DEBUG statement in a method has the same effect as setting a breakpoint at that line, with the advantage that it is saved with your source code so you don't have to keep resetting it as you would a breakpoint. The setting for this item is saved with the form.
Enable Ctrl+Break to Debugger	Lets you use Ctrl+Break to suspend execution and open a Debugger window containing the active method or procedure just as if a breakpoint had been encountered. When this item is not checked, pressing Ctrl+Break halts execution.
View Source	Displays another method's code in an Editor window.
Origin	Displays the method containing the current breakpoint and places the cursor on the line containing the breakpoint. Useful when you're looking at several different methods in several different windows and want to return quickly to the breakpoint. Available only when execution stops at a breakpoint.
Step Over	Lets you single-step through a method skipping procedure and function calls. Available only when execution stops at a breakpoint.
Step Into	Lets you single-step through a method including procedure and function calls. Available only when execution stops at a breakpoint.
Quit This Method	Halts execution and closes any Debugger windows. Available only when execution is suspended at a breakpoint.
Run	Exits the Debugger, saves changes, and switches from Design to View Data mode.

For more information on each command, select the command you want and press F1.

Debug | Inspect

Debug | Inspect displays and changes (optional) the value of a variable. This is available only when execution stops at a breakpoint.

See Also
Debug | Set Breakpoint

Debug | Stack Backtrace

Debug | Stack Backtrace lists the called methods and procedures since the form started running. The most recently called routine is listed first, followed by its caller, and so on, all the way back to the first method or procedure. This option is available only when execution stops at a breakpoint.

See Also
Debug | Set Breakpoint

Debug | Set Breakpoint

Use Debug | Set Breakpoint to set breakpoints in a method to halt execution at specified lines. You can set as many breakpoints as your system memory allows.

You cannot set breakpoints in procedures you write, but you can step through them by choosing Debug | Step Into.

When you choose Debug | Set Breakpoint, the Set Breakpoint dialog box opens.

See Also
Set Breakpoint dialog box

Debug | Step Into

Set Breakpoint Dialog Box

Use the Set Breakpoint dialog box to halt execution of methods at specified lines.

The Object list tells you which method you are setting your breakpoint in.

Enter the line number for the breakpoint in the Line Number text box and click OK. The default line number is the current location of the cursor.

Breakpoints must be set at executable lines. If you tell Paradox to set the breakpoint at a non-executable line, it will place the breakpoint at the next available executable line. If it cannot find an executable line, an illegal line number error message appears.

Setting a breakpoint causes ObjectPAL to run the syntax checker on your code. The breakpoint will not be set if a syntax error is found.

When you save your code, all breakpoints are automatically removed. Breakpoints are also removed if you modify your source code.

To open the Set Breakpoint dialog box, choose Debug | Set Breakpoint or click the Set Breakpoint SpeedBar button.

Debug | List Breakpoints

Debug | List Breakpoints displays a dialog box listing the line numbers of breakpoints, and lets you delete any or all breakpoints.

Note: If you change your source code, all breakpoints are automatically deleted.

Debug | Trace Execution

When this item is checked, ObjectPAL Tracer opens a window and lists each line of code as it executes. Your setting for this item is saved with the form, so you don't have to check it every time you want to trace execution.

When this item is not checked, execution proceeds normally. Also, ObjectPAL provides procedures for controlling the Tracer.

Debug | Trace BuiltIns

Debug | Trace BuiltIns displays a dialog box listing all the built-in methods. Check a built-in method to display information about the method in the Tracer window as it executes. You must have code attached to at least one object for this option to take effect.

Note: Checking boxes in this dialog box does not automatically turn on the tracer. You must still choose Debug | Trace Execution, or Debug | Enable Debug and use the **tracerOn** procedure in ObjectPAL code attached to an object.

Checking a built-in method indicates that you want that method traced; unchecked methods are not traced. It does not matter whether or not the method has code attached to it---if it is checked it will be traced.

When the box labeled "form prefilter" is checked, methods are traced as they execute for the form and the intended target object; otherwise methods are traced only for the target object.

See Also

Debug | Trace Execution

Debug | Enable Debug

Select BuiltIn Methods for Tracing Dialog Box

Use the Select BuiltIn Methods for Tracing dialog box to display all of the built-in methods. Checking a built-in method in this dialog box indicates that you want that method traced whether or not it has any ObjectPAL code attached to it.

All	checks all of the methods
None	unchecks all of the methods

To open the Select BuiltIn Methods for Tracing dialog box, choose Debug | Trace BuiltIns.

Debug | Enable DEBUG Statement

When Debug | Enable DEBUG Statement is checked, execution stops whenever the DEBUG statement is encountered. Placing a DEBUG statement in a method has the same effect as setting a breakpoint at that line, with the advantage that it is saved with your source code so you don't have to keep resetting it as you would a breakpoint. The setting for this item is saved with the form.

When this item is not checked, DEBUG statements are ignored. Be sure to uncheck this item before delivering a form or report. Otherwise, the DEBUG statements will execute when the user runs the form or report.

Note: DEBUG statements have an effect only in methods you write and not in procedures.

Note: When this item is checked, Paradox provides more detailed error information. This item is useful even if you never use a DEBUG statement.

Debug | Enable Ctrl+Break to Debugger

When Debug | Enable Ctrl + Break to Debugger is checked, pressing Ctrl+Break suspends execution and opens a Debugger window containing the active method or procedure, just as if a breakpoint had been encountered.

When this item is not checked, pressing Ctrl+Break halts execution.

Note: Ctrl+Break halts execution of ObjectPAL methods and procedures. Other operations (for example, queries) are not affected.

Debug | View Source

Debug | View Source displays another method's code in an Editor window. This is a quick way to move to a specific method for a specific object. You can use the mouse or arrow keys to scroll through items in this dialog box even though it has no scroll bars.

Debug | Origin

Debug | Origin displays the method containing the current breakpoint with the cursor on the line containing the breakpoint.

When execution suspends at a breakpoint, an Editor window displays the method containing the breakpoint. At this time, you can also open other Editor windows containing other methods, and your screen may become cluttered. This function is useful when you have several Editor windows open and want to return quickly to the breakpoint. Debug | Origin is available only when execution is suspended at a breakpoint.

Debug | Step Over

Debug | Step Over lets you single-step through a method treating procedures and custom methods as single steps. This function is available only when execution is suspended at a breakpoint.

See Also

Debug | Step Into

Debug | Set Breakpoint

Debug | Step Into

Debug | Step Into lets you single-step through every line in a method and every line in the rocedures and custom methods the method calls. This function is available only when execution stops at a breakpoint.

See Also

Debug | Step Over

Debug | Set Breakpoint

Debug | Quit This Method

Debug | Quit This Method halts execution and closes any Debugger windows. This function is available only when execution is suspended at a breakpoint.

See Also
Debug | Set Breakpoint

Debug | Run

Choose Debug | Run to run the ObjectPAL Debugger. Paradox saves all attached methods, compiles the code, and leaves you in a View window. When Paradox encounters a breakpoint, execution halts, and a Debugger window opens.

If you are currently in a Debugger window, execution will resume from the breakpoint.

Note: You have to set breakpoints in the code to open a Debug window.

See Also
Debug | Set Breakpoint

Properties

Use the commands on the Properties menu to tailor the ObjectPAL window to your preferences.

Desktop	displays a dialog box for setting properties of the Desktop window.
Window Sizing	sets the default size of Editor and Debugger windows.
Alternate Editor	displays a dialog box where you can specify another editor for writing or editing methods.
Show Compiler Warnings	specifies whether the compiler displays warning messages when it encounters undeclared variables.

See Also

Properties | Desktop

Properties | Window Sizing

Properties | Alternate Editor

Properties | Show Compiler Warnings

Properties | Desktop

Choose Properties | Desktop to open the Desktop Properties dialog box where you can change the appearance of your Desktop.

Title	Type another title to appear on the Desktop title bar.
Background Bitmap	Type the name of a bitmap file, or choose Find to see a list. Find opens the Select a File dialog box. Choose Center Bitmap to display the bitmap in the center of the Desktop, or choose Tile Bitmap to repeat the bitmap until it fills the Desktop.
SpeedBar	Make the SpeedBar a floating palette shaped into one or two columns or rows. To return the floating SpeedBar to its position under the menu, choose Fix from its Control menu.
ObjectPAL Level	Change the level of ObjectPAL that you are working in.

Desktop Properties Dialog Box

Use the Desktop Properties dialog box to change the appearance of your Desktop.

Title	Type another title to appear on the Desktop title bar.
Background Bitmap	Type the name of a bitmap file, or choose Find to see a list. Find opens the Select a File dialog box.
SpeedBar	Make the SpeedBar a floating palette shaped into one or two columns or rows. To return the floating SpeedBar to its position under the menu, choose Fix from its Control menu.
ObjectPAL Level	Change the level of ObjectPAL that you are working in.The default level is Beginner.

To open the Desktop Properties dialog box, choose Properties | Desktop.

Properties | Window Sizing

Choose Properties | Window Sizing to change the default size of the Editor and Debugger windows. Resize the window to the size you want, then choose Properties Window Sizing to display the Editor Window Size dialog box. You can make the current size the default or return to the Windows default size.

See Also
Editor Window Size dialog box

Editor Window Size Dialog Box

Use the Editor Window Size dialog box to change the default size of the Editor and Debugger windows.

Use current size from now on	changes the default Editor and Debugger window size to the size of the currently selected window.
Return to default sizing	changes the default Editor and Debugger window size to the Windows default size.

To open the Editor Window Size dialog box, Choose Properties | Window Sizing .

Properties | Alternate Editor

Choose Properties | Alternate Editor to display a dialog box where you can specify another editor to use for writing and editing methods. Enter the full path to your editor of choice in the ObjectPAL editor text box and click OK.

ObjectPAL Editor Preferences

Use the ObjectPAL Editor Preferences dialog box to specify another editor to use for writing and editing ObjectPAL code.

Alternate ObjectPAL editor	displays the path of an alternate editor that you specify. Enter the full path of the editor you want to use in the text box, choose Use alternate, and click OK.
Use Built-in	select this to use the built-in ObjectPAL Editor.
Use alternate	select this if you have specified an alternate editor in the Alternate ObjectPAL editor text box. This option is not available until you have entered an alternate editor.

Properties | Show Compiler Warnings

Use Properties | Show Compiler Warnings warnings to toggle the display of warning messages from the compiler. When this item is checked, messages in the Status line warn you about undeclared variables and other conditions that might cause errors at run-time. These messages are suppressed when the menu item is not checked.

Types of Constants

ActionClasses
ActionDataCommands
ActionEditCommands
ActionFieldCommands
ActionMoveCommands
ActionSelectCommands
ButtonStyles
ButtonTypes
Colors
CompleteDisplay
ErrorReasons
EventErrorCodes
ExecuteOptions
FieldDisplayTypes
FileBrowserFileTypes
FontAttributes
FrameStyles
General
GraphBindTypes
GraphicMagnification
GraphLabelFormats
GraphLegendPosition
GraphMarkers
GraphTypeOverRide
GraphTypes
IdRanges
Keyboard
KeyBoardStates
LibraryScope
LineEnds
LineStyles
LineThickness
LineTypes
MenuChoiceAttributes
MenuCommands
MenuReasons
MouseShapes
MoveReasons

PatternStyles
QueryRestartOptions
RasterOperations
ReportOrientation
ReportPrintPanel
ReportPrintRestart
Status Reasons
TableFrameStyles
TextAlignment
TextDesignSizing
TextSpacing
UIObjectTypes
ValueReasons
WindowStyles

ActionClasses

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

ActionDataCommands

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 SpeedBar "first record" button, Record | First menu action.
DataBeginEdit	SmallInt	Used to enter Edit mode on the form. Normally always allowed.
DataBeginFirstField	SmallInt	Moves to the first field in the first record of the table associated with the given UIObject.
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 into Design mode. 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 SpeedBar "last record" button.
DataEndEdit	SmallInt	Used to exit Edit mode on the form. Invoked by (second) F9 Edit Data on the SpeedBar, or Form | End Edit menu action.
DataEndLastField	SmallInt	Moves to the last field of the last record of the table associated with a UIObject.
DataFastBackward	SmallInt	Moves backward one "set" of records (where a "set" is defined as the number of rows in a table frame or MRO).
DataFastForward	SmallInt	Moves forward one "set" of records (where a "set" is defined as the number of rows in a table frame or MRO).
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.
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 F6.
DataLookupMove	SmallInt	A special form of lookup which allows the user to choose a new master record for this detail.
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 menu choice, SpeedBar "next record" 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.
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.
DataPrint	SmallInt	Prints a Form or Table window.
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 menu choice, SpeedBar "prior record" 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, Record | Prior Set, Shift+F11, and so forth.
DataRecalc	SmallInt	Forces a calculated field to recalculate its value.
DataRefresh	SmallInt	Specifies a refresh of a value in a record displayed on the screen.
DataRefreshOutside	SmallInt	Specifies a refresh of a value in a record not displayed on the screen.
DataSaveCrosstab	SmallInt	Writes given crosstab to CROSSTAB.DB.
DataSearch	SmallInt	Brings up 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.
DataSearchReplace	SmallInt	Brings up a dialog 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 Ctl+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 bring up 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 on the SpeedBar or Form | 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 Edit Data on the SpeedBar.
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."
DataUnlock Record	SmallInt	Used to commit the record modifications to the table and then (if successful) to unlock the record. Error encountered will call **error** method.

ActionEditCommands

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.
EditCopyToFile	SmallInt	Invokes a dialog box to copy selection to a file.
EditCutSelection	SmallInt	Copies selected area of text to Clipboard and deletes it.
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. Invoked by Ctrl+Backspace.
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.
EditDeleteWord	SmallInt	Deletes word around the current position.
EditDropDownList	SmallInt	Used by drop-down edit fields. Will drop down the associated pick list.
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 dehighlighting it.
EditEnterMemoView	SmallInt	Enters "memo" view on memos or OLE fields.
EditEnterPersistFieldView	SmallInt	Enters "persistent Field View," meaning arrow keys always move within character positions within a field, even when moving to new fields.
EditExitFieldView	SmallInt	Exits Field View (meaning arrow keys will move between fields again) and highlights entire field.
EditExitMemoView	SmallInt	Exits "memo" view on memos or OLE fields, meaning Enter and Tab will once again move between fields.
EditExitPersistField View	SmallInt	Exits "presistent Field View" meaning arrow keys move between fields.
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.
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).
EditPasteFromFile	SmallInt	Invokes a dialog box, allowing user to select file to insert at

		current position.
EditProperties	SmallInt	Invokes the property inspection menu for given object. Currently only graphic fields and formatted memo fields support this.
EditReplace	SmallInt	Toggles overstrike mode in a field object.
EditTextSearch	SmallInt	Invokes a dialog box to allow user to search and replace text within current field.
EditToggleFieldView	SmallInt	Reverses current state of Field View. Recursively calls action (EditEnterFieldView) or action (EditExitFieldView). Invoked by F2 Field View on the SpeedBar.
EditUndoField	SmallInt	Discards current fields modifications and reverts to value in current record buffer. Invoked by Esc.

ActionFieldCommands

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+xTab.
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 or by Home (when not in Field View).
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+ ↑.

ActionMoveCommands

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. Invoked by Ctrl+PgUp.
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).
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).
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. Invoked by PgDn.
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. Invoked by PgUp.
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. Invoked by Ctrl+PgDn.
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 ▪.

ActionSelectCommands

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 displa 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 recod. 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+Crtl+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+Ctrl+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+ ▪.

ButtonStyles

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

ButtonTypes

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.

Colors

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

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

ErrorReasons

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

EventErrorCodes

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	Grants permission to arrive at an object (blocks the move).
CanNotDepart	SmallInt	Grants permission to leave an object (blocks the move).

ExecuteOptions

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

FieldDisplayTypes

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.

FileBrowserFileTypes

Constant	Data type	Description
fbASCII	LongInt	Plain text files.
fbAllTables	LongInt	All table types supported by Paradox.
fbBitmap	LongInt	Bitmap graphics.
fbDBase	LongInt	dBASE tables.
fbExcel	LongInt	Excel worksheets.
fbFiles	LongInt	All files.
fbForm	LongInt	Paradox forms.
fbGraphic	LongInt	Graphic files.
fbIni	LongInt	File whose extension is .INI
fbLibrary	LongInt	ObjectPAL libraries.
fbLotus1	LongInt	Lotus 1-2-3 version 1 worksheets.
fbLotus2	LongInt	Lotus 1-2-3 version 2 worksheets.
fbParadox	LongInt	Paradox tables.
fbQuattro	LongInt	Quattro worksheets.
fbQuattroPro	LongInt	Quattro Pro worksheets.
fbQuattroProWindows	LongInt	Quattro Pro for Windows notebooks.
fbQuery	LongInt	Query files.
fbReport	LongInt	Paradox reports.
fbScript	LongInt	ObjectPAL scripts.
fbTable	LongInt	Paradox tables.
fbTableView	LongInt	Paradox table view files.
fbText	LongInt	All text files.

FontAttributes

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

FrameStyles

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.

General

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

GraphBindTypes

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

GraphicMagnification

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

GraphLabelFormats

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

GraphLegendPosition

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

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 graph.
GraphBar	SmallInt	Displays specified series as a bar graph.
GraphDefault	SmallInt	Displays specified series in the default graph type.
GraphLine	SmallInt	Displays specified series as a line graph.

GraphTypes

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

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

Constant	Data type	Description
VK_ADD	SmallInt	Add 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

KeyBoardStates

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

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.

LineEnds

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.

LineStyles

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

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.

LineTypes

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

MenuChoiceAttributes

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.

MenuCommands

Constant	Data type	Description
MenuBuild	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.
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 Control menu.
MenuControlPrevWindow	SmallInt	Same as choosing Prev Window from the Control menu.
MenuControlRestore	SmallInt	Same as choosing Restore from the Control menu.
MenuControlSize	SmallInt	Same as choosing Size from the Control menu.
MenuEditCopy	SmallInt	Edit | Copy
MenuEditCopyTo	SmallInt	Edit | Copy To
MenuEditCut	SmallInt	Edit | Cut
MenuEditDelete	SmallInt	Edit | Delete
MenuEditPaste	SmallInt	Edit | Paste
MenuEditPasteFrom	SmallInt	Edit | Paste From
MenuEditSearchText	SmallInt	Edit | Search Text
MenuEditUndo	SmallInt	Edit | Undo
MenuFileAliases	SmallInt	File | Aliases
MenuFileExit	SmallInt	File | Exit
MenuFileExport	SmallInt	File | Utiltities | Export
MenuFileImport	SmallInt	File | Utilitites | Import
MenuFileMultiBlankZero	SmallInt	File | System Settings | Blank As Zero
MenuFileMultiUserAutoRefresh	SmallInt	File | System Settings | Auto Refresh
MenuFileMultiUserDrivers	SmallInt	File | System Settings | Drivers
MenuFileMultiUserLock	SmallInt	File | Multiuser | Set Locks
MenuFileMultiUserLockInfo	SmallInt	File | Multiuser | Display Locks
MenuFileMultiUserRetry	SmallInt	File | Multiuser | Set Retry

MenuFileMultiUserUserName	SmallInt	File | Multiuser | User Name
MenuFileMultiUserWho	SmallInt	File | Multiuser | Who
MenuFilePrint	SmallInt	File | Print
MenuFilePrinterSetup	SmallInt	File | Printer Setup
MenuFilePrivateDir	SmallInt	File | Private Directory
MenuFileTableAdd	SmallInt	File | Utilties | Add
MenuFileTableCopy	SmallInt	File | Utilties | Copy
MenuFileTableDelete	SmallInt	File | Utilties | Delete
MenuFileTableEmpty	SmallInt	File | Utilties | Empty
MenuFileTableInfoStructure	SmallInt	File | Utilties | Info Structure
MenuFileTablePasswords	SmallInt	File | Utilties | Passwords
MenuFileTableRename	SmallInt	File | Utilties | Rename
MenuFileTableRestructure	SmallInt	File | Utilties | Restructure
MenuFileTableSort	SmallInt	File | Utilties | Sort
MenuFileTableSubtract	SmallInt	File | Utilties | Subtract
MenuFileWorkingDir	SmallInt	File | Working Directory
MenuFolderOpen	SmallInt	File | Open | Folder
MenuFormDesign	SmallInt	Form | Design
MenuFormEditData	SmallInt	Form | Edit Data
MenuFormFieldView	SmallInt	Form | Field View
MenuFormNew	SmallInt	File | New | Form
MenuFormOpen	SmallInt	File | Open | Form
MenuFormOrderRange	SmallInt	Form | Order/Range
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
MenuFormShowDeleted	SmallInt	Form | Show Deleted
MenuFormTableView	SmallInt	Form | Table View
MenuHelpAbout	SmallInt	Help | About
MenuHelpContents	SmallInt	Help | Contents
MenuHelpKeyboard	SmallInt	Help | Keyboard
MenuHelpSpeedBar	SmallInt	Help | SpeedBar
MenuHelpSupport	SmallInt	Help | Support Info
MenuHelpUsingHelp	SmallInt	Help | Using Help
MenuInit	SmallInt	Generated by clicking a menu item
MenuLibraryNew	SmallInt	File | New | Library
MenuLibraryOpen	SmallInt	File | Open | Library
MenuPasteLink	SmallInt	Edit | Paste Link

MenuPropertiesCurrent	SmallInt	Properties | Current Object
MenuPropertiesDesigner	SmallInt	Properties | Designer
MenuPropertiesDesktop	SmallInt	Properties | Desktop
MenuPropertiesExpandedRuler	SmallInt	Properties | Expanded Ruler
MenuPropertiesFormRestoreDefaults	SmallInt	Properties | Form Options | Restore Defaults
MenuPropertiesFormSaveDefaults	SmallInt	Properties | Form Options | Save Defaults
MenuPropertiesHorizontalRuler	SmallInt	Properties | Form Options | Horizontal Ruler
MenuPropertiesVerticalRuler	SmallInt	Properties | Form Options | Vertical Ruler
MenuPropertiesZoom100	SmallInt	Properties | Zoom | 100%
MenuPropertiesZoom200	SmallInt	Properties | Zoom | 200%
MenuPropertiesZoom25	SmallInt	Properties | Zoom | 25%
MenuPropertiesZoom400	SmallInt	Properties | Zoom | 400%
MenuPropertiesZoom50	SmallInt	Properties | Zoom | 50%
MenuPropertiesZoomBestFit	SmallInt	Properties | Zoom | Best Fit
MenuPropertiesZoomFitHeight	SmallInt	Properties | Zoom | Fit Height
MenuPropertiesZoomFitWidth	SmallInt	Properties | 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
MenuReportNew	SmallInt	File | New | Report
MenuReportOpen	SmallInt	File | Open | Report
MenuSave	SmallInt	File | Save
MenuScriptNew	SmallInt	File | New | Script
MenuScriptOpen	SmallInt	File | Open | Script

MenuSelectAll	SmallInt	Edit | Select All
MenuTableNew	SmallInt	File | New | Table
MenuTableOpen	SmallInt	File | Open | Table
MenuWindowArrangeIcons	SmallInt	Window | Arrange Icons
MenuWindowCascade	SmallInt	Window | Cascade
MenuWindowCloseAll	SmallInt	Window | Close All
MenuWindowTile	SmallInt	Window | Tile

MenuReasons

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 SpeedBar button.

MouseShapes

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).
MouseUpArrow	LongInt	Pointer is an arrow pointing up.
MouseWait	LongInt	Pointer is an hourglass.

MoveReasons

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.

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
RightDiagonalLinePattern	SmallInt
ScalesPattern	SmallInt
StaggeredDashPattern	SmallInt
ThickHorizontalLinePattern	SmallInt
ThickStripesDownPattern	SmallInt
ThickStripesUpPattern	SmallInt
ThickVerticalLinesPattern	SmallInt
VerticalLinesPattern	SmallInt
VeryHeavyDotPattern	SmallInt
WeavePattern	SmallInt
ZigZagPattern	SmallInt

QueryRestartOptions

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.

RasterOperations

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 unchaged 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

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

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

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.

StatusReasons

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.

TableFrameStyles

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

Constant	Data type	Description
TextAlignBotton	SamllInt	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

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

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.

UIObjectTypes

Constant	Data type	Description
BoxTool	SmallInt	Creates a box.
ButtonTool	SmallInt	Creates a button.
ChartTool	SmallInt	Creates a graph.
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.
RecordTool	SmallInt	Creates a record.
TableFrameTool	SmallInt	Creates a table frame.
TextTool	SmallInt	Creates a text box.
XtabTool	SmallInt	Creates a crosstab object.

ValueReasons

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.

WindowStyles

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 miximize 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

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.

The basic language elements are:

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

=

Keyword

Syntax **itemSpec =** *expression*

Description The **=** statement 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.

In a statement containing more than one **=** statement, the first **=** does the assignment, all others compare two values or expressions.

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
```

See Also

Containers
Properties
Variables

const

Keyword 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.

Example

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

See Also var

disableDefault

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

Syntax **disableDefault**

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

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 window.

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

See Also doDefault
enableDefault
passEvent

doDefault

Keyword Executes the default code for a built-in 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.

Example 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
```

See Also disableDefault
enableDefault
passEvent

enableDefault

Keyword Enables the default code for a built-in 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.

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
```

See Also disableDefault
doDefault
passEvent

for

Keyword Executes a sequence of statements a specified number of times.

Syntax **for** *counter* [**from** *startVal*] [**to** *endVal*] [**step** *stepVal*]

Statements

endFor

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.

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.

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*.

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
```

See Also loop
quitLoop
while

forEach

Keyword

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. 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 *DynArrayName* does not exist, the **forEach** statement causes an error when the method is compiled.

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.

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
```

See Also

for

loop

quitLoop

while

if

Keyword 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
```

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
```

See Also for

iif

switch

while

iif

Keyword	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.
Example	`a = iif(x > 1, b, c) ; if x > 1, a = b; else a = c`
See Also	if

loop

Keyword 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.

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
```

See Also

for
forEach
quitLoop
scan
while

method

Keyword Defines an ObjectPAL method.

Syntax **method** *Name* (*parameterDesc* [,*parameterDesc*]*]) [*returnType*]

[**type** *section*]

[**const** *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, 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

Example

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

See Also proc

passEvent

Keyword 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.

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
```

See Also disableDefault
doDefault
enableDefault

proc

Keyword 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, 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
```

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.

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
```

See Also method

quitLoop

Keyword 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.

Example In this 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
```

See Also loop
return

return

Keyword 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. The following apply to **return**:

- The procedure must be declared to return a value before you can use **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.

Example Following is a simple example that 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 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
```

See Also quitLoop

scan

Keyword 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**.

Example This 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()
```

See Also if
for
while

switch

Keyword 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**.

Example This 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
```

See Also if
while

try

Keyword 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.

Example This 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
```

See Also System::fail

type

Keyword Declares data types.

Syntax **type**

[*newTypeName* **=** *existingType*]*

endType

Description Using **type**, you can define new data types. Once defined, you can use these types to declare variables in methods.

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
```

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
```

See Also var

uses

Keyword Declares external library routines to use in a method or procedure.

Syntax **uses** [*library* | *ObjectPAL*]

routineName (*parameterList*)

endUses

Description The **uses** block, declared in an object's Uses window, makes routines stored in external libraries available to ObjectPAL methods. The routines must fit one of the following descriptions:

- Routines written in ObjectPAL and stored in an ObjectPAL library.
- Routines written in ObjectPAL and attached to a form. The syntax for calling methods attached to a form is the same as for calling them from a library.
- Routines written in assembly language, C, C++, or Pascal and stored in an ObjectPAL library 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.

A **uses** block for an ObjectPAL library differs slightly from a **uses** block for a DLL, so they're discussed in separate sections.

To use methods stored in an ObjectPAL library or attached to a form, write a **uses** block in an object's Uses window. Here's the syntax:

uses ObjectPAL

[*methodName* ([**var | const**] *argList*) [*returnType*]]*

endUses

The keyword **ObjectPAL** is required to indicate that you're calling methods from an ObjectPAL library or a form, rather than from a DLL.

Important: You must open a library before calling a method from it; you must open and run a form before calling a method from it.

Next, *methodName* represents the name of the method to call, and *argList* represents a comma-separated list of argument/data type pairs, optionally preceded by the keywords **var** and **const**, as appropriate.

The optional argument *returnType* specifies the data type of the value (if any) returned by the method.

Within a single Uses window, you can declare more than one library method, and methods from more than one library.

Important: Arguments and data types declared in the Uses window must be declared exactly as they are in the library.

The following example declares two library methods, **buildMenu** and **calcInterest. buildMenu** takes one argument, a String constant named *pageName*. **calcInterest** takes two arguments: a Number variable named *intRate* (passed by reference) and a SmallInt variable named *nPeriods* (passed by value).

```
uses ObjectPAL
   buildMenu(var pageName String)
   calcInterest(var intRate Number, nPeriods SmallInt) Number
endUses]
```

The following code, attached to a button's uses window, declares the **calcInterest** method so the button can use it. Notice that you don't have to declare every method

in a library, just the ones you want to use.

```
uses ObjectPAL
      calcInterest(var intRate Number, nPeriods SmallInt)
Number
endUses
```

The following code, attached to a button's built-in **pushButton** method, opens the library and calls these methods.

```
method pushButton(var eventInfo Event)
   var
      mathLib Library
      intRate Number
      nPeriods SmallInt
      interest Number
   endVar

   if mathiLib.open("mathlib.lsl") then
      intRate = mortgage.intRate.value
      nPeriods = mortgage.nYears.value * 12
      interest = mathLib.calcInterest(intRate, 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 *mastLib*.

```
interest = mathLib.calcInterest(intRate, 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 example assumes that the Form variable *codeForm* has been previously declared, the form has been opened, and the method **getObjHelp** has been declared in an appropriate uses window.

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

To use routines stored in a DLL, write a **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 pseudocode) is the same:

uses *libraryName*

routineName* (*parameterList*) *returnType

endUses

The argument *libraryName* specifies the DLL file name, where *libraryName* is a valid DOS filename of up to eight characters. Paradox assumes a file-name extension of .DLL or .EXE. Windows searches for the file in this order:

1. The current directory.

2. The Windows directory (the directory containing WIN.COM). You can use the FileSystem procedure **windowsDir** to get this information (typically, it's C:\WINDOWS).
3. The Windows system directory (the directory containing such system files as KERNEL.EXE). You can use the FileSystem procedure **windowsSystemDir** to get this information (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.

Note to 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 **uses** block can contain one or more *routineNames*, and each *routineName* can have its own *parameterList*. A *parameterList* specifies one 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.

Declare a **uses** block in an object's Uses window, and within that window, declare one **uses** block for each library or DLL you want to use. You don't have to declare every routine the library or DLL contains, just the ones you want to use. Once declared, routines are available to all methods attached to that object, and to all objects that object contains.

In a **uses** block, declare data types using the following keywords:

Data type	ObjectPAL	C	Pascal
16-bit integer	CWORD	int	Integer
32-bit integer	CLONG	long	Longint
64-bit floating-point number	CDOUBLE	double	Double
80-bit floating-point number	CLONGDOUBLE	long double	Extended
pointer	CPTR	char far *	String
binary or graphic data	CHANDLE	Handle (Windows)	THandle (Windows)

These keywords are valid only within a **uses** block. Don't use them anywhere else.

To use a 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
```

In the previous example, arguments in the **uses...endUses** block are declared using CLONG and CDOUBLE, and variables in the method were declared using LongInt and Number. That's because ObjectPAL data types are more complex (and powerful) than corresponding data types in C or Pascal:

- CWORD corresponds to SmallInt.
- CLONG corresponds to LongInt.
- CDOUBLE and CLONGDOUBLE correspond to Number.
- CPTR corresponds to String.
- CHANDLE corresponds to Binary and Graphic.

Note: Do not modify the contents of a passed CPTR.

Example

This example uses routines from MINMAX.DLL, written using Borland's Turbo Pascal for Windows. The Pascal code for the DLL is:

```
{ this is the Pascal code that defines the DLL }
library MinMax;

function Min(X, Y: Integer): Integer; export;
begin
   if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; export;
begin
   if X > Y then Max := X else Max := Y;
end;

exports
   Min index 1,
   Max index 2;

begin
end.
```

Following is the ObjectPAL code to use the routines in the DLL. First is the code for the Uses window, then comes 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 CWORD, y CWORD) CWORD ; declare the routines to use
   Max (x CWORD, y CWORD) CWORD
endUses
```

The following code modifies a button's built-in **pushButton** method:

```
method pushButton(var eventInfo Event)
   var
      x, y, z SmallInt
   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
```

Calling C routines

Windows uses the same calling convention used by Pascal. The Pascal calling convention entails the following:

- Parameters are pushed onto the stack in the order in which they appear in the function call.
- The code that restores the stack is part of the called function (rather than the calling function).

The Pascal calling convention differs from the calling convention used in C. In C, parameters are pushed onto the stack in reverse order, and the calling function is responsible for restoring the stack. When writing a DLL in a language that does not ordinarily use the Pascal calling convention, such as C, you must ensure that the Pascal calling convention is used for any function that is called by Windows. In C, this requires the use of the **PASCAL** keyword when the function is declared.

The examples in the following tables assume that these ObjectPAL variables have been declared:

si	SmallInt
li	LongInt
nu	Number
gr	Graphic
st	String

When passing a value to a C procedure, the ObjectPAL variable must be declared and typed explicitly. However, AnyType is not allowed.

If you are using C++, surround your C code with the following syntax:

```
extern "C"
{
// Your procs
}
```

All C functions that you want ObjectPAL to call must be exported in the .DEF file.

See Also

Passing by value

Passing by pointer

Returning values

Notes on using graphic and binary data

Passing by value

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, pass values by pointer.

C data type	C syntax	In USES block	ObjectPAL call
Long double	void pascal far _loadds cproc(long double value)	cproc(value CLONGDOUBLE)	cproc(si)
			cproc(li)
			cproc(nu)
Double	void pascal far _loadds cproc(double value)	cproc(value CDOUBLE)	cproc(si),
			cproc(li)
			cproc(nu)
Long int	void pascal far _loadds cproc(long int value)	cproc(value CLONG)	cproc(si)
			cproc(li)
Int	void pascal far _loadd cproc(int value)	cproc(value CWORD)	cproc(si)
String	void pascal far _loadds cproc(char * value)	cproc(value CPTR)	cproc(st)
Graphic	void pascal far _loadds cproc(HANDLE value)	cproc(value CHANDLE)	cproc(gr)
Binary	void pascal far _loadds cproc(HANDLE value)	cproc(value CHANDLE)	cproc(gr)

See also

Calling C Routines

Passing by pointer

To pass information if the C procedure takes pointers to information, the pointer points directly to the corresponding value in the ObjectPAL variable. For example, if you want an int* and you pass a SmallInt, you will get a pointer that points directly to the int inside the SmallInt variable. You will be able to modify the value of the SmallInt. This could be extremely dangerous for novice C programmers, since you can corrupt ObjectPAL by overwriting memory (writing past the bounds of the memory pointer).

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.

C data type	C syntax	In USES block	ObjectPAL call
LONG DOUBLE *	void pascal far _loadds cproc(long double * value)	cproc(value CPTR)	cproc(nu)
LONG INT *	void pascal far _loadds cproc(long int * value)	cproc(value CPTR)	cproc(li)
INT *	void pascal far _loadds cproc(int * value)	cproc(value CPTR)	cproc(si)
STRING *	void pascal far _loadds cproc(char * value)	cproc(value CPTR)	cproc(st)

See also

Calling C Routines

Returning Values

The following table presents the syntaxes for returning values of various data types from a C procedure.

C data type	C syntax	In USES block	ObjectPAL call
Long double	long double pascal far _loadds cproc(void)	cproc() CLONGDOUBLE	nu = cproc()
Double	double pascal far _loadds cproc(void)	cproc() CDOUBLE	nu = cproc()
Long int	long int pascal far _loadds cproc(void)	cproc() CLONG	nu = cproc()
			li = cproc()
Int	long int pascal far _loadds cproc(void)	cproc() CWORD	nu = cproc()
			li = cproc()
			sm = cproc()
Char	char * pascal far _loadds cproc(void)	cproc() CPTR	st = cproc()

See also

Calling C Routines

Notes on Graphic and Binary data(CHANDLE)

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 it, include:

```
void pascal far _loadds cproc(HANDLE value)
{
   huge * ptr = (huge *) GlobalLock(value);
   // .... use ptr.
   // ...  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. You can read or modify the data, but you cannot change its size.

For a Graphic variable, HANDLE is Windows Bitmap.

Use this as you would any other bitmap HANDLE.

var

Keyword 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.

Note: A variable's scope depends on where it is declared.

Example

```
var
   myChars, xx String
   myNum Number
   orders, sales, parts TCursor
   proteus AnyType
   myBox UIObject
   a, b Array[5] SmallInt
   myOtherNum Number
endVar
```

See Also method

while

Keyword 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.

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
```

See Also

for
forEach
if
loop
quitLoop
scan

Type Reference

ObjectPAL methods and procedures fall into six categories. 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. Choose a category below to see the types it contains.

Data model objects	Design objects	Basic language	Workgroup Methods
System data objects	Display managers		
Data types	Events		

The level of ObjectPAL that you are working in determines the number of methods available for each type. You can change the ObjectPAL level in the Desktop Properties dialog box.

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

- From the Help window menu choose Edit | Copy, then select the block of code you want and click Copy.
- Place the cursor in your code where you want to insert the example, then choose Edit | Paste from the Paradox menu.

See Also

The ObjectPAL Editor

The ObjectPAL Debugger

Scripts

Libraries

Alphabetic list of methods

Pasting Sample Code Into Your Methods

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

- From the Help window menu choose Edit | Copy, then select the block of code you want and click Copy.
- Place the cursor in your code where you want to insert the example, then choose Edit | Paste from the Paradox menu.

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

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

System

TextStream

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

Design 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.

Menu

PopupMenu

UIObject

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

TableView

Events

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

DataBase Type

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 myDB 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.

close

delete

executeQBE

executeQBEFile

executeQBEString

isAssigned

isTable
open
writeQBE

Query Type

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.

Most methods for working with queries are defined for the Database type, because in certain applications you will need to specify the database that contains the tables to query.

executeQBE

isAssigned

query

writeQBE

Table Type

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.

add

attach

cAverage

cCount

cMax

cMin

cNpv

compact

copy

create

cSamStd

cSamVar

cStd

cSum

cVar

delete

dropIndex

empty

enumFieldNames

enumFieldNamesInIndex

enumFieldStruct

enumIndexStruct

enumRefIntStruct

enumSecStruct

familyRights

fieldName

fieldNo

fieldType

index

isAssigned

isEmpty

isEncrypted

isShared

isTable

lock
nFields
nKeyFields
nRecords
protect
reIndex
reIndexAll
rename
setExclusive
setFilter
setIndex
setReadOnly
showDeleted
sort
subtract
tableRights
type
unAttach
unlock
unProtect
usesIndexes

TCursor Type

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.

For information about related objects, refer to the Table and TableView types.

add

atFirst

atLast

attach

attachToKeyViol

bot

cancelEdit

cAverage

cCount

close

cMax

cMin

cNpv

compact

copy

copyFromArray

copyRecord

copyToArray

cSamStd

cSamVar

cStd

cSum

currRecord

cVar

deleteRecord

didFlyAway

dropIndex

edit

empty

end

endEdit

enumFieldNames

enumFieldNamesInIndex

enumFieldStruct

enumIndexStruct

enumLocks
enumRefIntStruct
enumTableProperties
eot
familyRights
fieldNo
fieldRights
fieldSize
fieldType
fieldUnits2
fieldValue
getLanguageDriver
getLanguageDriverDesc
home
initRecord
insertAfterRecord
insertBeforeRecord
insertRecord
isAssigned
isEdit
isEmpty
isEncrypted
isRecordDeleted
isShared
isShowDeletedOn
isValid
locate
locateNext
locateNextPattern
locatePattern
locatePrior
locatePriorPattern
lock
lockRecord
lockStatus
moveToRecord
moveToRecNo
nextRecord
nFields
nKeyFields

nRecords
open
postRecord
priorRecord
qLocate
recNo
recordStatus
reIndex
reIndexAll
seqNo
setFlyAwayControl
setFieldValue
setFilter
showDeleted
skip
subtract
switchIndex
tableName
tableRights
type
unDeleteRecord
unlock
unlockRecord
updateRecord

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 PDOXWIN.

Note: Paradox and ObjectPAL also support OLE, another protocol for sharing data between applications. Refer to the OLE type and to the *ObjectPAL Developer's Guide* for more information.

execute

open

setItem

FileSystem Type

FileSystem variables provide access to and information about disk files, drives, and directories. A FileSystem variable provides a handle, a variable you can use in ObjectPAL statements to work with a directory or a file. In many cases, the first step in working with FileSystem variables is using **findFirst** to see if any information is present. It may be helpful to think of this step as initializing the FileSystem variable.

accessRights
copy
delete
deleteDir
drives
enumFileList
existDrive
findFirst
findNext
size
freeDiskSpace
fullName
getDir
getDrive
getFileAccessRights
getValidFileExtensions
isDir
isFile
isFixed
isRemote
isRemovable
makeDir
name
privDir
rename
setDir
setDrive
setFileAccessRights
splitFullFileName
startUpDir
time
totalDiskSpace
windowsDir
windowsSystemDir
workingDir

Library Type

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 methods. You add code to a library just as you do to a form, using the Methods dialog box 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, you can't place design objects in the library.

close

enumSource

enumSourceToFile

execMethod

open

Session Type

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.

addAlias

addPassword

advancedWildcardsInLocate

blankAsZero

close

enumDataBaseTables

enumDriverCapabilities

enumDriverInfo

enumDriverNames

enumDriverTopics

enumEngineInfo

enumFolder

enumOpenDatabases

enumUsers

getAliasPath

getNetUserName

ignoreCaseInLocate

isAdvancedWildcardsInLocate

isAssigned

isBlankZero

isIgnoreCaseInLocate

lock

open

removeAlias

removeAllPasswords

removePassword

retryPeriod

saveCFG

setAliasPath

setRetryPeriod

unLock

System Type

The System type contains procedures for displaying messages, finding out about the user's system, manipulating the File Browser, working with the Help system, and more.

beep

close

constantNameToValue

constantValueToName

cpuClockTime

dataModelAddTable

dataModelHasTable

dataModelRemoveTable

debug

dlgAdd

dlgCopy

dlgCreate

dlgDelete

dlgEmpty

dlgNetDrivers

dlgNetLocks

dlgNetRefresh

dlgNetRetry

dlgNetSetLocks

dlgNetSystem

dlgNetUserName

dlgNetWho

dlgRename

dlgRestructure

dlgSort

dlgSubtract

dlgTableInfo

enumDesktopWindowNames

enumFonts

enumFormNames

enumReportNames

enumRTLClassNames

enumRTLConstants

enumRTLMethods

enumWindowNames

errorClear

errorCode

errorLog
errorMessage
errorPop
errorShow
errorTrapOnWarnings
execute
exit
fail
fileBrowser
formatAdd
formatDelete
formatExist
formatSetCurrencyDefault
formatSetDateDefault
formatSetDateTimeDefault
formatSetLogicalDefault
formatSetLongIntDefault
formatSetNumberDefault
formatSetSmallIntDefault
formatSetTimeDefault
getMouseScreenPosition
helpOnHelp
helpQuit
helpSetIndex
helpShowContext
helpShowIndex
helpShowTopic
helpShowTopicInKeywordTable
message
msgAbortRetryIgnore
msgInfo
msgQuestion
msgRetryCancel
msgStop
msgYesNoCancel
pixelsToTwips
play
readEnvironmentString
readProfileString
setMouseScreenPosition

setMouseShape
sleep
sound
sysInfo
tracerClear
tracerHide
tracerOff
tracerOn
tracerSave
tracerShow
tracerToTop
tracerWrite
version
winGetMessageID
winPostMessage
winSendMessage
writeEnvironmentString
writeProfileString

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).

advMatch

close

commit

create

end

eof

home

open

position

readChars

readLine

setPosition

size

writeLine

writeString

AnyType Type

An AnyType value can be any one of the data types listed in the following table.

Type	Description
AnyType	A catch-all for basic data types
Array	An indexed collection of data
Binary	Machine-readable data
Currency	Used to manipulate currency values
Date	Calendar data
DateTime	Calendar and clock data combined
DynArray	A dynamic array
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
Record	A user-defined structure
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.)

blank

dataType

isAssigned

isBlank

isFixedType

view

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. The Array type also includes methods defined for the AnyType type. Refer to AnyType for more information.

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 also method and procedures for the DynArray type.

addLast

append

contains

countOf

empty

exchange

fill

grow

indexOf

insert

insertAfter

insertBefore

insertFirst

isResizeable

remove

removeAllItems

removeItem

replaceItem

setSize

size

view

Binary Type

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 table, or from a disk file or a table to a method or procedure.

The Binary type also includes methods defined for the AnyType type. Refer to AnyType for more information.

readFromFile

size

writeToFile

Currency Type

Currency values can range from 3.4 * ± 10-4930 to 1.1 * ± 104930 precise to six 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 six decimal places. The Currency type also includes methods defined for the AnyType type and the Number type. Refer to AnyType and Number for more information.

Currency

Date Type

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.

Although you can use ObjectPAL to perform calculations on any valid date, date values stored in a Paradox table must range from Jan. 1, 100, to Dec. 31, 9999.

Dates in the 20th century can be specified using two digits for the year, as in

```
myDay = date("11/09/59")
```

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.

The Date type includes several methods defined for the DateTime type. Refer to DateTime for more information.

date

dateVal

today

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** method (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.

The DateTime type includes methods defined for the AnyType type. Refer to AnyType for more information. Also, both the Date and Time types include methods defined for the DateTime type.

dateTime

day

daysInMonth

dow

dowOrd

doy

hour

isLeapYear

milliSec

minute

month

moy

second

year

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 released from it. A DynArray's size is limited only by system memory.

Note: ObjectPAL also supports fixed-size and resizeable arrays.

Unlike fixed-size arrays, the indexes of dynamic arrays are not integers; dynamic array indexes 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 also includes methods defined for the AnyType type. Refer to AnyType for more information.

contains

getKeys

removeItem

size

view

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 also includes methods defined for the AnyType type. Refer to AnyType for more information.

readFromClipboard

readFromFile

writeToClipboard

writeToFile

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,

Is that table empty?

Is that form displayed as an icon?

Did that operation successfully create a text file?

logical

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        ; causes an error
```

To work with boundary values, store the result in a Number variable.

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 also includes methods defined for the AnyType type. Refer to AnyType for more information.

bitAND

bitIsSet

bitOR

bitXOR

LongInt

Memo Type

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 also includes methods defined for the AnyType type. Refer to AnyType for more information.

readFromFile

memo

writeToFile

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 ± 3.4 * 10 -4930 to ± 1.1 * 10 4930. An attempt to assign a value outside of this range to a Number variable causes an error.

Note: The Number type includes methods defined for the AnyType type. Refer to AnyType for more information. Also, 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()
```

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.

abs

acos

asin

atan

atan2

ceil

cos

cosh

exp

fraction

floor

fv

ln

log

max
min
mod
number
numVal
pmt
pow
pow10
pv
rand
round
sinh
sin
sqrt
tanh
tan
truncate

OLE Type

OLE is an acronym for Object Linking and Embedding, a protocol that provides access to the function 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 function of the paint program available to your users (assuming, of course, that your users have the paint program installed on their systems).

The OLE type also includes methods defined for the AnyType type. Refer to AnyType for more information.

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 not an OLE server.

OLE client is an application that can use the OLE mechanism to access documents created by an OLE server. Paradox is an OLE client.

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.

canReadFromClipboard

edit

enumVerbs

getServerName

readFromClipboard

writeToClipboard

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 an 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 Point type also includes methods defined for the AnyType type Refer to AnyType for more information.

distance

isAbove

isBelow

isLeft

isRight

point

setX

setXY

setY

x

y

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. 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 also includes methods defined for the AnyType type. Refer to AnyType for more information.

view

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 ; causes an error
```

To work with boundary values, store the result in a variable of a type that can accommodate it. For example,

```
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 = x + y

z.view() ; displays 32768 because z is a LongInt
         ; and can handle the large value
```

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.

Note: The SmallInt type includes methods defined for the AnyType type. Refer to AnyType for more information. Also, 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.

bitAND

bitIsSet

bitOR

bitXOR

int

smallInt

String Type

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.

The String type also includes methods defined for the AnyType type. Refer to AnyType for more information.

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.

advMatch

ansiCode

breakApart

chr

chrOEM

chrToKeyName

fill

format

ignoreCaseInStringCompares

isIgnoreCaseInStringCompares

isSpace

keyNameToChar

keyNameToVKCode

lower

lTrim

match

oemCode

rTrim

search

size

space

string

strVal

substr

toANSI
toOEM
upper
vkCodeToKeyName

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 methods defined for the AnyType and DateTime types. Refer to the DateTime and AnyType sections for more information.

time

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 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 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.

addArray

addBreak

addPopUp

addStaticText

addText

contains

count

empty

getMenuChoiceAttribute

getMenuChoiceAttributeById

hasMenuChoiceAttribute

remove

removeMenu

setMenuChoiceAttribute

setMenuChoiceAttributeById

show

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. However, the PopUpMenu type includes methods defined for the Menu type. Refer to the "Menu" seciton for more information.

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

addArray

addBar

addBreak

addPopUp

addSeparator

addStaticText

addText

show

switchMenu

UIObject Type

UIObjects (the UI stands for User Interface) create the user interface for an application: anything you can place in a form is a UIObject. Only UIObjects have built-in methods, The different UIObjects are the bitmap, box, button, crosstab, ellipse, field object, form, graph, line, 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 methods, you can attach code to those built-in methods, and a form responds to events. There is also a separate type, Form, for methods and procedures that work only with forms.

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.

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.**

enumUIClasses
enumUIObjectNames
enumUIObjectProperties
execMethod
getBoundingBox
getPosition
getProperty
getPropertyAsString
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
setFilter
setPosition
setProperty
setTimer
skip
switchIndex
twipsToPixels
unDeleteRecord
unlockRecord
view
wasLastClicked
wasLastRightClicked

Application Type

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. The Application type includes methods defined for the Form type. Refer to the Form type for more information.

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
   x, y, w, h    LongInt
endVar
thisApp.getPosition(x, y, w, h)                    ; get current position
thisApp.setPosition(x, y, w * 0.9, h * 0.9)    ; shrink desktop by 10%
endmethod
```

All of the methods for the Application type are defined for the Form type. See Form for more information.

Form Type

A Form variable provides a handle for working with a Paradox form. Form type methods let you

- Load a form in a design window and save a design
- Open and close a form
- Attach to an open form
- Add and remove tables in a form's data model
- 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.

action

attach

bringToTop

close

create

delayScreenUpdates

design

disableBreakMessage

dmAddTable

dmGet

dmHasTable

dmPut

dmRemoveTable

enumSource

enumSourceToFile

enumTableLinks

enumUIObjectNames

enumUIObjectProperties

formCaller

formReturn

getPosition

getTitle

hide

hideSpeedBar

isMaximized

isMinimized

isSpeedBarShowing

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
setPosition
setTitle
show
showSpeedBar
wait
windowClientHandle
windowHandle

Report Type

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 Design window; use **open** to open the report in the View Data window, and use **print** to open a report and print it. You cannot attach methods to objects in a report.

The Report type includes several methods defined for the Form type. Refer to Form for more information.

attach

close

currentPage

design

enumUIObjectNames

enumUIObjectProperties

load

moveToPage

open

print

run

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 three 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

Refer to Form for more information.

action

close

open

wait

ActionEvent Type

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 method that is triggered by an ActionEvent is **action**. This method, along with the rest of the built-in methods, is discussed in Chapter 2. For information about the Event model, see Chapter 6 in the *ObjectPAL Developer's Guide*.

Typically, when you work with ActionEvents, you'll also work with ObjectPAL's 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(1)
endif
endMethod
```

Action constants are listed in the Constants dialog box. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose an item beginning with Action; for example, ActionDataCommands. The constants appear in the Constants column.

For more information and examples, refer to Chapter 6 in the *ObjectPAL Developer's Guide*

The ActionEvent type includes several methods defined for the Event type. Refer to Event for more information.

actionClass

id

setId

ErrorEvent Type

The ErrorEvent type provides methods you can use to get and set information about errors that occur as ObjectPAL code executes. The ErrorEvent type includes several methods defined for the Event type. Refer to Event for more information.

The only built-in method triggered by an ErrorEvent is **error**.

reason

setReason

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 in this section are used by the other event types.

The following built-in methods are triggered by Events: **open**, **close**, **setFocus**, **removeFocus**, **newValue**, and **pushButton**.

errorCode

getTarget

isFirstTime

isPreFilter

isTargetSelf

reason

setErrorCode

setReason

KeyEvent Type

A KeyEvent object gets and sets information about keystroke events.

The following built-in methods are triggered by KeyEvents: **keyChar**, and **keyPhysical**.

The KeyEvent type includes several methods defined for the Event type. Refer to Event for more information.

char

charAnsiCode

isAltKeyDown

isControlKeyDown

isFromUI

isShiftKeyDown

setAltKeyDown

setChar

setControlKeyDown

setShiftKeyDown

setVChar

setVCharCode

vChar

vCharCode

MenuEvent Type

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 **menuAction** method. By modifying an object's built-in **menuAction** method, you can define how the object responds.

The MenuEvent type includes several methods defined for the Event type. Refer to Event for more information.

data

id

isFromUI

menuChoice

reason

setData

setId

setReason

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 MouseEvent type includes several methods defined for the Event type. Refer to Event for more information.

The following built-in methods are triggered by MouseEvents: **mouseClick**, **mouseDown**, **mouseUp**, **mouseDouble**, **mouseRightUp**, **mouseRightDown**, **mouseRightDouble**, **mouseMove**, **mouseEnter**, and **mouseExit**.

getMousePosition

getObjectHit

isControlKeyDown

isFromUI

isLeftDown

isMiddleDown

isRightDown

isShiftKeyDown

setControlKeyDown

setInside

setLeftDown

setMiddleDown

setMousePosition

setRightDown

setShiftKeyDown

setX

setY

x

y

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 MoveEvent type includes several methods defined for the Event type. Refer to Event for more information.

The following built-in methods are triggered by MoveEvents: **arrive**, **canArrive**, **canDepart**, and **depart**.

getDestination

reason

setReason

StatusEvent Type

StatusEvent type methods control messages that appear in the Desktop status bar. The StatusEvent type includes several methods defined for the Event type. Refer to Event for more information.

Using StatusEvent type methods, you can attach code to the built-in 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 StatusWindows constants ModeWindow1, ModeWindow2, ModeWindow3, and StatusWindow to refer to the areas of the status bar.

reason

setReason

setStatusValue

statusValue

TimerEvent Type

Methods in the TimerEvent type process information used by the timer method built into every design object. The TimerEvent type includes several methods defined for the Event type. Refer to Event for more information.

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 *custRecord*.

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 **action** method for *custRecord* 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 *custRecord*, as follows:

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

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

```
; custRecord::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 *custRecord*.

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

See the killTimer, setTimer, and action methods from the UIObject type for more information.

ValueEvent Type

ValueEvent methods control field value changes. The **changeValue** built-in 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.

The ValueEvent type includes several methods defined for the Event type. Refer to Event for more information.

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.

newValue

setNewValue

SQL Type

An ObjectPAL SQL variable represents a 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 a SQL file, a SQL statement, or a string. Some methods for working with SQL are defined for the Database type, because in certain applications you will need to specify the database that contains the tables to query.

executeSQL

executeSQLFile

executeSQLString

writeSQL

beginTransaction

Method Starts a transaction.

Type Database

Syntax **beginTransaction**() Logical

Description Starts a transaction using the highest (most isolated) isolation level on a server when transactions are supported. Only one transaction is allowed for each database. Returns True if successful; otherwise 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.

Example This 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
      myQBE                Query
      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

   enumAliasNames("aliasNam.db")   ; list aliases known to the
system

   aliasNamTC.open("aliasNam.db")
   if aliasNamTC.locate("DBName", myAlias) then
      db.open(myAlias)      ; use alias to get database handle
to server
   else
      msgStop("Problem",
              "The alias " + myAlias + " has not been
defined.")
      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
```

See also commitTransaction

rollbackTransaction

transactionActive

close

Method Closes a database.

Type 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.

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
```

See Also open

commitTransaction

Method	Commits all changes within a transaction.
Type	Database
Syntax	**commitTransaction**() Logical
Description	Commits all changes made within a transaction on a server when transactions are supported. 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.
Example	See the example for beginTransaction.
See also	beginTransaction rollbackTransaction transactionActive

delete

Method/ Procedure	Deletes a table from a database.
Type	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.
Example	In this 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"
myDb.open("megadata")
if isTable(tableName) then
  myDb.delete(tableName, "dBASE") ; removes OldTable.dbf from
megadata
endif
endmethod
```

executeQBE

Method/ Procedure Executes a QBE query.

Type Database

Syntax

1. executeQBE (const ***qbeVar*** Query) Logical

2. executeQBE (const ***qbeVar*** Query, const ***ansTbl*** String) Logical

3. executeQBE (const ***qbeVar*** Query, const ***ansTbl*** Table) Logical

4. executeQBE (const ***qbeVar*** Query, var ***ansTbl*** TCursor) Logical

Description

executeQBE executes a query created in an ObjectPAL method and writes the results to *ansTbl*. In syntax 1, where *ansTbl* is not specified, **executeQBE** writes to ANSWER.DB in the user's private directory. In syntax 2, you specify the answer table as a string; if you don't include a file extension, *ansTbl* is a Paradox table by default. In syntax 3, where *ansTbl* is a Table variable, the Table variable must be assigned and valid. If you use syntax 4 to write query results to a TCursor, the results are stored in system 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.

A query in a method 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**. Double backslashes are not required to specify a path.

Because this kind of query is not a quoted string, it can contain tilde variables. (Compare this method with **executeQBEString**). You can use absolute paths or aliases in the query definition.

Example

The code in this example modifies the built-in pushButton method for *findName*. When the button is pushed, the method defines a Query variable, then uses **executeQBE** to execute the query and store the results in the CUSTNAME.DB table.

```
; findName::pushButton
method pushButton(var eventInfo Event)
var
  qq Query
  cName String
  tv TableView
endVar

cName = "Unisco"
qq = Query
     c:\pdoxwin\sample\customer | Customer No | Name         |
                                | Check       | Check ~cName |

     EndQuery
executeQBE(qq, "CustName.db") ; put results into Custname.db
tv.open("CustName")           ; view the table
endmethod
```

The next example adds an alias, uses the alias to open a database, then executes a query on a table in that database:

```
; thisButton
method pushButton(var eventInfo Event)
```

```
var
  db Database
  qq Query
  tc TCursor
  tv TableView
endVar

qq = Query
     contacts.db | Last Name  | First Name | Company | Phone
|
                 | Check      | Check      | Check   | Check
808..  |
     EndQuery

; create the sampData alias then open it
addAlias("sampData", "Standard", "C:\\pdoxwin\\sample")
db.open("sampData")

; now query CONTACTS.DB in the sampData database and write
; results to PHONE.DBF in the default database
db.executeQBE(qq, ":work:phone.dbf")
tv.open("phone.dbf")     ; open the table in the default
database
endmethod
```

See Also executeQBEFile
executeQBEString

executeQBEFile

Method/ Procedure Opens and executes a QBE file.

Type Database

Syntax

1. executeQBEFile (const ***qbeFileName*** String) Logical

2. executeQBEFile (const ***qbeFileName*** String, const ***ansTbl*** String) Logical

3. executeQBEFile (const ***qbeFileName*** String, const ***ansTbl*** Table) Logical

4. executeQBEFile (const ***qbeFileName*** String, var ***ansTbl*** TCursor) Logical

Description **executeQBEFile** opens *qbeFileName* (created with the **writeQBE** method or interactively using the Query Editor), executes it, and writes the results to the table specified in *ansTbl*. In syntax 1, where *ansTbl* is not specified, **executeQBEFile** writes to ANSWER.DB in the user's private directory. In syntax 2, you specify the answer table as a string; if you don't include a file extension, *ansTbl* is a Paradox table by default (use .DBF to write to a dBASE table). In syntax 3, where *ansTbl* is a Table variable, the Table variable must be assigned and valid. If you use syntax 4 to write query results to a TCursor, the results are stored in system memory only; a table is not created on disk.

If **executeQBEFile** is successful---if *ansTbl* or ANSWER.DB is created---this method returns True (even if the resulting table is empty); otherwise it returns False.

Example This code demonstrates how each form of the syntax works:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  db Database
  tb Table
  tc TCursor
endVar
addAlias("myData",  "Standard", "c:\\PdoxWin\\MyTables")
db.open("myData")
tb.attach("Cust1.db", db)

; this writes results into :priv:ANSWER.DB
executeQBEFile("GetCust.qbe")

; this writes results into MyCust.db in myData database
executeQBEFile("GetCust.qbe", "MyCust.db")

; this writes results into Cust1.db in myData database
executeQBEFile("GetCust.qbe", tb)

; this writes results into the tc TCursor
executeQBEFile("GetCust.qbe", tc)

endmethod
```

See Also executeQBE
executeQBEString

executeQBEString

Method/ Procedure Executes a QBE string.

Type Database

Syntax

1. executeQBEString (const ***QBEString*** String) Logical

2. executeQBEString (const ***QBEString*** String, const ***ansTbl*** String) Logical

3. executeQBEString (const ***QBEString*** String, const ***ansTbl*** Table) Logical

4. executeQBEString (const ***QBEString*** String, var ***ansTbl*** TCursor) Logical

Description **executeQBEString** executes a QBE string, and writes the results to *ansTbl*. In syntax 1, where *ansTbl* is not specified, **executeQBEString** writes to ANSWER.DB in the user's private directory. In syntax 2, you specify the answer table as a string; if you don't include a file extension, *ansTbl* is a Paradox table by default. In syntax 3, where *ansTbl* is a Table variable, the Table variable must be assigned and valid. If you use syntax 4 to write query results to a TCursor, the results are stored in system 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.

A QBE string can be a combination of quoted strings and string variables.

executeQBEString is useful when you're building a QBE string from smaller strings. Double backslashes are required when specifying a path.

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 **executeQBE**.

Example For this example, the pushButton method for findName defines a query as a string value, then uses **executeQBEString** to execute the query.

```
; findName::pushButton
method pushButton(var eventInfo Event)
var
  db Database
  qs String
  tv TableView
  tc TCursor
  stkNo String
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

  if executeQBEString(qs) then       ; writes to :PRIV:ANSWER
    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
```

See Also executeQBE
executeQBEFile

getQueryRestartOptions

Method Returns a value representing the user's query restart option setting.

Type Database

Syntax **getQueryRestartOptions**( ) SmallInt

Description 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.

Example See the example for setQueryRestartOptions.

See also setQueryRestartOptions

isAssigned

Method Reports whether a Database variable has been assigned a value.

Type Database

Syntax **isAssigned** ( ) Logical

Description **isAssigned** returns True if the Database variable has been assigned a value; otherwise, it returns False.

Example For this 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, an 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
```

See Also isTable

isExecuteQBEFileLocal

Method Reports whether a QBE query will be executed locally or on a server.

Type Database

Syntax **isExecuteQBEFileLocal**(const ***fileName*** String) Logical

Description Returns True if the query stored in the file specified by *fileName* will be executed locally; otherwise, returns False. If a QBE query generates SQL code, the query executes on the server; otherwise the query executes locally. A query that executes on the server may take a long time, so it can be useful to know beforehand where it will execute.

Example The following code calls **isExecuteQBEFileLocal** to find out where a QBE query will execute. If the query will execute on the server, the code asks the user whether or not to continue.

```
method pushButton (var eventInfo Event)
   var
      qbeFileName,
      dlgTitleText,
      dlgBodyText,
      doQBE         String
   endVar

   qbeFileName  = ":PRIV:getCust.qbe"
   dlgTitleText = "Remote query"
   dlgBodyText  = "This query may take a long time. \n" +
                  "Do you want to continue?"

   if isExecuteQBEFileLocal(qbeFileName) then
      doQBE = msgQuestion(dlgTitleText, dlgBodyText)

      if doQBE = "Yes" then
         executeQBEFile(qbeFileName)
      else
         msgInfo(dlgTitleText, "Query canceled")
         return
      endIf

   endIf

endMethod
```

See also isExecuteQBELocal

isExecuteQBEStringLocal

isExecuteQBELocal

Method Reports whether a QBE query will be executed locally or on a server.

Type Database

Syntax **isExecuteQBELocal**(const ***qbeVar*** Query) Logical

Description Returns True if the query represented by *qbeVar* will be executed locally; otherwise, returns False. If a QBE query generates SQL code, the query executes on the server; otherwise the query executes locally. A query that executes on the server may take a long time, so it can be useful to know beforehand where it will execute.

Example The following code calls **isExecuteQBELocal** to find out where a QBE query will execute. If the query will execute on the server, the code asks the user whether or not to continue.

```
method pushButton (var eventInfo Event)
   var
      qbeVar        Query
      dlgTitleText,
      dlgBodyText,
      doQBE         String
   endVar

   dlgTitleText = "Remote query"
   dlgBodyText  = "This query may take a long time. \n" +
                  "Do you want to continue?"

   qbeVar = Query

            :WestData:orders.db  | CustName | Qty         |
                                 | Check    | Check > 10  |

            endQuery

   if isExecuteQBELocal(qbeVar) then
      doQBE = msgQuestion(dlgTitleText, dlgBodyText)

      if doQBE = "Yes" then
         executeQBE(qbeVar)
      else
         msgInfo(dlgTitleText, "Query canceled")
         return
      endIf

   endIf

endMethod
```

See also isExecuteQBEFileLocal
isExecuteQBEStringLocal

isExecuteQBEStringLocal

Method Reports whether a QBE query will be executed locally or on a server.

Type Database

Syntax **isExecuteQBEStringLocal**(const ***qbeString*** String) Logical

Description Returns True if the query represented by *qbeString* will be executed locally; otherwise, returns False. If a QBE query generates SQL code, the query executes on the server; otherwise the query executes locally. A query that executes on the server may take a long time, so it can be useful to know beforehand where it will execute.

Example The following code calls **isExecuteQBEStringLocal** to find out where a QBE query will execute. If the query will execute on the server, the code asks the user whether or not to continue.

```
method pushButton (var eventInfo Event)
   var
      qbeString,
      dlgTitleText,
      dlgBodyText,
      doQBE         String
   endVar

   dlgTitleText = "Remote query"
   dlgBodyText  = "This query may take a long time. \n" +
                  "Do you want to continue?"

   qbeString = "Query

               :WestData:orders.db  | CustName | Qty         |
                                    | Check    | Check > 10  |

               endQuery"

   if isExecuteQBEString(qbeString) then
      doQBE = msgQuestion(dlgTitleText, dlgBodyText)

      if doQBE = "Yes" then
         executeQBEString(qbeString)
      else
         msgInfo(dlgTitleText, "Query canceled")
         return
      endIf

   endIf

endMethod
```

See also isExecuteQBEFileLocal

isExecuteQBELocal

isTable

Method/ Procedure Reports whether a table exists in a database.

Type Database

Syntax

1. isTable (const ***tableName*** String [, const ***tableType*** String]) Logical

2. isTable (const ***tableVar*** Table) Logical

Description

isTable returns True if a specifed 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.

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
```

See Also isAssigned

open

Method/ Procedure Opens a database.

Type 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 respresents 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.

Example For this 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("current") ; 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
```

See Also close
Session

rollBackTransaction

Method	Rolls back (undoes) all changes within a transaction on a server when transactions are supported.
Type	Database
Syntax	**rollbackTransaction**() Logical
Description	Rolls back (undoes) the effects of all operations within a transaction. Returns True if successful; otherwise, returns False.
Example	See the example for beginTransaction.
See also	beginTransaction commitTransaction transactionActive

setQueryRestartOptions

Method Specifies what to do with the underlying tables while running a query.

Type Database

Syntax **setQueryRestartOptions**(const ***qryRestartType*** SmallInt) Logical

Description 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 QueryRestartOptionsconstants:

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.

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)
  if getQueryRestartOptions <> QueryRestart then
     setQueryRestartOptions(QueryRestart)
  endIf

  executeQBEFile("newcust.qbe")
endMethod
```

See also **getQueryRestartOptions**

transactionActive

Method	Reports whether a transaction is currently active in a specified database.
Type	Database
Syntax	**transactionActive**() Logical
Description	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.
Example	See the example for beginTransaction.
See also	beginTransaction commitTransaction rollbackTransaction

writeQBE

Method/ Procedure Writes a query statement or a query string to a file.

Type Database

Syntax

1. writeQBE (const ***qbeVar*** Query, const ***fileName*** String) Logical

2. writeQBE (const ***str*** String, const ***fileName*** String) Logical

Description **writeQBE** writes a previously defined query statement or query string to the file specified in *fileName*. If *fileName* exists, it is overwritten without asking for confirmation. **writeQBE** returns True if the write succeeds; otherwise it returns False.

Example For this example, assume a form has a button named *getNames*. When the form opens, this example determines whether the GETNAMES.QBE file exists in the current directory (or in the private directory). If the file does not exist, the form's built-in **open** method uses **writeQBE** to write a query string to GETNAMES.QBE. The built-in **pushButton** for *getNames* runs the query with **executeQBEFile**, then views the results in a TableView. The code immediately following is attached to the form's **open** method.

```
; thisForm::open
method open(var eventInfo Event)
Var
  qs String    ; a query string
endVar

if eventInfo.isPreFilter() then
  ;code here executes for each object in form
else
  ;code here executes just for form itself

  if not isfile("GetDest.qbe") then    ; if the query file
dosn't exist
                                       ; construct a query
string

    qs = "Query\n\n" +
         ":mastApp:Dest | Destination Name | Avg Temp (F)  |
\n" +
                        " |  Check | Check 70  | \n\n" +
         "EndQuery"

    ; write the query string to GetNames.qbe
    writeQBE(qs, "GetDest.qbe")
  endif

endif
endmethod
```

The following code is attached the built-in **pushButton** method for the *getNames* button. This code does not check whether GETNAMES.QBE exists because the form's **open** method ensures the file is available.

```
; getDest::pushButton
method pushButton(var eventInfo Event)
var
```

```
  tv TableView
endVar

; execute the query file and store results in MyDest.db
executeQBEFile("GetDest.qbe", "MyDest.db")
; display the table
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.

See Also

executeQBE
executeQBEFile
executeQBEString

executeQBE

Method Executes a QBE query.

Type Query

Syntax

1. executeQBE ( ) Logical

2. executeQBE (const ***ansTbl*** String) Logical

3. executeQBE (const ***ansTbl*** Table) Logical

4. executeQBE (var ***ansTbl*** TCursor) Logical

Description **executeQBE** executes a query created in an ObjectPAL method and writes the results to *ansTbl*. In syntax 1, where *ansTbl* is not specified, **executeQBE** writes to ANSWER.DB in the user's private directory. In syntax 2, specify the answer table as a string; if you don't include a file extension, *ansTbl* is a Paradox table by default. In syntax 3, where *ansTbl* is a Table variable, *ansTbl* must be assigned and valid. If you use syntax 4 to write query results to a TCursor, the results are stored in system 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.

A query in a method 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**.

Because this kind of query is not a quoted string, it can contain tilde variables. (Compare this method with **executeQBEString** in the Database type.) You can use absolute paths or aliases in the query definition.

Example For this 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.

```
; getReceivables::pushButton
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 a keyword, but since it's also the abbreviation for
; Oregon, 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(qVar, "myCust.db")   ; put results into myCust.db
tv.open("myCust.db")            ; view the table

endmethod
```

See Also

QUERY
writeQBE
Database::executeQBEFile
Database::executeQBEString

isAssigned

Method Reports whether a Query variable has an assigned value.

Type Query

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.

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.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  qVar Query
endVar

qVar = Query

     This is not a query

     endQuery

msgInfo("Assigned?", qVar.isAssigned())    ; displays True

endmethod
```

See Also query

query

Keyword Begins a query statement.

Type Query

Syntax **query**

tableName | *fieldName* | [*fieldName* |]*
| *criteria* | [*criteria* |]*
[*tableName* | *fieldName* | [*fieldName* |]*
| *criteria* | [*criteria* |]*] *

endQuery

Description **query** marks the beginning of a QBE statement. 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.

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 neccessary 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
```

Because this kind of query is not a quoted string, it can contain tilde variables. (Compare this kind of query with a query string in the Database type). You can use absolute paths or aliases in the query definition.

Example For this 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.

```
; getReceivables::pushButton
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 a keyword, but since it's also the abbreviation for
; Oregon, 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(qVar, "myCust.db")   ; put results into myCust.db
tv.open("myCust.db")            ; view the table

endmethod
```

See Also executeQBE

DataBase::executeQBEFile
DataBase::executeQBEString

writeQBE

Method Writes a query statement to a specified file.

Type Query

Syntax **writeQBE** (const ***fileName*** String) Logical

Description **writeQBE** writes a previously defined query statement to the file specified in *fileName*. If *fileName* exists, it is overwritten without asking for confirmation. **writeQBE** returns True if the write succeeds; otherwise it returns False.

Example In this 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 with **executeQBEFile**, 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*:

```
; pageOne::open
method open(var eventInfo Event)
Var
  qVar Query
endVar

; if the GetDest.qbe query file dosn'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
  writeQBE(qVar, "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.

```
; getDest::pushButton
method pushButton(var eventInfo Event)
var
  tv TableView
endVar

; execute the query file and store results in MyDest.db
executeQBEFile("GetDest.qbe", "MyDest.db")
; open the table
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.

See Also

executeQBE
Database::executeQBEFile
Database::executeQBEString

add

Beginner

Method/ Procedure Adds the data in one table to another table.

Type 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.

Key violations (including validity check violations), if any, are listed in KEYVIOLS.DB in the user's private directory. This method overwrites an existing KEYVIOLS.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.

```
myTable.add(yourTable, False, True) ; specifies update
myTable.add(yourTable)              ; specifies update
and append by default
```

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.

Example

For this 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"

if executeQBEFile("getCust.qbe") then   ; if the query
succeeds
  ansTbl.attach("Answer.db")

  ; attempt to add Answer.db records to Customer.db
```

```
  if isTable(destTbl) then
    if NOT ansTbl.add(destTbl) then
      msgStop("Error", "Can't write lock " + destTbl + "
table.")
    endif
  else
    msgStop("Error", "Can't find " + destTbl + ".")
  endif
else
  msgStop("Error", "Query failed.")
endIf

endmethod
```

See Also

copy
subtract

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.

Adds the data in one table to another table.

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

attach

Beginner

Method Associates a Table variable with a table on disk.

Type Table

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 data 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 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.

Example In this 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
```

See Also create

unAttach

cAverage

Beginner

Method/ Procedure Returns the average value of a field (column) in a table.

Type 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 numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method handles blank values as specified by 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.

Example

This 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
```

See Also

cCount
cMax
cMin
cSum
cStd

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.

Returns the average value of a field (column) in a table.

Syntax

1. **cAverage** (const ***tableName*** String, const ***fieldName*** String) Number

2. **cAverage** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cCount

Beginner

Method/ Procedure Returns the number of nonblank values in a field (column) of a table.

Type Table

Syntax

1. cCount (const ***fieldName*** String) Number

2. cCount (const ***fieldNum*** SmallInt) Number

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.

This method respects the limits of restricted views displayed in a linked table frame or multi-record object.

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.

Example

For this 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, numItems Number
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", "Average quantity: " +
            String(qvgQty, "\nbased on ", numItems, " items.")
  else
    msgStop("Stop", "Can't lock Lineitem table.")
  endif
else
  msgStop("Sorry", "Can't attach to Lineitem table.")
endif

endmethod
```

See Also

cAverage

cMax

cMin

cStd

cSum

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.

Returns the number of nonblank values in a field (column) of a table.

Syntax

1. **cCount** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cCount** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cMax

Beginner

Method/ Procedure Returns the maximum value of a field (column) in a table.

Type Table

Syntax

1. cMax (const ***fieldName*** String) Number

2. cMax (const ***fieldNum*** SmallInt) Number

Description

cMax returns the maximum value in the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **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.

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
```

See Also

cAverage
cMin
cStd
cSum
cCount

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.

Returns the maximum value of a field (column) in a table.

Syntax

1. **cMax** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cMax** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cMin

Beginner

Method/ Procedure Returns the minimum value in a field (column) of a table.

Type Table

Syntax

1. cMin (const ***fieldName*** String) Number

2. cMin (const ***fieldNum*** SmallInt) Number

Description

cMin returns the minimum value in the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **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 write lock on the table. If a lock cannot be placed, the method fails.

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
```

See Also

cAverage
cCount
cStd
cSum
cMax

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.

Returns the minimum value in a field (column) of a table.

Syntax

1. **cMin** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cMin** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cNpv

Beginner

Method/ Procedure Returns the net present value of a field (column), based on a specifed discount or interest rate.

Type Table

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 numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **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 write lock on the table. If a lock cannot be placed, the method fails.

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
```

See Also

cAverage
cMax
cMin
cSum
cVar

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.

Returns the net present value of a field (column), based on a specifed discount or interest rate.

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

compact

Beginner

Method Removes deleted records from a table.

Type 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 fix them. When *regIndex* is True, this method regenerates all maintained indexes associated with the table: indexes specified by **usesIndexes**, and the .MDX index whose name matches the table name. When *regIndex* is False, only maintained indexes are 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.

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.

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
```

See Also showDeleted
usesIndexes

copy

Beginner

Method/ Procedure Copies a table.

Type 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 (*destTable*). The source table and destination table can be different types of tables, but both tables must have compatible field types. **copy** fails if the source table and destination tables have incompatible field types.

If the destination table already exists, **copy** overwrites it without asking for confirmation. 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.

Example

For this 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
NewCust.db

endmethod
```

See Also

add

subtract

rename

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.

Copies a table.

Syntax

1. **copy** (const ***sourceTable*** String, const ***destTable*** String) Logical
2. **copy** (const ***sourceTable*** String, const ***destTable*** Table) Logical

create

Keyword Creates a table.

Type Table

Syntax

create "*tableName"* [**as** "*tableType"*] [**database** *db*]
[[**like** *likeObject*]
[**with** "*fieldName*" : "*type*" [, "*fieldName*" : "*type*"] *]
[**where** *fieldDesc* **is** "newName" [, *fieldDesc* **is** "newname"] *]
[**without** *fieldDesc* [, *fieldDesc*] *]
[**struct** *fieldStructTable*]
[**indexStruct** *indexStructTable*]
[**refIntStruct** *refIntStructTable*]
[**secStruct** *secStructTable*]
] *
[**key** *fieldDesc* [, *fieldDesc*] *]
endCreate

Description **create** creates a table, where

tableName is the file name of the table to create. For example:

"Orders.db"

If *tableName* exists, **create** tries, for the duration of the retry period, to place a full (exclusive) lock on *tableName*. If the lock cannot be placed, **create** fails.

as *tableType* specifies the table format. Example:

AS "Paradox"

If **as** is omitted, *tableType* is Paradox by default. **as** infers *tableType* from the *fileName* extension if given. (.DB is a Paradox table and .DBF is a dBASE table.)

database *db* is a database variable (opened before creating the new table) that specifies where the table will reside. If omitted, *tableName* 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 and field types. 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 "*fieldID*" : "*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 table lists valid field specifications for Paradox and dBASE tables.

Field type	Paradox table	dBASE table
Alphanumeric	A*nn*	(none)
Character	(none)	C*nn*
Date	D	D
Integer	S	N
Floating-point value	N	You cannot create a floating-point field with create.
Graphic	G	(none)
Logical	(none)	L
Money	$	(none)
Memo	M*nn*	M
Formatted memo	F*nn*	(none)
Binary	B*nn*	(none)
OLE object	O	(none)

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 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 the **enumFieldStruct** method 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 the **enumIndexStruct** method 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 the **enumRefIntStruct** method 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 the **enumSecStruct** method 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 to the *User's Guide* for information about master passwords

key *fieldID* specifies one or more key fields (Paradox tables only). 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()
```

Instead, use = to assign the **create** structure to a Table variable.

Example 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
```

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

  ; include from Customer field names, ValChecks,
  ; and key fields in new table CustFlds
  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
```

See Also

copy
enumFieldStruct
enumIndexStruct

enumRefIntStruct
enumSecStruct

cSamStd

Beginner

Method/ Procedure Returns the sample standard deviation of a field (column) of a table.

Type 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 write lock on the table. If a lock cannot be placed, the method fails.

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
  empTbl Table
  tblName String
  calcSalary, calcYears Number
endVar
tblName = "Answer"

empTbl.attach(tblName)
calcSalary = empTbl.cSamStd("Salary")  ; get sample std
deviation for Salaries
calcYears  = empTbl.cSamStd(2)         ; assume "Years in
service" is field 2
msgInfo("Sample Std Deviation",        ; display info in a
dialog box
        "Salaries : " + String(calcSalary,
        "\nYears in service : ", calcYears))

endmethod
```

See Also

cSamVar
cAverage
cCount
cMax

cMin
cNpv
cSum
cVar
cStd

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.

Returns the sample standard deviation of a field (column) of a table.

Syntax

1. cSamStd (const ***tableName*** String, const ***fieldName*** String) Number

2. cSamStd (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cSamVar

Beginner

Method/ Procedure Returns the sample variance of a field (column) in a table.

Type Table

Syntax

1. cSamVar (const ***fieldName*** String) Number

2. cSamVar (const ***fieldNum*** SmallInt) Number

Description

cSamVar returns the sample variance of the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **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 lock on the table. If a lock cannot be placed, the method fails.

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
```

See Also

cAverage
cCount
cMax
cMin
cNpv
cSamStd
cStd

cSum
cVar

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.

Returns the sample variance of a field (column) in a table.

Syntax

1. cSamVar (const ***tableName*** String, const ***fieldName*** String) Number

2. cSamVar (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cStd

Beginner

Method/ Procedure Returns the standard deviation of a field (column) in a table.

Type 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 numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. 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 write lock on the table. If a lock cannot be placed, the method fails.

Example

For this 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
```

See Also

cSamStd
cSamVar
cSum
cAverage
cCount
cMax
cMin
cNpv
cVar

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.

Returns the standard deviation of a field (column) in a table.

Syntax

1. cStd (const ***tableName*** String, const ***fieldName*** String) Number

2. cStd (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cSum

Beginner

Method/ Procedure Rturns the sum of the values in a field (column) of a table.

Type 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 numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **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 write lock on the table. If a lock cannot be placed, the method fails.

Example

For this 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
```

See Also

cAverage
cCount
cMax
cMin
cNpv
cSamStd
cSamVar
cStd
cVar

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.

Returns the sum of the values in a field (column) of a table.

Syntax

1. cSum (const ***tableName*** String, const ***fieldName*** String) Number

2. cSum (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cVar

Beginner

Method/ Procedure Returns the variance of a field in a table.

Type Table

Syntax

1. cVar (const ***fieldName*** String) Number

2. cVar (const ***fieldNum*** SmallInt) Number

Description

cVar returns the population variance of the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **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 write lock on the table. If a lock cannot be placed, the method fails.

Example

For this 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
```

See Also

cAverage
cCount
cMax
cMin
cNpv
cSamVar
cSamStd
cSum

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.

Returns the variance of a field in a table.

Syntax

1. cVar (const ***tableName*** String, const ***fieldName*** String) Number

2. cVar (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

delete

Beginner

Method/ Procedure Deletes a table.

Type 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.

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
```

See Also empty

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.

Deletes a table.

Syntax **delete** (const ***tableName*** String) Logical

dropIndex

Method Deletes an index file associated with a table.

Type 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 cannot 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.

You must obtain exclusive rights to the table (by calling the Table method **setExclusive** before opening the table) before calling **dropIndex**.

dropIndex fails if the index you're trying to delete is currently being used, or if the table is open.

Example For this 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
```

See Also setIndex
usesIndexes

empty

Beginner

Method/ Procedure Removes all records from a table in a database.

Type Table

Syntax **empty** ( ) Logical

Description **empty** removes all records from a table without asking for confirmation. This operation cannot be undone. **empty** fails if the table is open.

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 expensive 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.)

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"

tblVar.attach(tblName)
if isTable(tblName) then
  if msgYesNoCancel("Confirm", "Empty " + tblName + " table?")
= "Yes" then
    if tblVar.empty() then
      message("All " + tblName + " records have been
deleted.")
    else
      message("Empty failed." +
        tblname + " has " + String(tblVar.nRecords()) + "
records.")
    endif
  endif
else
  msgInfo("Error", "Can't find " + tblName + " table.")
endif
endmethod
```

See Also delete

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.

Removes all records from a table in a database.

Syntax **empty** (const ***tableName*** String) Logical

enumFieldNames

Method Fills an array with the names of fields in a table.

Type 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.

Example For this 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.

```
; showIndexFlds::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  fieldNames Array[] AnyType
endVar

tbl.attach("Sales.dbf")
if tbl.isTable() then
  tbl.enumFieldNamesInIndex("DateIndx", "byDate", fieldNames)
  fieldNames.view()
else
  msgStop("Stop", "Couldn't find Sales.dbf.")
endIf

endmethod
```

See Also enumFieldNamesInIndex
enumFieldStruct
enumIndexStruct
enumRefIntStruct
enumSecStruct

enumFieldNamesInIndex

Method Fills an array with the names of fields in a table's index.

Type Table

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.

Example For this 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
```

See Also enumFieldNames
enumFieldStruct
enumIndexStruct
enumRefIntStruct
enumSecStruct

enumFieldStruct

Method Creates a Paradox table listing the field structure of a table.

Type Table

Syntax **enumFieldStruct** (const ***tableName*** String) Logical

Description **enumFieldStruct** creates the Paradox table specified in *tableName* listing the structure of a Table. 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 **struct** option in a **create** statement to borrow a table's field structure (including primary keys and validity checks) for use in the new table. The structure for *tableName* is listed in the following table.

Field Name	Field Type
Field Name	A31
Type	A31
Size	S
Dec	S
Key	A1
_Required Value	A1
_Min Value	A255
_Max Value	A255
_Default Value	A255
_Picture Value	A175
_Table Lookup	A81
_Table Lookup Type	A1
_Invariant Field ID	S

Once *tableName* is created, you can modify values in the table, then use it with the **struct** option in the **create** command.

Example For this 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

  ; include from Customer field names, ValChecks,
```

```
  ; and key fields in new table CustFlds
  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
```

See Also

enumFieldNames
enumFieldNamesInIndex
enumIndexStruct
enumRefIntStruct
enumSecStruct

enumIndexStruct

Method Creates a Paradox table listing the structure of a table's secondary indexes.

Type Table

Syntax **enumIndexStruct** (const ***tableName*** String) Logical

Description **enumIndexStruct** creates the Paradox table specified in *tableName* listing the structure of a table's secondary indexes. 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 **indexStruct** option in a **create** statement to borrow secondary indexes for use in the new table.

The structure of *tableName* is listed in the following table.

Field Name	Field Type
infoHeader	A1
szName	A127
szTagName	A31
szFormat	A31
bPrimary	A1
bUnique	A1
bDescending	A1
bMaintained	A1
bCaseInsensitive	A1
bSubset	A1
bExpIdx	A1
bKeyExpType	N
szKeyExp	A220
szKeyCond	A220
FieldNo	N
FieldName	A31

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.

Example For this 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
```

See Also enumFieldNames
enumFieldNamesInIndex
enumFieldStruct
enumRefIntStruct
enumSecStruct

enumRefIntStruct

Method Creates a Paradox table listing a table's referential integrity information.

Type Table

Syntax **enumRefIntStruct** (const *tableName* String) Logical

Description **enumRefIntStruct** writes the referential integrity information for the current table to *tableName*. If *tableName* exists, this method overwrites it without 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. The structure for *tableName* is listed in the following table.

Field Name	Type	Size
infoHeader	A	1
Ref Name	A	31
Other Table	A	81
Slave	A	1
Modify	A	1
Delete	A	1
FieldNo	N	
aiThisTabField	A	31
Other FieldNo	N	
aiOthTabField	A	31

Example This 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")    ; write ref. integ. info
to CustRef
tb1.enumFieldStruct("CustFlds.db")    ; write field structure
to CustFlds

tb2 = CREATE "NewCust.db"
        STRUCT "CustFlds.db"          ; use field-level info.
from CustFlds
        REFINTSTRUCT "CustRef.db"     ; use ref. integ. infor
from CustRef
        KEY "Customer No", "Order No" ; key NewCust on these
fields
      ENDCREATE

endmethod
```

See Also

enumFieldNames
enumFieldNamesInIndex
enumFieldStruct
enumIndexStruct
enumSecStruct

enumSecStruct

Method Creates a Paradox table listing a table's security information.

Type Table

Syntax **enumSecStruct** (const ***tableName*** String) Logical

Description **enumSecStruct** creates the Paradox table specified in *tableName* listing security information (access rights) of a table. 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. The structure of *tableName* is listed in the following table.

Field Name	Type	Size
infoHeader	A	1
iSecNum	N	
eprvTable	N	
eprvTableSym	A	10
iFamRights	N	
iFamRightsSym	A	10
szPassword	A	31
fldNum	N	
aprvFld	N	
aprvFldSym	A	10

Example This 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 "Secrets.db"
      ENDCREATE

endmethod
```

See Also enumFieldNames
enumFieldNamesInIndex
enumFieldStruct
enumIndexStruct

enumRefIntStruct

familyRights

Method Tests for a user's ability to create or modify objects in a table's family.

Type Table

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.

Example This 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
```

See Also tableRights

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.

Tests for a user's abitlity to create or modify object's in a table's family.

Syntax **familyRights**(const ***tableName*** String, ***rights*** AnyType) Logical

fieldName

Method/ Procedure Returns the name of field in a table, given a field number.

Type Table

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.

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
```

See Also fieldNo

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.

Returns the name of field in a table, given a field number.

Syntax **fieldName** (const ***tableName*** String, const ***fieldNum*** SmallInt) String

fieldNo

Method/ Procedure Returns the position of a field in a table.

Type 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.

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
```

See Also fieldName

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.

Returns the position of a field in a table.

Syntax **fieldNo** (const ***tableName*** String, const ***fieldName*** String) SmallInt

fieldType

Method/ Procedure Returns the type of a field in a table.

Type 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 the "Unknown". The following table list the possible return values:

Field type	Paradox table	dBASE table
Alphanumeric	ALPHA	(none)
Character	(none)	CHARACTER
Date	DATE	DATE
Integer	SHORT	(none)
Floating-point value	NUMERIC	FLOAT (IV)
		NUMERIC (III+ or IV)
Graphic	GRAPHIC	(none)
Logical	(none)	BOOLEAN
Money	MONEY	(none)
Memo	MEMO	MEMO
Formatted memo	FMTMEMO	(none)
Binary	BINARYBLOB	(none)
OLE object	OLEOBJ	(none)

Example This 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
```

See Also fieldNo

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.

Returns the type of a field in a table.

Syntax

1. fieldType (const ***tableName*** String, const ***fieldName*** String) String

2. fieldType (const ***tableName*** String, const ***fieldNum*** SmallInt) String

index

Keyword Creates a primary or secondary index on the specified fields of a table.

Type Table

Syntax

1. index

[**maintained**] *tableDesc* **on** *fieldID*

endIndex

2. index *tableDesc*

[**maintained**] (Paradox)

[**descending**] (dBASE)

[**unique**] (dBASE)

[**primary**] (Paradox)

[**caseInsensitive**] (Paradox)

on

{ *fieldDesc* [, *fieldDesc*] [**to** *indexName*]

|

{ *keyExp*

to *ndxFileName* | **tag** *tagName* [**of** *mdxFileName*]

|

for *condition* } }

endIndex

Description If you are building an index on a keyed table, the **maintained** keyword specifies to Paradox that you want the index you are creating to be incrementally maintained. Incremental maintenance means that after changes made to a table are saved, only that portion of the index affected by the change will be updated. Tables that are keyed and have incrementally maintained secondary indexes typically result in far better performance than those that are not set up this way, although there is a hidden performance cost because maintaining the index takes resources.

If you use the **maintained** keyword for Paradox tables and specify a non-keyed table to index, **index** fails. Without the **maintained** keyword, a secondary index is not maintained automatically. Instead, you must use **reIndex** or **reIndexAll**. For dBASE tables, all opened index files are automatically maintained.

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 index file to write to. For Paradox tables, secondary index files have extensions .X*nn* and .Y*nn*, where *nn* refers to the structural position of the field being indexed, expressed in hexadecimal notation.

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.

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.

Because keyed tables have primary indexes on their key fields, you don't have to create a secondary index for these fields; doing so would only duplicate the primary index. However, you can use INDEX to create a primary index. Memo fields, OLE, and Graphic fields cannot be indexed.

It's 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.

Example

The following example builds a primary index for a Paradox table named CUSTOMER.DB. If the Customer table can not be found, or can not 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
```

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
```

See Also

create
enumIndexStruct
setIndex
usesIndexes

isAssigned

Method Reports whether a Table variable has been assigned a value.

Type Table

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 variable is attached to a table, or, if attached, 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.
```

Example This 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
```

See Also attach

isTable

isEmpty

Method/ Procedure Reports whether a table contains any records. This can be an expensive operation for dBASE tables.

Type Table

Syntax **isEmpty** ( ) Logical

Description **isEmpty** returns True if there are no records in a table; otherwise, it returns False.

Example For this 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
```

See Also empty
isTable

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.

Reports whether a table contains any records.

Syntax **isEmpty** (const ***tableName*** String) Logical

isEncrypted

Method/ Procedure Reports whether a table is encrypted.

Type Table

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 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.

Example This example uses **isEncrypted** to determine whether the *Secrets* table is protected (encrypted); if it is the code calls **unProtect** to permanently remove password-protection from the table.

```
; thisForm::open
method open(var eventInfo Event)
var
  tblVar TCursor
  tblName String
endVar
if eventInfo.isPreFilter() then
  ;code here executes for each object in form
else
  ;code here executes just for form itself

  tblName = "Secrets.db"
  if isEncrypted(tblName) then   ; if table is protected
    getPassword(tblName)         ; run a custom method and
                                 ; verify that password was
effective
    if isEncrypted(tblName) then ; if table is still protected
      close()                    ; close this form
    endif
  endif
   ; at this point a valid password has been issued,
   ; so go ahead and open the TCursor
  tblVar.open(tblName)

endif
endmethod
```

See Also protect
tableRights
Session::addPassword
Session::removePassword

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.

Reports whether a table is encrypted.

Syntax **isEncrypted** (const ***tableName*** String) Logical

isShared

Method/ Procedure Reports whether a table is currently shared.

Type Table

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.

Example In this 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
```

See Also tableRights

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.

Reports whether a table is being shared by another user.

Syntax **isShared** (const ***tableName*** String) Logical

isTable

Method/ Procedure Reports whether a table exists in a database.

Type Table

Syntax **isTable** ( ) Logical

Description **isTable** returns True if the Table variable represents a table that can be opened; otherwise, it returns False.

Example This 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
```

See Also isAssigned

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.

Reports whether a table exists in a database.

Syntax **isTable** (const ***tableName*** String) Logical

lock

Beginner

Method Locks a specified table.

Type 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: Write, Read, Full, or Exclusive. If successful, this method returns True; otherwise, it returns False.

If a TCursor is opened onto a dBASE table, a Full or Exclusive lock cannot be obtained.

Example The following example attaches a Table variable to *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
  tblVar Table
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

if isTable(pdoxTbl) then
  tblVar.attach(pdoxTbl)
  if tblVar.lock("Full") then    ; attempt to gain exclusive
access
    tblVar.reIndex("Phone_Zip")  ; rebuild Phone_Zip index
    tblVar.unLock("Full")        ; unlock the table
  else
    msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
  endif
else
  msgStop("Sorry", "Can't find " + pdoxTbl + " table.")
endif
endmethod
```

See Also unlock

Session::lock

Session::unlock

nFields

Method/ Procedure Returns the number of fields in a table.

Type Table

Syntax **nFields** ( ) LongInt

Description **nFields** returns the number of fields in a table.

Example For this 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
```

See Also nKeyFields
nRecords

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.

Returns the number of fields in a table.

Syntax **nFields** (const ***tableName*** String) LongInt

nKeyFields

Method/ Procedure Returns the number of fields in the primary or current index for a table.

Type Table

Syntax **nKeyFields** ( ) LongInt

Description **nKeyFields** returns the number of fields in the current index for a table. When used on a Paradox table, this method works with the primary index; when used on a dBASE table, it works with the index specified by the **setIndex** method.

Example This example reports the number of primary key fields in a Paradox table (ORDERS.DB) and the number of primary key fields in the LASTNAME.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("LastName.MDX") ; number of key fields in
LastName.MDX index
nkf = dBaseTbl.nKeyFields()
msgInfo("Scores.dbf", "Scores.dbf has " + String(nkf) + " key
fields.")

endmethod
```

See Also nFields

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.

Returns the number of key fields in the table.

Syntax **nKeyFields** (const ***tableName*** String) LongInt

nRecords

Method/ Procedure Returns the number of records in a table.

Type Table

Syntax **nRecords** ( ) LongInt

Description **nRecords** returns the number of records in a table. For dBASE tables, this may be an expensive operation.

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
```

See Also nFields
nKeyFields

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.

Returns the number of records in a table.

Syntax **nRecords** (const ***tableName*** String) LongInt

protect

Method/ Procedure Encrypts and assigns an owner password to a table.

Type Table

Syntax **protect** (const ***password*** String) Logical

Description **protect** assigns an owner password to a table. 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.

Example For this 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
```

See Also isEncrypted

Session::addPassword

Session::removePassword

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.

Encrypts and assigns an owner password to a table.

Syntax **protect** (const ***tableName*** String, const ***password*** String) Logical

reIndex

Method Rebuilds specified index files.

Type Table

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.

Example The following example attaches a Table variable to Customer (a Paradox table), places a full 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("Full") then    ; attempt to gain exclusive
access
  tblVar.reIndex("Phone_Zip")  ; rebuild Phone_Zip index
  tblVar.unLock("Full")        ; unlock the table
else
  msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
endif

endmethod
```

See Also reIndexAll

reIndexAll

Method Rebuilds all index files associated with a table.

Type 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.

Example For this example, the **pushButton** method for a button attempts to place a full 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("Full") then    ; attempt to lock Customer.db
  tblVar.reIndexAll()          ; rebuild all Customer.db
indexes
  tblVar.unLock("Full")        ; unlock the table
else
  msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
endif

endmethod
```

See Also reIndex

rename

Beginner

Method/ Procedure Renames a table.

Type 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.

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
```

See Also copy

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.

Renames a table.

Syntax **rename** (const ***tableName*** String, const ***destTableName*** String) Logical

setExclusive

Method Specifies whether to give the user exclusive rights to a table when it is opened.

Type Table

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 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, a value of No allows the table to be opened in shared mode. If omitted, *yesNo* defaults to Yes.

Example This 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 sucessfully 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
```

See Also

attach
setIndex
setReadOnly

setFilter

Method Specifies a range of records to include.

Type Table

Syntax **setFilter** ([const ***exactMatchVal*** AnyType,] * const ***minVal*** AnyType, const ***maxVal*** AnyType) Logical

Description **setFilter** specifies conditions for including a range of records. Records that meet the conditions are included when the table is opened, records that don't are filtered out.

This method compares the criteria you specify with values in the corresponding fields of a table's index; **setFilter** fails if the table is not indexed. To filter records based on the value of a single field, specify values in *minVal* and *maxVal.* For example, the following statement checks values in the first field of the index of each record.

```
tableVar.setFilter(14, 88)
```

If a value is less than 14 or greater than 88, that record is filtered out. To specify an exact match on a single field, assign *minVal* and *maxVal* the same value. For example, the following statement filters out all values except 55.

```
tableVar.setFilter(55, 55)
```

You can filter records 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're the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field.

```
tcVar.setFilter("Borland", "Paradox", 100, 500)
```

Calling **setFilter** without any arguments resets the filter to include the entire table.

Example In this 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.setFilter(1005, 1005)

  ; now display the total for Order No. 1005
msgInfo("Total for Order 1005", tblVar.cSum("Total"))

endmethod
```

See Also setIndex

setIndex

Method Specifies an index for a table.

Type 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.

Example In this 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 **setFilter**. 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.setFilter("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
```

See Also setFilter

setReadOnly

Method Specifies whether to give the user read-only rights to a table when it is opened.

Type Table

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 allows full rights to the table. If omitted, *yesNo* is Yes by default.

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

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

endmethod
```

See Also setExclusive

showDeleted

Method Specifies whether to display deleted records in a dBASE table.

Type 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.

Example For this 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
```

See Also compact
delete

sort

Keyword Sorts a table.

Type 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.

Example This example sorts *Customer* on the Last Name and First Name fields, and places the results in in the *CustSort* table.

```
; sortCustTable::pushButtton
method pushButton(var eventInfo Event)
var
  custTbl Table
  tv TableView
endVar

custTbl.attach("Customer.db")

sort custTbl
  on "Last Name" D, "First Name" D     ; sort in descending
order
  to "CustSort.db"
endSort

tv.open("CustSort.db")                 ; open the sorted table

endmethod
```

See Also index

TCursor::sortTo

subtract

Beginner

Method/ Procedure Subtracts the records in one table from another table.

Type 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 *destTableName* is not keyed, **subtract** deletes all records that exactly match any record in the source table. If *destTableName* is keyed, **subtract** deletes all records with keys that exactly match values in corresponding key fields in the source table. Whether *destTableName* is keyed or not, this method considers only fields that *could* be keyed. For example, numeric fields are considered, but formatted memos are not.

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.

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
```

See Also add

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.

Subtracts the records in one table from another table.

Syntax

1. subtract (const ***sourceTableName*** String, const ***destTableName*** String) Logical

2. subtract (const ***sourceTableName*** String, const ***destTableName*** Table) Logical

tableRights

Method/ Procedure Specifies whether the user has rights to perform certain operations on a table.

Type Table

Syntax **tableRights** (const ***rights*** String) Logical

Description **tableRights** reports about a user's rights to a table, where *rights* is one of:

"ReadOnly" (read from the table, but not change it)

"Modify" (enter or change data)

"Insert" (add new records)

"InsDel" (add and delete records)

"All" (perform all operations)

Example This 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
```

See Also familyRights

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.

Tells whether the user has rights to perform certain operations on a table.

Syntax **tableRights** (const ***tableName*** String, const ***rights*** String)

type

Method Returns the type of a table.

Type Table

Syntax **type** ( ) String

Description **type** returns the string value "PARADOX" or "DBASE" to report the type of a table.

For this example, assume a form has a button named *compactButton* and a table frame bound to the *Orders* table. The following code (attached to the **pushButton** method) compacts (removes deleted records) from the table if **type** returns DBASE; otherwise a message informs the user that *Orders* is a Paradox table.

```
; 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
```

See Also attach

unAttach

Method Ends the association between a Table variable and a table on disk.

Type Table

Syntax **unAttach** ( ) Logical

Description **unAttach** ends the association (created using **attach**) between a Table variable and a table in a file. 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.

Example In this 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

if q1 = q2 then
  msg = "Sales must increase."
else
  msg = "Sales are up."
endIf
msgInfo("Sales", msg)

endmethod
```

See Also attach

unlock

Beginner

Method Unlocks a specified table.

Type Table

Syntax **unlock** (const ***lockType*** String) Logical

Description **unlock** attempts to remove locks explicitly placed on a table. *lockType* must be an expression that evaluates to one of the following string values: Write, Read, Full, or Exclusive. If successful, this method returns True; otherwise, it returns False.

Example For this 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"

if executeQBEFile("getCust.qbe") then  ; if the query succeeds
      ansTbl.attach("Answer.db")
  if destTbl.lock("Write") then        ; attempt to write lock
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
```

See Also lock

unProtect

Method/ Procedure Decrypts and removes an owner password from a table.

Type 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.

Example This 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
```

See Also isEncrypted
protect
Session::addPassword
Session::removePassword

usesIndexes

Method Specifies index files to use and maintain with a dBASE table.

Type 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. You don't need to use this method to open production file (such as .MDX files) for a dBASE table---these files are automatically opened.

This method fails if all of the specified index files do not exist.

Example This 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
```

See Also reIndex
reIndexAll
setIndex

add

Method Adds the records of one table to another.

Type TCursor

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 KEYVIOLS.DB in the user's private directory. This method overwrites an existing KEYVIOLS.DB or creates one, if necessary. **add** does not respect the limits of restricted views displayed in a linked table frame or multi-record object.

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.

Example

In this 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 multi-
record object

   sTC.close()                  ; close both TCursors
   dTC.close()
else
   msgStop("Stop!", "Could not open one or more tables.")
endif

endmethod
```

See Also

copy
subtract

atFirst

Method Reports whether the TCursor is pointing to the first record of a table.

Type TCursor

Syntax **atFirst** ( ) Logical

Description **atFirst** returns True if the TCursor is pointing to the first record of a table; otherwise, it returns False.

Example This 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
```

See Also atLast
bot
eot

atLast

Method Reports whether the TCursor is pointing to the last record of a table.

Type TCursor

Syntax **atLast** ( ) Logical

Description **atLast** returns True if the TCursor is pointing to the the last record of a table; otherwise, it returns False.

Example This 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
```

See Also atFirst
bot
eot

attach

Method Binds a TCursor to a UIObject.

Type TCursor

Syntax

1. attach (const ***object*** UIObject) Logical

2. attach (const srcTCursor TCursor) Logical

3. attach (const ***tv*** TableView) Logical

Description **attach** binds a TCursor to a UIObject (*object* in syntax 1), another TCursor (*srcTCursor* in syntax 2) or a TableView object (*tv* in syntax 3). The data 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 added). **attach** returns True if successful; otherwise, it returns False.

Example In this 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  Number
  orderTC   TCursor
  OrderNum  Number
endVar

stockNum = LINEITEM."Stock No" ; get Stock No from current
LineItem
; 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
```

See Also isValid

attachToKeyViol

Method Attaches a TCursor to the existing record that has the same key as the record you attempted to post.

Type TCursor

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.

Example This 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
```

See Also postRecord
updateRecord

bot

Method Tests for a move past the beginning of a table.

Type TCursor

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.

Example This 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
```

See Also atFirst
atLast
end
eot
home

cancelEdit

Beginner

Method Ends Edit mode without saving changes to the current record.

Type TCursor

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.

Example The code for this 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")                      ; open TCursor for
Customer
if tc.locate("Customer No", 1231) then      ; if 1231 exists in
Customer No field
  tc.edit()                                 ; go into edit mode
  tc."Customer No" = 1221                   ; attempt to change
key value
  if not tc.endEdit() then                  ; if endEdit fails
    msgStop("Error", "Can't complete operation.")
    tc.cancelEdit()                         ; restore record
and leave edit mode
    message("Record left intact.")
  else
    message("Key value changed.")
  endif
else
  msgStop("Error", "Can't find Customer 1231")
endif

endmethod
```

See Also edit
endEdit

cAverage

Method Returns the average value of a field (column) in a table.

Type TCursor

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 displayed in a linked table frame or multi-record object. **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.

Example This 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, avgOrders 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
```

See Also cCount
cSum
cMax
cMin
cStd
Session::blankAsZero

cCount

Method Returns the number of values in a field (column) of a table.

Type TCursor

Syntax **1. cCount** (const ***fieldName*** String) Number

2. cCount (const ***fieldNum*** SmallInt) Number

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.

This method respects the limits of restricted views displayed in a linked table frame or multi-record object.

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.

cCount is useful for returning the number of entries used by another column function.

Example This 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, numRecs Number
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
```

See Also cAverage
cSum
cMax
cMin
cStd
Session::blankAsZero

close

Beginner

Method Closes a table.

Type TCursor

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.

Example This 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
```

See Also isAssigned

open

cMax

Method Returns the maximum value of a field (column) in a table.

Type TCursor

Syntax **1. cMax** (const ***fieldName*** String) Number

2. cMax (const ***fieldNum*** SmallInt) Number

Description **cMax** returns the maximum value in the column of numeric fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **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.

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"))
; syncronize the table frame to the TCursor
ORDERS.moveToRecord(ordTC)

endmethod
```

See Also cAverage
cMin
cSum
cCount
cStd
Session::blankAsZero

cMin

Method Returns the minimum value in a field (column) of a table.

Type TCursor

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*. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **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 write lock on the table. If a lock cannot be placed, the method fails.

Example This 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
```

See Also cMax
cAverage
cCount
cStd
cSum
Session::blankAsZero

cNpv

Method Returns the net present value of a field (column), based on a specified discount or interest rate.

Type TCursor

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 nonblank 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 lock on the table. If a lock cannot be placed, the method fails.

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.

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
```

See Also cAverage

cSum

compact

Method Removes deleted records from a dBASE table.

Type TCursor

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 fix them. When *regIndex* is True, this method regenerates all maintained indexes associated with the TCursor and frees any unused space in the indexes. 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.

Example The following example inspects each record in a table and deleted records where the value of the Date field is more than 30 days old. It compacts the table after appropriate records have been deleted. This code is attached the **pushButton** method for the *purgeTable* button.

```
; purgeTable::pushButton
method pushButton(var eventInfo Event)
var
  tb Table
  tc TCursor
  recsDeleted SmallInt
endVar
tb.attach("OldData.dbf")
tb.setExclusive()              ; get exclusive rights to table
tc.open(tb)                    ; associate TCursor with
OldData table
tc.edit()
scan tc for tc.Date  (today() - 30) :
  tc.deleteRecord()            ; delete old records
endScan
tc.compact()                   ; remove all deleted records
tc.close()                     ; close TCursor
message("Old records purged.")
endmethod
```

See Also deleteRecord

showDeleted

Table::compact

copy

Method Copies a table.

Type TCursor

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 (exclusive) 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 displayed in a linked table frame or multi-record object.

Example This 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
    sourceTC.copy(destTb)
  endIf
endIf

endmethod
```

See Also add

copyRecord

copyFromArray

Method Copies data from an array to the fields of the current record.

Type TCursor

Syntax **1. copyFromArray** (const ***ar*** Array[] AnyTpe) Logical

2. copyFromArray (const ***ar*** DynArray[] AnyType) Logical

Description **copyFromArray** copies the elements of the array *ar* to the record pointed to by a TCursor, which must be in Edit mode. 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.

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**.

Example In this 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
```

See Also copyRecord
copyToArray
insertRecord
insertAfterRecord
insertBeforeRecord

copyRecord

Method Copies a record pointed to by one TCursor into the record pointed to by another TCursor.

Type TCursor

Syntax **copyRecord** (const ***pointer*** 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 the record pointed to by the *copyFromThisTC* TCursor into the record pointed to by the *pointerTC* TCursor:

```
copyToThisTC.copyRecord(pointerTC)
```

The TCursor specified in *pointer* does not have to be in Edit mode; the TCursor you're copying to does.

You can not use **copyRecord** to copy a record into an empty table. To copy a new record into an empty table, use **insertRecord**.

Example This 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
  ui TableView
endVar

ordTC.open("Orders.db")
newOrdTC.open("NewOrdrs.db")
newOrdTC.edit()                ; copyRecord only works in Edit
mode

; scan Orders.db table for records dated ten days ago
scan ordTC for ordTC."Sale Date" >= today() - 10:
  newOrdTC.insertRecord()      ; insert a new record in
NewOrdrs.db
  newOrdTC.copyRecord(ordTC)   ; copy record from Orders.db
into NewOrdrs.db
endScan
newOrdTC.endEdit()             ; end Edit mode for TCursor

ui.open("NewOrdrs.db")
endmethod
```

See Also copyToArray
insertAfterRecord
insertBeforeRecord
insertRecord

copyToArray

Method Copies the fields of the current record to an array.

Type TCursor

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.

Example In this 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
```

See Also copyFromArray

cSamStd

Method Returns the sample standard deviation of a field (column) of a table.

Type TCursor

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 displayed in a linked table frame or multi-record object. 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 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*)

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
```

See Also cAverage
cCount
cMax
cMin
cNpv
cSamVar

cStd
cSum
cVar

cSamVar

Method Returns the sample variance of a field (column) in a table.

Type TCursor

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 displayed in a linked table frame or multi-record object. 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 write 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*)

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
```

See Also

cAverage
cCount
cMax
cMin
cNpv
cSamStd
cStd
cSum

cVar

cStd

Method Returns the standard deviation of a field (column) in a table.

Type TCursor

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 numeric fields. The calculation is based on the variance. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. 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 write lock on the table. If a lock cannot be placed, the method fails.

Example In this 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
```

See Also cAverage
cCount
cMax
cMin
cNpv
cSamStd
cSamVar
cSum
cVar

cSum

Method Returns the sum of the value in a field (column) of a table.

Type TCursor

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 numeric fields. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. 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 write lock on the table. If a lock cannot be placed, the method fails.

Example In this 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
```

See Also

cAverage
cCount
cMax
cMin
cNpv
cSamStd
cSamVar
cStd
cVar

currRecord

Method Reads the current record into the record buffer.

Type TCursor

Syntax **currRecord** ( ) Logical

Description **currRecord** reads values of the current record into the record buffer. This method ensures you're working with the most recently updated version of the *Customer* table. Before posting the new *Customer* record, the code gives the user a chance to confirm the changes or cancel them.

Example This example copies a record from the *CustBak* table and inserts it into the record, particularly on a network.

```
; updateFromBackup::pushButton
method pushButton(var eventInfo Event)
var
  custBakTC, custTC TCursor
endVar

custBakTC.open("CustBak.db")
custTC.open("Customer.db")

if custBakTC.locate("Customer No", 6312) then
  custTC.edit()
  custTC.insertRecord(custBakTC)
  if msgYesNoCancel("Confirm",
     "Do you want to overwrite Customer information?") = "Yes"
then
    ; user confirmed, so overwrite the existing record
    custTC.updateRecord()
    custTC.endEdit()
  else
    ; user did not confirm, so cancel changes to record
    custTC.currRecord()
  endif
else
  msgStop("Error", "Can't find Customer 6312")
endif

endmethod
```

See Also

home
end
nextRecord
priorRecord
skip
moveToRecord

cVar

Method Returns the variance of a field (column) in a table.

Type TCursor

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 displayed in a linked table frame or multi-record object. **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 write lock on the table. If a lock cannot be placed, the method fails.

Example In this 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
```

See Also cAverage
cCount
cMax
cMin
cNpv
cSamVar
cSamStd
cStd
cSum

deleteRecord

Beginner

Method Deletes the record pointed to by a TCursor.

Type TCursor

Syntax **deleteRecord** ( ) Logical

Description **deleteRecord** deletes the record pointed to by a TCursor without prompting for confirmation. The operation cannot be undone. 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.

Example In this 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
```

See Also empty

didFlyAway

Method Reports whether the current record moved to a different position as the result of a key value change.

Type TCursor

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 (Yes); otherwise, **didFlyAway** returns False (even if the record moved to its sorted position).

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)
; now, a call to unlockRecord will make tc point
; to a record that "flies away"

if tc.locate("Key Field", 120) then
  tc.edit()

  ; change the key value so that the record
  ; changes relative position
  tc."Key Field" = 145

  ; Unlock the record. Because setFlyAwayControl
  ; is set to Yes, tc still points to the record
  tc.unlockRecord()

  ; this 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
```

See Also setFlyAwayControl
unLockRecord

dropIndex

Method Deletes an index file associated with a table.

Type TCursor

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 calling the Table method **setExclusive** *before* opening the table) before calling **dropIndex**.

Example In this example, the **pushButton** method for a button deletes the *CustName* tag from a .MDX file and the primary index from a Paradox table:

```
method pushButton(var eventInfo Event)
var
  tc1, tc2 TCursor
  tb1, tb2 Table
endVar

if isTable("Sales.dbf") then
  tb1.setExclusive (Yes)
  tb1.attach("Sales.dbf")                  ; Sales.dbf is a
dBASE table
  tc1.open(tb1)
  tc1.dropIndex("index2.mdx", "CustName") ; delete CustName
tag from index2 file
  msgInfo("", "custname dropped")
else
  msgStop("Stop!", "Could not find Sales.dbf table.")
endif

if isTable("Orders.db") then
  tc2.open("Orders.db")                    ; Orders.db is a
Paradox table
tb2.attach("Orders.db")
tb2.setExclusive (Yes)
tc2.open(tb2)
  if tc2.dropIndex() then
    msgInfo("", "Primary index for Orders.db was deleted")
  else
    msgStop("", "Could not delete primary index for
Orders.db")
  endif
else
  msgStop("Stop!", "Could not find Orders.db table.")
endif
endmethod
```

See Also switchIndex

edit

Beginner

Method Puts a TCursor into Edit mode.

Type TCursor

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** 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.

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
```

See Also cancelEdit
endEdit
postRecord

empty

Method Deletes all records from a table.

Type TCursor

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 is required. This operation cannot be undone.

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
```

See Also deleteRecord
nRecords

end

Beginner

Method Moves a TCursor to the last record in a table.

Type TCursor

Syntax **end** ( ) Logical

Description **end** sets the current record (and the record buffer) to the last record in a table.

Example This 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",
        "Oustanding balance: " + tc."Balance Due")

endmethod
```

See Also

home
nextRecord
priorRecord
currRecord
skip
moveToRecord

endEdit

Beginner

Method Exits Edit mode and accepts changes made to the current record.

Type TCursor

Syntax **endEdit** ( ) Logical

Description **endEdit** accepts changes made 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.)

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
```

See Also cancelEdit

edit

enumFieldNames

Method Fills an array with the names of fields in a table.

Type TCursor

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.

Example For this 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
```

See Also enumFieldNamesInIndex

enumFieldNamesInIndex

Method Fills an array with the names of fields in a table's index.

Type TCursor

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.

Example In this 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", "byDate", fieldNames)
  ; display the index field names for byDate in DateIndx
  fieldNames.view()
else
  msgStop("Stop", "Couldn't open Sales.dbf.")
endIf
endmethod
```

See Also enumIndexStruct

enumFieldStruct

Method Creates a Paradox table listing the structure of a TCursor.

Type TCursor

Syntax **enumFieldStruct** (const ***tableName*** String) Logical

Description **enumFieldStruct** creates the Paradox table specified in *tableName* listing the structure of a TCursor. 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 a table's field structure (including primary keys and validity checks) for use in the new table.

The structure of the Paradox table is listed in the following table:

Field Name	Field Type
FieldName	A31
Type	A31
Size	S
Dec	S
Key	A1
_Required Value	A1
_Min Value	A255
_Max Value	A255
_Default Value	A255
_Picture Value	A175
_Table Lookup	A81
_Table Lookup Type	A1
_Invariant Field ID	S

Example For this 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
```

```
  ; include from Customer field name, ValChecks,
  ; and key fields in new table CustFlds
  tc.enumFieldStruct(structName)

  ; now 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
```

See Also

enumFieldNames
enumFieldNamesInIndex
enumIndexStruct
enumRefIntStruct
enumSecStruct

enumIndexStruct

Method Creates a Paradox table listing the structure of a TCursor's secondary indexes.

Type TCursor

Syntax **enumIndexStruct** (const ***tableName*** String) Logical

Description **enumIndexStruct** creates the Paradox table specified in *tableName* listing the structure of a table's secondary indexes. 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.

The structure of *tableName* is listed in the following table.

Field Name	Field Type
infoHeader	A1
szName	A127
szTagName	A31
szFormat	A31
bPrimary	A1
bUnique	A1
bDescending	A1
bMaintained	A1
bCaseInsensitive	A1
bSubset	A1
bExpIdx	A1
bKeyExpType	N
szKeyExp	A220
szKeyCond	A220
FieldNo	N
FieldName	A31

Example For this 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
```

See Also

enumFieldNames
enumFieldNamesInIndex
enumFieldStruct
enumRefIntStruct
enumSecStruct

enumLocks

Method Creates a Paradox table listing the locks currently applied to a table.

Type TCursor

Syntax **enumLocks** (const ***fileName*** 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* is listed below:

Field Name	Field Type
UserName	A15
Lock Type	A32
Net Session	N
Session	N
Record Number	N

Example In this 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
```

See Also lock
lockStatus

enumRefIntStruct

Method Creates a Paradox table listing referential integrity information for a TCursor.

Type TCursor

Syntax **enumRefIntStruct** (const ***tableName*** String) Logical

Description **enumRefIntStruct** writes referential integrity information for a TCursor to the 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.

The structure of *tableName* is listed in the following table:

Field Name	Field Type	Size
infoHeader	A	1
RefName	A	31
Other Table	A	81
Slave	A	1
Modify	A	1
Delete	A	1
FieldNo	N	
aiThisTabField	A	31
Other FieldNo	N	
aiOthTabField	A	31

Example This 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
```

See Also

enumFieldNames
enumFieldNamesInIndex
enumFieldStruct
enumIndexStruct
enumSecStruct
Form::enumTableLinks

enumSecStruct

Method Writes table security information to a Paradox table.

Type TCursor

Syntax **enumSecStruct** (const ***tableName*** String) Logical

Description **enumSecStruct** creates the Paradox table specified in *tableName* listing security information (access rights) of a TCursor. 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.

The structure of *tableName* is listed in the following table.

Field Name	Field Type	Size
infoHeader	A	1
iSecNum	N	
eprvTable	N	
eprvTableSym	A	10
iFamRights	N	
iFamRightsSym	A	10
szPassword	A	31
fldNum	N	
aprvFld	N	
aprvFldSym	A	10

Example This 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

tc.open("Secrets.db") ; associate tc with SECRETS.DB
tc.enumSecStruct("SecInfo.db") ; write security information to
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
```

See Also enumFieldNames

enumFieldNamesInIndex
enumFieldStruct
enumIndexStruct
enumRefIntStruct

enumTableProperties

Method Writes the properties of a TCursor to a Paradox table.

Type TCursor

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
TableName	A32
PropertyName	A64
PropertyValue	A255

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
```

See Also enumFieldNames

eot

Method Tests for a move past the end of a table.

Type TCursor

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 1000 to 10,000. If you call **setFilter** 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 **setFilter** can affect **eot** and **bot**:

```
var tc TCursor endvar
tc.open("Customer.db")
tc.setFilter(1, 10)           ; filter ranges from 1 to 10
  ; tc.eot() and tc.bot() are True at this point
```

Similarly, if a call to **switchIndex** forces the TCursor to point to a nonexistent record, **eot** and **bot** methods return True.

Example In this 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
```

See Also bot
end
home

familyRights

Method Tests for a user's ability to create or modify objects in a table's family.

Type TCursor

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.

Example This 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
```

See Also tableRights

fieldName

Method Returns the name of a field.

Type TCursor

Syntax **fieldName** (const ***fieldNum*** SmallInt) String

Description **fieldName** returns the name of field *fieldNum*. Fields are numbered from left to right, beginning with 1.

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
```

See Also fieldNo
fieldType
fieldValue

fieldNo

Method Returns the position of a field in a table.

Type TCursor

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.

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
```

See Also fieldValue

fieldRights

Method Reports whether a user has rights to read or modify a field in a table.

Type TCursor

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.

Example This 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
```

See Also tableRights

fieldSize

Method Returns the size of a field.

Type TCursor

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.

Example This 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
```

See Also fieldName
fieldType
fieldUnits2

fieldType

Method Returns the data type of a field.

Type TCursor

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 table lists the possible return values:

Field type	Paradox table	dBASE table
Alphanumeric	ALPHA	(none)
Character	(none)	CHARACTER
Date	DATE	DATE
Integer	SHORT	(none)
Floating-point value	NUMERIC	FLOAT (IV)
		NUMERIC (III+ or IV)
Graphic	GRAPHIC	(none)
Logical	(none)	BOOLEAN
Money	MONEY	(none)
Memo	MEMO	MEMO
Formatted memo	FMTMEMO	(none)
Binary	BINARYBLOB	(none)
OLE object	OLEOBJ	(none)

Example This 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
```

See Also fieldNo

fieldUnits2

Method Returns the number of decimal places defined for a numeric field in a dBASE table.

Type TCursor

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. 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.

Example For this 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
```

See Also fieldName
fieldNo
fieldSize
fieldType

fieldValue

Method Reads the value of a specified field.

Type TCursor

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)
```

Example For this 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
```

See Also setFieldValue

forceRefresh

Method Makes TCursor point to the current data in the underlying table.

Type TCursor

Syntax **forceRefresh**() Logical

Description 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.

Example This 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.fName = 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.fName = fVal_2:
      flQty = flQty + tc.Qty
   endScan

   msgInfo("CA Qty and FL Qty:",
           "CA = " + caQty + "\n" + FL = " + flQty)

endmethod
```

See also currRecord

getLanguageDriver

Method Returns the name of the current language driver for a table.

Type TCursor

Syntax **getLanguageDriver** ( ) String

Description **getLanguageDriver** returns a String value indicating the language driver for a table.

Example This 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
```

See Also getLanguageDriverDesc

getLanguageDriverDesc

Method Returns the name of the current language driver description for a table.

Type TCursor

Syntax **getLanguageDriverDesc** ( ) String

Description **getLanguageDriverDesc** returns a String value indicating the language driver description for a table.

Example This 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
```

See Also getLanguageDriver

home

Beginner

Method Moves to the first record of a table.

Type TCursor

Syntax **home** ( ) Logical

Description **home** sets the current record (and the record buffer) to the first record in a table.

Example For this 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."Order No"
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
```

See Also

end
moveToRecord
moveToRecNo
nextRecord
priorRecord
skip

initRecord

Method	Empties the record buffer.
Type	TCursor
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.
See Also	copyRecord currRecord

insertAfterRecord

Method Inserts a record into a table after the current record.

Type TCursor

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).

Example For this 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
```

See Also insertRecord
insertBeforeRecord

insertBeforeRecord

Method Inserts a record into a table before the current record.

Type TCursor

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).

Example For this 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
```

See Also insertRecord
insertAfterRecord

insertRecord

Beginner

Method Inserts a record into a table.

Type TCursor

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).

Example For this 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
```

See Also insertAfterRecord
insertBeforeRecord

isAssigned

Method Reports whether a TCursor variable has been assigned a value.

Type TCursor

Syntax **isAssigned** ( ) Logical

Description **isAssigned** returns True if a TCursor variable has a value assigned using **open** or **attach**; otherwise, it returns False.

Example This 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
```

See Also open
close

isEdit

Method Reports whether a TCursor is in Edit mode.

Type TCursor

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.

Example For this 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
```

See Also edit
endEdit

isEmpty

Method Determines whether a table contains any records.

Type TCursor

Syntax **isEmpty** ( ) Logical

Description **isEmpty** returns True if there are no records in the table associated with the TCursor; otherwise, it returns False.

Example For this 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
```

See Also empty
nRecords
insertRecord

isEncrypted

Method Reports whether a table is password-protected.

Type TCursor

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.

Example This 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.")
else
  msgStop("Error", "Can't open the Customer table.")
endif

endmethod
```

See Also tableRights
Session::addPassword
Table::protect

isOnSQLServer

Method Reports whether a TCursor is associated with a table on a SQL server.

Type TCursor

Syntax **isOnSQLServer**() Logical

Description Returns True if the TCursor is associated with a table on a SQL server; otherwise, returns False.

Example This example is a custom method that uses **isOpenOnSQLServer** to find out whether a TCursor is associated with a remote table. If **isOpenOnSQLServer** 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?",
                     "Do you want to lock a remote table?") =
"Yes" then
         return True
      else
         return False
      endIf

   endIf

endMethod
```

See also isOpenOnUniqueIndex

isOpenOnUniqueIndex

Methods Reports whether a TCursor is open on a unique index.

Type TCursor

Syntax **isOpenOnUniqueIndex**() Logical

Description 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) will fail unless the TCursor is opened on a unique index.

Example This 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 Reports whether the current record has been deleted (dBASE tables only).

Type TCursor

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**.

Example This 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
```

See Also isShowDeletedOn

showDeleted

isShared

Method Reports whether a table is currently shared.

Type TCursor

Syntax **isShared** ( ) Logical

Description **isShared** returns True if another user has opened the table pointed to by a TCursor; otherwise, it returns False.

Example For this 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
```

See Also isAssigned
isValid

isShowDeletedOn

Method Reports whether deleted records in a dBASE table are shown.

Type TCursor

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.

Example In this 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
```

See Also compact
isRecordDeleted
showDeleted

isValid

Method Reports whether the contents of a field are legitimate and complete.

Type TCursor

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.

Example This 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 Number
endVar
tryValue = 100
tc.open("Orders.db")
if NOT tc.isValid("Sale Date", tryValue) then  ; 100 is not a
valid date
  msgStop("Error",
  String(tryValue) + " is not valid for this field.")
else                                           ; this
condition is never met
  tc."Sale Date" = tryValue
  tc.postRecord()
endif

endmethod
```

See Also postRecord

setFieldValue

locate

Beginner

Method Searches for a specified field value.

Type TCursor

Syntax **1. locate** (const ***fieldName*** String, const ***searchValue*** AnyType [, const ***fieldName*** String, const ***searchValue*** AnyType]*) Logical

2. locate (const ***fieldNum*** SmallInt, const ***searchValue*** AnyType [, const ***fieldNum*** SmallInt, const ***searchValue*** AnyType]*) Logical

Description **locate** searches a table for records whose values match the criteria specified in one or more field/value pairs. Specify the value to search for in *searchValue* and the field to search in *fieldName* or *fieldNum*. This method guarantees that the first value matching *searchValue* 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).

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 "Profesional Divers, Ltd." in the Name field
if ordTC.locate("Name", "Profesional 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 \nProfesional Divers, Ltd.")
endIf
ordTC.endEdit()
endmethod
```

See Also locateNext
locatePattern
locateNextPattern

locateNext

Beginner

Method Searches for a specified field value.

Type TCursor

Syntax **1. locateNext** (const ***fieldName*** String, const ***searchValue*** AnyType [, const ***fieldName*** String, const ***searchValue*** AnyType]*) Logical

2. locateNext (const ***fieldNum*** SmallInt, const ***searchValue*** AnyType [, const ***fieldNum*** SmallInt, const ***searchValue*** AnyType]*) Logical

Description **locateNext** searches a table for records whose values match the criteria specified in one or more field/value pairs. Specify the value to search for in *searchValue* and the field to search in *fieldName* or *fieldNum*. This method guarantees that the next value matching *searchValue* 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).

Example This 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
```

See Also locate
locateNextPattern
locatePattern

locateNextPattern

Method Locates the next record containing a field that has a specified pattern of characters.

Type TCursor

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*, and specify a pattern of characters in *pattern*.

You can include the 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** in String for more information about advanced match pattern operators, and **advancedWildCardsInLocate** and isAdvancedWildCardsInLocate in Session.

For example, the following statement checks values the 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*")
```

Example

In this 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 array
  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
```

See Also

locatePattern
locateNext
String::advMatch
Session::advancedWildCardsInLocate
Session::isAdvancedWildCardsInLocate

locatePattern

Method Locates a record containing a field that has a specified pattern of characters.

Type TCursor

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 to the 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*, and specify a pattern of characters in *pattern*.

You can include the 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** in String for more information about advanced match pattern operators, and **advancedWildCardsInLocate** and isAdvancedWildCardsInLocate in Session.

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*")
```

Example

In this 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 array
  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
```

See Also

locateNextPattern
locate
String::advMatch
Session::advancedWildCardsInLocate
Session::isAdvancedWildCardsInLocate

locatePrior

Method Searches for a specified field value.

Type TCursor

Syntax **1. locatePrior** ([const fieldName String, const searchValue AnyType] 1+) Logical

2. locatePrior (const ***fieldNum*** SmallInt const ***searchValue*** AnyType [, const ***fieldNum*** String, const ***searchValue*** AnyType]*) Logical

Description **locatePrior** searches a table for records whose values 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*. This method guarantees that the previous value matching *searchValue* 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 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.

Example In this 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
```

See Also locateNext

locateNextPattern

locatePattern

locatePriorPattern

Method Locates the previous record containing a field that has a specified pattern of characters.

Type TCursor

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*, and specify a pattern of characters in *pattern*.

You can include the 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** in String for more information about advanced match pattern operators, and **advancedWildCardsInLocate** and **isAdvancedWildCardsInLocate** in Session.

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*")
```

Example In this 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
```

See Also

locatePattern
locateNextPattern
String::advMatch
Session::advancedWildCardsInLocate
Session::isAdvancedWildCardsInLocate

lock

Beginner

Method Places specified locks on a specified table.

Type TCursor

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: Write, Read, Full, or Exclusive. If successful, this method returns True; otherwise, it returns False.

If a TCursor is opened onto a dBASE table, a Full or Exclusive lock cannot be obtained.

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
```

See Also lockStatus

unlock

lockRecord

Method Puts a write lock on the current record.

Type TCursor

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.

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
```

See Also unLockRecord

lockStatus

Method Returns the number of times you have placed a lock on a table.

Type TCursor

Syntax **lockStatus** (***lockType*** String) SmallInt

Description **lockStatus** returns the number of times you have placed a lock of type *lockType* on a table, 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, not on locks placed by Paradox or by other users or applications.

Example This example uses **lockStatus** to determine whether you've placed any write locks on the *Customer* table; if so, all write locks are removed.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tblName String
endVar

tc.open("Customer.db")

; loop until all write locks are removed from Customer
while tc.lockStatus("Write") > 0
  tc.unlock("Write")
endWhile
message("All write locks removed from Customer.db")

endmethod
```

See Also lock
unLockRecord

moveToRecNo

Method Moves a TCursor to a specific record in a table.

Type TCursor

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.

Example This example moves a TCursor to the sixth record in a dBASE table.

```
; gotoRecSix::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.dbf")     ; suppose Orders.db has 10 records
if not tc.moveToRecNo(6) then
  msgStop("Sorry", "Can't post current record.")
endif
endmethod
```

See Also

currRecord
end
home
moveToRecord
nextRecord
nRecords
priorRecord
skip

moveToRecord

Method Moves a TCursor to a specific record in a table.

Type TCursor

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 the method **nRecords** or examine the NRecords property 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).

Example In this example, the **pushButton** method attempts to move a TCursor to the sixth record in the *Orders* table. If the TCursor can't be moved (because the current record can't be posted) this method displays a warning message.

```
; gotoRecSix::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.db")      ; suppose Orders.db has 10 records
if not tc.moveToRecord(6) then
  msgStop("Sorry", "Can't post current record.")
endif
endmethod
```

See Also nRecords
home
end
nextRecord
priorRecord
currRecord
skip

nextRecord

Beginner

Method Moves to the next record in a table.

Type TCursor

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.

Example In this 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
```

See Also home

end

priorRecord
skip
moveToRecord

nFields

Method Returns the number of fields in a table.

Type TCursor

Syntax **nFields** ( ) LongInt

Description **nFields** returns the number of fields in the table associated with a TCursor.

Example In this 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
```

See Also nKeyFields
nRecords

nKeyFields

Method Returns the number of fields in the current index of a table.

Type TCursor

Syntax **nKeyFields** ( ) LongInt

Description **nKeyFields** returns the number of fields the current index of the table associated with a TCursor. When used on a Paradox table, this method works with the primary index; when used on a dBASE table, it works with the index specified by the Table type method **setIndex**.

Example This example reports the number of key fields in a Paradox table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  pdoxTC, dBASETC TCursor
  nkf LongInt
  pdoxTbl, dBASETbl String
  tb Table
endVar
pdoxTbl = "Orders.db"
dBASETbl = "Scores.dbf"

if pdoxTC.open(pdoxTbl) then
  nkf = pdoxTC.nKeyFields()      ; number of key fields in the
primary index
  msgInfo(pdoxTbl, pdoxTbl + " has " + String(nkf) + " key
fields.")
else
  msgInfo("Sorry", "Can't open " + pdoxTbl + " table.")
endif

endmethod
```

See Also nFields
nRecords

nRecords

Beginner

Method Returns the number of records in a table.

Type TCursor

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 large tables.

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.

Example In this example, the **pushButton** method for *thisButton* runs a custom method if there are more than 10000 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 10000
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
```

See Also nFields
nKeyFields

open

Beginner

Method Opens a table.

Type TCursor

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*. If you use syntax 1, where *tableName* is a String, you can use arguments *db* and *indexName* to specify a database and an index. If you use 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**.

Example 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
tc1.open("Customer.db", samp, "NameAndState")

endmethod
```

The next example achieves the same as the previous example, 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
tc1.open(tbl)

endmethod
```

See Also close

postRecord

Beginner

Method Posts changes to a record.

Type TCursor

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.

Example For this 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
```

See Also unLockRecord

priorRecord

Beginner

Method Moves to the previous record in a table.

Type TCursor

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).

Example In this 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
```

See Also

home
end
nextRecord
skip
moveToRecord

qLocate

Method Searches an indexed table for a specified field value.

Type TCursor

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 operation fails if the current record cannot be posted or if the number of search values exceeds the number of fields in the current index.

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
```

See Also

locate
locateNext
locateNextPattern
locatePattern
locatePrior
locatePriorPattern

recNo

Method Returns the record number of the current record.

Type TCursor

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.

Example In this 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
```

See Also nRecords

recordStatus

Method Reports about the status of a record.

Type TCursor

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.

Example This 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
```

See Also lockRecord
unlockRecord

reIndex

Method Rebuilds specified index files.

Type TCursor

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.

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
```

See Also reIndexAll

reIndexAll

Method Rebuilds all index files for a table.

Type TCursor

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.

Example For this example, the **pushButton** method for a button attempts to place a full 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
  tc      TCursor
  pdoxTbl String
  tb      Table
endVar
pdoxTbl = "Customer.db"

tb.attach(pdoxTbl)
tb.setExclusive(Yes)

if tc.open(tb) then
  tc.reIndexAll()                ; rebuild all Customer
indexes
  message("Indexes rebuilt.")
else
  msgStop("Sorry", "Can't open " + pdoxTbl + " table.")
endif
endmethod
```

See Also reIndex

seqNo

Method Returns the record number of the current record.

Type TCursor

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.

Example In this 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
```

See Also moveToRecNo
moveToRecord
recNo

setFieldValue

Method Assigns a value to a specified field.

Type TCursor

Syntax **setFieldValue** (const ***fieldName*** String, const ***value*** AnyType) Logical

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)
```

Example In this 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
```

See Also fieldValue

setFilter

Method Sets the range of records a TCursor can point to.

Type TCursor

Syntax **setFilter** ([const ***exactMatchVal*** AnyType,] * const ***minVal*** AnyType, const ***maxVal*** AnyType) Logical

Description **setFilter** specifies conditions for including a range of records. Records that meet the conditions are included, records that don't are filtered out. This operation fails if the current record cannot be committed or if the TCursor does not point to a keyed table.

This method compares the criteria you specify with values in the corresponding fields of a table's index. To filter records based on the value of a single field, specify values in *minVal* and *maxVal.* For example, the following statement checks values in the first field of the index of each record. If a value is less than 14 or greater than 88, that record is filtered out.

```
tcVar.setFilter(14, 88)
```

To specify an exact match on a single field, assign *minVal* and *maxVal* the same value. For example, the following statement filters out all values except 55:

```
tcVar.setFilter(55, 55)
```

You can filter records 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're the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field:

```
tcVar.setFilter("Borland", "Paradox", 100, 500)
```

Calling **setFilter** without any arguments resets the filter to include the entire table. If you don't call this method before opening a TCursor for a dBASE table, deleted records are not shown.

Example For this 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 **setFilter**, 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.setFilter(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
```

See Also

reIndex
reIndexAll
switchIndex

setFlyAwayControl

Method Controls whether the TCursor points to a record whose position has changed as the result of an **unlockRecord**.

Type TCursor

Syntax **setFlyAwayControl** ([const ***yesNo*** Logical]) Logical

Description **setFlyAwayControl** specifies in *yesNo* whether the TCursor will stay on the record after a successful call to **unlockRecord**.

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 the **setFlyAwayControl** method to control the behavior of **unlockRecord** and record flyaway. If the optional argument *yesNo* is Yes, **setFlyAwayControl** guarantees that the TCursor will point to the posted record after a call to **unlockRecord**; if set to No, the TCursor points to the record following the original position of the record. You can use the **didFlyAway** method to test whether the record did, infact, fly away.

When **setFlyAwayControl** is set to Yes, Paradox performs record-level checking for many pointer level operations. Because this extra work can slow an application down, **setFlyAwayControl** should be used with caution. The **postRecord** method is usually preferred over **setFlyAwayControl** and **unlockRecord**. See postRecord for more information.

Example See the example for didFlyAway.

See Also didFlyAway
postRecord
unLockRecord

setShowDeleted

Method Specifies whether to show deleted records (dBASE tables only).

Type TCursor

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.

Example

```
var
   dbfTable TCursor
endVar
if dbfTable.open("orders.dbf") then
   dbfTable.setShowDeleted(Yes)
endIf
```

See Also open

showDeleted

Method Specifies whether to show deleted records in a dBASE table.

Type TCursor

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.

Example In this 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
```

See Also compact
isRecordDeleted
isShowDeletedOn

skip

Method Moves forward or backward a specified number of records in a table.

Type TCursor

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.

Example This 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
```

See Also

home
end
nextRecord
priorRecord
currRecord
moveToRecord

sortTo

Method Sorts a table.

Type TCursor

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.

Example This 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 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] = 0                ; in ascending order

  sortFlds[2] = "Country"         ; then by Country
  sortOrder[2] = 0                ; 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?",
         "Do you want to replace " + destTbl + " table?")
"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
```

See Also

add
copy
subtract

subtract

Method Subtracts the records in one table from another table.

Type TCursor

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. 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 expensive.

Example In this 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("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
```

See Also add

copy

switchIndex

Beginner

Method Specifies another index to use to view the records in a table.

Type TCursor

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.

Example In this 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()                          ; make sure we're on the
first record
msgInfo("First Record", tc.Name)   ; display value in Name
field

endmethod
```

See Also reIndex

reIndexAll

Table::setIndex

tableName

Method Returns the name of the table associated with a TCursor.

Type 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.

Example In this 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
```

See Also tableRights

tableRights

Method Reports about the operations you can perform on a table.

Type TCursor

Syntax **tableRights** (const ***rights*** String) Logical

Description **tableRights** reports about a user's rights to a table, where *rights* is one of:

"ReadOnly" (read from the table, but don't change it)

"Modify" (enter or change data)

"Insert" (add new records)

"InsDel" (add and delete records)

"All" (perform all operations)

Example This 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
```

See Also fieldRights

type

Method Returns the type of a table.

Type TCursor

Syntax **type** ( ) String

Description **type** returns a string describing the type of a table, either Paradox or dBASE.

Example This 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
```

See Also isAssigned
tableName

unDeleteRecord

Beginner

Method Undeletes the current record from a dBASE table.

Type TCursor

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.

Example This 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
```

See Also deleteRecord
isRecordDeleted
isShowDeletedOn
showDeleted

unlock

Beginner

Method Removes specified locks from a TCursor.

Type TCursor

Syntax **unlock** (const ***lockType*** String) Logical

Description **unlock** attempts to remove locks explicitly placed on the table pointed to by a TCursor. *lockType* must be an expression that evaluates to one of the following string values: Write, Read, Full, or Exclusive. If successful, this method returns True; otherwise, it returns False.

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
```

See Also lockStatus
lock

unLockRecord

Beginner

Method Unlocks the current record.

Type TCursor

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.

You can use the **setFlyAwayControl** method to control the behavior of **unLockRecord** and record flyaway. If **setFlyAwayControl** is set to True, the TCursor will continue to point to the posted record after a call to **unLockRecord**. You can also use the **didFlyAway** method to test whether the record did, in fact, fly away. Refer to the entry for **didFlyAway** method for more information regarding record flyaway and the **unLockRecord** method.

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
```

See Also didFlyAway
lockRecord
postRecord
setFlyAwayControl

updateRecord

Beginner

Method	Updates the existing record with data from the new record when a key violation exists.
Type	TCursor
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**.
Example	See the example for attachToKeyViol
See Also	attachToKeyViol lockRecord postRecord unlockRecord

close

Method Closes a DDE link.

Type DDE

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.

Example The following code opens the ObjectVision form ADDRESS.OVD, gets the values of the Name field and the Company field, then calls **close** to close the DDE link:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  getNames DDE
  name, company AnyType
endVar

; link to an ObjectVision form
getNames.open("vision", "C:\\vision\\sample\\Address.ovd")

getNames.setItem("Name")          ; item is field "Name"
name = getNames                   ; set name = field "Name"

getNames.setItem("Company")       ; item is field "Company"
company = getNames                ; sets company = field
"Company"

msgInfo("From Address.ovd",       ; display info from
ObjectVision
        "Name : " + name + "\n" +
        "Company : " + company )

getNames.close()                  ; close the link

endmethod
```

See Also open
setItem

execute

Method Sends a command via a DDE link.

Type DDE

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.

Example The following code uses the ObjectVision function @SETTITLE to specify the text to display in the ObjectVision title bar.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  d1 DDE
endVar

; open a link to the ObjectVision form named Address
d1.open("vision", "C:\\vision\\forms\\Address.ovd")

; execute the ObjectVision @SETTITLE command
d1.execute("[@SETTITLE(\"I'm in charge!\")]")

endmethod
```

See Also setItem
open
close

open

Method Opens a DDE link to another application.

Type DDE

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.

Example The following code opens two DDE links (represented by the DDE variables *d1* and *d2*) to the ObjectVision form ADDRESS.OVD.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  d1, d2 DDE
  name, company AnyType
endVar

d1.open("vision", "C:\\vision\\forms\\Address.ovd", "Name")
d2.open("vision", "C:\\vision\\forms\\Address.ovd", "Company")

name = d1             ; name takes value from "Name"
company = d2          ; company takes value from "Company"

                      ; display info from ObjectVision fields
msgInfo("Address.ovd Info",
        "Name : " + name + "\n" +
        "Company : " + company )

d1.close()                          ; close DDE links
d2.close()

endmethod
```

See Also close
setItem
execute

setItem

Method Specifies an item in a DDE conversation.

Type DDE

Syntax **setItem** (const ***server*** String)

Description **setItem** is used in a DDE link with application and topic established. *server* specifies a new item. The nature of *server* 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.

Example The following code opens a DDE link to the ObjectVision form ADDRESS.OVD, then calls **setItem** to link first to the Name field, then to the Company field in the ObjectVision form.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  getNames DDE
  name, company AnyType
endVar

; link to an ObjectVision form
getNames.open("vision", "C:\\vision\\sample\\Address.ovd")

getNames.setItem("Name")          ; item is field "Name"
name = getNames                   ; set name = field "Name"

getNames.setItem("Company")       ; item is field "Company"
company = getNames                ; sets company = field
"Company"

msgInfo("From Address.ovd",       ; display info from
ObjectVision
        "Name : " + name + "\n" +
        "Company : " + company )

getNames.close()                  ; close the link

endmethod
```

See Also execute
close

accessRights

Method Reports access rights (also called file attributes) of a file.

Type FileSystem

Syntax **accessRights** ( ) String

Description **accessRights** returns a String describing access rights. 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. You must use **findFirst** before using **accessRights**.

Example This example checks the attributes of the file MEMO14.TXT. It calls **findFirst** to make sure the file exists, then calls **accessRights**. If the file is not marked read-only, this code calls a custom procedure named **launchEditor** (which we assume is defined elsewhere) 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
```

See Also getFileAccessRights
setFileAccessRights

copy

Method Copies a file.

Type FileSystem

Syntax **copy** (const ***srcName***, String ***dstName*** String) Logical

Description **copy** returns True if **copy** has been successful in copying source file *srcName* to destination file *dstName*; otherwise, it returns False. If *dstName* exists, **copy** overwrites it without asking for confirmation. **copy** can copy only one file at a time. You can't use DOS wildcard characters with **copy**.

Example The code in this example searches the current directory for the file specified in the variable *sourceFile*. If the file exists, this code copies it and gives the new file the name specified in *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
```

See Also rename
delete

delete

Method Deletes a file.

Type FileSystem

Syntax **delete** (const ***name*** String) Logical

Description **delete** returns True if **delete** has been successful in deleting the file specified in *name*; otherwise, it returns False. **delete** can delete only one file at a time. You can use DOS wildcard characters, but **delete** will delete only the first file it encounters that matches the file specification.

Example The first example displays a dialog box asking if the user wants to delete the file specified in the variable *fileName*. If the user chooses Yes, the call to **delete** 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
```

The next example 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
```

See Also deleteDir

deleteDir

Method Deletes a directory.

Type FileSystem

Syntax **deleteDir** (const ***name*** String) Logical

Description **deleteDir** returns True if **deleteDir** was successful in deleting the directory specified in *name*; otherwise, it returns False. **deleteDir** does not prompt for confirmation.

Example The first example tries to delete the directory C:\DOS. If the operation fails (for example, because the directory is not empty), a dialog box 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
```

You can delete a directory only if the directory is empty (contains no files). The following examples use **enumFileList** to find if a directory is empty. As the example shows, the way to use **enumFileList** depends on the directory path and file specification. In the first example, **enumFileList** creates an array containing one item (the directory name) when the directory is empty:

```
; 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
```

In the second example, **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.

```
; 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 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
```

See Also

delete
makeDir

drives

Method Returns the letters of the drives attached to the system and known to Windows.

Type FileSystem

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.

Example The following example displays a dialog box listing the drive 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
```

See Also existDrive

enumFileList

Method Writes information about files to a table or an array.

Type FileSystem

Syntax **1. enumFileList** (const ***fileSpec*** String, var ***arrayName*** Array[] String)

2. enumFileList (const ***fileSpec*** String, const ***tableName*** String)

Description **enumFileList** writes information about files matching the criteria in *fileSpec* to the array named in *arrayName* (syntax 1) or to the table named in *tableName* (syntax 2).

If *fileSpec* is *.*, the array or table includes records for the current directory (.) and the parent directory (..).

You must declare the array named in *arrayName* before calling this method. The resulting array contains file names and extensions only.

If *tableName* does not exist, it is created automatically. If *tableName* is created in the directory you're listing, the table does not appear in the list. If *tableName* does exist, information is added to it, and it appears in the list.

Here is the structure of the table:

Field name	Type	Size
Name	A	20
Size	N	
Attributes	A	10
Date	A	10
Time	A	10

File names are listed in the order they're listed in the directory---not necessarily in alphabetical order.

Example The following 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\\*.fsl"

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
```

See Also getValidFileExtensions
isFile

existDrive

Method Reports whether a drive is attached to the system.

Type FileSystem

Syntax **existDrive** (const ***driveLetter*** String) Logical

Description **existDrive** returns True if *driveLetter* is attached to the system; otherwise, it returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example This example calls **existDrive** to test for the existence of drive P. If P exists, the call to **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
```

See Also

drives
setDrive
getDrive
isRemote
isRemovable

findFirst

Method Searches a file system for a file name.

Type FileSystem

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 find if a file or directory exists, and to initialize a FileSystem variable before calling another FileSystem method or procedure.

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.

Example This 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
fs.findFirst("c:\\pdoxwin")

; this displays PDOXWIN (findFirst finds the directory)
msgInfo("findFirst test", fs.name())


; search in the root directory for a file
; or directory named PDOXWIN
fs.findFirst("c:\\pdoxwin\\")

; this displays PDOXWIN (findFirst finds the directory)
msgInfo("findFirst test", fs.name())


; search in the root directory for a file
; or directory named PDOXWIN
fs.findFirst("c:\\pdoxwin\\*.*")

; this displays one dot (.) because the
; first file in a directory is a single dot (.)
msgInfo("findFirst test", fs.name())

endmethod
```

See Also findNext

name

findNext

Method Searches a file system for multiple instances of a file name.

Type FileSystem

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.

Example The first example calls **findFirst** and **findNext** to fill a list with the names of the tables in the current directory. The example assumes that a drop-down list object has already been placed in the form:

```
; fillList::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endvar

if fs.findFirst("c:\\pdoxwin\\sample\\*.db") then
  ; initialize the list in the drop-down edit box
  fileListField.fileList.list.count = 0

  ; this while loop fills the list in the drop-down edit
  ; box with *.db files in the default sample directory
  while fs.findNext()
    fileListField.fileList.list.selection =
    fileListField.fileList.list.selection + 1
    fileListField.fileList.list.value = fs.name()
  endWhile
else
  msgStop("*.db?", "File not found.")
endIf

endmethod
```

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
```

See Also findFirst

freeDiskSpace

Method Returns the amount of free space on a drive.

Type FileSystem

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:").

Example The first 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
```

The next 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
endvar

; if MEMO14.TXT exists in current directory
if fs.findFirst("memo14.txt") then

  ; if there's enough disk space for a a copy of MEMO14.TXT
  if fs.size()  fs.freeDiskSpace(fs.getDrive()) then

    ; copy the file
    fs.copy("memo14.txt", "memo14.bak")

  else
    msgStop("Copy", "Not enough disk space to copy file.")
  endIf
else
  msgStop("MEMO14.TXT", "File not found.")
endIf

endmethod
```

See Also totalDiskSpace
findFirst
getDrive

fullName

Method Returns the full path to a file.

Type FileSystem

Syntax **fullName** ( ) String

Description 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.

Example This example calls **fullName** to get the full name of the first form listed in the current directory. Then it calls **splitFullFileName** to split the name into its component parts and store them in a DynArray. Then it 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
```

See Also name
findFirst
findNext
splitFullFileName

getDir

Method Returns the directory path that the FileSystem variable is pointing to.

Type FileSystem

Syntax **getDir** ( ) String

Description **getDir** returns a string representing the path of the directory that the FileSystem variable is pointing to. The string does not include the drive letteruse **getDrive** for that.

Example This example copies the form DIVEPLAN.FSL from the directory specified in *sourcePath* to the directory specified in *destPath.* This example uses **getDir** to extract the directory from *destPath*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs1, fs2 FileSystem
  sourceFile, destPath String
endvar

sourceFile = "c:\\pdoxwin\\diveplan\\diveplan.fsl"
destPath   = "c:\\pdoxwin\\myforms\\*.*"

if fs1.findFirst(sourceFile) then       ; if the source file
exists
  if fs2.findFirst(destPath) then       ; if the destination
dir exists
    fs1.copy(sourceFile, fs2.getDir())  ; copy file to
destination directory
  else
    msgStop(destPath, "Directory not found.")
  endIf
else
  msgStop(sourceFile, "File not found")
endIf

endmethod
```

See Also setDir
getDrive

getDrive

Method Returns the drive letter pointed to by the FileSystem variable.

Type FileSystem

Syntax **getDrive** ( ) String

Description **getDrive** returns a string representing the drive letter pointed to by the FileSystem variable.

Example This example calls **getDrive** to return the alias of the working directory. Then the example 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
```

See Also existDrive
getDir
setDrive

getFileAccessRights

Procedure Reports access rights (also called file attributes) of a file.

Type FileSystem

Syntax **getFileAccessRights** (const ***fileName*** String) String

Description **getFileAccessRights** 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. This procedure is similar to the **accessRights** method. However, **getFileAccessRights** does not require you to call the **findFirst** method first.

Example This example displays in a dialog box the file attributes for the C:\CONFIG.SYS file.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fileName String
endvar

fileName = "C:\\CONFIG.SYS"

msgInfo(fileName, getFileAccessRights(fileName))

endmethod
```

See Also accessRights
setFileAccessRights

getValidFileExtensions

Procedure Returns the valid file extensions for a specified object.

Type FileSystem

Syntax **getValidFileExtensions** (const ***objectType*** String) String

Description **getValidFileExtensions** returns a string containing the valid file extensions for the object specified in objectType, where objectType is one of: Form, Library, MailMerge, Report, or Script.

Example This 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
```

See Also getDir
getDrive
splitFullFileName

isDir

Procedure Reports whether a specfied string represents the name of a directory.

Type FileSystem

Syntax **isDir** (const ***dirName*** String) Logical

Description **isDir** returns True if *dirName* is a valid directory name; otherwise, it returns False.

Example This example calls **isDir** to make sure that the directory specified by the variable *newDir* is valid. If it is, the call to **setDir** makes *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, and so on.

```
; 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
```

See Also deleteDir
getDir
makeDir
setDir

isFile

Procedure Reports whether a specified string is the name of a file in the current file system.

Type FileSystem

Syntax **isFile** (const ***fileName*** String) Logical

Description **isFile** returns True if *fileName* is a file in the current file system; otherwise, it returns False.

Example The first example 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\\xxxx.xxx"))   ; displays False
sleep(1500)

endmethod
```

The second example prompts the user to enter the full path and file name of a file to delete. A call to **isFile** tests if the file exists, and if it does, a call to **delete** deletes 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
```

See Also isDir

isFixed

Method Reports whether a drive is fixed.

Type FileSystem

Syntax **isFixed** (const ***driveLetter*** String) Logical

Description **isFixed** returns True if *driveLetter* represents a fixed (not removable or network) drive; otherwise, it returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

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
```

See Also isRemote
isRemovable

isRemote

Method Reports whether a drive is remote (a network drive).

Type FileSystem

Syntax **isRemote** (const ***driveLetter*** String) Logical

Description **isRemote** returns True if *driveLetter* represents a remote (network) drive; otherwise, it returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example In this 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 if drive H is a network drive.

```
var
   h FileSystem
endVar
if h.existDrive("h") then ; if drive H is attached
   if h.isRemote() then
      msgInfo("Drive H: ", "Remote Drive")
   else
      msgInfo("Drive H:", "Not a Remote Drive.")
   endIf
else
   msgStop("Drive H", "Drive is not attached.")
endIf
```

See Also isFixed

isRemovable

isRemovable

Method Reports whether a drive is removable.

Type FileSystem

Syntax **isRemovable** (const ***driveLetter*** String) Logical

Description **isRemovable** returns True if *driveLetter* represents a removable drive; otherwise, it returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example In this example, drive D is a removable drive. This code calls **existDrive** to make sure drive D is attached, then calls **isRemovable** to find out if 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
```

See Also isFixed
isRemote

makeDir

Method Creates a new directory.

Type FileSystem

Syntax **makeDir** (const ***name*** String) Logical

Description **makeDir** creates all directories and subdirectories specified in *name*. This method returns True if successful in creating *name*; otherwise it returns False. This method also returns True if the directory already exists.

Example The code in this example tries to create a new directory on drive M. The example displays a dialog box to report whether the method succeeded or failed.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
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
```

See Also deleteDir
isDir

name

Method Returns the name of a file.

Type FileSystem

Syntax **name** ( ) String

Description After a successful **findFirst** or **findNext**, **name** returns the name of the file whose name matches the pattern.

Example This 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
```

See Also fullName
findFirst
findNext

privDir

Procedure Returns the name of the user's private directory.

Type FileSystem

Syntax **privDir** () String

Description **privDir** 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 can use **setAliasPath**, defined for Session to specify the path to the private directory.

Example This example calls **privDir** to display the path to :PRIV: in the status bar.

```
method pushButton(var eventInfo Event)
   message("Your private directory is: ", privDir())
endmethod
```

See Also getDir
startUpDir
workingDir

rename

Method Renames a file.

Type FileSystem

Syntax **rename** (const ***oldName***, String ***newName*** String) Logical

Description **rename** 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. **rename** returns True if it succeeds; otherwise, it returns False. This method is independent of **findFirst** and **findLast**.

Example The following example searches the current directory for the file specified in the variable *oldName*. If it exists, the call to **rename** tries to rename it. A dialog box appears to report errors, if any.

```
; 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
```

See Also copy
delete
findFirst
findNext

setDir

Method Sets the directory path for a FileSystem variable.

Type FileSystem

Syntax **setDir** (const ***name*** String) Logical

Description **setDir** sets the directory path to *name* for a FileSystem variable. Compare this method to **setDrive**, which sets the default drive.

Example This example calls **isDir** to find if the directory specified in the variable *newDir* is valid. If it is, the code 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
```

See Also getDir
setDrive

setDrive

Method Makes a specified drive the default drive.

Type FileSystem

Syntax **setDrive** (const ***name*** String) Logical

Description **setDrive** returns True if successful in setting the default drive to *name*; otherwise, it returns False. You can specify *name* using a letter ("C") or a letter and a colon ("C:"), or an alias (":MAST:").

Example The first example 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, the call to **setDrive** makes 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
```

The next example shows how to use an alias with **setDrive**. It assumes that the alias :MAST: has already been defined.

```
; setDrive::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endVar

fs.setDrive(":MAST:")

endmethod
```

See Also getDrive
setDir

setFileAccessRights

Procedure Sets access rights (also called attributes) of a file.

Type FileSystem

Syntax **setFileAccessRights** (const ***fileName*** String , const ***rights*** String) Logical

Description **setFileAccessRights** sets the attributes (access rights) of *fileName* to the attributes specified in *rights*. *rights* is a string that evaluates to one or more or the following: A, D, H, R, S, V (for archive, directory, hidden, read only, system, and volume, respectively). If you specify an empty string ("") for *rights*, all attributes for *fileName* are removed. You don't have to declare a FileSystem variable (or use the **findFirst** method) before calling **setFileAccessRights**.

Example This example sets file attributes 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
```

See Also accessRights
getFileAccessRights

size

Method Returns the size of a file.

Type FileSystem

Syntax **size** ( ) LongInt

Description After a successful **findFirst** or **findNext**, **size** returns the number of bytes in a found file.

Example This 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
```

See Also freeDiskSpace
totalDiskSpace

splitFullFileName

Procedure Breaks a full path name into its component parts.

Type FileSystem

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 **splitFullFileName** divides a full file path (obtained using **fullName**) into its component parts. You can use syntax 1 to store the results in a DynArray, or use syntax 2 to store the results in separate variables.

If you use syntax 1, you must declare the DynArray before you call **splitFullFileName**. The DynArray has the following keys: DRIVE, EXT, NAME, and PATH.

Example The first example calls **fullName** to get the full name of the first form listed in the current directory. Then it calls **splitFullFileName** to split the name into its component parts and store them in a DynArray, then it 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
```

The next example 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
```

See Also fullName

startUpDir

Procedure Returns a string containing the path to the user's start-up directory.

Type FileSystem

Syntax **startUpDir** () String

Description **startUpDir** returns a string containing the full path (including the drive ID letter) to the user's start-up directory; that is the directory from which Paradox was started.

Example This example displays a dialog box listing the path to the directory from which the user started Paradox:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

msgInfo("Start-up directory", startUpDir())

endmethod
```

See Also privDir
workingDir

time

Method Returns the time and date a file was last modified.

Type FileSystem

Syntax **time** ( ) DateTime

Description **time** returns a DateTime value representing the time and date of the last modification to a file.

Example This example calls time to get the time and date of the most recent changes to the *Customer* table. Then, statements compare 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 Returns the capacity of a drive.

Type FileSystem

Syntax **totalDiskSpace** (const ***driveLetter*** String) LongInt

Description **totalDiskSpace** returns the total number of bytes drive *driveLetter* can hold. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example This example calls **totalDiskSpace** and **freeDiskSpace** to calculate the amount of space in use. It 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
```

See Also freeDiskSpace

windowsDir

Procedure Returns the path to the WINDOWS directory.

Type FileSystem

Syntax **windowsDir** ( ) String

Description **windowsDir** returns the path to the WINDOWS directory.

Example This 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
```

See Also windowsSystemDir
workingDir
privDir
startUpDir

windowsSystemDir

Procedure Returns the path to the Windows system directory.

Type FileSystem

Syntax **windowsSystemDir** ( ) String

Description **windowsSystemDir** returns the path to the Windows system directory.

Example This 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
```

See Also windowsDir

workingDir

Procedure	Returns the name of the current working directory.
Type	FileSystem
Syntax	**workingDir** ( ) String
Description	**workingDir** returns the name of the current working directory (:WORK:).

Example	This example displays a dialog box containing the path to the current working directory:
	`; thisButton::pushButton`
	`method pushButton(var eventInfo Event)`
	
	`message("Working directory is: " + workingDir())`
	
	`endmethod`

See Also	privDir windowsDir windowsSystemDir

close

Method Closes a library.

Type Library

Syntax **close** ( )

Description **close** closes a library, and ends the association between a Library variable and the underlying library file.

Example This 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
```

See Also open

enumSource

Method Writes the code from a library to a Paradox table.

Type Library

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 call **open** to open the library before calling this method.

Example This 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.TXT--
  ; goes to :WORK: by default
  lib.enumSource("LIBSRC.DB")

else
  msgStop("TOOLS.LSL", "Could not open library.")
endIf

endmethod
```

See Also enumSourceToFile

open

enumSourceToFile

Method Writes the code from a library to a text file.

Type Library

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 in the user's private directory (:PRIV:); if the file does exist, Paradox overwrites it without asking for confirmation.

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 Var window and built-in **open** method:

```
Var
   myMsgCursor TCursor
endVar

method open(var eventInfo Event)
if not myMsgCursor.open("Msghelp.db")
   then msgStop("Error", "Couldn't open MsgHelp.db")
      fail()
endif
endmethod
```

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** to open the library before calling this method.

Example This 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
```

See Also enumSource

open

execMethod

Method Calls a custom method that takes no arguments.

Type Library

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. However, the method must be declared in the appropriate Uses window.

Example This 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
```

See Also open

open

Method Associates a Library variable with a library, and makes the library code available.

Type Library

Syntax **open** (const ***libraryName*** String [, const ***libScope*** SmallInt]) Logical

Description **open** associates a Library variable with a library, and makes the library code available to one or more forms, depending on the value of *libScope*. ObjectPAL defines two constants for specifying the scope of a library: PrivateToForm and GlobalToDesktop:

PrivateToForm: only the form that opened the library has access to its code.

GlobalToDesktop: every form in the Desktop (Paradox session) has access to the library.

To open a library and make it available to every form in the current session of Paradox, use the argument 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)
```

Example This 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.

```
; formOne::open
method open(var eventInfo Event)
var
  libOne, libTwo Library
endVar

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 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.

```
; formTwo::open
method open(var eventInfo Event)
var
   kitLib Library
endVar

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
```

See Also close

addAlias

Method/ Procedure	Adds a database alias to a session.
Type	Session
Syntax	**addAlias** (const ***aliasName*** String, const ***type*** String, const ***path*** String) Logical
Description	**addAlias** adds a database alias to a session. Specify the alias name in *aliasName*, the alias type ("Standard") in *type*, and the full DOS path in *path*. An added alias using this method is known only to the session for which it is defined, and exists only until the session is closed.
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. 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
```

See Also	getAliasPath

addPassword

Method/ Procedure Presents a password allowing access to a protected table

Type Session

Syntax **addPassword** (const ***password*** String)

Description **addPassword** presents to a Paradox session the password specified in *password*. 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.

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 to Add")
  ses.addPassword(newPass)
else
  msgStop("Help!","Session variable is not Assigned!")
endif
endmethod
```

See Also removePassword

removeAllPasswords

advancedWildcardsInLocate

Procedure Specifies whether this session can use advanced wildcards in locate operations.

Type Session

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.

Example This 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
```

See Also isAdvancedWildcardsInLocate

blankAsZero

Method/ Procedure Specifies whether to treat blank values as zeros in calculations.

Type Session

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 **setBlankZero** to set it.

Example This 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 isBlankZero() then
    blankAsZero(True)
  endif

  msgInfo("Average Amount Paid", tc.cAverage("Amount Paid"))

else
  msgStop("Error", "Can't open Orders table.")
endif

endmethod
```

See Also isBlankZero

close

Method Closes a session.

Type 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.

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
```

See Also open

enumAliasLoginInfo

Method Writes data about a specified alias to a table.

Type Session

Syntax **enumAliasLoginInfo**(const ***tableName*** String, const ***aliasName*** String) Logical

Description Writes information about the alias specified in *aliasName* to the Paradox table specified in *tableName*. Returns True if successful; otherwise returns False. **enumAliasLoginInfo** operates on aliases stored in ODAPI.CFG and on new aliases opened and stored in system memory. This method fails if the table specified in *tableName* is already open.

The structure of the resulting table is:

Field name	Type	Description
DBName	A32*	Name of the database
Property	A32*	Name of the property
PropertyValue	A82	Value of the property

Examples of properties are OPEN MODE, NET PROTOCOL, SERVER NAME, and USER NAME.

Example This 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 Creates a Paradox table listing the names of database aliases available to a session.

Type Session

Syntax **enumAliasNames** (const ***tableName*** String) Logical

Description **enumAliasNames** creates a Paradox table *tableName* listing the aliases of databases available to a session. 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 *tableName* in the working directory.

The structure of tableName is

Field Name	Type	Size
DBName	A	32*
DBType	A	32
DBPath	A	82

Example In this example, the **pushButton** method for *getAliasButton* writes the alias names active for the *ses* session to the table *AliasNam*, then it views the table.

```
; getAliasButton::pushButton
method pushButton(var eventInfo Event)
var
  dbName   String
  infoType String
  tv1      TableView
endvar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  ses.enumAliasNames("AliasNam.db")   ; create the table
  tv1.open("AliasNam.db")             ; view the table
else
  msgStop("Help!","Session variable is not Assigned!")
endif
endmethod
```

See Also addAlias

enumDataBaseTables

Method/ Procedure Creates a Paradox table listing the tables and other files in a database.

Type Session

Syntax **enumDataBaseTables** (const ***tableName*** String, const ***databaseName*** String, const ***fileSpec*** String)

Description **enumDataBaseTables** creates the Paradox table *tableName* listing the tables and other files in the database specified in *databaseName*. *fileSpec* specifies a DOS file specification (it can include the wildcards * and ?).

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	Size
DBName	A	32*
TableName	A	32*

Example This 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
```

See Also enumDriverCapabilities

enumDriverCapabilities

Procedure Creates three Paradox tables that list the capabilities of the current driver.

Type Session

Syntax **enumDriverCapabilities** (const ***drvCapName*** String, const ***tblCapName*** String, const ***fldCapName*** String

Description **enumDriverCapabilities** creates three Paradox tables that list the capabilities of the current driver. The tables are overwritten (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	Size
DriverType	A	32*
Description	A	32
Category	A	32
DB	A	4
DBType	A	32
MultiUser	A	4
ReadWrite	A	4
Transactions	A	4
PassThruSQL	A	4
Login	A	4
CreateDb	A	4
DeleteDb	A	4
CreateTable	A	4
DeleteTable	A	4
MultiPasswords	A	4

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	Size
DataType	A	32*
TableType	A	32*
Format	A	32*
ReadWrite	A	4
Create	A	4
Restructure	A	4
ValChecks	A	4
Security	A	4
RefInt	A	4
PrimaryKey	A	4
Indexing	A	4

NoFieldType	A	6
MaxRecSize	A	6
MaxFlds	A	6

Field capabilities are written to the table *fldCapName*, which has the following structure:

Field Name	Type	Size
DriverType	A	32*
TableType	A	32*
Format	A	32*
FieldType	A	32*
Description	A	32
NativeType	A	6
XType	A	6
XSubType	A	6
MaxUnits1	A	6
MaxUnits2	A	6
Size	A	6
Required	A	4
Default	A	4
Min	A	4
Max	A	4
RefInt	A	4
Other	A	4
Key	A	4
Multi	A	4

Example

In this 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
```

See Also

enumDriverInfo
enumDriverNames
enumDriverTopics

enumDriverInfo

Procedure Lists the available drivers.

Type Session

Syntax **enumDriverInfo** (const ***tableName*** String)

Description **enumDriverInfo** lists the driver types currently available. Driver types are 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	Size
DriverType	A	32*
Topic	A	32*
Property	A	32*
PropertyValue	A	68

Example This 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
```

See Also enumDriverCapabilities
enumDriverNames
enumDriverTopics

enumDriverNames

Method/ Procedure Creates a Paradox table listing the names of the drivers available in the current session.

Type 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*.

Example This 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
```

See Also enumDriverCapabilities

enumDriverInfo

enumDriverTopics

enumDriverTopics

Procedure Creates a Paradox table listing the topics currently available for each driver type.

Type Session

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	Size
DriverType	A	32*
Topic	A	32*

Example This 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
```

See Also enumDriverCapabilities
enumDriverInfo
enumDriverNames

enumEngineInfo

Procedure Creates Paradox table listing the current ODAPI engine properties.

Type Session

Syntax **enumEngineInfo** (const ***tableName*** String)

Description **enumEngineInfo** creates a Paradox table that describes the contents of the ODAPI 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	Size
Property	A	32*
PropertyValue	A	68

Example This 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
```

See Also enumDriverInfo

enumFolder

Procedure Creates a Paradox table or array listing files in a folder.

Type Session

Syntax **1. enumFolder** (const ***tableName*** String [, const fileSpec String]) Logical

2. enumFolder (var ***result*** Array[] String [, const fileSpec String]) Logical

Description **enumFolder** creates the Paradox table *tableName*, or array *result* listing the files in a session's folder. 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.

Optionally, you can 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 is

Field Name	Type	Size
Name	A	128
LocalName	A	68
IsReference	A	4
IsPrivate	A	4
IsTemp	A	4
Position	A	10

Example In this 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 matched 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 matched your specification.")
tv1.open("PartCat")
endmethod
```

See Also enumDataBaseTables

enumOpenDatabases

Method/ Procedure Creates a Paradox table listing the open databases.

Type Session

Syntax **enumOpenDatabases** (const ***tableName*** String) Logical

Description **enumOpenDatabases** creates the Paradox table *tableName* listing the databases open in the current session. 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	Size
DBName	A	32*
DBType	A	32
ShareMode	A	32
OpenMode	A	32

See Also enumDataBaseTables

enumUsers

Procedure Creates a Paradox table listing all known users with an open channel to the ODAPI engine.

Type Session

Syntax **enumUsers** (const ***tablename*** String) LongInt

Description **enumUsers** creates the table *tableName* that lists all users with an open path to ODAPI. 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	Size
UserName	A	15
NetSession	N	
ProductClass	N	
SerialNumber	A	22

Example This 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
```

See Also getNetUserName

getAliasPath

Method/ Procedure Returns the path for a specified alias.

Type Session

Syntax **getAliasPath** (const ***aliasName*** String) String

Description **getAliasPath** returns the path for the alias *aliasName*.

Example This 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
  alname string
  alpath string
endvar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  alname.view("Enter Alias Name")    ; prompt for an alias
name
  alpath = ses.getAliasPath(alname)  ; get the path
  alpath.view("The Answer is...")    ; show the path
else
  msgStop("Help!","Session variable is not assigned!")
endif
endmethod
```

See Also addAlias

setAliasPath

getAliasProperty

Method Returns the value of a specified property for a specified alias.

Type Session

Syntax **getAliasProperty**(const ***aliasName*** String, const ***property*** String) String

Description Returns a string representing the value of the property specified in *property* for the alias specified in *aliasName*. If the property is not valid for the alias, this method returns an error. **getAliasProperty** operates on aliases stored in ODAPI.CFG and on new aliases opened and stored in system memory.

Example This 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
```

See also setAliasProperty

getNetUserName

Method/ Procedure	Returns the network user name for a session.
Type	Session
Syntax	**getNetUserName** ( ) String
Description	**getNetUserName** returns the name of the current network user.
Example	This 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
```

See Also enumUsers

ignoreCaseInLocate

Procedure Specifies whether to ignore case-sensitivity in locates.

Type Session

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, this subsequent locate operations will ignore case in string comparisons; if *yesNo* is No, locate operations will be case-sensitive.

Example This 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
endvar

if tc.open("Customer.db") then
  ; if case-sensitivity is turned off
  if isIgnoreCaseInLocate() then

    ; locate values based on value as entered
    ; (do not ignore case in string compares)
    ignoreCaseInLocate()
  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
else
  msgStop("Error", "Can't open Customer table.")
endif

endmethod
```

See Also isIgnoreCaseInLocate

isAdvancedWildcardsInLocate

Procedure Reports whether this session is using advanced wildcards in locate operations.

Type Session

Syntax **isAdvancedWildcardsInLocate** ( ) Logical

Description **advancedWildcardsInLocate** reports whether the current session is using advanced wildcards during locate operations that include pattern strings.

Example This 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
```

See Also advancedWildcardsInLocate

isAssigned

Method	Reports whether a session variable is assigned.
Type	Session
Syntax	**isAssigned** ( ) Logical
Description	**isAssigned** reports whether a Session variable is assigned.
Example	See the example for close.
See Also	open close

isBlankZero

Method/ Procedure	Reports whether blank values are being treated as zero in calculations.
Type	Session
Syntax	**isBlankZero** ( ) Logical **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.
Example	See the example for blankAsZero.
See Also	blankAsZero

isIgnoreCaseInLocate

Procedure	Reports whether the current session is ignoring case-sensitivity in locate operations.
Type	Session
Syntax	**isIgnoreCaseInLocate** ( ) Logical
Description	**isIgnoreCaseInLocate** reports whether the current session is ignoring case-sensitivity during locate operations.
Example	See the example for ignoreCaseInLocate.
See Also	ignoreCaseInLocate

lock

Procedure Locks one or more tables.

Type Session

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 a 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: Write, Read, and Full. ("Read" and "Full" apply only to Paradox tables.)

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.

Example This example attempts to place a write lock on the *Orders* table and a Full 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")

custTC.open("Customer.db", "samples")
ordTB.attach("Orders.db", "samples")

if lock(custTC, "Read", ordTB, "Write") then
  if custTC.locate("Name", "Unisco") then
    custNo = custTC."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
  msgStop("Error", "Can't lock one or more tables.")
endif

endmethod
```

See Also unLock

open

Method Opens a session (a channel to the database engine).

Type Session

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.

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
```

See Also close

setAliasPassword

Method Sets the in-memory password for a specified alias.

Type Session

Syntax **setAliasPassword**(const ***aliasName***, const ***password*** String) Logical

Description Sets the in-memory password for the alias specified in *aliasName* to the value specified in *password*. 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.

Example This 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"

   ; 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
```

See also getAliasProperty

setAliasProperty

setAliasProperty

Method	Sets the value of a specified property for a specified alias.
Type	Session
Syntax	**setAliasProperty**(const ***aliasName*** String, const ***property*** String, const ***propertyValue*** String) Logical
Description	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 ODAPI.CFG. Use the Session procedure **saveCFG** to save alias properties to a file.
Example	See the example for getAliasProperty.
See also	getAliasProperty setAliasPassword

removeAlias

Method/ Procedure Removes an alias from a session.

Type 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.

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,
; remove the alias from the current session

removeAlias("CustomerInfo")

endmethod
```

See Also addAlias

removeAllPasswords

Method/ Procedure	Removes all passwords presented to a session.
Type	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. Open tables are not affected by **removeAllPasswords**. To remove access to open tables, you must both close the table and withdraw the passwords that provide access to the tables.
Example	This 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
```

See Also	addPassword removePassword

removePassword

Method/ Procedure	Removes a password presented to a session.
Type	Session
Syntax	**removePassword** (const ***password*** String)
Description	**removePassword** reverses the effect of a **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.
Example	In this 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
```

See Also	addPassword removeAllPasswords

retryPeriod

Method/ Procedure	Returns the number of seconds to retry an operation on a locked record or table.
Type	Session
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.
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
```

See Also setRetryPeriod

saveCFG

Method/ Procedure	Saves the current session's alias information to a file.
Type	Session
Syntax	**saveCFG** (const ***fileName*** String) Logical
Description	**saveCFG** saves the ODAPI 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 ODAPI.CFG to set session information when you start Paradox for Windows.
Example	See the example for System::dlgNetSystem
See Also	System::dlgNetSystem

setAliasPath

Method/ Procedure	Sets the path for an alias.
Type	Session
Syntax	**setAliasPath** (const ***aliasName*** String, const ***aliasPath*** String) Logical
Description	**setAliasPath** sets the path *aliasPath* for the alias *aliasName*.
See Also	getAliasPath

setRetryPeriod

Method/ Procedure	Sets the number of seconds to retry an action on a locked table or record.
Type	Session
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.
Example	This 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
```

See Also retryPeriod

unLock

Procedure Unlocks one or more tables.

Type Session

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: Write, Read, and 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, make 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 type 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.

Example See the example for lock

See Also lock

beep

Beginner

Procedure Sounds the Windows default beep.

Type System

Syntax **beep** ( )

Description **beep** activates the Windows default beep sound. The beep will be audible only if the Enable System Sounds option is checked in the Control Panel's Sound dialog box.

To send a sound of specified pitch and duration to the system speaker, use **sound**.

Example The following code is attached to a button's **pushButton** method. It prompts the user to enter a number and beeps if the number is out of range.

```
; 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
```

See Also sound

close

Beginner

Procedure Closes the current form.

Type System

Syntax **close** ([const ***returnValue*** AnyType])

Description **close** posts a request to close the current form. If *returnValue* is specified, a value will be returned to the calling form (if there is one). Specifying a *returnValue* when there is no calling form does not result in an error. **close** starts the process of closing the form, which includes removing the focus and departing.

Example The following code closes the current form after asking the user 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
```

See Also exit

constantNameToValue

Procedure Returns the numeric value of a constant.

Type System

Syntax **constantNameToValue** (const ***constantName*** String) AnyType

Description **constantNameToValue** returns the value represented by the ObjectPAL constant specified in *constantName*. **constantNameToValue** returns values only for predefined ObjectPAL constants; it will not return a value for a constant you have defined yourself.

Note: For readability and portability (among other benefits), we recommend using constant names rather than numeric values.

Example The following code 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
```

See Also constantValueToName
enumRTLConstants

constantValueToName

Procedure	Reports on the name of a constant.
Type	System
Syntax	**constantValueToName** (const ***groupName*** String, const ***value*** AnyType, var ***constName*** String) Logical
Description	**constantValueToName** writes to *constName* the name of a constant with the *value* specified that belongs to the group *groupName*, where *groupName* is one of the Types of Constants. The constant name is written to the variable *constName*. This method returns True if it succeeds; otherwise, it returns False. **constantValueToName** works for names of predefined ObjectPAL constants only; it will not work for a constant you have defined yourself.
Example	See the example for constantNameToValue.
See Also	constantNameToValue enumRTLConstants

cpuClockTime

Beginner

Procedure Returns the number of seconds since the computer was booted.

Type System

Syntax **cpuClockTime** ( ) LongInt

Description **cpuClockTime** returns the number of milliseconds since the computer was booted. The minimum clock increment is 55 milliseconds. This procedure is useful for measuring the interval between two events.

Example This example uses **cpuClockTime** to compare execution times for two **for** loops: one with an undeclared variable, one with a declared variable. Although execution times vary by system, the loop executes significantly faster when the variable is declared.

```
; 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
```

See Also Time::time

dataModelAddTable

Procedure	Adds a table to a form's data model.
Type	System
Syntax	**dataModelAddTable** (const ***tableName*** String) Logical
Description	**dataModelAddTable** adds the table *tableName* to a form's data model.

Example	`= if not dataModelHasTable("order.db") then`
	`  dataModelAddTable(oder.db")`
	`endIf`

See Also	dataModelHasTable dataModelRemoveTable

dataModelHasTable

Procedure Reports whether a table is part of a form's data model.

Type System

Syntax **dataModelHasTable** (const ***tableName*** String) Logical

Description **dataModelHasTable** reports whether *tableName* is a table associated with a form.

Example

```
 = if not dataModelHasTable("order.db") then
   dataModelAddTable("order.db")
endIf
```

See Also dataModelAddTable
dataModelRemoveTable

dataModelRemoveTable

Procedure Removes a table from a form's data model.

Type System

Syntax **dataModelRemoveTable** (const ***tableName*** String) Logical

Description **dataModelRemoveTable** removes *tableName* from the list of tables associated with a form.

Example

```
 = if dataModelHasTable("order.db") then
   dataModelRemoveTable("order.db")
endIf
```

See Also dataModelAddTable
dataModelHasTable

debug

Procedure Halts execution of a method and invokes the Debugger.

Type System

Syntax **debug** ( )

Description Placing **debug** in a method 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. Also, **debug** statements only take effect when you choose Debug | Enable Debug Statement; otherwise, they are ignored.

Note: debug works only in methods and procedures that you write, not for methods and procedures in the ObjectPAL run-time library.

debug is handy for setting persistent breakpoints in methods while you are developing an application. You can test the application with Debug | Enable DEBUG Statement turned on, and deliver the application with it turned off.

Example In this example, assume the Debug | Enable DEBUG Statement ObjectPAL Editor menu command is selected. The following code executes a **for** loop. Halfway through the loop, the call to **debug** suspends execution and opens an Editor window containing the code. Choose Debug | Run to resume execution, or use the other Debugger features.

```
; 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 Debug | Enable DEBUG
                  ; ObjectPAL Editor menu command is checked
    endIf
endFor
endmethod
```

See Also execute

dlgAdd

Procedure Invokes the Table Add dialog box.

Type System

Syntax **dlgAdd** (const ***tableName*** String)

Description **dlgAdd** displays the Table Add dialog box (described in the *User's Guide*), just as if you had chosen File | Table Utilities | Add. The variable *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 is up to the user to fill in any necessary information and close the dialog box.

Example The following example invokes the Table Add dialog box and fills in the *Customer* table name as the source table. The user must fill in the destination table name and close the dialog box.

```
; showAddDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Add dialog box with Customer as the source
dlgAdd("customer.db") ; same as File | Table Utilities | Add
endmethod
```

See Also dlgCopy
dlgEmpty
dlgSubtract

dlgCopy

Procedure Invokes the Table Copy dialog box.

Type System

Syntax **dlgCopy** (const ***tableName*** String)

Description **dlgCopy** displays the Table Copy dialog box (described in the *User's Guide*), just as if you had chosen File | Table Utilities | Copy. The variable *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 is up to the user to fill in any necessary information and close the dialog box.

Example The following example invokes the Table Copy dialog box and fills in the *Customer* table name as the source table. The user must fill in the destination table name and close the dialog box.

```
; showCopyDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Copy dialog box with the Customer table as
the source
DlgCopy("customer.db") ; same as File | Table Utilities | Copy
endmethod
```

See Also dlgAdd
dlgEmpty
dlgSubtract

dlgCreate

Procedure Invokes the Create Table dialog box.

Type System

Syntax **dlgCreate** (const ***tableName*** String)

Description **dlgCreate** displays the Create Table dialog box (described in the *User's Guide*), just as if you had chosen File | New | Table. The variable *tableName* specifies the name of table to create.

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 fill in any necessary information and close the dialog box.

Example The following example invokes the Table Create dialog box. The user must choose the table type, fill out the field roster, and save the created table.

```
; showCreateDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Create dialog box -- table name is not used
dlgCreate("sometbl.db") ; same as  File | New | Table
endmethod
```

See Also dlgCopy
dlgDelete

dlgDelete

Procedure Invokes the Table Delete dialog box.

Type System

Syntax **dlgDelete** (const ***tableName*** String)

Description **dlgDelete** displays the Table Delete dialog box, just as if you had chosen File | Utilities | Delete. The variable *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 fill in any necessary information and close the dialog box.

Example The following example invokes the Table Delete dialog box and fills in the *Customer* table name as the table to delete. The user must close the dialog box and confirm the deletion.

```
; showDeleteDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Delete dialog box for the Customer table
DlgDelete("Customer.db")  ; same as File | Table Utilities |
Delete
endmethod
```

See Also dlgCreate
dlgEmpty

dlgEmpty

Procedure Invokes the Table Empty dialog box.

Type System

Syntax **dlgEmpty** (const ***tableName*** String)

Description **dlgEmpty** displays the Table Empty dialog box (described in the *User's Guide*), just as if you had chosen File | Table Utilities | Empty. The variable *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 fill in any necessary information and close the dialog box.

Example The following example invokes the Table Empty dialog box and fills in the *Customer* table name as the table to empty. The user must close the dialog box and confirm the data loss.

```
method pushButton(var eventInfo Event)
; invokes the Table Empty dialog box for Customer table
DlgEmpty("Customer.db") ; same as File | Table Utilities |
Empty
endmethod
```

See Also dlgDelete

dlgSubtract

dlgExport

Method Invokes the Table Export dialog box.

Type System

Syntax **dlgExport**(const ***tableName*** String)

Description Invokes the Table Export dialog box, just as if you had chosen File | Utilities | Export. *tableName* specifies the name of the table to export. Refer to the User's Guide or the online Help for more information about exporting data.

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 fill in any necessary information and close the dialog box.

Example The following example invokes the Table Export dialog box for the Orders table.

```
method pushButton(var eventInfo Event)
   var
      tableName String
   endVar

   tableName = "orders.db"

   ; invoke the Table Export dialog box
   dlgExport(tableName)
endMethod
```

See also dlgImportASCIIFix
dlgImportASCIIVar
dlgImportSpreadsheet
exportASCIIFix
exportASCIIVar
exportSpreadsheet

dlgImportASCIIFix

Procedure Invokes the Fixed Length ASCII Import dialog box.

Type System

Syntax **dlgImportASCIIFix**(const ***fileName*** String)

Description Invokes the Fixed Length ASCII Import dialog box, just as if you had chosen File | Utilities | Import, specified Fixed Length for the type, and entered a file name. *fileName* specifies both the name of the source file and the name of the destination table for the imported data. If you specify a file-name extension, Paradox uses it to find the source file, but the file-name extension of the destination table depends on the table type. The default type is Paradox, so the default file-name extension is .DB (for dBASE tables it's .DBF). Dates and numbers are formatted as specified in the Control Panel. Refer to the User's Guide or the online Help for more information about importing data.

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 fill in any necessary information and close the dialog box.

Example The following example invokes the Fixed Length Import 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 Fixed Length ASCII Import dialog box.
   ; by default, Paradox will use ORDERS.TXT as the source
file
   ; and ORDERS.DB as the destination table
   dlgImportASCIIFix(fileName)
endMethod
```

See also dlgExport
dlgImportASCIIVar
dlgImportSpreadsheet
importASCIIFix

dlgImportASCIIVar

Procedure Invokes the Delimited ASCII Import dialog box.

Type System

Syntax **dlgImportASCIIVar**(const ***fileName*** String)

Description Invokes the Delimited ASCII Import dialog box, just as if you had chosen File | Utilities | Import, specified Delimited for the type, and entered a file name. *fileName* specifies both the name of the source file and the name of the destination table for the imported data. If you specify a file-name extension, Paradox uses it to find the source file, but the file-name extension of the destination table depends on the table type. The default type is Paradox, so the default file-name extension is .DB (for dBASE tables it's .DBF). Dates and numbers are formatted as specified in the Control Panel. Refer to the User's Guide or the online Help for more information about importing data.

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 fill in any necessary information and close the dialog box. Following are the default settings: fields are separated by commas, fields are delimited by quotes, only text fields are delimited, and the OEM character set is used.

Example The following example invokes the Delimited Import 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 Fixed Length ASCII Import dialog box.
   ; by default, Paradox will use ORDERS.TXT as the source
file
   ; and ORDERS.DB as the destination table.
   dlgImportASCIIVar(fileName)
endMethod
```

See also dlgExport
dlgImportASCIIFix
dlgImportSpreadsheet
importASCIIVar

dlgImportSpreadsheet

Procedure Invokes the Spreadsheet Import dialog box.

Type System

Syntax **dlgImportSpreadsheet**(const ***fileName*** String)

Description Invokes the Spreadsheet Import dialog box, just as if you had chosen File | Utilities | Import, specified a spreadsheet type, and entered a file name. The *fileName* argument specifies the name of the source file, the spreadsheet type of the source file, and the name of the destination table for the imported data. You must specify a file-name extension; Paradox uses it to find and identify the type of the source file, but the file-name extension of the destination table depends on the table type. The default type is Paradox, so the default file-name extension is .DB (for dBASE tables it's .DBF). Refer to the User's Guide or the online Help for more information about importing data.

Note: The extension you supply as part of *fileName* specifies the format of the spreadsheet file. The following list shows extensions and file formats:

Extension	Format
WB1	Quattro Pro Win
WQ1	Quattro Pro DOS
WKQ	Quattro
WK1	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box. Following are the default settings: 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.

Example The following example invokes the Spreadsheet Import dialog box to import the 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 Spreadsheet Import dialog box
   ; y default, Paradox will use ORDERS.WB1 as the source file
   ; and ORDERS.DB as the destination table
   dlgImportSpreadsheet(fileName)
endMethod
```

See also dlgExport
dlgImportASCIIFix
dlgImportASCIIVar
importSpreadsheet

dlgNetDrivers

Procedure Invokes the Drivers dialog box.

Type System

Syntax **dlgNetDrivers** ( )

Description **dlgNetDrivers** displays the Drivers dialog box (described in the *User's Guide*), just as if you had chosen File | System Settings | Drivers.

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 fill in any necessary information and close the dialog box.

Example The following code opens the Drivers dialog box.

```
; showNetDrivers::pushButton
method pushButton(var eventInfo Event)
; invoke the Drivers dialog box
DlgNetDrivers()  ; same as File | System Settings | Drivers
endmethod
```

See Also Session::enumDriverCapabilities

Session::enumDriverInfo

Session::enumDriverNames

dlgNetLocks

Procedure Creates and displays a table of lock information.

Type System

Syntax **dlgNetLocks** ( )

Description **dlgNetLocks** invokes the Select File dialog box (described in the *User's Guide*), and prompts you to choose a table, just as if you had chosen File | Multiuser | Display Locks. When you choose a table and click OK, Paradox creates a Paradox table named LOCKS.DB in your private directory. If the table 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
Type	S	
UserName	A	14
NetSession	S	
OurSession	S	
RecordNum	N	
Count	S	

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 fill in any necessary information and close the dialog box.

After Paradox creates the *Locks* table, Paradox displays it in a Table window.

Example The following example opens the Select File dialog box. Once the user chooses a file, a *Locks* table is created and displayed.

```
; showNetLocks::pushButton
method pushButton(var eventInfo Event)
; creates a table of lock info :PRIV:LOCKS.DB, then displays
it
DlgNetLocks()  ; same as File | Multiuser | Display Locks
endmethod
```

See Also dlgNetSetLocks
dlgNetRetry
TCursor::enumLocks

dlgNetRefresh

Procedure Invokes the Network Refresh Rate dialog box.

Type System

Syntax **dlgNetRefresh** ( )

Description **dlgNetRefresh** displays the Network Refresh Rate dialog box (described in the *User's Guide*), just as if you had chosen File | System Settings | Auto Refresh.

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 fill in any necessary information and close the dialog box.

Example This example opens the Network Refresh Rate dialog box.

```
; showNetRefresh::pushButton
method pushButton(var eventInfo Event)
; invoke the Network Refresh Rate dialog
DlgNetRefresh()  ; same as File | System Settings | Auto
Refresh
endmethod
```

See Also dlgNetRetry
dlgNetWho

dlgNetRetry

Procedure Invokes the Network Retry Period dialog box.

Type System

Syntax **dlgNetRetry** ( )

Description **dlgNetRetry** displays the Network Retry Period dialog box (described in the *User's Guide*), just as if you had chosen File | Multiuser | Set Retry.

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 fill in any necessary information and close the dialog box.

Example This example opens the Network Retry Period dialog box.

```
; showNetRetryDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Network Retry Period dialog box
DlgNetRetry() ; same as File | Multiuser | Set Retry
endmethod
```

See Also dlgNetLocks

dlgNetSetLocks

Procedure Invokes the Table Locks dialog box.

Type System

Syntax **dlgNetSetLocks** ( )

Description **dlgNetSetLocks** displays the Table Locks dialog box (described in the *User's Guide*), just as if you had chosen File | Multiuser | Set Locks.

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 fill in any necessary information and close the dialog box.

Example This example opens the Table Locks dialog box.

```
; showSetLocks::pushButton
method pushButton(var eventInfo Event)
DlgNetSetLocks()  ; invoke the Table Locks dialog box
                  ; same as choosing File | Multiuser | Set
Locks
endmethod
```

See Also dlgNetLocks

dlgNetSystem

Procedure Invokes the ODAPI System Information dialog box.

Type System

Syntax **dlgNetSystem** ( )

Description **dlgNetSystem** displays the ODAPI System Information dialog box (described in the *User's Guide*), just as if you had chosen File | System | ODAPI.

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 fill in any necessary information and close the dialog box.

Example This code opens the ODAPI System Information dialog box.

```
; showNetSystem::pushButton
method pushButton(var eventInfo Event)
; invoke the ODAPI System Information dialog box
DlgNetSystem()    ; same as File | System | ODAPI
endmethod
```

See Also dlgNetDrivers

Session::enumDriverCapabilities

Session::enumDriverInfo

Session::enumDriverNames

dlgNetUserName

Procedure Invokes the Network User Name dialog box.

Type System

Syntax **dlgNetUserName** ( )

Description **dlgNetUserName** displays the Network User Name dialog box (described in the *User's Guide*), just as if you had chosen File | Multiuser | User 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 fill in any necessary information and close the dialog box.

Example The following code opens the Network User Name dialog box, which shows the current user's network name.

```
; showUserName::pushButton
method pushButton(var eventInfo Event)
; invoke the Network User Name dialog box
DlgNetUserName()   ; same as File | Multiuser | User Name
endmethod
```

See Also dlgNetWho

dlgNetWho

Procedure Invokes the Current Users dialog box.

Type System

Syntax **dlgNetWho** ( )

Description **dlgNetWho** displays the Current Users dialog box (described in the *User's Guide*), just as if you had chosen File | Multiuser | Who.

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.

Example The following code opens the Current Users dialog box.

```
; showUserList::pushButton
method pushButton(var eventInfo Event)
; invoke the Current Users dialog box
DlgNetWho()  ; same as File | Multiuser | Who
endmethod
```

See Also dlgNetUserName

dlgRename

Procedure Invokes the Table Rename dialog box.

Type System

Syntax **dlgRename** (const ***tableName*** String)

Description **dlgRename** displays the Table Rename dialog box (described in the *User's Guide*), just as if you had chosen File | Table Utilities | Rename. The variable *tableName* specifies the name of 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 fill in any necessary information and close the dialog box.

Example The following example invokes the Table Rename dialog box and fills in the *Customer* table name as the table to rename. The user must enter a new name and close the dialog box.

```
; showRenameDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Rename dialog box
dlgRename("customer.db") ; same as File | Table Utilities |
Rename
endmethod
```

See Also dlgCopy
dlgDelete
dlgSort

dlgRestructure

Procedure Invokes the Table Restructure dialog box.

Type System

Syntax **dlgRestructure** (const ***tableName*** String)

Description **dlgRestructure** displays the Table Restructure dialog box (described in the *User's Guide*), just as if you had chosen File | Table Utilities | Restructure. The variable *tableName* specifies the name of table to restructure.

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 fill in any necessary information and close the dialog box.

Example The following example invokes the Table Rename dialog box and fills in the *Customer* table name as the table to restructure. The user must modify the structure and close the dialog box.

```
; showRestructureDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Restructure dialog box for Customer table
DlgRestructure("customer.db")   ; same as File | Table
Utilities | Restructure
endmethod
```

See Also dlgCreate

dlgSort

Procedure Invokes the Table Sort dialog box.

Type System

Syntax **dlgSort** (const ***tableName*** String)

Description **dlgSort** displays the Table Sort dialog box (described in the *User's Guide*), just as if you had chosen File | Table Utilities | Sort. The variable *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 fill in any necessary information and close the dialog box.

Example The following example invokes the Table Sort dialog box and chooses the *Customer* table name as the table to sort. The user must create a sort specification and close the dialog box.

```
; showSortDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Sort dialog box
DlgSort("customer.db")      ; same as File | Table Utilities |
Sort
endmethod
```

See Also dlgRename

dlgSubtract

Procedure Invokes the Table Subtract dialog box.

Type System

Syntax **dlgSubtract** (const ***tableName*** String)

Description **dlgSubtract** displays the Table Subtract dialog box (described in the *User's Guide*), just as if you had chosen File | Table Utilities | Subtract. The variable *tableName* specifies the name of table to subtract records from.

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 fill in any necessary information and close the dialog box.

Example The following example invokes the Table Subtract dialog box and fills in the *Customer* table name as the source table---the table with records to subtract. The user must fill in the name of the table to subtract records from and close the dialog box.

```
; showSubtractDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Subtract dialog box
DlgSubtract("customer.db")  ; File | Table Utilities |
Subtract
endmethod
```

See Also dlgAdd

dlgDelete

dlgTableInfo

Procedure Invokes the Structure Information dialog box.

Type System

Syntax **dlgTableInfo** (const ***tableName*** String)

Description **dlgTableInfo** invokes the Structure Information dialog box, just as if you had chosen File | Table Utilities | Info Structure. The variable *tableName* specifies the name of the table from which to subtract records.

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 fill in any necessary information and close the dialog box.

Example The following example invokes 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") ; same as File | Table Utilities |
Info Structure
endmethod
```

See Also dlgCreate
dlgRestructure

enumDesktopWindowNames

Procedure Creates a table listing open Paradox windows.

Type System

Syntax **1. enumDesktopWindowNames** (const ***tableName*** String)

2. enumDesktopWindowNames (const ***windowNames*** Array[] String)

Description **enumDesktopWindowNames** lists open Paradox window names. Syntax 1 creates the Paradox table named in *tableName* listing the name, class, position, and size of each open window in the user's system. The table lists all applications opened by Paradox. By default, the table is created 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
WindowName	A	32
ClassName	A	32
Position	A	12
Size	A	12
Handle	S	

Syntax 2 fills the array named in *winArray* with the names of the applications. You must declare the array 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 application in the second layer is listed second, and so on.

Compare this method to **enumWindowNames**, which lists all Windows applications running on the user's system.

Example This example writes the open desktop window titles to an array and shows the array. Next, the method 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
```

See Also enumFormNames

enumReportNames

enumWindowNames

enumFonts

Procedure Creates a table listing fonts in the user's system.

Type System

Syntax **enumFonts** (const ***tableName*** String)

Description **enumFonts** creates a Paradox table *tableName* listing fonts in the user's system. By default, this method creates the table in the user's working directory. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

Here is the table structure:

Field Name	Type	Size
FaceName	A	64
FontSize	A	8
Attribute	A	64

Example This example calls **enumFonts** to list the fonts to a table named FONTS.DB. Then it searches a TCursor for a font named Modern. If Modern is in the table, it 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
```

See Also enumRTLConstants

enumFormNames

Procedure Creates an array listing open forms.

Type System

Syntax **enumFormNames** (var ***formNames*** Array[] String)

Description **enumFormNames** fills the array named in *formNames* with the names of the forms opened in the user's desktop. You must 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.

Example The code in this example writes the open forms, reports, and libraries to an array named *openForms*, then views the *openForms* array.

```
; getFormNames::pushButton
method pushButton(var eventInfo Event)
var
  openForms Array[] String
endVar
enumFormNames(openForms)
openForms.view()          ; lists forms, reports, and libraries
endmethod
```

See Also enumWindowNames
enumReportNames
enumDesktopWindowNames

enumReportNames

Procedure Creates an array listing open reports.

Type System

Syntax **enumReportNames** (var ***reportNames*** Array[] String)

Description **enumReportNames** fills the array named in *reportNames* with the names of the reports open in the user's desktop. You must 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.

Example The code in this example writes the open forms, reports, and libraries to an array named *openReports*, then views the *openReports* array.

```
; getReportNames::pushButton
method pushButton(var eventInfo Event)
var
  openReports Array[] String
endVar
enumReportNames(openReports)
openReports.view()           ; lists forms, reports
endmethod
```

See Also enumFormNames
enumDesktopWindowNames
enumWindowNames

enumRTLClassNames

Procedure Creates a table listing the object classes known to ObjectPal.

Type System

Syntax **enumRTLClassNames** (const ***tableName*** String) Logical

Description **enumRTLClassNames** creates a Paradox table *tableName* listing the names of all object classes in the ObjectPAL run-time library. By default, this table is created 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
ClassName	A	32*.

Example This example writes the run-time library class names to a table named *Rtlclass* and views the table.

```
; 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
```

See Also enumRTLConstants
enumRTLMethods

enumRTLConstants

Procedure Creates a table listing the constants defined by ObjectPal.

Type System

Syntax **enumRTLConstants** (const ***tableName*** String) Logical

Description **enumRTLConstants** creates a Paradox table *tableName* listing all the constants defined in the ObjectPAL run-time library. By default, this table is created 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
GroupName	A	32*
ConstantName	A	48*
Type	A	48
Value	A	48

Note: Although Paradox provides the values of constants, you should not use the constant's value in code: refer to constants by name. Use the **constantValueToName** and **constantNameToValue** methods to convert values to constants, if necessary.

Example This example writes the run-time library constant descriptions to a table named *Rtlconst* and views the table.

```
; getRTLConsts::pushButton
method pushButton(var eventInfo Event)
var
  tempTV TableView
endVar
enumRTLConstants("rtlconst.db")     ; write cconstants names
to table
tempTV.open("rtlconst")             ; show the table
endmethod
```

See Also constantValueToName
constantNameToValue
enumRTLClassNames
enumRTLMethods

enumRTLMethods

Procedure Creates a table listing the methods in ObjectPAL.

Type System

Syntax **enumRTLMethods** (const ***tableName*** String) Logical

Description **enumRTLMethods** creates a Paradox table *tableName* listing all the methods defined in the ObjectPAL run-time library. By default, this table is created 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 table structure:

Field Name	Type	Size
ClassName	A	32*
MethodType	A	8*
MethodName	A	64*
MethodArgs	A	255*
ReturnType	A	32*

Example This example writes the run-time library method descriptions to a table named *Rtlmeth* and views the table.

```
; 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
```

See Also enumRTLClassNames
enumRTLConstants

enumWindowNames

Procedure Creates a table or an array listing open windows.

Type System

Syntax **1. enumWindowNames** (const ***tableName*** String) Logical

2. enumWindowNames (var ***windowNames*** Array[] String)

Description **enumWindowNames** creates a list of the applications currently running under Windows on the user's system.

Syntax 1 creates the Paradox table named in *tableName* listing the name, class, position, and size of each open window in the user's system. The table lists windows opened by any application, not just Paradox. By default, the table is created 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
WindowName	A	32
ClassName	A	32
Position	A	12
Size	A	12
Handle	S	

Syntax 2 fills the array named in *winArray* with the names of the applications. You must declare the array 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 application in the second layer is listed second, and so on.

Compare this method to **enumDesktopWindowNames**, which lists open windows for Paradox only.

Example In this example, the **pushButton** method for a button named *getWindowNames* writes and displays open window information in two ways. First, the method fills an array with the titles of the open windows and displays the array. Then, the method fills a table with descriptions of the open windows and shows the table, as well as the table's 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
```

See Also enumDesktopWindowNames

errorClear

Procedure Clears the error stack.

Type System

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 *ObjectPAL Developer's Guide*.

Example This code clears the error stack.

```
; clearError::pushButton
method pushButton(var eventInfo Event)
errorClear()            ; clear the error stack
endmethod
```

See Also errorCode
errorMessage
errorPop

errorCode

Beginner

Procedure Returns a number describing the most recent run-time error or error condition.

Type System

Syntax **errorCode** ( ) SmallInt

Description **errorCode** returns an integer describing the most recent run-time error or error condition. ObjectPAL provides constants for these integers (for example, peObjectNotFound), so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose Errors. The constants appear in the Constants column.

Example The method in this example uses a try clause to attempt to attach to an object named *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
```

See Also errorLog
errorMessage
errorPop

errorLog

Procedure Adds information to the error stack.

Type System

Syntax **errorLog** (const ***errorCode*** SmallInt, const ***errorMessage*** String)

Description **errorLog** adds (pushes) the error information specified in *errorCode* and *errorMessage* onto the error stack.

For more information about the error stack, refer to the *ObjectPAL Developer's Guide*.

Example The method in this example uses a try clause to attempt to attach to an object named *boxOne* on 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
```

See Also errorCode
errorMessage
errorPop

errorMessage

Beginner

Procedure	Returns the text of the most recent error message.
Type	System
Syntax	**errorMessage** ( ) String
Description	**errorMessage** returns a string containing the message displayed by the most recent run-time error or error condition. If no error occurred, **errorMessage** returns the empty string (""). This method is useful for logging error messages during a session.
Example	See the example for errorLog.
See Also	errorLog errorCode errorPop

errorPop

Procedure	Removes the top layer of information from the error stack.
Type	System
Syntax	**errorPop** ( ) Logical
Description	**errorPop** removes the top layer (most recently added error code and error message) from the error stack, giving access to the layer below. For more information about the error stack, refer to the *ObjectPAL Developer's Guide*.
Example	See the example for errorLog.
See Also	errorLog errorCode errorMessage errorShow

errorShow

Procedure Displays the error dialog box.

Type System

Syntax **errorShow** ([const ***topHelp*** String [, const ***bottomHelp*** String]]) Logical

Description **errorShow** opens the Error dialog box and displays the current error information. The string supplied in *topHelp* is used to label the top portion of the dialog box; *bottomHelp* labels the bottom portion of the dialog box.

For more information about the error stack, refer to the *ObjectPAL Developer's Guide*.

Example In this example the *tryAnError* button logs several errors to the error stack, then displays them with **errorShow**.

```
; 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
```

See Also errorCode
errorLog
errorMessage

errorTrapOnWarnings

Beginner

Procedure Specifies whether to handle warning errors as critical errors.

Type System

Syntax **errorTrapOnWarnings** (const ***yesNo*** Logical)

Description **errorTrapOnWarnings** specifies whether to handle warning errors as critical errors. By default, warning errors cannot be trapped in a **try...onFail** block. Specifying **errorTrapOnWarnings(Yes)** tells Paradox to trap warning errors as well as critical errors. For more information about warning errors and critical errors, refer to the *ObjectPAL Developer's Guide*.

Example The following code attempts to open a bogus form: when **errorTrapOnWarnings** is set to No (the default), no error results from this attempt. Once **errorTrapOnWarnings** is set to Yes, the same code generates an error message.

```
; 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
```

See Also errorLog
errorCode
errorMessage
errorShow

execute

Beginner

Procedure Executes a DOS command.

Type System

Syntax **execute** (const ***programName*** String [, const ***wait*** Logical [, const ***displayMode*** SmallInt]]) Logical

Description **execute** runs the DOS command specified in *programName*. The optional argument *wait* specifies whether ObjectPAL suspends execution until the user closes the program. The optional argument *displayMode* specifies the video display mode to use when executing the command.

If the command or program is not in the user's path, you must specify the path in *programName*. Use double backslashes in path names.

Example The following example launches the Windows Clock application with the default window style and waits the user to close it before resuming execuction.

```
; showClock::pushButton
method pushButton(var eventInfo Event)
execute("clock.exe", Yes, WinStyleDefault)  ; execute Windows
Clock
endmethod
```

See Also play

exit

Beginner

Procedure Closes Paradox and exits to Windows.

Type System

Syntax **exit** ( )

Description **exit** closes Paradox and exits to Windows. If a Paradox application has changed, the user is prompted to save it.

Example In this example, the **menuAction** method for the form traps for a MenuFileExit or MenuControlClose action. If the user attempts to exit Paradox, the method first asks for confirmation; if the user confirms, the method uses **exit** to close Paradox.

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    ; trap for File | Exit and confirm before closing
    if eventInfo.id() = MenuFileExit OR
       eventInfo.id() = MenuControlClose  then
      if msgYesNoCancel("Exit", "Do you want to quit?") =
"Yes" then
        exit()          ; leave
      else
        disableDefault  ; if user escapes, or chooses Cancel
or No, then
                        ; block the action
      endIf
    endIf
  else
    ; code here executes just for form itself
  endif
endmethod
```

See Also close

exportASCIIFix

Procedure Exports data from a table to a text file with fixed-length fields.

Type System

Syntax **exportASCIIFix**(const ***tableName*** String, const ***fileName*** String, const ***specTableName*** String [, const ***ANSI*** Logical])

Description Exports data from a table to an ASCII (text) file in which fields of each record are the same length. Using this procedure, you can write ObjectPAL code to duplicate the functionality of the Fixed Length ASCII Export dialog box.

tableName specifies the table to export from, and *fileName* specifies the file to export to. *TableName* is the name of a table that specifies the layout of the exported data. The structure of this table is

Field Name A25, Start S, Length S.

This table serves the same purpose as EXPORT.DB, which is created automatically in your private directory when you use the Fixed Length ASCII Export Dialog box to export a table interactively. Use this table to define the length of each field in the exported file. For each field you export, enter a Start position (the column in the exported file where you want the field value to begin) and a Length (how many characters to display).

ANSI 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. (OEM stands for Original Equipment Manufacturer and refers to the character set your computer uses; ANSI stands for American National Standards Institute and refers to the character set supported by Windows.)

Example The following code exports the data in the Orders table to a text file named ORDERS.TXT. It reads the export format from the Orderexp table, and it exports the data using the ANSI character set.

```
method pushButton(var eventInfo Event)
   exportASCIIFixed("orders.db", "orders.txt", "orderexp.db",
True)
endMethod
```

See also dlgExport
exportASCIIVar
exportSpreadsheet

exportASCIIVar

Procedure Exports a table to a delimited (variable field length) text file.

Type System

Syntax **exportASCIIVar**(const ***tableName*** String, const ***fileName*** String, const ***separator*** String [, const ***delimiter*** String, const ***allFieldsDelimited*** String , const ***ANSI*** Logical])

Description Exports data from a table to an ASCII (text) file in which the table's field values determine the length of each line. Using this procedure, you can write ObjectPAL code to duplicate the functionality of the Delimited ASCII Export dialog box.

tableName specifies the table to export from, and *fileName* specifies the file to export to. Use *separator* to specify the character that separates field values in the exported file. You can choose a comma or any other single character, including special characters such as tabs. Use *delimiter* to specify the characters that surround values in the exported file. You can specify the empty string if you do not want any characters to enclose the values. Use *allFieldsDelimited* to specify whether to surround data from all field types (True) or only from text field types (alphanumeric or character) with the specified delimiter character (False).

Note: Paradox cannot export memo (Paradox or dBASE), formatted memo, graphic, OLE, or binary field types to delimited text.

ANSI 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. (OEM stands for Original Equipment Manufacturer and refers to the character set your computer uses; ANSI stands for American National Standards Institute and refers to the character set supported by Windows.)

Following are the default settings for the optional arguments:

separator is "," (comma)

delimiter is "\"" (double quote)

allFieldsDelimited is False

ANSI is False

Example The following code exports the data from the Orders table to a text file named ORDERS.TXT. It uses tabs to delimit field values, it uses percent signs to enclose each value, it delimits only text fields, and it uses the ANSI character set.

```
method pushButton(var eventInfo Event)
   exportASCIIVar("orders.db", "orders.txt", "\t", "%", False,
True)
endMethod
```

See also dlgExport
exportASCIIFix
exportSpreadsheet

exportSpreadsheet

Procedure Exports the data from a table to a spreadsheet file.

Type System

Syntax **exportSpreadsheet**(const ***tableName*** String, const ***fileName*** String [, const ***makeRowHeaders*** Logical])

Description Exports the data from a table to a spreadsheet file. Using this procedure, you can write ObjectPAL code to duplicate the functionality of the Spreadsheet Export dialog box. 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.

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 to export from, and *fileName* specifies the file to export to. *makeRowHeaders* specifies whether to use the table's column headers to label the corresponding rows of values in the spreadsheet file: set it to True to use column headers as labels, or set it to False if you don't want labels. If you omit this argument, it is True by default.

Note: The extension you supply as part of *fileName* specifies the format of the spreadsheet file. The following list shows extensions and file formats:

Extension	Format
WB1	Quattro Pro Win
WQ1	Quattro Pro DOS
WKQ	Quattro
WK1	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0

Example The following code exports the data from the Orders 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
```

See also dlgExport
exportASCIIFix
exportASCIIVar

fail

Procedure Causes a method to fail.

Type System

Syntax **fail** ([const ***errorNumber*** SmallInt, const ***errorMessage*** String])

Description **fail** causes a method to fail. Executing this method in the **onFail** section of a **try...onFail** block forces a jump to the next highest block, if it exists, or to the implicit **try...onFail** block ObjectPAL wraps around every method. The optional argument *errorNumber* specifies an error code for the failure. The optional argument *errorMessage* specifes a message to display.

ObjectPAL provides constants for error codes, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window, choose Language | Constants, then, from the Types of Constants column, choose Errors. The constants appear in the Constants column.

For more information about using **try...onFail** blocks, refer to the *ObjectPAL Developer's Guide*.

Example In this example, assume that you want to vary the position of a box called *rateBox*. The values of an unbound field object named *rateField* range from 1 to 10; the position of *rateBox* is determined by the value in *rateField*. The following code is attached to the **changeValue** method for *rateField*. This method uses a **try...onFail** block to trap for and handle an improper data type entered in an unbound field.

```
; 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
```

See Also errorCode
errorMessage
errorShow

fileBrowser

Procedure Displays the Paradox File Browser and returns the names of one or more files selected by the user.

Type System

Syntax **1. fileBrowser** (var ***selectedFile*** String [, var ***browserInfo***FileBrowserInfo]) Logical

2. fileBrowser (var ***selectedFiles*** Array[] String [, var ***browserInfo*** FileBrowserInfo]) Logical

Description **fileBrowser** displays the Paradox File Browser and returns the names of one or more files returned by the user. ObjectPAL execution suspends until the user closes the File Browser. Use syntax 1 to return one file name in *selectedFile*, and use syntax 2 to return an array of file names in *selectedFiles*, where *selectedFiles* is declared as a resizeable array. For either syntax, you can provide a BrowserParms record containing information to pass to the File Browser (see the second example, following).

Example The first example calls **fileBrowser** twice: the first time, it returns one file name, and if it is the name of a table, 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 the user 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 the user select multiple files
and store
                       ; the file names in an array
manyFiles.view()       ; displays the user's choices
endmethod
```

You can also pass a record as an argument to **fileBrowser** to specify what data the File Browser displays. For example, you can make the File Browser display Paradox tables only, or forms only, or forms and reports.

ObjectPAL provides a special data type, called FileBrowserInfo, that you can use only with the **fileBrowser** procedure. FileBrowserInfo is a Record with the following structure:

```
TYPE FileBrowserInfo =
  Record
    x, y, w, h   SmallInt      ; size of Browser window in
twips
    WindowStyle  LongInt       ; window style (see WindowStyle
constants)
```

```
    AllowableTypes LongInt     ; file type (see following
table)
    SelectedType   LongInt     ; one of the AllowableTypes
    FileFilters    String      ; the filespec in edit box
    Alias          String      ; alias or drive name
    Path           String      ; path relative to Alias
    PathOnly       Logical     ; return path only, no file
name
  endRecord
ENDTYPE
```

This record structure is predefined and built into ObjectPAL, so all you have to do is declare a variable of type FileBrowserInfo and assign values to its fields, as shown in the example---you don't have to declare the type yourself each time you want to use it.

After the call to **fileBrowser**, the Alias, Path, and FileFilter fields are filled in with the values that were in the File Browser dialog box. In other words, you can find out what the user entered in those areas of 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. The following table lists valid ObjectPAL constants to use in the SelectedType and AllowableTypes fields.

Constant	Extension	Description
fbFiles	*.*	All files
fbTable	*.db	Paradox tables
fbQuery	*.qbe	QBE files
fbForm	*.fsl, *.fdl	Paradox forms
fbReport	*.rsl, *.rdl	Paradox reports
fbScript	*.ssl, *.sdl	Paradox scripts
fbGraphic	*.bmp	Bitmap graphics
fbText	*.txt	Text files
fbSQL	*.sql	SQL files
fbAllTables	*.db, *.dbf	User and system tables
fbTableView	*.tv	Table view files
fbParadox	*.db	Paradox tables
fbDBase	*.dbf	dBASE tables
fbASCII	*.txt	Text files
fbQuattroProWindows	*.wt1	Quattro Pro for Windows worksheets
fbQuattroPro	*.wq1	Quattro Pro for DOS worksheets
fbQuattro	*.wkq	Quattro worksheets
fbLotus2	*.wk1	Lotus worksheets (version 2)
fbLotus1	*.wks	Lotus worksheets (version 1)
fbExcel	*.xls	Excel worksheets
fbConfig	*.cfg	Configuration files
fbLibrary	*.lsl	Paradox libraries

The **fileBrowser** procedure looks only at the names given in the structure. You can pass a different record structure to it and it finds the fields with the appropriate names and uses them. In other words, you can define a simpler record structure with only the items you are interested in.

Attach the following code to a button's built-in **pushButton** method. When it executes, it invokes 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

; Display the Browser and process the user's selection
if fileBrowser(selectedFile, fbi) then
   message("You selected ", selectedFile," with the path ",
fbi.path)
else
   message("You selected cancel")
endif

endMethod
```

See Also FileSystem

formatAdd

Procedure Adds a format.

Type System

Syntax **formatAdd** (const ***formatName*** String, const ***formatSpec*** String) Logical

Description **formatAdd** creates the format described by *formatSpec* and named *formatName* available to the current session.

Note: Field width (W*n*), alignment (AR, AL, AC), and case specifiers (CU, CL, CC) are not saved with a new format definition; but decimal precision (W.*n*) is saved. See **format** in the String type for a complete description of format specifiers.

Example The following 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", "W15.4, 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
```

See Also formatDelete
formatExist

formatDelete

Procedure Deletes a format.

Type System

Syntax **formatDelete** (const ***formatName*** String) Logical

Description **formatDelete** deletes the named format *formatName* from the current session.

Example This 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
```

See Also formatAdd
formatExist

formatExist

Procedure Reports whether a format exists.

Type System

Syntax **formatExist** (const ***formatName*** String) Logical

Description **formatExist** checks if the format named *formatName* is available for the current session. The method returns True if the format is available; otherwise, it returns False.

Example In this example, the method checks if a custom format named *FourCurrency* exists; if not, the method adds the new 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", "W15.4, 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
```

See Also formatAdd
formatDelete

formatSetCurrencyDefault

Procedure	Sets the default display format for Currency values.
Type	System
Syntax	**formatSetCurrencyDefault** (const ***formatName*** String) Logical
Description	**formatSetCurrencyDefault** sets the default format for displaying Currency values. This setting remains in effect for the duration of the session.
Example	See the example for formatExist.
See Also	formatSetNumberDefault

formatSetDateDefault

Procedure Sets the default display format for Date values.

Type System

Syntax **formatSetDateDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying Date values. This setting remains in effect for the duration of the session.

Example In the following example, the **pushButton** method for the *setDateFormat* button sets the default display format for Date values to the Windows Long format. The method then uses **view** to display a date; the date is shown in the new default 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
```

See Also formatSetTimeDefault
formatSetDateTimeDefault

formatSetDateTimeDefault

Procedure Sets the default display format for DateTime values.

Type System

Syntax **formatSetDateTimeDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying DateTime values. This setting remains in effect for the duration of the session.

Example In the following example, the **pushButton** method for the *setDateTimeFormat* button sets the default display format for DateTime values. The method then uses **view** to display a DateTime value; the value is shown in the new default 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
```

See Also formatSetDateDefault
formatSetTimeDefault

formatSetLogicalDefault

Procedure Sets the default display format for Logical values.

Type System

Syntax **formatSetLogicalDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying Logical values. This setting remains in effect for the duration of the session.

Example In the following example, the **pushButton** method for the *setLogicalFormat* button sets the default display format for Logical values to the Male/Female format. The method then uses **view** to display a logical value; the value is shown in the new default 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
```

See Also formatAdd

formatSetLongIntDefault

Procedure Sets the default display format for LongInt values.

Type System

Syntax **formatSetLongIntDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying LongInt values. This setting remains in effect for the duration of the session.

Example In the following example, the **pushButton** method for the *setIntegerFormat* button sets the default display format for LongInt values to the Integer format. The method then uses **view** to display a long integer; the value is shown in the new default 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
```

See Also formatSetSmallIntDefault

formatSetNumberDefault

Procedure Sets the default display format for Number values.

Type System

Syntax **formatSetNumberDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying Number values. This setting remains in effect for the duration of the session.

Example In the following example, the **pushButton** method for the *setNumberFormat* button sets the default display format for Number values to the Scientific format. The method then uses **view** to display a number; the value is shown 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 3.4893+e3
else
  msgStop("Stop", "Requested format does not exist.")
endif
endmethod
```

See Also formatSetCurrencyDefault

formatSetSmallIntDefault

Procedure Sets the default display format for SmallInt values.

Type System

Syntax **formatSetSmallIntDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying SmallInt values. This setting remains in effect for the duration of the session.

Example In the following example, the **pushButton** method for the *setSmallIntFormat* button sets the default display format for SmallInt values to the Integer format. The method then uses **view** to display a small integer; the value is shown 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
```

See Also formatSetLongIntDefault

formatSetTimeDefault

Procedure Sets the default display format for Time values.

Type System

Syntax **formatSetTimeDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying Time values. This setting remains in effect for the duration of the session.

Example In the following example, the **pushButton** method for the *setTimeFormat* button sets the default display format for Time values to the format *hh:mm:ss am*. The method then uses **view** to display a time; the value is shown 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
```

See Also formatSetDateDefault
formatSetDateTimeDefault

getMouseScreenPosition

Procedure Returns the mouse position as a Point.

Type System

Syntax **getMouseScreenPosition** ( ) Point

Description **getMouseScreenPosition** returns the coordinates (in twips) of the pointer relative to the screen, not the Desktop. Use Point type methods (for example **getX** and **getY**) to get more information.

This method gets the mouse position at the time of the event; the current mouse position may be different.

Example In this example, when the user clicks the *nervousMouse* button, the mouse jumps one inch down and one inch to the left.

```
; 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
```

See Also setMouseScreenPosition

helpOnHelp

Procedure	Displays information about how to use the Help system.
Type	System
Syntax	**helpOnHelp** ( ) Logical
Description	**helpOnHelp** displays information explaining how to use the Windows Help application, opening the application if necessary.
Example	See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.
See Also	helpShowContext helpShowIndex

helpQuit

Procedure	Tells the Help application it is no longer needed.
Type	System
Syntax	**helpQuit** (const ***helpFileName*** String) Logical
Description	**helpQuit** tells the Help application that Help is no longer needed. If no other applications have asked for Help, Windows closes the Help application.
Example	See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.
See Also	helpOnHelp

helpSetIndex

Procedure	Sets the help index.
Type	System
Syntax	**helpSetIndex** (const ***helpFileName*** String, const ***indexId*** LongInt) Logical
Description	**helpSetIndex** tells the Windows Help application to set the current index to *indexID* in *helpFileName*.
Example	See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.
See Also	helpShowIndex

helpShowContext

Procedure	Displays help for a particular context.
Type	System
Syntax	**helpShowContext** (const ***helpFileName*** String, const ***helpId*** LongInt) Logical
Description	**helpShowContext** tells the Windows Help application to search *helpFileName* for *helpId* and display the associated help.
Example	See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.
See Also	helpShowIndex helpShowTopic

helpShowIndex

Procedure	Displays the index of a specifed Help file.
Type	System
Syntax	**helpShowIndex** (const ***helpFileName*** String) Logical
Description	**helpShowIndex** displays the index to the file *helpFileName*.
Example	See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.
See Also	helpShowContext helpShowTopic

helpShowTopic

Procedure	Displays help for a specified context.
Type	System
Syntax	**helpShowTopic** (const ***helpFileName*** String, const ***topicKey*** String) Logical
Description	**helpShowTopic** tells the Windows Help application to search *helpFileName* for *topicKey* and display the associated help.
Example	See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.
See Also	helpShowTopicInKeywordTable helpShowContext helpShowIndex

helpShowTopicInKeywordTable

Procedure	Displays help for a topic identified by a keyword in an alternate keyword table.
Type	System
Syntax	**helpShowTopicInKeywordTable** (const ***helpFileName*** String, const ***keyTableLetter*** String, const ***topicKey*** String) Logical
Description	**helpShowTopicInKeywordTable** tells the Windows Help application to search the file *helpFileName* for *keyTableLetter* and *topicKey* and display the associated help.
Example	See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.
See Also	helpShowTopic helpShowContext helpShowIndex

importASCIIFix

Procedure Imports data to a table from an ASCII (text) file in which fields of each record are the same length.

Type System

Syntax **importASCIIFix**(const ***fileName*** String, const ***tableName*** String [, const ***spec*** String, const ***ANSI*** Logical])

Description Imports data to a table from an ASCII (text) file in which fields of each record are the same length. Using this procedure, you can write ObjectPAL code to duplicate the functionality of the Fixed Length ASCII Import dialog box.

fileName specifies the file to import from, and *tableName* specifies the table to import to. Dates and numbers are formatted as specified in the Control Panel.

Note: The extension you supply as part of the *tableName* argument specifies the table type. Use .DB to specify a Paradox table, use .DBF to specify a dBASE table. If you omit the extension, the data is imported to a Paradox table by default.

spec is the name of a table that specifies the layout for the imported data. The structure of this table is

Field Name A25, Type A4, Start S, Length S.

This table serves the same purpose as IMPORT.DB, which is created automatically in your private directory when you use the Fixed Length ASCII Import Dialog box to import a table interactively. Use this table to define the structure of the table that will receive the imported data. For each field you import, enter a name, a type (must be a valid Paradox or dBASE field specification; see Table::create for details), a Start position (the column where you want the field value to begin) and a Length (the field size).

ANSI 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. If you omit this argument, Paradox uses the OEM character set by default. (OEM stands for Original Equipment Manufacturer and refers to the character set your computer uses; ANSI stands for American National Standards Institute and refers to the character set supported by Windows.)

Example The following code imports the data from a text file named ORDERS.TXT to the Orders table. It reads structure information from the Orderimp table, and it imports the data using the OEM character set.

```
method pushButton(var eventInfo Event)
   importASCIIFixed("orders.txt", "orders.db", "orderimp.db",
False)
endMethod
```

See also importASCIIVar

importSpreadsheet

importASCIIVar

Procedure Imports data from a delimited (variable field length) text file to a table.

Type System

Syntax **importASCIIVar**(const ***fileName*** String, const ***tableName*** String [, const ***separator*** String, const ***delimiter*** String, const ***allFieldsDelimited*** String, const ***ANSI*** Logical])

Description Imports data from a delimited ASCII (text) file in which the field values are delimited by a specified character. Using this procedure, you can write ObjectPAL code to duplicate the functionality of the Delimited ASCII Import dialog box.

fileName specifies the file to import from, and *tableName* specifies the table to import to. Dates and numbers are formatted as specified in the Control Panel.

Note: The extension you supply as part of *tableName* specifies the table type. Use .DB to specify a Paradox table, use .DBF to specify a dBASE table. If you omit the extension, the data is imported to a Paradox table by default.

Use the *separator* argument to specify the character that separates field values in the text file. You can choose a comma or any other single character, including special characters such as tabs. Use delimiter to specify the characters that surround values in the exported file. You can specify the empty string if you do not want any characters to enclose the values. Use *allFieldsDelimited* to specify whether to surround data from all field types (True) or only from text field types (alphanumeric or character) with the specified delimiter character (False). These are the default settings: fields are separated by commas, fields are delimited by quotes, only text fields are delimited, and the OEM character set is used.

Note: Paradox trims strings longer than 255 characters when it imports them.

ANSI 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. If you omit this argument, Paradox uses the OEM character set by default. (OEM stands for Original Equipment Manufacturer and refers to the character set your computer uses; ANSI stands for American National Standards Institute and refers to the character set supported by Windows.)

Example The following code imports the data from a text file named ORDERS.TXT to the Orders table. It uses commas to delimit field values, it does not enclose each value, it delimits all fields, and it uses the ANSI character set.

```
method pushButton(var eventInfo Event)
   importASCIIVar("orders.txt", "orders.db", ",", "", True,
True)
endMethod
```

See also importASCIIFix

importSpreadsheet

importSpreadsheet

Procedure Imports the data from a table to a spreadsheet file.

Type System

Syntax **importSpreadsheet**(const ***fileName*** String, const ***tableName*** String, const ***fromCell*** String, const ***toCell*** String [, const ***getFieldNames*** Logical])

Description Imports the data from a spreadsheet file to a table. Using this procedure, you can write ObjectPAL code to duplicate the functionality of the Spreadsheet Import dialog box. When you import data from a spreadsheet, Paradox converts each row to a record and each column to a field.

fileName specifies the file to import from, and *tableName* specifies the table to import to. *fromCell* specifies the upper left cell of the block to import, and *toCell* specifies the lower right cell. *getFieldNames* specifies whether to use the top row of the spreadsheet as column headers for the table (True), or not (False). If you omit this argument, the default value is True.

Note: The extension you supply as part of the fileName argument specifies the format of the spreadsheet file. The following list shows extensions and file formats:

Extension	Format
WB1	Quattro Pro Win
WQ1	Quattro Pro DOS
WKQ	Quattro
WK1	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0

Note: The extension you supply as part of *tableName* specifies the table type. Use .DB to specify a Paradox table, and use .DBF to specify a dBASE table. If you omit the extension, the data is imported to a Paradox table by default.

When you import data from a spreadsheet, 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	Alphanumeric	Character
Integer	Short number	Float number (5,0)
Number	Numeric	Float number (20,4)
Currency	Currency	Float number (20,4)
Date	Date	Date

The following rules determine which category a column falls into:

- Any column that contains a label (text) is converted to an alphanumeric field (or character field if you import to a dBASE table).
- A column containing both dates and numbers is converted to an alphanumeric field (or character field if you import to a dBASE table).
- A column containing only values formatted as currency is converted to a currency 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 numbers in unedited

spreadsheets as alphanumeric fields. For example, spreadsheet columns often have rows of hyphens separating sections of numbers. Since only an alphanumeric field can have both numbers and hyphens, the column is converted to an alphanumeric field even though it contains mostly numbers.

To avoid conversion problems, edit the spreadsheet before importing it. This eliminates any ambiguities. 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 column titles 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 on the spreadsheet, uncheck the Get field names from first row check box in the Spreadsheet Import dialog box.)

If the table does not have the format you want after you import it, you can restructure it in Paradox.

Example

The following code imports the data from a Quattro Pro for Windows file to the Orders table. It 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
```

See also

importASCIIFix

importASCIIVar

message

Beginner

Procedure Displays a message in the status line.

Type System

Syntax **message** (const ***message*** String [, const ***message*** String]*)

Description **message** displays a message composed of up to six strings in the status line.

Example The following method 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
```

See Also msgInfo
msgQuestion
msgStop

msgAbortRetryIgnore

Procedure Displays a dialog box containing a message and three buttons: Yes, No, and Ignore.

Type System

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 to the user. The return value is a string corresponding to the button clicked: "Abort", "Retry", or "Ignore". If the user presses Esc or clicks the Close box, the return value is "Cancel".

Example In this example, the *showAbortRetryIgnore* button warns the user that an operation may take a long time, and asks the user 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()
; execute a custom method based on the user's choice
switch
   case doThis = "Abort"  : message("Aborting operation.")
   case doThis = "Retry"  : message("Retrying operation.")
   case doThis = "Ignore" : message("Ignoring problem.")
   case doThis = "Cancel" : message("Cancelling operation.")
endSwitch
endmethod
```

See Also msgRetryCancel
msgYesNoCancel

msgInfo

Beginner

Procedure Displays a dialog box containing the information icon, a message, and an OK button.

Type System

Syntax **msgInfo** (const ***caption*** String, const ***text*** String)

Description **msgInfo** displays a one-button dialog box. The text in *caption* is displayed in the title bar, and *text* is displayed in the box itself. The user must click OK or press Esc to close the box. This method does not return a value.

Example This example uses the **msgInfo** method to display a fact to the user.

```
; showMsgInfo::pushButton
method pushButton(var eventInfo Event)
msgInfo("Trivia", "The capital of Oregon is Salem.")
endmethod
```

See Also msgQuestion
msgStop

msgQuestion

Beginner

Procedure Displays a dialog box containing a message, a question mark icon, a Yes button, and a No button.

Type System

Syntax **msgQuestion** (const ***caption*** String, const ***text*** String) String

Description **msgQuestion** displays a two-button dialog box. The text in *caption* is displayed in the title bar, and *text* is displayed in the box itself. The return value corresponds to the button the user clicks to close the dialog box: "Yes" or "No". If the user presses Esc or clicks the Close box, the return value is "Cancel".

Example The following code asks the user whether to change the desktop title. If the user presses the Yes button, 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 the
desktop title
     sleep(2000)                              ; pause
     thisApp.setTitle("Paradox for Windows")  ; restore it
   case userChoice = "No"  :
     message("Application title not changed.")
endSwitch
endmethod
```

See Also msgInfo
msgStop

msgRetryCancel

Procedure Displays a dialog box containing a message and two buttons: Retry and Cancel.

Type System

Syntax **msgRetryCancel** (const ***caption*** String, const ***text*** String) String

Description **msgRetryCancel** displays a two-button dialog box where *caption* specifies the text in the dialog box title bar, and *text* specifies a message to display to the user. The return value corresponds to the button the user clicks: "Retry" or "Cancel". If the user presses Esc or clicks the Close box, the return value is "Cancel".

Example The following example poses a question to the user in response to a problem, then displays a message on the status line depending on how the user answers.

```
; 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
```

See Also msgYesNoCancel
msgAbortRetryIgnore

msgStop

Procedure Displays a dialog box containing a stop sign icon, a message, and an OK button.

Type System

Syntax **msgStop** (const ***caption*** String, const ***text*** String)

Description **msgStop** displays a one-button dialog box. The text in *caption* is displayed in the title bar, and *text* is displayed in the box itself, along with a Stop icon. The user must click OK or press Esc to close the box. This method does not return a value.

Example In this example, the **pushButton** method for *showMsgStop* alerts the user of 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
```

See Also msgInfo
msgQuestion

msgYesNoCancel

Procedure Displays a dialog box containing a message and three buttons: Yes, No, and Cancel.

Type System

Syntax **msgYesNoCancel** (const ***caption*** String, const ***text*** String) String

Description **msgYesNoCancel** displays a three-button dialog box where *caption* specifies the text in the dialog box title bar, and *text* specifies text to display to the user. The return value corresponds to the button the user clicks: "Yes", "No", or "Cancel". If the user presses Esc or clicks the Close box, the return value is "Cancel".

Example In this example, **msgYesNoCancel** is used to find out whether the user wants 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
```

See Also msgRetryCancel
msgAbortRetryIgnore

pixelsToTwips

Procedure Converts screen coordinates from pixels to twips.

Type System

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; a twip is a device-independent unit equal to 1/1440 of an inch (1/20 of a printer's point).

Example The following example shows the position of the button (*self*) first in twips, then in pixels. The method goes on to find the display resolution for the system (given 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
```

See Also twipsToPixels

play

Beginner

Procedure Plays a standalone script.

Type System

Syntax **play** (const ***scriptName*** String) AnyType

Description **play** executes the statements in the script specified in *scriptName*. A standalone script consists of one or more ObjectPAL statements attached to an object that never appears. You can think of a standalone script as a special kind of custom method. For more information, refer to the *ObjectPAL Developer's Guide*.

Example The following code plays a script named TESTSCR.SSL, which is assumed to be in the working directory.

```
; playAScript::pushButton
method pushButton(var eventInfo Event)
play("Testscr.ssl")
endmethod
```

See Also execute

readEnvironmentString

Procedure Reads an item from the DOS environment.

Type System

Syntax **readEnvironmentString** (const ***key*** String) String

Description **readEnvironmentString** returns a string containing information about the DOS environment variable specified in *key*. 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.

Example This example uses **readEnvironmentString** and **writeEnvironmentString** to get and 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 = fs.getDir()                    ; get the current
directory
myDir = "c:\\pdoxwin\\refwork\\pc0syall"
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
```

See Also readProfileString

writeEnvironmentString

readProfileString

Procedure Reads an item from the user's WIN.INI file.

Type System

Syntax **readProfileString** (const ***fileName*** String, const ***section*** String, const ***key*** String) String

Description **readProfileString** returns a value from a specified section of a specified file on the user's system. This method searches the user's WINDOWS directory by default. Typically, you use this method to read the user's WIN.INI file, so *fileName* is WIN.INI. A section in WIN.INI is marked by a string bounded by square brackets on a line by itself (for example, [windows]). However, omit the brackets when you specify *section*; that is, to specify the [windows] section, use "windows." Within a section, a value marker is a string followed by an equal sign (for example, Beep =), but don't include the = when you specify *key*.

Example This example uses **readProfileString** and **writeProfileString** to get and 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
```

See Also writeProfileString
readEnvironmentString

setMouseScreenPosition

Procedure	Displays in a the mouse pointer at a specified position.
Type	System
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 *x* and *y* (syntax 2). Coordinates should be specified in twips.
Example	See the example for getMouseScreenPosition.
See Also	getMouseScreenPosition

setMouseShape

Procedure Specifies the shape of the mouse pointer.

Type System

Syntax **setMouseShape** (const ***mouseShapeId*** LongInt) LongInt

Description **setMouseShape** specifies in *mouseShapeId* the shape of the mouse pointer. ObjectPAL provides constants for mouse shapes, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose MouseShapes. The constants appear in the Constants column.

Example This example cycles through the available mouse shapes when the user clicks a field.

```
; changeMouseField::mouseUp
method mouseUp(var eventInfo MouseEvent)
beep()
message("Watch carefully--changing mouse shapes.")
setMouseShape(MouseIBeam)
sleep(1000)
setMouseShape(MouseCross)
sleep(1000)
setMouseShape(MouseWait)
sleep(1000)
setMouseShape(MouseUpArrow)
sleep(1000)
setMouseShape(MouseArrow)
endmethod
```

See Also setMouseScreenPosition
getMouseScreenPosition

sleep

Beginner

Procedure Produces a delay of a specified duration.

Type System

Syntax **sleep** ([const ***numberOfMilliSeconds*** LongInt])

Description **sleep** suspends execution of a method for the interval specified in *numberOfMilliSeconds*. Programs other than Paradox continue to execute. Printing continues, too. The Paradox Desktop, including TimerEvents, is suspended for the duration of the sleep.

Note: When called with no argument, **sleep** causes the current process (method) to release control of the processor for an instant so that other processes can complete their tasks. This is called a yield. The Windows environment occasionally requires a yield before it will finish a simultaneous (time-sliced) process.

Example The following code 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
```

See Also Form::wait

sound

Beginner

Procedure Creates a sound of specified frequency and duration.

Type System

Syntax **sound** (const ***freqHertz***, const ***durationMilliSecs*** LongInt)

Description **sound** creates a sound of frequency (in Hertz) *freqHertz* for a time (in milliseconds) specified in *durationMilliSecs*. Frequency values can range between 1 and 50,000 Hertz (the audible limit is approximately 20,000 Hertz).

Example In this 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
```

See Also beep

sysInfo

Procedure Creates a dynamic array of information about the system running Paradox.

Type System

Syntax **sysInfo** (var ***info*** DynArray[] AnyType)

Description **sysInfo** creates a dynamic array *info* of information about the system running Paradox. You must declare the DynArray before calling this method. The DynArray contains indexes for system attributes and their values. The indexes are described in the table below.

Index	Definition
AreMouseButtonsSwapped	Reports whether functions of the left and right mouse buttons are reversed
ConfigFile	The path to ODAPI.CFG
CodePage	Reports which code page is currently loaded by Windows
CPU	Processor type
EngineDate	Creation date of database engine
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
KeyboardSubType	Keyboard subtype is an OEM-dependent value
KeyboardType	Type and manufacturer of the keyboard
LanguageDriver	Default language driver
LocalShare	Reports whether Local Share is active
MathCoprocessor	Reports whether a math coprocessor is present
Memory	Available memory, including swap file (if present) in bytes
Mouse	The number of mice attached to the system
NetDir	The path to PARADOX.NET
NetProtocol	Network protocol
NetShare	Reports whether Net Share is active
NetType	Network type
NumTasks	The number of active tasks (programs)
Protected	Reports whether the system is running in protected mode
ScreenHeight	Total height of screen (in pixels)
ScreenWidth	Total width of screen (in pixels)
UserName	Network user name
WindowsDir	Path to the WINDOWS directory

WindowsSystemDir	Path to the WINDOWS\SYSTEM directory
WindowsVersion	Version number of Windows

Example The following code writes system information to a dynamic array named *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
```

See Also readProfileString

tracerClear

Procedure Clears the Tracer window.

Type System

Syntax **tracerClear** ( )

Description **tracerClear** empties the Tracer window. The Tracer window can be opened by the **tracerOn** procedure at run time, or by selecting the ObjectPAL Editor menu's Debug | Trace Execution command.

Example The following code clears the Tracer window. For this example, assume that the Tracer window is open and contains information.

```
; wipeTracer::pushButton
method pushButton(var eventInfo Event)
tracerClear()                    ; clear the Tracer window
endmethod
```

See Also tracerHide
tracerOn

tracerHide

Procedure Hides the Tracer window.

Type System

Syntax **tracerHide** ( )

Description **tracerHide** makes the Tracer window invisible, but it does not clear or close the Tracer window. To make the Tracer window visible again, use **tracerShow**.

Example The following code hides the Tracer window, pauses, then displays it again. For this example, 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
```

See Also tracerOn

tracerShow

tracerOff

Procedure Closes the Tracer window.

Type System

Syntax **tracerOff** ( )

Description **tracerOff** stops writing code traces to the Tracer window. You can resume tracing code with the **tracerOn** procedure. Tracing is on by default when the Tracer window is first opened.

Example This code turns off code tracing.

```
; stopTracer::pushButton
method pushButton(var eventInfo Event)
tracerOff()                    ; close the Tracer window
endmethod
```

See Also tracerOn
tracerSave

tracerOn

Procedure	Activates code tracing.
Type	System
Syntax	**tracerOn** ( )
Description	**tracerOn** resumes writing code traces to the Tracer window.
Example	This example reactivates code tracing.

```
; startTracer::pushButton
method pushButton(var eventInfo Event)
tracerOn()                          ; reactivate the Tracer window
endmethod
```

See Also tracerOff
tracerShow

tracerSave

Procedure Saves the contents of the Tracer window to a file.

Type System

Syntax **tracerSave** (const ***fileName*** String)

Description **tracerSave** saves the contents of the Tracer window to *fileName*.

Example This example saves the contents of the Tracer window to a file named MYTRACE.TXT.

```
; saveTracerToFile::pushButton
method pushButton(var eventInfo Event)
tracerSave("mytrace.txt")          ; save the Tracer window to
a file
endmethod
```

See Also tracerWrite

tracerShow

Procedure	Makes the Tracer window visible.
Type	System
Syntax	**tracerShow** ( )
Description	**tracerShow** makes the Tracer window visible. You can make the Tracer window invisible with the **tracerHide** procedure.
Example	See the example for tracerHide.
See Also	tracerHide tracerToTop

tracerToTop

Procedure	Makes the Tracer window the topmost window.
Type	System
Syntax	**tracerToTop** ( )
Description	**tracerToTop** brings the Tracer window to the top of the desktop.
Example	See the example for tracerWrite.
See Also	tracerHide tracerShow

tracerWrite

Procedure Writes a message to the Tracer window.

Type System

Syntax **tracerWrite** (const ***message*** String [, const ***message*** String]*)

Description **tracerWrite** writes a message to the Tracer window.

Example The following code 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
```

See Also tracerSave

twipsToPixels

Procedure	Converts screen coordinates from twips to pixels.
Type	System
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; the number of pixels per inch is device-dependent. A twip is a device-independent unit equal to 1/1440 of an inch (1/20 of a printer's point).
Example	See the example for pixelsToTwips.
See Also	pixelsToTwips

version

Procedure Returns the Paradox version number.

Type System

Syntax **version** ( ) String

Description **version** returns a string containing the version number of Paradox currently being used.

Example In this example, the **pushButton** method for *showVersion* tells the user which Paradox version is in use.

```
; showVersion::pushButton
method pushButton(var eventInfo Event)
msgInfo("FYI", "You are running version " + String(version())
+ ".")
endmethod
```

See Also sysInfo

winGetMessageID

Procedure	Returns the ID of a Windows message.
Type	System
Syntax	**winGetMessageID** (const ***msgName*** SmallInt) **SmallInt**
Description	**winGetMessageID** gets the ID of the message **msgName**. This method should be used by Windows programmers only. See your Windows programming documentation for more information.
See Also	winPostMessage winSendMessage MenuEvent**::**data MenuEvent**::**id Form**::**windowClientHandle Form**::**windowHandle

winPostMessage

Procedure	Posts a message to Windows.
Type	System
Syntax	**winPostMessage** (const ***hWnd*** SmallInt, const ***msg*** SmallInt, const ***wParam*** SmallInt, const ***lParam*** LongInt) Logical
Description	**winPostMessage** posts a message to Windows. This method should be used by Windows programmers only. See your Windows programming documentation for more information.
See Also	winGetMessageID winSendMessage MenuEvent**::**data MenuEvent**::**id Form**::**windowClientHandle Form**::**windowHandle

winSendMessage

Procedure	Sends a message to Windows.
Type	System
Syntax	**winSendMessage** (const ***hWnd*** SmallInt, const ***msg*** SmallInt, const ***wParam*** SmallInt, const ***lParam*** LongInt) Logical
Description	**winSendMessage** sends a message to Windows. This method should be used by Windows programmers only. See your Windows programming documentation for more information.
See Also	winGetMessageID winPostMessage MenuEvent**::**data MenuEvent**::**id Form**::**windowClientHandle Form**::**windowHandle

writeEnvironmentString

Procedure	Writes information ato the DOS environment.
Type	System
Syntax	**writeEnvironmentString** (const ***key*** String, const ***value*** String) Logical
Description	**writeEnvironmentString** sets a DOS environment variable. 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.
Example	See the example for readEnvironmentString.
See Also	readEnvironmentString writeProfileString

writeProfileString

Procedure	Writes information about the user's system to a file.
Type	System
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 the user's system. Typically, you use this method to modify the user's WIN.INI file, so *fileName* is WIN.INI. A section in WIN.INI is marked by a string bounded by square brackets on a line by itself (for example, [windows]). However, omit the brackets when you specify *section*; 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.
Example	See the example for readProfileString
See Also	readProfileString writeEnvironmentString

advMatch

Method Searches for a pattern of characters in a text file.

Type TextStream

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 System 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" section.)

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 any of the above 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 line (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.

Example This 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
```

See Also

home
end
setPosition
position
String::advMatch

close

Method Closes a text file.

Type TextStream

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.

Example This 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
```

See Also open
commit

commit

Method Writes the contents of the text buffer to disk.

Type TextStream

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.

Example In this 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

endmethod
```

See Also writeString

create

Method Creates a text file for reading and writing.

Type TextStream

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")
```

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()
  ts.create(":PRIV:rick.txt")  ; creates file, open for
writing and reading
  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
```

See Also open

close

end

Method Sets the pointer to the end of a text file.

Type TextStream

Syntax **end** ( )

Description **end** sets the pointer to the last character of a text file.

Example This 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 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
```

See Also home

setPosition
eof

eof

Method Tests for a move past the end of a text file.

Type TextStream

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.

Example This 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 **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
```

See Also end
position
setPosition

home

Method	Sets the pointer to the beginning of a text file.
Type	TextStream
Syntax	**home** ( )
Description	**home** sets the file pointer to the first character of a text file.
Example	See the example for end.
See Also	end eof setPosition

open

Method Opens a text file in a specified mode.

Type TextStream

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

"a", append and read

"r", read only

"w", write and read

"nw", new file,, read and write

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.

Example This 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
```

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 ts2.
                            ; This is line 2.

endmethod
```

See Also create
close

position

Method Returns the pointer's position in a text file.

Type TextStream

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).

Example It may be helpful to think of position as returning the number of the next character in the file. As this 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)
```

See Also setPosition

size

readChars

Method Reads a specified number of characters from a text file.

Type TextStream

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.

Example This 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
```

See Also readLine

setPosition

The example for advMatch.

readLine

Method Reads a line from a text file.

Type TextStream

Syntax **readLine** (var ***value*** String) Logical

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, and moves the file pointer to the first position after the CR/LF pair. **readLine** begins reading from the current pointer position.

You can store a single line in *value*, or store the entire file in *stringArray,* where *stringArray* is a resizable array of strings and each array item stores one line from the file. In either case, the CR/LF pair is not stored.

This method returns True if successful; otherwise, it returns False.

Example The first 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(1234)
message(ts.position()) ; displays 7 (skips over CR/LF)
sleep(1234)
endmethod
```

The next 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.

```
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()
```

See Also readChars
writeLine

setPosition

Method Positions the pointer in a text file.

Type TextStream

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.

Example In this 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
```

This 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
```

See Also position

size

eof

size

Method Returns the number of characters in a text file.

Type TextStream

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).

Example This 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
```

See Also position
setPosition
eof

writeLine

Method	Writes a string to a text file.
Type	TextStream
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.
Example	See the example for create.
See Also	readLine writeString

writeString

Method Writes a character string to a text file.

Type TextStream

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.

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, Sell High. = 10
myText.close()
endmethod
```

See Also writeLine

blank

Beginner

Method/ Procedure Returns a blank value.

Type AnyType

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.

Example This 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
```

See Also isBlank

dataType

Method Returns a string representing the data type of a variable.

Type AnyType

Syntax **dataType** ( ) String

Description **dataType** returns a string representing the data type of a variable or expression: Binary, Currency, Date, DateTime, Graphic, Logical, LongInt, OLE, Memo, Number, 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.

Example This 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
```

See Also isAssigned

isAssigned

Method Reports whether a variable has been assigned a value.

Type AnyType

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.

Example This 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
```

See Also dataType
unAssign

isBlank

Beginner

Method Reports whether an expression has a blank value.

Type AnyType

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.

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
```

See Also blank

isFixedType

Method Reports whether a variable's data type has been explicitly declared.

Type AnyType

Syntax **isFixedType** ( ) Logical

Description **isFixedType** returns True if the variable has been declared using a **var...Endvar** block; otherwise, it returns False.

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
```

See Also dataType
isAssigned

unAssign

Method Sets a variable's state to unAssigned.

Type AnyType

Syntax **unAssign** ( ) Logical

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.

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
```

See Also blank
isAssigned

view

Beginner

Method Displays in a dialog box, the value of a variable.

Type AnyType

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
Array	yes	no
Binary	no	no
Currency	yes	yes
Date	yes	yes
DateTime	yes	yes
DynArray	yes	no
Graphic	no	no
Logical	yes	yes
LongInt	yes	yes
Memo	no	no
Number	yes	yes
OLE	no	no
Point	yes	yes
Record	yes	no
SmallInt	yes	yes
String	yes	yes
Time	yes	yes

Example This example uses **view** to display in a dialog box, the value of several variables. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dy DynArray[] AnyType
  x AnyType
endVar

dy["Name"]      = "Jasbir Atwal"
dy["Hire Date"] = Date("5/2/84")
dy["Zip"]       = 97509
```

```
dy["Title"]     = "Engineer"

dy.view()          ; displays DynArray indexes and values

x = 5
x.view()           ; displays 5
x = "Hello"
x.view("Hi")       ; displays Hello, title is Hi
endmethod
```

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
```

See Also isAssigned
unAssign

addLast

Method Inserts an item at the end of a resizeable array.

Type 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
```

Example This 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
```

See Also

append
insert
insertAfter
insertBefore
insertFirst

append

Method Appends the contents of one array to another.

Type Array

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.

Example The following code creates two resizeable arrays, *addMe* and *baseArray*, and loads them with numeric values. This 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
```

See Also addLast
insert
insertAfter
insertFirst

contains

Method Searches the items of an array for a pattern of characters.

Type Array

Syntax **contains** (const ***value*** AnyType) Logical

Description **contains** returns True if any item of an array exactly matches *value*; otherwise, it returns False.

Example This 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
    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
  dogs.view("dogs size: " + strVal(dogs.size()))
else                          ; "Bandit" must already exist
  msgInfo("Once is enough", "The dogs array already contains
Bandit.")
endif
```

```
endmethod
```

See Also countOf

countOf

Method Counts the occurrences of a value in an array.

Type 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.

Example This code (attached to a button's **pushButton** method) creates and loads a fixed 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
```

See Also contains

empty

Method Removes all items from an array.

Type 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.

Example This example shows how empty functions for a fixed array. The code immediately following declares a fixed 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 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 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
```

See Also remove
removeAllItems

exchange

Method	Swaps the contents of two cells in an array.
Type	Array
Syntax	**exchange** (const ***index1*** LongInt, const ***index2*** LongInt)
Description	**exchange** swaps the contents of the cells at *index1* and *index2* in an array.
Example	See the example for indexOf.
See Also	contains countOf indexOf

fill

Method Fills an array with a value.

Type Array

Syntax **fill** (const ***value*** AnyType)

Description **fill** assigns *value* to every item of an array.

Example This code creates a fixed 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
```

See Also append
insert

grow

Method Increases the size of a resizeable array.

Type 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.

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
```

See Also addLast
insert
insertFirst
isResizeable

indexOf

Method Returns the position of an item in an array.

Type 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.

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
```

See Also contains
countOf

insert

Method Inserts one or more empty cells into an array.

Type 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.

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
```

See Also insertAfter
insertBefore

insertAfter

Method Inserts an item into an array after a specified item.

Type Array

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.

Example This 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
```

See Also insert
insertBefore

insertBefore

Method Inserts an item into an array before a specified item.

Type Array

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.

Example This 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
```

See Also insert
insertAfter

insertFirst

Method Inserts an item at the beginning of an array.

Type 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.

Example This 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
```

See Also

addLast
append
insert
insertAfter
insertBefore

isResizeable

Method Reports whether an array can be resized.

Type Array

Syntax **isResizeable** ( ) Logical

Description **isResizeable** returns True if an array can be resized; otherwise, it returns False.

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
```

See Also grow
size

remove

Method Removes one or more items from an array.

Type 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).

Example This 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 diasplays myZoo[lion, bear]
myZoo.view("myZoo size: " + strVal(myZoo.size()))

endmethod
```

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
```

See Also

insert
removeItem
removeAllItems

removeAllItems

Method Removes all occurrences of an array item.

Type Array

Syntax **removeAllItems** (const ***value*** AnyType)

Description **removeAllItems** deletes all occurrences of *value* from an array. Indexes of subsequent items are decreased by 1.

Example This 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
```

See Also remove
removeItem

removeItem

Method Deletes a specified item from an array.

Type 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.

Example This 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
```

See Also remove
removeAllItems
replaceItem

replaceItem

Method Overwrites an item in an array with another item.

Type Array

Syntax **replaceItem** (const ***keyItem*** AnyType, const ***newItem*** AnyType)

Description **replaceItem** searches an array for *keyItem*, and replaces the first occurrence of *keyItem* with *newItem*.

Example This 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
```

See Also removeItem

setSize

Method Specifies the size of an array.

Type 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.

Example This 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
```

See Also grow
isResizeable

size

Method	Returns the number of items in an array.
Type	Array
Syntax	**size** ( ) LongInt
Description	**size** returns the total number of items in an array, even if one or more elements are blank.
Example	See the example for setSize.
See Also	grow setSize

view

Method Displays in a dialog box the contents of an array.

Type 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" earlier in this chapter for more information regarding other data types and the **view** method.

Example This 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
```

See Also contains

readFromFile

Method Reads data from a file and stores it in a Binary variable.

Type Binary

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.

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
```

See Also size
writeToFile
Methods and procedures defined for FileSystem.

size

Method Returns the number of bytes in a Binary variable.

Type Binary

Syntax **size** ( ) LongInt

Description **size** returns a value representing the number of bytes stored in a Binary variable.

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
```

See Also readFromFile
writeToFile
Methods and procedures defined for FileSystem.

writeToFile

Method Writes the data stored in a Binary variable to a disk file.

Type Binary

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.

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
```

See Also readFromFile
size
Methods and procedures defined for FileSystem.

currency

Procedure Casts a value as Currency.

Type Currency

Syntax **currency** (const ***value*** AnyType) Currency

Description **currency** casts (converts) the data type of *value* to Currency.

Example In this 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
```

In the next 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
```

See Also Currency

date

Beginner

Procedure Casts a value as a Date.

Type date

Syntax **1. date** (const ***value*** AnyType) Date

2. date ( ) Date

Description **date** casts (converts) *value* as a date. If the the date supplied in *value* is invalid, the method fails. If you do not supply *value*, **date** returns the current date as a Date data type.

Example This 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

d.view()         ; show value of d in a dialog box (11/18/99)
                 ; dialog box title displays "Date"
endmethod
The next example
```

See Also dateVal

dateVal

Procedure Returns a value as a date.

Type Date

Syntax **dateVal** (const ***value*** AnyType) Date

Description **dateVal** returns a value as a date.

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
```

See Also date

today

Procedure	Returns the current date.
Type	Date
Syntax	**today** ( ) Date
Description	**today** returns the current date, according to the system clock/calendar.
Example	This 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
```

See Also	date DateTime::day DateTime::month DateTime::year

dateTime

Beginner

Method Casts a value as a DateTime data type.

Type dateTime

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 time and date as a DateTime data type.

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 the **formatSetDateTimeDefault** method (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.

See Also

day
hour
month
year

day

Beginner

Method Extracts the day of the month from a DateTime.

Type DateTime

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.

Example In this 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
```

See Also

dow
dowOrd
doy
month
moy
year

daysInMonth

Beginner

Method Returns the number of days in a month.

Type DateTime

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.

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
```

See Also

day
dow
dowOrd
moy

dow

Beginner

Method Returns the day of the week of a DateTime.

Type DateTime

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.

Example This 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
```

See Also

dateTime
day
dowOrd
doy
moy

dowOrd

Beginner

Method Returns the number of a day of the week.

Type DateTime

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.

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
```

See Also dateTime
day
dow
doy
moy

doy

Beginner

Method Returns the number of a day of the year.

Type DateTime

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.

Example This 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
```

See Also dow
moy

hour

Beginner

Method Extracts as a number the hour from a DateTime.

Type 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.

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
```

See Also

day
month
minute
milliSec
year

isLeapYear

Beginner

Method Reports whether a year has 366 days.

Type DateTime

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.

Example For this 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
```

See Also year

milliSec

Beginner

Method Extracts as a number the milliseconds from a DateTime.

Type 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.

Example This 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
```

See Also hour
minute
second

minute

Beginner

Method Extracts as a number the minutes from a DateTime.

Type 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.

Example For this 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
```

See Also hour
milliSec
second

month

Beginner

Method Extracts as a number the month from a DateTime.

Type 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.

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
```

See Also moy

moy

Beginner

Method Extracts as a string the month from a DateTime.

Type 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.

Example For this 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
```

See Also month

second

Beginner

Method Extracts as a number the seconds from a DateTime.

Type 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.

Example This 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
```

See Also hour
milliSec
minute

year

Beginner

Method Extracts as a number the year from a DateTime.

Type 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.

Example For this 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
```

See Also

day
isLeapYear
month
moy

contains

Method Searches the indexes in a DynArray for a value.

Type DynArray

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.

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
```

See Also getKeys

AnyType::view

empty

Method Removes all items from a dynamic array.

Type DynArray

Syntax **empty** ( )

Description **empty** removes all items from an dynamic array. The size of the DynArray becomes 0.

Example This 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"]  = "Porshe"  ; 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
```

See Also contains

getKeys

Method Loads a resizeable array with indexes of an existing DynArray.

Type 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.

Example This 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"]  = "Porshe"  ; 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
```

See Also contains

removeItem

Method Deletes a specified item from a DynArray.

Type DynArray

Syntax **removeItem** (const ***value*** AnyType)

Description **removeItem** deletes the element (specified by its index) in *value* from a DynArray. **removeItem** is case insensitive.

Example The following example concatenates two values in a dynamic array, then uses remove 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
```

See Also empty
contains

size

Method Returns the number of elements in a DynArray.

Type DynArray

Syntax **size** ( ) LongInt

Description **size** returns the number of elements in a DynArray.

Example For this 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
```

See Also contains

view

Method Displays the contents of a DynArray in a dialog box

Type DynArray

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 can not be changed interactively. See "AnyType" earlier in this chapter for more information regarding other data types and the **view** method.

Example For this 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
```

See Also contains

readFromClipboard

Method Reads a graphic from the Clipboard.

Type Graphic

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.

Example In this 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
```

See Also readFromFile
writeToClipboard

readFromFile

Method Reads a graphic from a file.

Type Graphic

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: bitmap (BMP), encapsulated Postscript (EPS), graphic interchange format (GIF), Paintbrush (PCX), and tagged information file format (TIF).

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
```

See Also readFromClipboard

writeToFile

writeToClipboard

Method Writes a bitmap to the Clipboard.

Type Graphic

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).

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
```

See Also writeToFile
readFromClipboard

writeToFile

Method Writes a bitmap to a file.

Type Graphic

Syntax **writeToFile** (const ***fileName*** String) Logical

Description **writeToFile** writes a bitmap to a disk file specified in *fileName*. **writeToFile** returns True if the file specified can be created; otherwise it returns False.

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
```

See Also writeToClipboard

logical

Beginner

Procedure Casts a value as type Logical.

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.

Example In this 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
```

See Also Logical

bitAND

Method Performs a bitwise AND operation on two values.

Type LongInt

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

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
```

See Also bitOR
bitXOR

bitIsSet

Method Reports whether a bit is 1 or 0.

Type LongInt

Syntax **bitIsSet** (const ***value*** LongInt) Logical

Description **bitIsSet** examines the binary respresentation 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).

Example 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
```

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
```

See Also

bitAND
bitOR
bitXOR

bitOR

Method Performs a bitwise OR operation on two values.

Type LongInt

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

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
```

See Also bitAND
bitXOR

bitXOR

Method Performs a bitwise XOR operation on two values.

Type LongInt

Syntax **bitXOR** (const ***value*** LongInt) LongInt

Description **bitXOR** performs a bitwise XOR (exclusive OR) operation on *value*. **bitXOR** operates on the binary respresentations 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

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
```

See Also bitAND

bitOR

LongInt

Beginner

Procedure Casts a value as a LongInt.

Type 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.

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
```

See Also AnyType::view

memo

Procedure Casts a value as a Memo.

Type Memo

Syntax **memo** (const ***value*** AnyType [, const ***value*** AnyType]*) String

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 concatentate them to one Memo.

Example This 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 particluar 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
```

readFromFile

Method Reads a memo from a file.

Type Memo

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.

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 project notes 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
```

See Also writeToFile

writeToFile

Method Writes a memo to a file.

Type Memo

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.

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 project
notes 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
```

See Also readFromFile

abs

Beginner

Method Returns the absolute value of a number.

Type Number

Syntax **abs** ( ) Number

Description **abs** removes the sign from a numeric value

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
```

See Also number

acos

Beginner

Method Returns the 2-quadrant arc cosine of a number.

Type 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*).

Example The **pushButton** method for the *findArcCos* button calculates and displays the arc cosine of a 30 degree angle.

```
; findArcCos::pushButton
method pushButton(var eventInfo Event)
var
  thirtyDegrees Number
endVar
thirtyDegrees = PI / 3.0
msgInfo("The arc cosine of 30 degrees", thirtyDegrees.acos())
; displays 1.02
endmethod
```

See Also asin
atan

asin

Beginner

Method Returns the 2-quadrant arc sine of a number.

Type 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.

Example For this 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
```

See Also

acos

atan

cos

sin

atan

Beginner

Method Returns the 2-quadrant arctangent of a number.

Type 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*).

Example In this 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
```

See Also

cos
sin
tan
tanh
atan2

atan2

Beginner

Method Returns the 4-quadrant arctangent of a number.

Type Number

Syntax **atan2** (const ***x*** Number) Number

Description Given a sine in radians, **atan** returns an angle in radians whose cosine is *x*. **atan2** is called the 4-quadrant arctangent because it returns values in all four quadrants.

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
```

See Also

cos

sin

tan

tanh

ceil

Beginner

Method Rounds a numeric expression up to the nearest whole number.

Type Number

Syntax **ceil** ( ) Number

Description **ceil** rounds a numeric expression up (toward positive infinity) to the nearest whole number.

Example In this 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
```

See Also floor

cos

Beginner

Method Returns the cosine of an angle.

Type Number

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.

Example In this 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
```

See Also cosh
sin
tan

cosh

Beginner

Method Returns the hyperbolic cosine of an angle.

Type Number

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

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
```

See Also cos
sin
tan

exp

Beginner

Method Returns the exponential (base *e*) of a number.

Type Number

Syntax **exp** ( ) Number

Description **exp** computes *ex*, where *e* is the constant 2.7182845905. The inverse method is **ln**.

Example In this 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
```

See Also ln
log

floor

Beginner

Method Rounds a numeric expression down to the nearest whole number.

Type Number

Syntax **floor** ( ) Number

Description **floor** rounds a numeric expression down (toward negative infinity) to the nearest whole number.

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
```

See Also ceil

fraction

Beginner

Method Returns the fractional part of a number.

Type Number

Syntax **fraction** ( ) Number

Description **fraction** returns the fractional part of a number, the part to the right of the decimal.

Example In this 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
```

See Also mod

fv

Beginner

Method Returns the future value of a series of equal payments.

Type Number

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 forumla 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.

Example This 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
```

See Also pmt

pv

ln

Beginner

Method Returns the natural logarithm of a numeric expression.

Type Number

Syntax **ln** ( ) Number

Description **ln** calculates the natural logarithm to the base *e* of a positive value. The constant *e* is 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.

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
```

See Also exp
log

log

Beginner

Method Returns the base 10 logarithm of a numeric expression.

Type Number

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.

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
```

See Also exp

ln

max

Beginner

Procedure Returns the larger of two numbers.

Type Number

Syntax **max** (const ***x1*** AnyType, const ***x2*** AnyType) AnyType

Description **max** returns the larger of the two values *x1* and *x2*.

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
```

See Also min

min

Beginner

Procedure Returns the smaller of two numbers.

Type Number

Syntax **min** (const ***x1*** AnyType, const ***x2*** AnyType) AnyType

Description **min** returns the smaller of the two values, *x1* and *x2*.

Example In this 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
```

See Also max

mod

Beginner

Method Returns the remainder when one number is divided by another.

Type Number

Syntax **mod** (const ***modulo*** Number) Number

Description **mod** returns the remainder (or modulus) when a number is divided by the value of *modulo*. If *modulo* is 0, **mod** returns 0.

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, m 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 3
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
```

See Also fraction

number

Beginner

Procedure Casts a value as a Number.

Type 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**.

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 = "1,123.54"
; cast x to a Number before multiplying by 2
msgInfo("Double " + x + " is", Number(x) * 2) ; displays
2,247.08
endmethod
```

See Also numVal

numVal

Procedure Casts a value as a Number.

Type 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**.

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 = "1,123.54"
; cast x to a Number before multiplying by 2
msgInfo("Double " + x + " is", numVal(x) * 2) ; displays
2,247.08
endmethod
```

See Also number

pmt

Beginner

Method Returns the periodic payment required to pay off a loan.

Type Number

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).

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
```

See Also fv

pv

pow

Beginner

Method Raises a number to a power.

Type Number

Syntax **pow** (const ***exponent*** Number) Number

Description **pow** returns the value of a number raised to the power specified in *exponent*. If the result is larger than 10308 or smaller than 10-*307*, you'll get an error.

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
```

See Also pow10
ln
log

pow10

Beginner

Method Calculates 10 to a specified power.

Type Number

Syntax **pow10** ( ) Number

Description **pow10** returns the value of 10 raised to a specified power.

Example In this example, the **pushButton** method for the *raiseTen* button calculates 10*expn* 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
```

See Also pow
ln
log

pv

Beginner

Method Returns the present value of a series of equal payments.

Type Number

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.

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
```

See Also fv
pmt

rand

Beginner

Procedure Generates a random value ranging from 0 to 1.

Type Number

Syntax **rand** ( ) Number

Description **rand** generates a random value ranging from 0 to 1.

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
```

See Also truncate

round

Beginner

Method Rounds a number to a specified number of decimal places.

Type Number

Syntax **round** (const ***places*** SmallInt) Number

Description **round** returns a number rounded to the number of decimal places specified in *places*.

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
```

See Also truncate
ceil
floor

sin

Beginner

Method Returns the sine of an angle.

Type Number

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.

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
```

See Also asin
cos
tan

sinh

Beginner

Method Returns the hyperbolic sine of an angle.

Type Number

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

Example In this 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
```

See Also sin

cos

tan

sqrt

Beginner

Method Returns the square root of a number.

Type Number

Syntax **sqrt** ( ) Number

Description **sqrt** returns the square root of a positive value or numeric expression.

Example In this 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
```

See Also exp

tan

Beginner

Method Returns the tangent of an angle.

Type Number

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.

Example In this 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
```

See Also

atan

atan2

cos

sin

tanh

Beginner

Method Returns the hyperbolic tangent of an angle.

Type Number

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)*

Example In this 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
```

See Also atan

cos

sin

truncate

Beginner

Method Truncates a number to a specified number of decimal places.

Type Number

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.

Example In this 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
```

See Also ceil
floor
round

canReadFromClipboard

Method Reports whether an OLE object can be pasted from the clipboard into an OLE variable.

Type OLE

Syntax **canReadFromClipboard** ( ) Logical

Description **canReadFromClipboard** returns True if an OLE object can be read (pasted) from the Clipboard into an OLE variable; otherwise, it returns False. This method is useful in a routine that informs the user whether an operation is possible. For example, you can make a menu item dimmed and inactive when this method returns False.

Example In this example, a form has an OLE object named oleObject and a button named *updateOLE*. The **pushButton** method for *oleObject* tests the contents of the Clipboard to determine whether an OLE document can be read into an OLE variable. If the method **canReadFromClipboard** returns True, this example pastes the contents of the clipboard into *oleObject*; otherwise it displays an error message.

```
; updateOLE::pushButton
method pushButton(var eventInfo Event)
var
  oleVar   OLE
endVar

; if an OLE object can be read from the clipboard
if oleVar.canReadFromClipboard() then

  ; read OLE object from clipboard to OLE variable
  oleVar.readFromClipboard()

  ; update the form's OLE object with contents of clipboard
  oleObject = oleVar

else
  beep()
  msgStop("Error", "Can't read from clipboard.")
endif

endmethod
```

See Also readFromClipboard

edit

Method Launches the OLE server and lets the user edit the object or take some other action.

Type OLE

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. With the **enumVerbs** method, you can 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.

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.

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
        olePic = picTC.PicData
        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
```

See Also enumVerbs

enumVerbs

Method Creates a DynArray listing the actions supported by the OLE server.

Type OLE

Syntax **enumVerbs** (var ***verbs*** DynArray[] SmallInt) Logical

Description **enumVerbs** creates a DynArray listing the action commands (called *verbs*) supported by the OLE server.

When you associate an OLE variable with an OLE object (sometimes called an OLE document), Paradox knows from what server application the object was generated. 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 values 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 to learn more about menus and special characters.

Example For this example, 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. The following example uses **enumVerbs** to create a pop-up menu that lists the verbs (actions) for Sound Recorder. 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.

```
; editSounds::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 OLE variable with
OLE field value
    oleVar.enumVerbs(verbs)       ; load DynArray with Sound
Recorder actions
    forEach tagName in verbs      ; create a pop-up menu from
verb listing
      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
Play is selected
    if not mChoice.isBlank() then
      oleVar.edit("PdoxWin", verbs[mChoice])
    endif

  else
    msgStop("Sorry", "Can't find " + soundName + ".")
  endif
else
  msgStop("Sorry", "Can't open " + tblName + ".")
endif

endmethod
```

See Also readFromClipboard

getServerName

Method Reports the name of the OLE server for an OLE object.

Type OLE

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.

Example For this 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
```

See Also edit
enumVerbs

readFromClipboard

Method	Pastes an OLE object from the Clipboard into an OLE variable.
Type	OLE
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.
Example	See the example for canReadFromClipboard.
See Also	canReadFromClipboard

writeToClipboard

Method Copies an OLE variable to the clipboard.

Type OLE

Syntax **writeToClipboard** ( ) Logical

Description **writeToClipboard** copies the original OLE object to the Clipboard as if the server had done the copy (because Paradox is not an OLE server).

This method returns True if an OLE object is successfully written (pasted) to the Clipboard; otherwise, it returns False.

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
```

See Also readFromClipboard

distance

Method Returns the distance between two points.

Type Point

Syntax **distance** (const ***pt*** Point) Number

Description **distance** returns the number of twips between a point and *pt*.

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
```

See Also

isAbove

isBelow

isLeft

isRight

x

y

isAbove

Method Reports whether a point is above another point.

Type 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.

Example In this 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
```

See Also isBelow
isLeft
isRight

isBelow

Method Reports whether a point is below another point.

Type 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.

Example In this 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
```

See Also isAbove
isLeft
isRight

isLeft

Method	Reports whether a point is to the left of another point.
Type	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.
Example	See the example for isAbove.
See Also	isAbove isBelow isRight

isRight

Method	Reports whether a point is to the right of another point.
Type	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.
Example	See the example for isBelow.
See Also	isAbove isBelow isLeft

point

Procedure Casts an expression as a Point.

Type 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.

Example In this example, you want to vary the position of a box called *rateBox*. The values of an unbound field object named *rateField* range from 1 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
```

See Also setX
setY

setX

Method Specifies the x-coordinate of a point.

Type 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.

Example In this 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
```

See Also setY

setXY

Method Specifies the x- and y-coordinates of a point.

Type 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.

Example In this 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
```

See Also setX
setY

setY

Method Specifies the y-coordinate of a point.

Type 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.

Example In this 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
```

See Also setX

setXY

x

Method	Returns the x-coordinate of a point.
Type	Point
Syntax	**x** ( ) LongInt
Description	**x** returns the x-coordinate of a point.
Example	See the example for setX.
See Also	setX setY y

y

Method	Returns the y-coordinate of a point.
Type	Point
Syntax	**y** ( ) LongInt
Description	**y** returns the y-coordinate of a point.
Example	See the example for setY.
See Also	setY setX x

view

Method Displays in a dialog box the value of a variable.

Type Record

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 can specify the dialog box's title in *title*, or you can omit *title* to display the variable's data type. Unlike many data types, values in a Record can't be changed when displayed in a **view** dialog box. Refer to AnyType earlier in this chapter for more information regarding **view** and other data types.

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
```

See Also

Array
DynArray

bitAND

Method Performs a bitwise AND operation on two values.

Type SmallInt

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

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
```

See Also bitOR
bitXOR

bitIsSet

Method Reports whether a bit is 1 or 0.

Type SmallInt

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).

Example 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
```

The next 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
```

See Also

bitAND
bitOR
bitXOR

bitOR

Method Performs a bitwise OR operation on two values.

Type SmallInt

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

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
```

See Also bitAND
bitXOR

bitXOR

Method Performs a bitwise XOR operation on two values.

Type SmallInt

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

Example In this 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
```

See Also bitAND
bitOR

int

Procedure Casts a value as an interger.

Type SmallInt

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.

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
```

See Also smallInt

smallInt

Beginner

Procedure Casts a value as a small integer.

Type SmallInt

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.

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
```

See Also int

advMatch

Beginner

Method Searches text for a specified string.

Type 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. This method is case-sensitive. To specify *pattern*, use a string and the optional symbols listed in the table.

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 any of the above 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 line (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.

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
```

See Also match
search

ansiCode

Procedure Returns the ANSI code of a one-character string.

Type String

Syntax **ansiCode** (const ***char*** String) SmallInt

Description **ansiCode** returns the ANSI code of *char*. The ANSI code returned is an integer between 1 and 255.

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
```

See Also chr
chrToKeyName

breakApart

Beginner

Method Splits a string into substrings.

Type String

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 in between parse as a token and result in an empty array element.

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
```

See Also substr

chr

Beginner

Procedure Returns the one-character string represented by an ANSI code.

Type String

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.

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
```

See Also chrOEM
chrToKeyName
string

chrOEM

Procedure Returns the one-character string of an OEM code.

Type String

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 *ObjectPAL Developer's Guide* for more information.

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
```

See Also chr
string

chrToKeyName

Procedure	Returns the virtual key-code string of a one-character string.
Type	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 is returned 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*. To display the list online, open an ObjectPAL Editor window, then from the Types of Constants column, choose Keyboard. The constants appear in the Constants column.
Example	See the example for chr.
See Also	vkCodeToKeyName

fill

Beginner

Procedure Returns a string containing repeated instances of a character.

Type String

Syntax **fill** (const ***fillCharacter*** String, const ***fillNumber*** SmallInt) 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.

Example In this 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 "Quattro Pro" four times.

```
; fillAndView::pushButton
method pushButton(var eventInfo Event)
var
  str String
endVar
str = fill("X", 5)
str.View()                    ; displays XXXXX
str = fill("Quattro Pro
", 4)          ; add a line break after every occurrence
str.View()
; displays:   Quattro Pro
;             Quattro Pro
;             Quattro Pro
;             Quattro Pro
endmethod
```

See Also space

format

Beginner

Procedure Returns a formatted string for display or printing.

Type String

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-justified) for all numeric types, AL (left-justified) 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
	American or International separators		American
Sign	Format of positive and negative numbers	All numeric	See following
	Positive 999		No leading positive sign
	Negative		Leading minus sign -999
Date	Specify date formats hh:mm:ss am(pm),, mm/dd/yy(yy)	Date & DateTime	mm/dd/yy(yy) for Date or 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)
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 Truth value
	LF(*s*)	*s* specifies representation of logical False value
	LY	Logical values as Yes and No
	LO	Logical values as On and Off

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 procede 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("w6",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()

endmethod
; to display date and time in 24-hour format

fs <M>=<D> "\"TNA(), TNP(), TO(%H:%M:%S %D), DO(%W %M/%D/%Y)
\"," +

     " dateTime(\"2:30:00 pm 11/24/92\")"

formatField <M>=<D> 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")

endmethod
```

See Also string

ignoreCaseInStringCompares

Procedure Specifies whether to consider case when comparing strings.

Type String

Syntax **ignoreCaseInStringCompares** (const ***yesNo*** Logical)

Description **ignoreCaseInStringCompares** specifies whether to consider case when comparing strings. Normally, upper- and lowercase 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**.

Example In this 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
```

See Also isIgnoreCaseInStringCompares

isIgnoreCaseInStringCompares

Procedure	Reports whether case is considered when comparing strings.
Type	String
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**.
Example	See the example for ignoreCaseInStringCompares.
See Also	ignoreCaseInStringCompares

isSpace

Beginner

Method Reports whether a string contains only spaces (or is empty).

Type String

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.

Example This 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
```

See Also space

keyNameToChr

Procedure	Returns the one-character string represented by a virtual key-code string.
Type	String
Syntax	**keyNameToChr** (const ***keyName*** String) String
Description	**keyNameToChr** returns the one-character string represented by the virtual key code *keyName*. **keyName** is one of the virtual key codes (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*. To display the list online, open an ObjectPAL Editor window, then from the Types of Constants column, choose Keyboard. The constants appear in the Constants column.
Example	See the example for keyNameToVKCode.
See Also	chrToKeyName keyNameToVKCode

keyNameToVKCode

Procedure Returns the VK_Code of a virtual key-code string.

Type String

Syntax **keyNameToVKCode** (const ***keyName*** String) SmallInt

Description **keyNameToVKCode** returns the VK_Code of the character represented by the virtual key code *keyName*, given as a string.

keyName is one of the virtual key codes (such as VK_BACK for Backspace), but must be supplied as a string (such as "VK_BACK"), not a constant. Alphanumeric characters and many symbols have a key name that consists simply of the character, for instance, "J" for the letter *J*. To display the list online, open an ObjectPAL Editor window, then from the Types of Constants column, choose Keyboard. The constants appear in the Constants column.

Example In this 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
```

See Also chrToKeyName
keyNameToChr
vkCodeToKeyName

lower

Beginner

Method Converts a string to lowercase.

Type String

Syntax **lower** ( ) String

Description **lower** converts a string to lowercase letters. Use **upper** to convert a string to uppercase letters.

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
```

See Also upper

lTrim

Beginner

Method Removes leading blanks from a string.

Type String

Syntax **lTrim** ( ) String

Description **lTrim** removes spaces and Tab characters from the left end of a string.

Example In this 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
```

See Also rTrim

match

Beginner

Method Compares a string with a pattern.

Type String

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 case by default (but you can use **ignoreCaseInStringCompares(No)** to make **match** case-sensitive).

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.

To embed a quote in *pattern*, precede it with a backslash (\). To embed a period, surround it with double quotes and precede the quotes with backslashes (\".\").

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"
```

See Also advMatch

search

oemCode

Procedure	Returns the OEM code of a one-character string.
Type	String
Syntax	**oemCode** (const ***char*** String) SmallInt
Description	**oemCode** returns the OEM code of *char*. The OEM code returned is an integer between 1 and 255.
Example	See the example for ansiCode.
See Also	ansiCode chrToKeyName

rTrim

Beginner

Method Removes trailing blanks from a string.

Type String

Syntax **rTrim** ( ) String

Description **rTrim** removes spaces, tabs, carriage returns, and linefeed characters from the right end of a string.

Example In this 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
```

See Also lTrim

search

Beginner

Method Returns the position of one string inside another.

Type String

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.

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
```

See Also advMatch

match

size

Beginner

Method Returns the length of a string.

Type String

Syntax **size** ( ) SmallInt

Description **size** returns the number of characters (including spaces) in a string.

Example In this 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
```

See Also string

space

Beginner

Method	Creates a string of a specifed number of spaces.
Type	String
Syntax	**space** (const ***numberOfSpaces*** SmallInt) String
Description	**space** returns a string of *numberOfSpaces* spaces.
Example	See the example for isSpace.
See Also	isSpace

string

Beginner

Procedure Casts a value as a String.

Type 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 concatentate them to one 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
```

See Also format

strVal

Procedure	Converts a value to a string.
Type	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.
Example	See the example for size.
See Also	string size

subStr

Beginner

Method Returns a portion of a string.

Type String

Syntax **substr** (const ***startIndex*** Number [, const ***numberOfChars*** SmallInt]) 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*.

Example In this 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
```

See Also breakApart
search
size

toANSI

Method Converts a string of OEM characters to ANSI characters.

Type String

Syntax **toANSI** ( ) String

Description **toANSI** converts a string of OEM characters to ANSI characters.

Example In this 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
```

See Also toOEM

toOEM

Method Converts a string of ANSI characters to OEM characters.

Type String

Syntax **toOEM** ( ) String

Description **toOEM** converts a string of ANSI characters to OEM characters.

Example In this 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
```

See Also toANSI

upper

Beginner

Method Converts a string to uppercase.

Type String

Syntax **upper** ( ) String

Description **upper** converts a string to uppercase letters. Use **lower** to convert a string to lowercase letters.

Example In this 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
```

See Also lower

vkCodeToKeyName

Procedure Converts a virtual keycode constant to a virtual keycode string.

Type String

Syntax **vkCodeToKeyName** (const ***vkCode*** SmallInt) String

Description **vkCodeToKeyName** returns the virtual key-code name, as a String, of the character represented by *vkCode*.

A key name is one of the virtual key codes (such as VK_BACK for Backspace), but is returned 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*. To display the list online, open an ObjectPAL Editor window, then from the Types of Constants column, choose Keyboard. The constants appear in the Constants column.

Example See the example for ansiCode.

See Also ansiCode
chr
chrToKeyName
keyNameToChr

time

Beginner

Procedure Casts a value as a time, or returns the current time.

Type 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**.

Example The first 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)
```

The next 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
```

See Also DateTime::hour
DateTime::milliSec
DateTime::minute
DateTime::second

addArray

Method Appends elements of an array to a menu.

Type 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**.

Example This 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
```

See Also

addBreak

addPopUp

addStaticText

addText

OPAL11

addBreak

Method Starts a new row in a menu.

Type 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.

Example This 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
```

See Also addArray
addPopUp
addStaticText
addText

addPopUp

Method Adds a pop-up menu to a menu bar item.

Type Menu

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.

Example The code in this 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
```

See Also addArray
addBreak

addStaticText

Method Adds an unselectable text string to a menu.

Type Menu

Syntax **addStaticText** (const ***item*** String)

Description **addStaticText** appends *item* to a menu as unselectable text.

Example In this 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
```

See Also addText

addText

Method Adds a selectable text string to a menu.

Type 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 the item *menuName* to a menu. Menu items are displayed from left to right across the menu bar. You can use *attrib* to preset the display attribute of *menuName*. ObjectPAL provides constants (like MenuEnabled) for display attributes, so you don't have to memorize numeric values.

In the third form of the syntax, you can specify an *id* number (a 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. When you specify a menu *id*, you should use the built-in constant UserMenu as a base constant, then add your own number to it. 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)
```

For more information regarding user-defined constants, refer to the *ObjectPAL Developer's Guide*.

You can use an ampersand in an item to designate an accelerator key. For example, the item "&File" would appear 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 this example, the return value is "&File", not "File".

If you want 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()
```

Example

The following examples demonstrate how **addText** syntax influences the way you test for the user's menu choice.

The first example uses the first form of **addText** syntax to create a simple menu. This example does not use *id* in the **addText** statements. The code attached to the built-in **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
```

The next 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
```

See Also addStaticText
addArray

contains

Method Reports whether an item is in a menu.

Type 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.

Example This 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 :
      ; add Undo and redisplay the menu
      m1.addText("&Undo")
      m1.show()

      ; when user posts the record (moves to another record)
    case eventInfo.id() = DataUnlockRecord :
      ; remove Undo redisplay the menu
      m1.remove("&Undo")
      m1.show()
  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) ; revert record to
original state
      m1.remove("&Undo")              ; remove Undo menu item
      m1.show()                       ; redisplay menu without
Undo
  endswitch

endif
endmethod
```

See Also count

count

Method Returns the number of items in a menu.

Type 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.

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
```

See Also contains

empty

Method Removes all items from a menu.

Type 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.

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 *showMenuTwo*'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
```

See Also remove

getMenuChoiceAttribute

Procedure Reports the display attributes of a menu item.

Type Menu

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 **getMenuChoiceAttribute** with **hasMenuChoiceAttribute** to determine whether a specific display attribute applies for a menu item.

ObjectPAL provides constants (like MenuEnabled) for display attributes, so you don't have to memorize numeric values. Constants are listed online. To display the list, open an ObjectPAL Editor window, then from the Types of Constants column, choose MenuChoiceAttributes. The constants appear in the Constants column.

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.

Example In this 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
  attrib SmallInt
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 enabled
msgInfo("Time enabled?", HasMenuChoiceAttribute(attrib,
MenuEnabled))
  ; this displays True because Time is grayed
```

```
msgInfo("Time grayed?", hasMenuChoiceAttribute(attrib,
MenuGrayed))

endmethod
```

See Also

getMenuChoiceAttributeById
hasMenuChoiceAttribute
setMenuChoiceAttribute
setMenuChoiceAttributeById

getMenuChoiceAttributeById

Procedure Reports the display attribute of a menu item specified by its menu ID.

Type Menu

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 **getMenuChoiceAttributeById** with **hasMenuChoiceAttribute** to determine whether a specific display attribute applies for a menu item.

ObjectPAL provides constants (like MenuEnabled) for display attributes, so you don't have to memorize numeric values.

Constants are listed online. To display the list, open an ObjectPAL Editor window, then from the Types of Constants column, choose MenuChoiceAttributes. The constants appear in the Constants column.

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.

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
```

See Also

getMenuChoiceAttribute
hasMenuChoiceAttribute
setMenuChoiceAttribute

hasMenuChoiceAttribute

Procedure Reports whether a menu item contains a given display attribute.

Type Menu

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 **hasMenuChoiceAttribute** with **getMenuChoiceAttribute** or **getMenuChoiceAttributeById** to determine whether a particular display attribute for a menu item is represented in *attribSet*.

ObjectPAL provides constants (like MenuEnabled) for display attributes, so you don't have to memorize numeric values.

Constants are listed online. To display the list, open an ObjectPAL Editor window, then from the Types of Constants column, choose MenuChoiceAttributes. The constants appear in the Constants column.

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
```

See Also getMenuChoiceAttribute
getMenuChoiceAttributeById

remove

Method Removes an item from a menu.

Type 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.

Example The code shown in this 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
```

See Also contains

empty

removeMenu

Procedure Removes a custom menu and restores the default menu.

Type Menu

Syntax **removeMenu** ( )

Description **removeMenu** replaces a menu built using ObjectPAL with Paradox's default menu.

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
```

See Also empty
remove

setMenuChoiceAttribute

Procedure Sets the display attribute of a menu item.

Type Menu

Syntax **setMenuChoiceAttribute** (const ***menuChoice*** String, const ***menuAttribute*** SmallInt)

Description **setMenuChoiceAttribute** sets the display attribute of *menuChoice* to *menuAttribute*. This procedure affects the currently displayed menu; if you have not created a custom menu, **setMenuChoiceAttribute** affects the built-in menu.

ObjectPAL provides constants (like MenuGrayed) for menu attributes, so you don't have to memorize numeric values.

Constants are listed online. To display the list, open an ObjectPAL Editor window. Choose Language | Constants. Then, from the Types of Constants column, choose MenuChoiceAttributes. The constants appear in the Constants column.

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
```

See Also

getMenuChoiceAttribute
getMenuChoiceAttributeById
hasMenuChoiceAttribute
setMenuChoiceAttributeById

setMenuChoiceAttributeById

Procedure Sets the display attribute of a menu item.

Type Menu

Syntax **setMenuChoiceAttributeById** (const ***menuId*** String, const ***menuAttribute*** SmallInt)

Description **setMenuChoiceAttributeById** sets the display attribute of *menuId* to *menuAttribute*. This procedure affects the currently displayed menu; if you have not created a custom menu, **setMenuChoiceAttributeById** affects the built-in menu.

ObjectPAL provides constants (like MenuGrayed) for menu attributes, so you don't have to memorize numeric values.

Constants are listed online. To display the list, open an ObjectPAL Editor window. Choose Language | Constants. Then, from the Types of Constants column, choose MenuChoiceAttributes. The constants appear in the Constants column.

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, MenuEnabled)

         ; when user posts the record (moves to another
record)
      case eventInfo.id() = DataUnlockRecord :
         ; disable and dim Undo menu item
        setMenuChoiceAttributeById(UndoMenu, 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
```

See Also

getMenuChoiceAttribute
getMenuChoiceAttributeById
hasMenuChoiceAttribute
setMenuChoiceAttribute

show

Method Displays a menu.

Type Menu

Syntax **show** ( )

Description **show** displays a menu.

The user's choice is handled using the built-in **menuAction** method and **menuChoice** from the MenuEvent type. Refer to the *ObjectPAL Developer's Guide* for more information about working with menus.

Example In this 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
```

See Also addText

addArray

Method Appends elements of an array to a pop-up menu.

Type PopupMenu

Syntax **addArray** (const ***items*** Array[] String)

Description **addArray** appends *items* from an array to a pop-up menu.

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
```

See Also

addBar
addBreak
addSeparator
addStaticText
addText

addBar

Method Adds a vertical bar to a pop-up menu.

Type PopupMenu

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**.

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
```

See Also addBreak
addSeparator

addBreak

Method Starts a new column in a pop-up menu.

Type PopupMenu

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.

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
```

See Also addBar
addSeparator

addPopUp

Method Adds a pop-up menu to the structure.

Type PopupMenu

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.

Example This 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
```

The next 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
```

See Also addBreak

addSeparator

Method Adds a horizontal bar to a pop-up menu.

Type PopupMenu

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.

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
```

See Also addBar
addBreak

addStaticText

Method Adds an unselectable text string to a pop-up menu.

Type PopupMenu

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.

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
```

See Also addText

addText

Method Adds a selectable text string to a pop-up menu.

Type PopupMenu

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 *menuName* to a pop-up menu as a selectable item. You can use *attrib* to preset the display attribute of *menuName*. ObjectPAL provides constants (like MenuDisabled) for display attributes, so you don't have to memorize numeric values.

The third form of **addText** syntax 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.

When you specify a menu *id*, you should use the built-in constant UserMenu as a base constant, then add your own number 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)
```

For more information regarding user-defined constants, refer to the *ObjectPAL Developer's Guide*.

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".

Example

The following examples demonstrate variations of **addText** syntax.

For the first 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
```

The next example demonstrates 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
  menuRed   = 1  ; define constant values for menu ids
  menuBlue  = 2
  menuWhite = 3
  menuTime  = 4
  menuDate  = 5
  menuAbout = 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 PopUpMenu
endVar

  ; add text to pop-up menus and use user-defined constants
clrMenu.addText("&Red", MenuEnabled, MenuRed + UserMenu)
clrMenu.addText("&Blue", MenuEnabled, MenuBlue + UserMenu)
clrMenu.addText("&White", MenuEnabled, MenuWhite + UserMenu)

subMenu1.addText("&Time", MenuEnabled, MenuTime + UserMenu)
subMenu1.addText("&Date", MenuEnabled, MenuDate + UserMenu)
subMenu1.addSeparator()
subMenu1.addPopUp("&Page colors", clrMenu)
subMenu1.addSeparator()
subMenu1.addText("&About", MenuEnabled, MenuAbout + UserMenu)
```

```
  ; attach pop-up menus to mainMenu and display the menu bar
mainMenu.addPopUp("&Utilities", subMenu1)
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 = MenuRed + UserMenu   : self.color = Red
  case menuId = MenuBlue + UserMenu  : self.color = Blue
  case menuId = MenuWhite + UserMenu : self.color = White
  case menuId = MenuTime  + UserMenu : msgInfo("Current Time",
time())
  case menuId = MenuDate + UserMenu  : msgInfo("Today's Date",
date())
  case menuId = MenuAbout + UserMenu :
eventInfo.setId(MenuHelpAbout)
endSwitch

endmethod
```

See Also addStaticText

show

Method Displays a pop-up menu and returns the item selected.

Type PopupMenu

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.

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
```

See Also Menu::empty

switchMenu

Procedure Builds and displays a pop-up menu, and handles the menu choice.

Type PopupMenu

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).

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
```

See Also addText

show

action

Beginner

Method Performs a specified action.

Type UIObject

Syntax **action** (const ***actionId*** SmallInt) Logical

Description **action** specifies an *actionId* to perform in response to an event. ObjectPAL provides a constant for each *actionId*, so you don't have to remember numeric values.

Constants are listed online. To display the list, open an Editor window, choose Language | Constants, then, from the Types of Constants column, choose an item beginning with "Action" (for example, ActionDataCommands). The constants appear in the Constants column.

Example The code in this 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 SpeedBar 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
```

See Also menuAction

ActionEvent::id

ActionEvent::setId

atFirst

Method Reports if the pointer is at the first record of a table.

Type UIObject

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.

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
```

See Also atLast

atLast

Method Reports if the pointer is at the last record in a table.

Type UIObject

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.

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
```

See Also atFirst

attach

Method Binds a UIObject variable to a specified design object.

Type UIObject

Syntax

1. attach ( ) Logical

2. attach (const ***object*** UIObject) Logical

3. attach (const ***form*** Form [, ***objectName*** String]) Logical

4. attach (const ***form*** Report [, ***objectName*** String]) Logical

Description **attach** binds a UIObject variable to self (syntax 1), to another UIObject (*object* in syntax 2), to a Form (*form* in syntax 3), or to a UIObject in another Form (*objectName* in syntax 3). You can also use **attach** to bind a UIObject to an open report, or to an object in an open report (syntax 4).

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 3 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 method, you need to attach a UIObject var (a Form variable won't work) to an open form.

Example The following example shows all three forms of the syntax. For syntax 1, the method attaches the variable *objBox* to the current object (self), then changes its color. For syntax 2, the method attaches *objBox* to another object, then changes that object's color via *objBox*. A second example for syntax 2 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 usually use syntax 2 instead of syntax 3. Syntax 2 can reference an object name on another form by including the form handle (previously obtained) in the object name. Syntax 3 supplies 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

; syntax 1
objBox.attach()           ; binds objBox to thisBox
objBox.color = DarkMagenta

; syntax 2
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

; syntax 3
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
```

See Also moveTo

broadcastAction

Method	Broadcasts an action to an object.
Type	UIObject
Syntax	**broadCastAction** (const ***actionID*** SmallInt)
Description	**broadCastAction** notifies an object and all of the objects contained by that object that an event has occurred.
See Also	action menuAction

cancelEdit

Beginner

Method Cancels record changes without ending Edit mode.

Type UIObject

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)
```

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 cancelled 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
```

See Also currRecord
edit
endEdit

convertPointWithRespectTo

Method Changes the frame of reference for calculating the coordinates of a point.

Type UIObject

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*.

Example This 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
```

See Also Point

copyFromArray

Method Copies data from an array to a record of a table.

Type UIObject

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.

Example In this 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) ; then copy the array into
the new record
CUSTNAME.endEdit()
endmethod
```

See Also copyToArray

copyToArray

Method Copies data from a record to an array.

Type UIObject

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.

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
```

See Also copyFromArray

create

Method Creates an object.

Type UIObject

Syntax **create** (const ***objectType*** SmallInt, const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt [, const ***container*** UIObject])

Description **create** creates the object specified in *objectType* 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 twips.) The optional argument *container* specifes a container object for the object you're creating.

ObjectPAL provides constants (like ButtonTool) for the object types, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose UIObjectTypes. The constants display in the Constants column.

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.

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
method mouseUp(var eventInfo MouseEvent)
var
  ui UIObject
endvar

; create a Blue box, called Fred, and make it visible
ui.create(BoxTool, 144, 144, 2880, 2880)
ui.Name = "Fred"
ui.Color = Blue
ui.Visible = True
; create a Green ellipse inside Fred, called Bill
ui.create (EllipseTool, 288, 288, 1440, 1440, self.Fred)
ui.Name = "Bill"
ui.Color = Green
ui.Visible = True

endmethod
```

See Also delete
methodSet

currRecord

Method Reads the current record into the record buffer.

Type UIObject

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)
```

Example In this 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 <M>=<D> 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
```

See Also cancelEdit

delete

Method Deletes an object from a form.

Type UIObject

Syntax **delete** ( )

Description **delete** deletes an object from a form at run time.

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 **create** example 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
```

See Also create
methodSet

deleteRecord

Beginner

Method Deletes the current record from the table.

Type UIObject

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.

deleteRecord has the same effect as the action constant DataDeleteRecord, so the following statements are equivalent:

```
obj.deleteRecord()
obj.action(DataDeleteRecord)
```

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
```

See Also empty
insertRecord
insertAfterRecord
insertBeforeRecord

edit

Beginner

Method Puts a table into Edit mode.

Type UIObject

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.

endEdit has the same effect as the action constant DataEndEdit, so the following statements are equivalent:

```
obj.endEdit()
obj.action(DataEndEdit)
```

Example In this 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 cancelled
endif

endmethod
```

See Also cancelEdit

endEdit

empty

Method Deletes all records from a table.

Type UIObject

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. This operation cannot be undone.

In multiuser applications, this method tries, for the duration of the retry period, to place a full lock on the table. If a lock can't be placed, the method fails.

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
TableFrame
ui.TableName = "TmpOrder.db" ; 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()                    ; deletes table frame
  DMRemoveTable("TmpOrder.db")   ; remove from data model
  tbl.attach("TmpOrder.db")
  tbl.delete()                   ; delete table
endif
endmethod
```

See Also deleteRecord

end

Beginner

Method Moves to the last record in a table.

Type UIObject

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)
```

Example This 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
```

See Also home
nextRecord
priorRecord
currRecord
skip
moveTo

endEdit

Beginner

Method Leaves Edit mode, and accepts changes to the current record.

Type UIObject

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)
```

Example See the example for edit.

See Also cancelEdit
edit

enumFieldNames

Method Fills an array with the names of the fields in a table.

Type UIObject

Syntax **enumFieldNames** (var ***fieldArray*** Array[] String)

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.

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
```

See Also enumObjectNames

enumLocks

Method Creates a Paradox table listing the locks currently applied to a UIObject; returns number of locks.

Type UIObject

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	Type	Size
UserName	A	15
Lock Type	A	32
Net Session	N	
Session	N	
Record Number	N	

Example In this 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
```

See Also lockStatus

TCursor::lock

TCursor::lockStatus

enumObjectNames

Method/ Procedure Fills an array with the names of the objects in a form.

Type UIObject

Syntax **enumObjectNames** (var objectNames Array[] String)

Description **enumObjectNames** fills an array with object names. If *arrayName* is resizeable, it grows automatically to hold the object names; if it is not resizeable, it holds as many as it can, and discards the rest. 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, put **enumObjectNames** in a method attached to the form. To enumerate all objects in a page, put it on the page. To enumerate all objects in a box, put it on the box.

Example In the following example, assume a custom method named **customGetObjectN** is defined for the form. The form also contains a button named *getObjectNames* and all of the fields from the *Customer* table. The **pushButton** method for *getObjectNames* calls the form's custom method to write all object names on the form to an array, then to a table.

Following is the code for the custom method **customGetObjectN**:

```
; form design::customGetObjectN  (custom method)
method customGetObjectN()
var
  objArray  Array [] String
endVar
enumObjectNames(objArray)          ; write names to array
objArray.view()                    ; show the array
enumUIObjectNames("objtable.db")   ; write names to table
endmethod
```

This code is for the **pushButton** method for *getObjectNames*:

```
; getObjectNames::pushButton
method pushButton(var eventInfo Event)
customGetObjectN()    ; call the custom method from the form
endmethod
```

See Also enumUIObjectProperties
enumUIClasses

enumSource

Method Fills a table with the source code of the methods on a form.

Type UIObject

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	Size
Object	A	128
MethodName	A	128
Source	M	64

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, put **enumSource** in a method attached to the form. To enumerate the source for all objects in a page, put it on the page. To enumerate the source for all objects in a box, put it on the box.

Example In the following example, assume a custom method named **customGetSourceToTable** is defined for the form. The form also contains a button named *getSourceToTable*. The **pushButton** method for *getSourceToTable* calls the form's custom method, which writes all source code for all objects on the form to the *CstSource* table.

Following is the **pushButton** method for *customGetSourceToTable*:

```
; customGetSourceToTable  (custom method)
method customGetSourceToTable()
self.enumSource("CstSourc.db", True)
endmethod
```

Following is the **pushButton** method for *getSourceToTable*:

```
; getSourceToTable::pushButton
method pushButton(var eventInfo Event)
customGetSourceToTable()
endmethod
```

See Also enumSourceToFile

enumUIObjectProperties

enumUIClasses

enumSourceToFile

Method Writes the source code for a form or an object to a text file.

Type UIObject

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 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, *enumSourceToFile* 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, put **enumSourceToFile** in a method attached to the form. To enumerate the source for all objects in a page, put it on the page. To enumerate the source for all objects in a box, put it on the box.

Example In the following example, assume that a custom method named **customGetSourceToFile** is defined for the form. The form also contains a button named *getSourceToFile*. The **pushButton** method for *getSourceToFile* calls the form's custom method, which writes all source code for all objects on the form to the CSTSOURCE.TXT file.

Following is the **pushButton** method for the *customGetSourceToFile* button:

```
; customGetSourceToFile  (custom method)
method customGetSourceToFile()
self.enumSourceToFile("CstSourc.txt", True)
endmethod
```

Following is the **pushButton** method for the *getSourceToFile* button:

```
; getSourceToFile::pushButton
method pushButton(var eventInfo Event)
customGetSourceToFile()
endmethod
```

See Also enumSource
enumUIObjectProperties
enumUIClasses

enumUIClasses

Method Writes a list of UIObject classes to a table.

Type UIObject

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 their property 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 table structure is

Field Name	Type	Size
ClassName	A	32
PropertyName	A	64

Example This example writes the types and properties to a table named *Tmpclass*.

```
; writeClasses::pushButton
method pushButton(var eventInfo Event)
enumUIClasses("TmpClass.db")
endmethod
```

See Also enumUIObjectNames
enumUIObjectProperties

enumUIObjectNames

Method/ Procedure Gets the names of each object in a form and writes them to a table.

Type UIObject

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	Size
ObjectName	A	128
ObjectClass	A	32

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, put **enumUIObjectNames** in a method attached to the form. To enumerate all objects in a page, put it on the page. To enumerate all objects in a box, put it on the box.

Example In the following example, assume a custom method named **customGetObjectN** is defined for the form. The form also contains a button named *getObjectNames* and all the fields from the *Customer* table. The **pushButton** method for *getObjectNames* calls the form's custom method to write all object names on the form first to an array, then to a table.

Following is the code for the custom method **customGetObjectN**:

```
; form design::customGetObjectN  (custom method)
method customGetObjectN()
var
  objArray  Array [] String
endVar
enumObjectNames(objArray)          ; write names to array
objArray.view()                    ; show the array
enumUIObjectNames("objtable.db")   ; write names to table
endmethod
```

Following is the code for *getObjectNames*' **pushButton** method:

```
; getObjectNames::pushButton
method pushButton(var eventInfo Event)
customGetObjectN()    ; call the custom method from the form
endmethod
```

See Also enumObjectNames
enumUIObjectProperties
enumUIClasses

enumUIObjectProperties

Method/ Procedure Gets the properties of each object in a form, and writes the data to a Paradox table.

Type UIObject

Syntax **enumUIObjectProperties** (const ***tableName*** String) Logical

Description **enumUIObjectProperties** gets the properties of each object in a form, and 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:).

The structure of the table is

Field Name	Type	Size
ObjectName	A	128
PropertyName	A	64
PropertyValue	A	255

Example 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
```

See Also enumUIObjectNames

enumUIClasses

execMethod

Method/ Procedure Calls a custom method that takes no arguments.

Type UIObject

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.

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)      ; find out who the
current target is
```

```
    targName = targObj.name           ; get the name of the
target
    foreach element in objPreProc     ; iterate through array
      if element = targName then      ; is there an element
for the target?
        execMethod(objPreProc[targName])  ; if so, execute the
corresponding
                                          ; custom method
      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
```

See Also methodDelete
methodGet
methodSet

forceRefresh

Method Makes an object display the current data in the underlying table, and makes a calculated field recalculate.

Type UIObject

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 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:

(1) 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.

(2) If a field object is active when you call **forceRefresh**, the table associated with that field refreshes, thereby repainting all fields dependent on it.

(3) You will not lose your current record position, provided the record still exists in the table.

(4) 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.

Example This 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 File | System Settings | AutoRefresh and entered a large value (at least 3,600 seconds) in the Network 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
```

See also currRecord

getBoundingBox

Method Returns the coordinates of the frame that bounds an object.

Type UIObject

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. Strictly speaking, 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.

Example This 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
```

See Also convertPointWithRespectTo

getPosition

Method Locates the position of an object.

Type UIObject

Syntax **getPosition** (const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt)

Description **getPosition** locates 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.

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
```

See Also setPosition

getProperty

Method Returns the value of a specified property.

Class UIObject

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
```

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
```

See Also getPropertyAsString

setProperty

getPropertyAsString

Method Returns the value of a specified property as a string.

Type UIObject

Syntax **getPropertyAsString** (const ***propertyName*** String) String

Description **getPropertyAsString** returns a string containing the value of the property specified in *propertyName*.

Example This 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
```

See Also getProperty
setProperty

getRGB

Method Finds the red, green, and blue components of a color.

Type UIObject

Syntax **getRGB** (const ***rgb*** LongInt, var ***red*** SmallInt, var ***green*** SmallInt, var ***blue*** SmallInt)

Description **getRGB** decomposes a composite color, *rgb*, to its component *red*, *green*, and *blue* values.

Example This 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
```

See Also rgb

hasMouse

Method Tells if the mouse is positioned over an object.

Type UIObject

Syntax **hasMouse** ( ) Logical

Description **hasMouse** returns True if the pointer is positioned inside the boundaries of an object; otherwise, it returns False.

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 speedbar
endif
endmethod
```

See Also MouseEvent

home

Beginner

Method Moves to the first record in a table.

Type UIObject

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)
```

Example This 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
```

See Also end
nextRecord
priorRecord

insertAfterRecord

Method Inserts a record into a table after the current record.

Type UIObject

Syntax **insertAfterRecord** ( ) Logical

Description insertAfterRecord inserts a record into a table after the current record. The table must be in Edit mode.

Example In this 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 the cursor in
CUSTSORT to the TC
  if NOT CUSTSORT.atLast() then
    CUSTSORT.insertBeforeRecord() ; inserts a blank record
before current
                                  ; behaves just like
insertRecord()
  else
    CUSTSORT.insertAfterRecord()  ; inserts a blank record
after current
  endif
  ; ... fill the record with the rest of the customer
information
else
  msgInfo("Sorry", "You must be in edit mode before inserting
a record.")
endif
```

```
endmethod
```

See also insertRecord
insertBeforeRecord

insertBeforeRecord

Method Inserts a record into a table before the current record.

Type UIObject

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)
```

Example In this 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 the cursor in
CUSTSORT to the TC
  if NOT CUSTSORT.atLast() then
    CUSTSORT.insertBeforeRecord() ; inserts a blank record
before current
                                  ; behaves just like
insertRecord()
  else
    CUSTSORT.insertAfterRecord()       ; inserts a blank
record after current
  endif
```

```
  ; ... fill the record with the rest of the customer
information
else
  msgInfo("Sorry", "You must be in edit mode before inserting
a record.")
endif
endmethod
```

See also insertRecord
insertAfterRecord

insertRecord

Beginner

Method Inserts a record into a table.

Type UIObject

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)
```

Example See the example for insertBeforeRecord.

See also insertAfterRecord
insertBeforeRecord

isContainerValid

Method Reports whether an object's container is valid.

Type UIObject

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.

Example In this example, the **arrive** built-in 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
```

See also isLastMouseClickedValid

isEdit

Beginner

Method	Reports whether an object is in Edit mode.
Type	UIObject
Syntax	**isEdit** ( ) Logical
Description	**isEdit** reports whether an object is in Edit mode.
Example	See the example for lockRecord.
See also	edit endEdit lockRecord

isEmpty

Method Reports if a table contains any records.

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.

Type UIObject

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.

Example The *cascadeDelete* button in this 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
```

See also empty

nRecords

isLastMouseClickedValid

Method Reports if the last object to receive a mouse click is valid.

Type UIObject

Syntax **isLastMouseClickedValid** ( ) Logical

Description **isLastMouseClickedValid** reports if the current form has received a mouse click since it opened.

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
```

See also islastMouseRightClickedValid

isLastMouseRightClickedValid

Method Reports if the last object to receive a right mouse click is valid.

Type UIObject

Syntax **isLastMouseRightClickedValid** ( ) Logical

Description **isLastMouseRightClickedValid** reports if the current form has received a right mouse click since it opened.

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
```

See also isLastMouseClickedValid

isRecordDeleted

Method	Reports whether the current record has been deleted (dBASE tables only).
Type	UIObject
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**.
See also	unDeleteRecord TCursor::isShowDeletedOn TCursor::showDeleted

keyChar

Beginner

Method Sends an event to an object's **keyChar** method.

Type UIObject

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** method of an object with that event.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

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
```

See also keyPhysical
KeyEvent

keyPhysical

Method Sends an event to an object's **keyPhysical** method.

Type UIObject

Syntax **keyPhysical** (const ***aChar*** SmallInt, const ***vChar*** SmallInt, const ***state*** SmallInt)

Description **keyPhysical** constructs an event and calls the built-in **keyPhysical** method of an object with that event.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

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
```

See also keyChar
KeyEvent

killTimer

Method Stops the timer associated with an object.

Type UIObject

Syntax **killTimer** ( )

Description **killTimer** stops a timer associated with an object.

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
```

See also setTimer

locate

Beginner

Method Searches for a specified value.

Type UIObject

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 match the criteria specified in one or more field/value pairs. 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 cursor returns to the current record. If a match is found, the cursor moves to that record. This operation fails if the current record cannot be posted and unlocked (for example, because of a key violation).

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
```

See also

locateNext

locatePattern

locateNextPattern

locateNext

Beginner

Method Searches forward from the current record for a specified field value.

Type UIObject

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 match the criteria specified in one or more field/value pairs. 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 cursor moves to that record. If no match is found, the cursor 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).

Example For this 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
```

See also locate

locateNextPattern

locateNextPattern

Method Locates the next record containing a field that has a specified pattern of characters.

Type UIObject

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 cursor moves to that record. If no match is found, the cursor 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*, and specify a pattern of characters in *pattern*.

You can include the 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** in String for more information about advanced match pattern operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in Session.

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..")
```

Example 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
```

This example is similar to the previous example, 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
```

See also

locate
locatePattern
locateNext
String::advMatch
String::match

locatePattern

Method Searches for a record containing a field that has a specified pattern of characters.

Type UIObject

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 cursor returns to the current record. If a match is found, the cursor 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*, and specify a pattern of characters in *pattern*.

You can include the 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** in String for more information about advanced match pattern operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in Session.

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..")
```

Example See the example for locateNextPattern.

See also

locateNextPattern

locate

String::advMatch

String::match

locatePrior

Method Searches backward for a specified field value.

Type UIObject

Syntax **1. locatePrior** (const ***fieldName*** String, const ***searchValue*** AnyType [,const ***fieldName*** String, const ***searchValue*** AnyType]*) Logical

2. locatePrior (const ***fieldNum*** SmallInt, const ***searchValue*** AnyType [,const ***fieldNum*** SmallInt, const ***searchValue*** AnyType]*) Logical

Description **locatePrior** searches backwards from the current record in a table for records whose values match the criteria specified in one or more field/value pairs. 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 cursor moves to that record. If no match is found, the cursor 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).

Example The method shown in this 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
```

See also locate

locatePriorPattern

locatePriorPattern

Method Locates the prior record containing a field that has a specified pattern of characters.

Type UIObject

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 cursor moves to that record. If no match is found, the cursor 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*, and specify a pattern of characters in *pattern*.

You can include the 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** in String for more information about advanced match pattern operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in Session.

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*")
```

Example

The method shown in this 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
```

See also

locatePattern
locatePrior
String::advMatch
String::match

lockRecord

Beginner

Method Puts a write lock on the current record.

Type UIObject

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.

Example This 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
```

The next 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 the previous example.

```
; 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
```

See also postRecord
recordStatus
unlockRecord

lockStatus

Method Returns the number of locks on a table.

Type UIObject

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.

Example This 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
```

See also enumLocks
lockRecord

menuAction

Method/ Procedure Sends an event to an object's menuAction method.

Type UIObject

Syntax **menuAction** (const ***action*** SmallInt)

Description **menuAction** constructs a MenuEvent and sends it to a built-in **menuAction** method. **action** is one of the MenuCommand constants, or a user-defined constant. ObjectPAL provides constants for **action**. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose MenuCommands. The constants appear in the Constants column. For more information on user-defined constants, see the *ObjectPAL Developer's Guide*.

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.

Example In this 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
```

See also action

methodDelete

Method Deletes a specified method.

Type UIObject

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.

Example This 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
```

See also

create
methodGet
methodSet

methodGet

Method	Returns the text of a specified method.
Type	UIObject
Syntax	**methodGet** (const ***methodName*** String) String
Description	**methodGet** returns a string containing the text of the method specified in *methodName*.
Example	See the example for methodDelete.
See also	create methodDelete methodSet

methodSet

Method	Sets the text of a specified method.
Type	UIObject
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.
Example	See the example for methodDelete.
See also	create methodDelete methodGet

mouseClick

Method	Generates a mouseClick MouseEvent and sends it to an object.
Type	UIObject
Syntax	**mouseClick** ( ) Logical
Description	**mouseClick** constructs a **mouseClick** MouseEvent and calls the built-in **mouseClick** method of an object with that event.
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
```

See also mouseUp

mouseDouble

Method Sends an event to an object's **mouseDouble** method.

Type UIObject

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** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

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
```

See also mouseRightDouble
mouseDown
mouseUp

mouseDown

Method Sends an event to an object's **mouseDown** method.

Type UIObject

Syntax **mouseDown** (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description **mouseDown** constructs an event and calls the built-in **mouseDown** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

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
```

See also mouseUp
mouseDouble
mouseRightDown
mouseRightUp

mouseEnter

Method Sends an event to an object's **mouseEnter** method.

Type UIObject

Syntax **mouseEnter** (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description **mouseEnter** constructs an event and calls the built-in **mouseEnter** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

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
```

See also mouseExit
mouseMove

mouseExit

Method Sends an event to an object's **mouseExit** method.

Type UIObject

Syntax **mouseExit** (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description **mouseExit** constructs an event and calls the built-in **mouseExit** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

Example This 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
```

See also mouseEnter

mouseMove

mouseMove

Method Sends an event to an object's **mouseMove** method.

Type UIObject

Syntax **mouseMove** (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description **mouseMove** constructs an event and calls the built-in **mouseMove** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

Example This 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
```

See also mouseEnter
mouseExit

mouseRightDouble

Method Sends an event to an object's **mouseRightDouble** method.

Type UIObject

Syntax **mouseRightDouble** (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description **mouseRightDouble** constructs an event and calls the built-in **mouseRightDouble** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard. Calling this method triggers the receiving object's **mouseRightDouble** method.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

Example This 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
```

See also mouseDouble
mouseRightUp
mouseRightDown

mouseRightDown

Method Sends an event to an object's **mouseRightDown** method.

Type UIObject

Syntax **mouseRightDown** (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description **mouseRightDown** constructs an event and calls the built-in **mouseRightDown** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

Example This 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
```

See also mouseRightUp
mouseRightDouble
mouseUp
mouseDown

mouseRightUp

Method Sends an event to an object's **mouseRightUp** method.

Type UIObject

Syntax **mouseRightUp** (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description **mouseRightUp** constructs an event and calls the built-in **mouseRightUp** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

Example This 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
```

See also mouseRightDown
mouseRightDouble
mouseDown
mouseUp

mouseUp

Method Sends an event to an object's **mouseUp** method.

Type UIObject

Syntax **mouseUp** (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description **mouseUp** constructs an event and calls the built-in **mouseUp** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyboardStates. The constants display in the Constants column. The keyboard state constants can be added together to create combined key states, such as Alt+Control.

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
```

See also mouseDown
mouseDouble
mouseRightDown
mouseRightUp
mouseRightDouble

moveTo

Beginner

Method Sets the focus to a specified object.

Type UIObject

Syntax **1.** (Method) **moveTo** ( ) Logical

2. (Procedure) **moveTo** (const ***bjectName*** String) Logical

Description **moveTo** moves the focus to a specified object. If you call **moveTo** as a procedure, specify the name of the object to move as a string in *objectName*.

Example In this 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 cursor to
LINEITEM detail
; move cursor 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
```

See also attach

moveToRecNo

Method Moves to a specific record in a dBASE table.

Type UIObject

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.

Example This 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
```

See also nRecords

moveToRecord

resync

moveToRecord

Method Moves to a specific record in a table.

Type UIObject

Syntax **1. moveToRecord** (const ***recordNum*** LongInt) Logical

2. moveToRecord (const ***tc*** TCursor) Logical

Description **moveToRecord** sets the current record to the record *recordNum*, or to the record pointed to by the TCursor *tc*. 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. This method can be very slow for dBASE tables; use **moveToRecNo** instead.

Example For an example of how to use **moveToRecord** with a TCursor, see **moveTo**.

This 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
```

See also nRecords
moveTo
moveToRecNo
resync

nextRecord

Beginner

Method Moves to the next record in a table.

Type UIObject

Syntax **nextRecord** ( ) Logical

Description **nextRecord** sets the current record to the next record in a table. It returns an error if the cursor 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)
```

Example This 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
```

See also home
end
priorRecord
moveToRecord

nFields

Method Returns the number of fields in a table.

Type UIObject

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.

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
```

See also nKeyFields

nRecords

nKeyFields

Method	Returns the number of key fields in a table.
Type	UIObject
Syntax	**nKeyFields** ( ) LongInt
Description	**nKeyFields** returns the number of key fields in a table.
Example	See the example for nFields.
See also	nFields nRecords

nRecords

Beginner

Method Returns the number of records in a table.

Type UIObject

Syntax **nRecords** ( ) LongInt

Description **nRecords** returns the number of records in a table. This operation can take a long time for dBASE tables. You can also examine the NRecords property for an object to find the number of records in the table bound to that object.

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.

Example This 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
```

See also moveToRecord
nKeyFields
nFields

pixelsToTwips

Method Converts screen coordinates from pixels to twips.

Type UIObject

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 an inch (1/20 of a printer's point).

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
```

See also twipsToPixels

postAction

Method	Posts an action to an action queue for delayed execution.
Type	UIObject
Syntax	**postAction** (const ***actionId*** SmallInt)
Description	**postAction** works like **action**, except that the action is not executed immediately. Instead, Paradox waits until the entire method has finished executing and Paradox is in a steady state. The action specified by *actionID* is posted to an action queue at the time of the method call; Paradox performs the action after the current method has finished executing.
See also	action Form**::**postAction

postRecord

Beginner

Method Posts a pending record to a table.

Type UIObject

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)
```

Example This 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
```

See also recordStatus

lockRecord

TCursor::attachToKeyViol

priorRecord

Beginner

Method Moves to the previous record in a table.

Type UIObject

Syntax **priorRecord** ( ) Logical

Description **priorRecord** sets the current record to the previous record in a table. It returns an error if the cursor 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)
```

Example This 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
```

See also home
end
nextRecord
currRecord
skip
moveToRecord

pushButton

Method	Generates a **pushButton** Event and sends it to an object.
Type	UIObject
Syntax	**pushButton** ( ) Logical
Description	**pushButton** constructs a **pushButton** Event and calls the built-in **pushButton** method of an object with that event.
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
```

See also	mouseClick
	mouseUp

recordStatus

Method Reports about the status of a record.

Type UIObject

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 examing the Inserting, Locked, Focus, and Touched properties for the record.

Example This 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
```

See also lockRecord
unlockRecord

resync

Method	Resynchronizes an object to a TCursor.
Type	UIObject
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.)
Example	See the example for insertBeforeRecord.
See also	attach moveToRecord insertRecord

rgb

Method Defines a color.

Type UIObject

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 constants (for example, LightBlue).

The color constants are listed online. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose Colors. The constants display in the Constants column.

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
```

See also getProperty

getRGB
setProperty

setFilter

Method Sets the range of records a table object can point to.

Type UIObject

Syntax **setFilter** ([const ***exactMatchVal*** AnyType,] * const ***minVal*** AnyType, const ***maxVal*** AnyType) Logical

Description **setFilter** specifies conditions for including a range of records. Records that meet the conditions are included, records that don't are filtered out. This operation fails if the current record cannot be committed or if the table object is not bound to a keyed table.

This method compares the criteria you specify with values in the corresponding fields of a table's index. To filter records based on the value of a single field, specify values in *minVal* and *maxVal.* For example, the following statement checks values in the first field of the index of each record. If a value is less than 14 or greater than 88, that record is filtered out.

```
tblObj.setFilter(14, 88)
```

To specify an exact match on a single field, assign *minVal* and *maxVal* the same value. For example, the following statement filters out all values except 55:

```
tblObj.setFilter(55, 55)
```

You can filter records 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're the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field:

```
tblObj.setFilter("Borland", "Paradox", 100, 500)
```

Calling **setFilter** without any arguments resets the filter to include the entire table.

Example For this 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.setFilter(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
```

See also switchIndex

TCursor::setFilter

setPosition

Method Sets the position of an object.

Type UIObject

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.

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)
```

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
```

See also getPosition

setProperty

Method Sets a property to a specified value.

Type UIObject

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, as in

```
myBox.Color = Red
```

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
```

See also getProperty
getPropertyAsString

setTimer

Method Starts the timer for an object.

Type UIObject

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.

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
```

See also killTimer

skip

Method Moves forward or backward a specified number of records in a table.

Type UIObject

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**).

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 cursor 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
```

See also home

end
nextRecord
priorRecord
currRecord
moveToRecord

switchIndex

Method Specifies another index to use to view the records in a table.

Type UIObject

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. 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.

Example For this 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
```

See also setFilter
TCursor::reIndex
TCursor::reIndexAll
Table::setIndex

twipsToPixels

Method	Converts screen coordinates from twips to pixels.
Type	UIObject
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 an inch (1/20 of a printer's point).
Example	See the example for pixelsToTwips.
See also	pixelsToTwips

unDeleteRecord

Method	Undeletes the current record from a dBASE table.
Type	UIObject
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.
See also	deleteRecord isRecordDeleted TCursor::isShowDeletedOn TCursor::showDeleted

unlockRecord

Beginner

Method Removes a write lock from the current record.

Type UIObject

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.

Example See the example for recordStatus.

See also

recordStatus

lockRecord

TCursor**::**attachToKeyViol

TCursor**::**didFlyAway

TCursor**::**setFlyAwayControl

view

Beginner

Method Displays the value of an object in a dialog box.

Type UIObject

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.

Example For this 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
```

See also attach

wasLastClicked

Method Tells if an object was the last object to receive a mouse click.

Type UIObject

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.

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
```

See also wasLastRightClicked
hasMouse

wasLastRightClicked

Method Tells if an object was the last object to receive a right mouse click.

Type UIObject

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.

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
```

See also wasLastClicked

hasMouse

action

Method/ Procedure Performs an action command.

Type Form

Syntax **action** (const ***actionId*** SmallInt) Logical

Description **action** performs the action represented by the constant *actionId*. ObjectPAL provides constants for actions, so you don't have to memorize numeric values. This **action** method constructs and sends an ActionEvent to the built-in **action** method of the object you specify.

Constants are listed online. To display the list, open an ObjectPAL Editor window, choose Language | Constants, then, from the Types of Constants column, choose an item beginning with Action (for example, ActionSelectCommands). The constants appear in the Constants column.

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 *ObjectPAL Developer's Guide*.

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 action event.

Note: When you call the **action** method as a procedure, ObjectPAL must make an educated guess about which object you want the action to affect. The event bubbles through the containership heirarchy until the event either reaches a container 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.

Example In this 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
```

See Also UIObject::action

attach

Method Associates a Form variable with an open form.

Type Form

Syntax **attach** ([const ***formName*** String]) Logical

Description **attach** associates a Form variable with an open form. You can use *formName* to specify a form's current title, or you can omit *formName* to attach to the form where **attach** is executing.

Example In this example, a form has two buttons: *openSites* and *attachToSites*. The **pushButton** method for *openSites* takes care of opening 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.fsl")   ; open Sitenote, default
                                 ; title will be "Form :
SITENOTE.FSL"
endmethod
```

This code is attached to the **pushButton** method for *attachToSites*:

```
; attachToSites::pushButton
method pushButton(var eventInfo Event)
var
  sitesForm  Form
endVar
sitesForm.attach("Form : SITENOTE.FSL")  ; attach to Sitenote
by its title
SITENOTE by its title

; Note that this won't work:   sitesForm.attach("Sitenote")

; 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
```

See Also getTitle
setTitle
open

bringToTop

Beginner

Method/ Procedure Brings the window to the top of the display stack and makes it active.

Type Form

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.

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
layer
endmethod
```

See Also isVisible
hide
show

close

Beginner

Method/ Procedure Closes a window.

Type Form

Syntax **1.** (Method) **close** ( )

2. (Procedure) **close** ([const ***returnValue*** AnyType])

Description **close** closes a form as if the user had chosen Close from the Control menu.

Example In this example, a form contains a button called *openAndClose*. The *Sitenote* form contains a button called *goBackButton*. The **pushButton** method for *goBackButton* uses **formReturn** to return control to the calling form:

```
; goBackButton::pushButton
method pushButton(var eventInfo Event)
formReturn()    ; return control to calling form or object
endmethod
```

The following code is attached to the **pushButton** method for a button on the current form named *openAndClose*. This method opens the *Sitenote* form to *siteForm*, then waits for it to return. Once *siteForm* returns (because the user clicked *goBackButton*), this method displays a message, pauses, then closes *siteForm*:

```
; openAndClose::pushButton
method pushButton(var eventInfo Event)
var
   siteForm Form
endVar
siteForm.open("SITENOTE")
message("To return, press the Go Back button.")
siteForm.wait()             ; wait for the form to return
msgInfo("Status", "Closing the SITENOTE form in three
seconds.")
sleep(3000)                 ; pause
; User may have used the Close box instead of the Go Back
button to close the form.
try
   siteForm.close()            ; close SITENOTE form
onFail
   return
endTry
endmethod
```

See Also open

create

Method Creates a blank form in a design window.

Type Form

Syntax **create** ( ) Logical

Description **create** creates a blank form and leaves it in a design window. You can use the UIObject type methods **create** and **methodSet** to place objects in the new form and attach methods to them. Use the Form type method **run** to open the form in a View Data window.

Example In this 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
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
```

See Also

design
load
open
UIObject::create
UIObject::methodGet
UIObject::methodSet

delayScreenUpdates

Procedure Turns delayed screen updates on or off.

Type Form

Syntax **delayScreenUpdates** (const ***yesNo*** Logical)

Description **delayScreenUpdates** postpones redrawing areas of the screen but doesn't lock the screen. You must specify Yes or No. Specifying Yes delays screen updates (redraws) until an operation is complete. Specifying No allows screen updates to occur normally.

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.

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
```

See Also System::sleep

deliver

Method Delivers a form.

Type Form

Syntax **deliver** ( ) Logical

Description **deliver** behaves like Form | Deliver. This method saves a copy of a form with an .FDL extension, which prevents users from editing the form in 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. For more information about saving and delivering forms, refer to the *User's Guide*.

Example In this 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 design window, **load** returns False, because a delivered form can't be loaded in a design window.

```
; createDeliver::pushButton
method pushButton(var eventInfo Event)
var
   newForm Form
endVar
newForm.create()              ; create a new blank form (a
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
  msgStop("Sorry", "Can't load a delivered form.")
endif
endmethod
```

See Also save

design

Method Switches a form from the View Data window to the Design window.

Type Form

Syntax **design** ( ) Logical

Description **design** switches a form from the View Data window to the 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 *ObjectPAL Developer's Guide*.

Use **run** to switch from the Design window to the View Data 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.

Example See the example for create.

See Also create
load
open
run

disableBreakMessage

Procedure Prevents program interruption by Ctrl+Break.

Type Form

Syntax **disableBreakMessage** (const ***yesNo*** Logical) Logical

Description **disableBreakMessage** lets you prevent or allow the user to interrupt a running program with Ctrl+Break.

Example In this 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
```

dmAddTable

Method/ Procedure Adds a table to a form's data model.

Type Form

Syntax **dmAddTable** (const ***tableName*** String) Logical

Description **dmAddTable** adds the table *tableName* to a form's data model.

Example In this 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
```

See Also dmHasTable
dmRemoveTable

dmGet

Method/ Procedure Retrieves a field value from a table in the data model.

Type Form

Syntax **dmGet** (const ***tableName*** String, const ***fieldName*** String, var ***datum*** AnyType) Logical

Description **dmGet** provides access to table data in a form's data model. dmGet writes to *datum* a field value from a specified table. The table specified by *tableName* must be one of the tables in the form's data model. *fieldName* must be a field in *tableName*.

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
```

See Also dmPut

dmHasTable

Method/ Procedure Reports whether a table is part of a form's data model.

Type Form

Syntax **dmHasTable** (const ***tableName*** String) Logical

Description **dmHasTable** reports whether *tableName* is a table associated with a form.

Example See the example for **dmAddTable** for an illustration of how to use **dmHasTable** as a procedure.

This 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
it
  thatForm.save()
else
  msgInfo("Status", "Stock is already in data model for
form.")
endif
thatForm.close()
endmethod
```

See Also dmAddTable
dmRemoveTable

dmLinkToFields

Method Links two tables in a form's data model based on lists of field names.

Type Form

Syntax **dmLinkToFields**(const ***masterTable*** String, const ***masterFields*** Array[] String, const ***detailTable*** String, const ***detailFields*** Array[] String) Logical

Description Links the tables specifed in *masterTable* and *detailTable* on the field names listed in *masterFields* and *detailFields* (resizeable arrays of strings). The linking fields cannot be any of the following types: Memo, Formatted Memo, Binary, Graphic, or OLE. The tables must already be in the form's data model. This method returns True if successful; otherwise, it returns False.

Example The first example creates a form, adds the Customer and Orders tables to the new form's 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's 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"

   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
```

The next 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
```

See also dmLinkToIndex
dmUnlink

dmLinkToIndex

Method Links two tables in a form's data model based on a list of field names and an index name.

Type Form

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*. The tables must already be in the form's data model. This method returns True if successful; otherwise, it returns False.

The linking fields cannot be any of the following types: Memo, Formatted Memo, Binary, Graphic, or OLE.

Example This example creates a form, adds the Customer and Orders tables to the new form's 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
```

See also dmLinkToFields
dmUnlink

dmPut

Method/ Procedure	Writes data to a table in a form's data model.
Type	Form
Syntax	**dmPut** (const ***tableName*** String, const ***fieldName*** String, const ***datum*** AnyType) Logical
Description	**dmPut** provides access to table data in a form's data model. dmPut writes *datum* to a field in a specified table. The table specified by *tableName* must be one of the tables in the form's data model. *fieldName* must be a field in *tableName*.
Example	See the example for dmGet.
See Also	dmGet

dmRemoveTable

Method/ Procedure	Removes a table from a form's data model.
Type	Form
Syntax	**dmRemoveTable** (const ***tableName*** String) Logical
Description	**dmRemoveTable** removes *tableName* from the list of tables associated with a form. Any objects on the form that depend on the table will be undefined when the table is removed.
Example	See the example for dmAddTable.
See Also	dmAddTable dmHasTable

dmUnlink

Method Unlinks two tables in a form's data model.

Type Form

Syntax **dmUnlink**(const ***masterTable*** String, const ***detailTable*** String) Logical

Description 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.

This method fails if the tables are not in the form's data model, and if they are in the form's data model but not linked.

This method returns True if successful; otherwise, it returns False.

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
```

See also dmLinkToFields
dmLinkToIndex

enumSource

Method/ Procedure Creates a table listing the methods for each object in a form.

Type Form

Syntax **enumSource** (const ***tableName*** String [, const ***recurse*** Logical]) Logical

Description **enumSource** creates a Paradox table listing every object you've 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 source code for methods on all objects contained by the form, *recurse* should be True.

Example In this 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
;enumSource("tempsorc.db", True) ; writes all source for the
                                 ; current form to TEMPSORC.DB
;tempTable.open("tempsorc.db")
;tempTable.wait()
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
```

See Also enumUIObjectNames
enumUIObjectProperties
enumSourceToFile

enumSourceToFile

Method/ Procedure Creates a file listing the methods for each object in a form.

Type 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 source code for methods on all objects contained by the form, *recurse* should be True.

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
```

See Also enumUIObjectNames
enumUIObjectProperties

enumTableLinks

Method/ Procedure Creates a table listing the tables linked to a form.

Type Form

Syntax **enumTableLinks** (const ***tableName*** String) Logical

Description **enumTableLinks** creates a Paradox table listing the names of tables linked to 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:).

The structure of tableName is

Field Name	Type	Size
Name	A	81*
Link	A	81*
LinkType	A	24*

Example In this 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.fsl")
tempTable.wait()
tempTable.close()
endmethod
```

See Also enumUIObjectNames

enumUIObjectProperties

TCursor::enumRefIntStruct

enumUIObjectNames

Method Creates a table listing the UIObjects contained in a form.

Type 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

ObjectName includes the entire path name of the object.

Example In this 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
```

See Also enumUIObjectProperties
enumSource
enumSourceToFile

enumUIObjectProperties

Method Creates a table listing the properties of each UIObject contained in a form.

Type 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

Example In this 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.")
thisForm.enumUIObjectProperties("tempprop.db")
tempTable.open("tempprop")
message("Close the table to continue.")
tempTable.wait()
tempTable.close()
endmethod
```

See Also UIObject::enumUIObjectProperties

formCaller

Procedure Creates a handle to the calling form.

Type Form

Syntax **formCaller** (var ***caller*** Form) Logical

Description 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.

Example In this 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
endmethod
```

See Also wait

formReturn

Procedure Returns control to a suspended method.

Type Form

Syntax **formReturn** ([const ***returnValue*** AnyType])

Description When a form has been called by **wait**, the calling method suspends execution until **formReturn** returns control. You can choose to return a value to the calling form in *returnValue*.

If no other form is waiting for the current form, **formReturn** closes the current form.

Example This 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.

```
; 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 otherForm. It
returns a value of "Cancel" when it returns control to the
method that called wait.
; cancelButton::pushButton
method pushButton(var eventInfo Event)
formReturn("Cancel")  ; return "Cancel" to calling form
endmethod
```

See Also formCaller

wait

getFileName

Method/ Procedure	Returns the path, file name, and extension of the associated form.
Type	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.
Example	The following example displays the file name of the current form in the status bar.

```
method pushButton(var eventInfo Event)
   message(getFileName())
endMethod
```

See also	getTitle

getPosition

Method/ Procedure	Reports the position of a window onscreen.
Type	Form
Syntax	**getPosition** (var ***x*** LongInt, var ***y*** LongInt, var ***w*** LongInt, var ***h*** LongInt)
Description	**getPosition** writes the position of a window onscreen to position and size arguments. 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).
Example	In this 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
```

See Also	setPosition

getTitle

Method/ Procedure Returns the text of the window title bar.

Type Form

Syntax **getTitle** ( ) String

Description **getTitle** returns the text in the title bar of the window containing the object.

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 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 Design window
sleep()                         ; yield!
titleText = siteForm.getTitle() ; get the Design window title
msgInfo("Title:", titleText)    ; displays "Form Design:
SITENOTE.FSL"
siteForm.close()
endmethod
```

See Also setTitle
attach

hide

Beginner

Method/ Procedure Makes a window invisible.

Type Form

Syntax **hide** ( )

Description **hide** makes a window invisible but doesn't close it.

Example In this 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
```

See Also

show
bringToTop
open
openAsDialog

hideSpeedBar

Procedure Makes the SpeedBar invisible.

Type Form

Syntax **hideSpeedBar** ( )

Description **hideSpeedBar** removes the SpeedBar from the Desktop. You must call **showSpeedBar** to restore it.

Example In this example, the **pushButton** method for the *toggleSpeedBar* button checks whether the SpeedBar is showing. If the SpeedBar is visible, this method hides it; if the SpeedBar isn't visible, this method shows it:

```
; toggleSpeedBar::pushButton
method pushButton(var eventInfo Event)
if isSpeedBarShowing() then   ; if speedbar is off
   hideSpeedBar()             ; hide it
else                          ; otherwise
   showSpeedBar()             ; show it
endif
endmethod
```

See Also isSpeedBarShowing

showSpeedBar

isMaximized

Method/ Procedure Reports whether a window is displayed at its maximum size.

Type Form

Syntax **isMaximized** ( ) Logical

Description **isMaximized** returns True if a form is displayed full screen; otherwise, it returns False.

Example In this 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
```

See Also maximize
minimize
isMinimized

isMinimized

Method/ Procedure	Reports whether a window is displayed as an icon.
Type	Form
Syntax	**isMinimized** ( ) Logical
Description	**isMinimized** returns True if a form is displayed as an icon; otherwise, it returns False.
Example	See the example for isMaximized.
See Also	isMaximized minimize maximize

isSpeedBarShowing

Procedure	Reports whether the SpeedBar is visible.
Type	Form
Syntax	**isSpeedBarShowing** ( ) Logical
Description	**isSpeedBarShowing** returns True if the SpeedBar is visible; otherwise, it returns False.
Example	See the example for hideSpeedBar.
See Also	hideSpeedBar showSpeedBar

isVisible

Method/ Procedure Reports whether any part of a window is displayed.

Type Form

Syntax **isVisible** ( ) Logical

Description **isVisible** returns True if any part of a window is displayed (not hidden); otherwise, it returns False.

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
```

See Also hide
show
bringToTop

keyChar

Method Sends an event to a form's **keyChar** method.

Type Form

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 constant in *state*. For syntax 2, you can specify a string of one or more characters and, optionally, include a keyboard state constant.

Example In this 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
```

See Also keyPhysical

keyPhysical

Method Sends an event to a form's **keyPhysical** method.

Type Form

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 constant in *state*.

Example In this 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.vCharAnsiCode(),
                            Control)
    otherwise:
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharAnsiCode(),
                            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
```

See Also keyChar

load

Method Opens a form in the Design window.

Type Form

Syntax **load** (const ***formName*** String) Logical

Description **load** opens *formName* in the Design window. 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 View Data window. To switch from the Design window to View Data window, use **run**. To switch from View Data window to the Design window, use **design**.

In either the Design window or the View Data 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 View Data 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** method in the System type for more information.

Example In the following example, the **pushButton** method for a button named *drawABox* loads the *Sitenote* form in a 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 View Data 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 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
```

See Also create
open
openAsDialog
design

maximize

Beginner

Method/ Procedure Maximizes a window.

Type Form

Syntax **maximize** ( )

Description **maximize** displays a window at its full size. Calling this method is equivalent to choosing Maximize from the Control menu.

Example In this 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.fsl")
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
```

See Also minimize
isMinimized
isMaximized
show

menuAction

Method/ Procedure Sends an event to a form's menuAction method.

Type Form

Syntax **menuAction** (const ***action*** SmallInt) Logical

Description **menuAction** constructs a MenuEvent and sends it to a form's built-in **menuAction** method. **action** is one of the MenuCommand constants, or a user-defined constant. ObjectPAL provides constants for **action**. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose MenuCommands. The constants appear in the Constants column. For more information on user-defined constants, see the *ObjectPAL Developer's Guide*.

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.

Example In this 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
```

See Also action

methodDelete

Method Deletes a form-level method from a form.

Type 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 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)

Example In this example, two forms are on the desktop in a 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 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
```

See Also

methodGet

methodSet

UIObject::create

UIObject::methodGet

UIObject::methodSet

methodGet

Method	Gets a form-level method.
Type	Form
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). For more information about saving and delivering forms, refer to the *User's Guide*.
Example	See the example for methodDelete.
See Also	methodDelete methodSet enumSourceToFile UIObject::create UIObject::methodGet UIObject::methodSet

methodSet

Method	Sets the definition of a method attached to a form.
Type	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). For more information about saving and delivering forms, refer to the *User's Guide*.
Example	See the example for methodDelete.
See Also	methodDelete methodGet UIObject::create UIObject::methodGet UIObject::methodSet

minimize

Method/ Procedure	Minimizes a window.
Type	Form
Syntax	**minimize** ( )
Description	**minimize** displays a window as an icon. Calling this method is equivalent to choosing Minimize from the Control menu.
Example	See the example for maximize.
See Also	maximize isMaximized isMinimized show

mouseDouble

Method Sends an event to a form's **mouseDouble** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this example, the form *Othermse* is open in the View Data 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
```

See Also mouseDown
mouseRightDouble

mouseDown

Method Sends an event to a form's **mouseDown** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this example, the form *Othermse* is open in the View Data 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
```

See Also mouseRightDown

mouseUp

mouseEnter

Method Sends an event to a form's **mouseEnter** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this example, the form *Othermse* is open in the View Data 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
```

See Also mouseExit

mouseExit

Method Sends an event to a form's **mouseExit** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this example, the form *Othermse* is open in the View Data 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 = "mouseDown"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endmethod
```

See Also mouseEnter

mouseMove

Method Sends an event to a form's **mouseMove** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this example, the form *Othermse* is open in the View Data 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
```

See Also mouseEnter
mouseExit

mouseRightDouble

Method Sends an event to a form's **mouseRightDouble** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this example, the form *Othermse* is open in the View Data 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
```

See Also mouseDouble
mouseRightDown

mouseRightDown

Method Sends an event to a form's **mouseRightDown** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this example, the form *Othermse* is open in the View Data 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
```

See Also mouseRightDouble

mouseDown

mouseRightUp

Method Sends an event to a form's **mouseRightUp** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this 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
```

See Also mouseUp
mouseRightDown

mouseUp

Method Sends an event to a form's **mouseUp** method.

Type Form

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. ObjectPAL provides constants for *state*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose KeyBoardStates. The constants appear in the Constants column.

Example In this example, the form *Othermse* is open in the View Data 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
```

See Also mouseRightUp
mouseDown

moveTo

Method Moves to a form.

Type Form

Syntax **moveTo** ([const ***objectName*** String]) Logical

Description **moveTo** moves the focus to a form. Optionally, it moves to the object specified in *objectName*.

Example In the following example, a form named *Sitenote* is already open in the View Data 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
```

See Also moveToPage

moveToPage

Method/ Procedure Displays a specified page of a form.

Type 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.

Example In this 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
```

See Also bringToTop

open

Beginner

Method Opens a window.

Type Form

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 View Data 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. 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", WinStyleVScroll + WinStyleHScroll)
```

Valid values for *windowStyle* are listed online. To display the list, open an ObjectPAL Editor window, choose Language | Constants, then from the Types of Constants column, choose WindowStyles. The constants appear in the Constants column.

Compare this method with **load**, which opens a form in a design window.

Syntax 3 lets you specify form settings from *openInfo*, a record of type FormOpenInfo. The FormOpenInfo record has been 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  ;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.

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.

Example

In this 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
```

This 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
```

See Also

close
create
load
openAsDialog
design

openAsDialog

Method Opens a Form window as a dialog box.

Type Form

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 View Data 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. 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.open("sales", WinStyleVScroll + WinStyleHScroll)
```

Valid values for *windowStyle* are listed online. To display the list, open an ObjectPAL Editor window, choose Language | Constants, then, from the Types of Constants column, choose WindowStyles. The constants appear in the Constants column.

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
```

Example

In this 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") ; give the dialog
helpForm.wait()
  helpForm.close()
  disableDefault                        ; don't call Help
system
endIf
endmethod
```

See Also

open
wait
formCaller
formReturn

postAction

Method Posts an action to an action queue for delayed execution.

Type Form

Syntax **postAction** (const ***actionId*** SmallInt)

Description **postAction** works like **action**, except that the action is not executed immediately. Instead, Paradox waits until the entire method has finished executing and Paradox is in a steady state. The action specified by *actionID* is posted to an action queue at the time of the method call; Paradox performs the action after the current method has finished executing.

Example In this 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 after the message dialog box appears and after this method ends:

```
; 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
```

See Also action

run

Method Switches a form from the Design window to the View Data window.

Type Form

Syntax **run** ( ) Logical

Description **run** switches a form from the Design window to the View Data 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 *User's Guide*.

To switch from the View Data window to the 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 informatiose **design**.

Example This example opens the *Sitenote* form in a 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
```

See Also design

save

Method	Saves a form to disk.
Type	Form
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 design window. You can use *newFormName* to specify a name for the form, or you can omit it. If you omit *newFormName* and the form doesn't have a name already, Paradox displays a dialog box to prompt the user to enter a name. If the form already has a name, Paradox saves it using that name.
Example	See the example for create.
See Also	create design

setPosition

Method/ Procedure	Positions a window onscreen.
Type	Form
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). For dialog boxes and for the Paradox desktop application, the position must be given relative to the entire screen; for forms, reports, and table windows, the position must be given relative to the Paradox Desktop.
Example	See the example for getPosition.
See Also	open

setTitle

Method/ Procedure	Sets the text in the window title bar.
Type	Form
Syntax	**setTitle** (const ***text*** String)
Description	**setTitle** changes the text of the window title bar to text. 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.)
Example	See the example for openAsDialog.
See Also	getTitle attach

show

Beginner

Method/ Procedure	Displays a minimized window at its previous size; makes a hidden form visible.
Type	Form
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.
Example	See the example for hide.
See Also	hide isVisible

showSpeedBar

Procedure	Makes the SpeedBar visible.
Type	Form
Syntax	**showSpeedBar** ( )
Description	**showSpeedBar** displays the SpeedBar.
Example	See the example for hideSpeedBar.
See Also	hideSpeedBar isSpeedBarShowing

wait

Beginner

Method	Suspends execution of a method.
Type	Form
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).
Example	See the example for formReturn.
See Also	formReturn formCaller openAsDialog

windowClientHandle

Method/ Procedure Returns the handle of a window.

Type Form

Syntax **windowClientHandle** ( ) SmallInt

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) written in C, C++, or Pascal.

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 CWORD)
endUses
doSomething(windowClientHandle())  ; call doSomething and
supply the
                                   ; handle of the client
portion
                                   ; of the current form
endmethod
```

See Also windowHandle

windowHandle

Method/ Procedure	Returns the handle of a window.
Type	Form
Syntax	**windowHandle** ( ) SmallInt
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) written in C, C++, or Pascal.
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 CWORD)
endUses
doSomething(windowHandle())  ; call doSomething and supply the
                             ; window handle of the current
form
endmethod
```

See Also	windowClientHandle

attach

Method Associates a Report variable with an open Report.

Type Report

Syntax **attach** (const ***reportTitle*** String) Logical

Description **attach** associates a Report variable with an open report. *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.

Example In this 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
```

See Also open

Form::getTitle

Form::setTitle

close

Method Closes a window.

Type Report

Syntax **close** ( )

Description **close** closes a Report window. Closing a report with **close** is equivalent to choosing Close from the Control menu.

Example In this 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
```

See Also open

currentPage

Method Returns the current page number of a report.

Type Report

Syntax **currentPage** ( ) SmallInt

Description **currentPage** returns the current page number of a report.

Example In this 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
```

See Also moveToPage

design

Method Switches a report from a View Data window to a Design window.

Type Report

Syntax **design** ( ) Logical

Description **design** switches a report from the View Data window to the Design window. This method works only with saved reports (.RSL); it does not work with delivered reports (.RDL). For more information about saving and delivering reports, refer to the *User's Guide*.

Use **run** to switch from a Design window to a View Data window, or load to open a report in a 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.

Example In this 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 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
```

See Also load
open
run

enumUIObjectNames

Method Creates a table listing the UIObjects contained in a report.

Type 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

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
```

See Also enumUIObjectProperties

enumUIObjectProperties

Method Creates a table listing the properties of each UIObject contained in a report.

Type 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

Example See the example for enumUIObjectNames.

See Also enumUIObjectNames

load

Method Opens a report in the Design window.

Type Report

Syntax **load** (const ***reportName*** String) Logical

Description **load** opens *reportName* in the Design window. This method works only with saved reports (.RSL); it does not work with delivered reports (.RDL). For more information about saving and delivering reports, refer to the *User's Guide*.

Compare this method to **open**, which opens a report in the View Data 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.

Example In this example, the **pushButton** method for the *loadOrders* button loads the ORDERS.RSL report in the 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
```

See Also design

run

open

moveToPage

Method	Displays a specified page of a report.
Type	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**).
Example	See the example for currentPage.
See Also	Form::bringToTop currentPage

open

Method Opens a report and prints it.

Type 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 and prints 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*).

ObjectPAL provides constants for valid values for *windowStyle*. To display the list, open an Editor window and choose Language | Constants. Then, from the Types of Constants column, choose WindowStyles. The constants appear in the Constants column.

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", WinStyleHScroll +
WinStyleVScroll)
```

Syntax 3 lets you specify form settings from *openInfo*, a record of type ReportOpenInfo. A ReportOpenInfo record 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
```

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.

Example In this 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
```

See Also

close

design

load

print

Beginner

Method Prints a report.

Type 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 and set restart options. Syntax 3 lets you set print settings with a ReportPrintInfo record. ReportPrintInfo records are predeclared and have the following structure:

```
name             String   ;run this report if not already open
masterTable      String   ;master table name
queryString      String   ;run this query (actual query string)
restartOptions   SmallInt ;do what when data changes while
printing report
                          ;one of the ReportPrintRestart
constants
printBackwards   Logical  ;forward FALSE, backward TRUE,
default false
makeCopies       Logical  ;Who does copies: Paradox or printer?
                          ;If True, Paradox make copies
panelOptions     SmallInt ;one of the ReportPrintPanel
                           constants
nCopies          SmallInt ;number of copies, default is 1
startPage        LongInt  ;starting page, default is 1
endPage          LongInt  ;ending page def: ending page
pageIncrement    SmallInt ;Page increment for multi-pass
                          ;printing, default is 1
xOffset          LongInt  ;horizontal page offset
yOffset          LongInt  ;vertical page offset
orient           SmallInt ;Landscape or Portrait
```

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
```

See Also open

run

run

Method Switches a report from the Design window to the View Data window.

Type Report

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). For more information about saving and delivering reports, refer to the *User's Guide*.

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** method in the System type for more information.

Example See the example for load.

See Also design
load

action

Method Performs an action command.

Type TableView

Syntax **action** (const ***actionID*** SmallInt) Logical

Description **action** performs the action represented by the constant *actionId*. ObjectPAL provides constants for actions, so you don't have to memorize numeric values.

Constants are listed online. To display the list, open an ObjectPAL Editor window, choose Language | Constants, then, from the Types of Constants column, choose an item beginning with Action (for example, ActionSelectCommands). The constants appear in the Constants column.

Example This 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 Closes a table window.

Type TableView

Syntax **close** ( )

Description **close** closes a table window, equivalent to choosing Close from the Control menu.

Example In this 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
```

See Also open

moveToRecord

Method Moves to a specific record in a table.

Type TableView

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.

Example This 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
```

See Also action

open

Method Opens a table window.

Type TableView

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.

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
```

See Also close

wait

Method	Suspends execution of a method.
Type	TableView
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.
Example	See the example for open.
See Also	close open

actionClass

Method Returns the class number of an ActionEvent.

Type ActionEvent

Syntax **actionClass** ( ) SmallInt

Description **actionClass** returns the class number of an ActionEvent. ObjectPAL defines constants for these class numbers (for example, DataAction), so you don't have to remember numeric values.

Constants are listed online. To display the list, open an ObjectPAL Editor window, choose Language | Constants. Then, from the Types of Constants column, select ActionClasses. The constants appear in the Constants column.

Example This 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(1)
    beep()
    message("Sorry. Can't make changes to this field.")
  endif
endif
endmethod
```

See Also id
setId
Event::getTarget

id

Beginner

Method Returns the ID number of an ActionEvent.

Type 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.

Constants are listed online. To display the list, open an ObjectPAL Editor window, choose Language | Constants. Then, from the Types of Constants column, select an item beginning with Action (for example, ActionDataCommands). The constants appear in the Constants column.

You can also send user-defined actions to a built-in action method. For more information about creating and using user-defined constants, see the *ObjectPAL Developer's Guide*.

Example This 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(1)     ; don't start Edit mode
      msgStop("Sorry", "View only - can't edit this form")
    endif
endif
endmethod
```

See Also actionClass

setId

setId

Method Specifies an ActionEvent.

Type 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.

Constants are listed online. To display a list of valid values for *actionId*, open an ObjectPAL Editor window. Choose Language | Constants. From the Types of Constants column, choose an item beginning with Action (for example, ActionDataCommands). The constants appear in the Constants column.

You can also send user-defined actions to a built-in action method.

Example In this example, the SpeedBar 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
```

See Also id

reason

Method Reports why an error occurred.

Type ErrorEvent

Syntax **reason** ( ) SmallInt

Description **reason** returns an integer value to report why an ErrorEvent occurred. ObjectPAL provides the following constants for testing the value returned by **reason**:

ErrorCritical indicates that an error message will appear in a dialog box.

ErrorWarning means that an error message will appear in the status line.

Note: Do not confuse **reason** with **errorCode** (see the Event type for a description of **errorCode**).

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
```

See Also Event::errorCode

setReason

setReason

Method Specifies a reason for generating an ErrorEvent.

Type ErrorEvent

Syntax **setReason** (const ***reasonId*** SmallInt)

Description **setReason** specifies a reason for generating an ErrorEvent. This method takes one of the following reason constants as an argument:

ErrorCritical indicates that an error message will appear in a dialog box.

ErrorWarning means that an error message will appear in the status line.

Example The following example creates an ErrorEvent, sets the reason to ErrorCritical, 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
```

See Also reason

errorCode

Beginner

Method Reports the status of an error flag.

Type Event

Syntax **errorCode** ( ) LongInt

Description **errorCode** returns an error code if there is an error; otherwise, **errorCode** returns 0.

Example In this example, assume that a form contains a table frame bound to the *Customer* table, a button named *startEdit*, and another button named *sendAnError*. The *startEdit* **pushButton** method examines the event; if the event doesn't have an error, the method starts Edit mode on *CUSTOMER*. For the purposes of example only, the *sendAnError* button creates an event, sets its error to 1, then calls the **pushButton** method of *startEdit* with the event. The following code is attached to the **pushButton** method for *startEdit*:

```
; startEditButton::pushButton
method pushButton(var eventInfo Event)
; check the event to see if it has an error
if eventInfo.errorCode() = 0 then
  CUSTOMER.action(DataBeginEdit)  ; if no error, then start
Edit mode
endif
endmethod
```

The following method is attached to the **pushButton** method for *sendAnError*:

```
; sendAnError::pushButton
method pushButton(var eventInfo Event)
var
  ev Event                      ; the event to dispatch
endVar
ev.setErrorCode(1)              ; set an error of 1
startEditButton.pushButton(ev)  ; send the event to the
startEditButton
endmethod
```

See Also setErrorCode

getTarget

Method Creates a handle to the target of an Event.

Type 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.

Example This 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
```

See Also isPreFilter

isFirstTime

Method Reports whether the form is handling an Event for the first time before dispatching it.

Type Event

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 methods attached to the form.

Example This 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 in this
situation.
  msgInfo("Status", "About to dispatch a pushButton event to "
                      + targObj.Name + ".")

  case NOT eventInfo.isTargetSelf() AND NOT
eventInfo.isFirstTime() :
    ; This happens when event has been explicitly bubbled back
to the form.
    ;   isPrefilter returns False in this situation.
  msgInfo("Status", "This dialog appears only when 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 heirarchy
endmethod
```

See Also isPreFilter

isPreFilter

Method Reports whether the form is handling an Event on its own behalf.

Type Event

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.

Note: Form methods are *not* prefiltered. In other words, when an Event occurs for the form, **isPreFilter** returns False.

Example See the example for getTarget.

See Also isFirstTime

isTargetSelf

Method	Reports whether an object is the target of an Event.
Type	Event
Syntax	**isTargetSelf** ( ) Logical
Description	**isTargetSelf** reports whether an object is the target of an Event. Use **isTargetSelf** in built-in methods attached to the form.
Example	See the example for isFirstTime.
See Also	isFirstTime

reason

Method Reports why an Event occurred.

Type Event

Syntax **reason** ( ) SmallInt

Description **reason** returns an integer value to report why an Event occurred. **reason** returns valid reason constants only for Events generated for the built-in **newValue** method (see the examples). ObjectPAL provides the following constants for testing the value returned by **reason**:

FieldValue means that the value of a field was changed by scrolling, by refresh across the network, by an ObjectPAL statement, or by a user changing the value of a field.

EditValue means that a value was specified by clicking a radio button or choosing an item from a list.

Note: Reason constants are also defined for ErrorEvents, MenuEvents, StartupValue means that a value was specified when the form was opened.

MoveEvents, and StatusEvents. See the entry for **reason** in those sections for examples. 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).

Example In this example, assume that a form contains a multi-record 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. 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
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
```

See Also

MoveEvent
StatusEvent
ErrorEvent
ValueEvent
setReason

setErrorCode

Beginner

Method	Sets the error code for an Event.
Type	Event
Syntax	**setErrorCode** (const ***errorId*** LongInt)
Description	**setErrorCode** sets the error code. If *errorId* is 0, it means "no error." Any nonzero value for *errorId* indicates an error. ObjectPAL provides constants for *errorId*, so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose EventReturns. The constants appear in the Constants column.
Example	See the example for errorCode.
See Also	errorCode

setReason

Method Specifies a Reason for generating a move.

Type Event

Syntax **setReason** (const ***reasonId*** SmallInt)

Description **setReason** specifies a Reason for generating an Event. This method takes one of the following Reason constants as an argument:

FieldValue means that the value of a field was changed by scrolling, by refresh across the network, by an ObjectPAL statement, or by a user changing the value of a field.

EditValue means that a value was specified by clicking a radio button or choosing an item from a list.

Note: Reason constants are also defined for ErrorEvents, MenuEvents, StartupValue means that a value was specified when the form was opened.

MoveEvents, and StatusEvents. See the entry for **setreason** in those sections for examples. The **setreason** method is valid for the other event types (ActionEvent, KeyEvent, MouseEvent, and ValueEvent), but you can use it only to set user-defined reason constants (an advanced technique).

Example In this example, assume that a form contains a multi-record 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
```

See Also reason

char

Method Returns the character assocated with a keypress.

Type KeyEvent

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**).

Example This 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)
var
  msg String
endVar
doDefault                    ; put character in the field
message(eventInfo.char())    ; then display character as a
message
endmethod
```

See Also charAnsiCode

setChar

vChar

String::ansiCode

String::chrToKeyName

String::toANSI

String::toOEM

charAnsiCode

Method Returns the ANSI value assocated with a keypress.

Type KeyEvent

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.

Example This example beeps when a user presses Backspace or Ctrl+H. This code is attached to a field object's built-in **keyPhysical** method.

```
; thisField::keyPysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.charAnsiCode() = 8 then  ; if user presses Ctrl+H
or Backspace
  beep()                              ; make a sound
endif
endmethod
```

See Also

char
vChar
setVChar
String::ansiCode
String::chrToKeyName
String::toANSI
String::toOEM

isAltKeyDown

Method Reports whether Alt was held down during a KeyEvent.

Type KeyEvent

Syntax **isAltKeyDown** ( ) Logical

Description **isAltKeyDown** returns True if Alt was held down at the time a KeyEvent occurred; otherwise, it returns False.

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
```

See Also isControlKeyDown
isShiftKeyDown
setAltKeyDown

isControlKeyDown

Method	Reports whether Ctrl was held down during a KeyEvent.
Type	KeyEvent
Syntax	**isControlKeyDown** ( ) Logical
Description	**isControlKeyDown** returns True if Ctrl was held down at the time a KeyEvent occurred; otherwise, it returns False.
Example	See the example for setControlKeyDown.
See Also	setControlKeyDown isAltKeyDown isShiftKeyDown

isFromUI

Method	Reports whether an event was generated by the user interacting with Paradox.
Type	KeyEvent
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.
See Also	Event::isPreFilter

isShiftKeyDown

Method	Reports whether Shift was held down during a KeyEvent.
Type	KeyEvent
Syntax	**isShiftKeyDown** ( ) Logical
Description	**isShiftKeyDown** returns True if Shift was held down at the time a KeyEvent occurred; otherwise, it returns False.
See Also	setShiftKeyDown isAltKeyDown isControlKeyDown

setAltKeyDown

Method Simulates pressing and holding Alt during a KeyEvent.

Type 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.

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
```

See Also isAltKeyDown

setControlKeyDown

setShiftKeyDown

setChar

Method Specifies an ANSI character for a KeyEvent.

Type 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).

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 keystokes
normally
endmethod
```

See Also

setVChar
setVCharCode
String::toOEM
String::chr
String::chrOEM
String::keyNameToChr
String::toANSI

setControlKeyDown

Method Simulates pressing and holding Ctrl during a KeyEvent.

Type 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.

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
```

See Also isControlKeyDown
setAltKeyDown
setShiftKeyDown

setShiftKeyDown

Method Simulates pressing and holding Shift during a KeyEvent.

Type 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.

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 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
```

See Also isShiftKeyDown
setAltKeyDown
setControlKeyDown

setVChar

Method	Specifies a Windows virtual character for a KeyEvent.
Type	KeyEvent
Syntax	**setVChar** (const ***char*** String)
Description	**setVChar** specifies in *char* a one-character string for a KeyEvent. Use **setVChar** to set a virtual character code string for a single letter. 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).
Example	See the example for setAltKeyDown.
See Also	setChar setVCharCode String::chr String::chrOEM String::chrToKeyName

setVCharCode

Method Specifies a Windows virtual character for a KeyEvent.

Type KeyEvent

Syntax **setVCharCode** (const ***VK_Constant*** SmallInt)

Description **setVCharCode** specifies in *VK_Constant* a Windows virtual character constant for a KeyEvent. Valid values for *VK_Constant* are also listed online. To display the list, open an ObjectPAL Editor window. Choose Language | Constants. Then, from the Types of Constants column, choose Keyboard. The constants appear in the Constants column.

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
```

See Also setChar
setVChar
String::keyNameToVKCode

vChar

Method Returns a Windows virtual character.

Type KeyEvent

Syntax **vChar** ( ) String

Description **vChar** returns a Windows virtual key name as a string. Windows virtual characters are listed in the *ObjectPAL Reference Guide.*

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 positon 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
postion of box
    x = posPt.x()                   ; x stores the horizontal
position
    y = posPt.y()                   ; y stores the vertical
postion

    ; 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
keystoke
    boxOne.position = Point(x,y)
    message("Value of vChar() was " + kp)

  else
    ;code here executes just for form itself
endif
endmethod
```

See Also

char
String::ansiCode
String::chrToKeyName
String::toANSI
String::toOEM

vCharCode

Method Returns the integer value of a Windows virtual character.

Type KeyEvent

Syntax **vCharCode** ( ) SmallInt

Description **vCharCode** returns the integer value of a Windows virtual character. Windows virtual characters are listed in the *ObjectPAL Reference Guide.*

ObjectPAL provides constants for virtual characters (for example VK_RETURN), so you don't have to remember numeric values. To display the list, open an ObjectPAL Editor window, choose Language | Constants, then, from the Types of Constants column, choose Keyboard. The constants display in the Constants column.

Example For this example, assume a form has a field named *thisField*. When the user types a value in *thisField* and presses Return, the code creates an 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
  qs 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
  qs = Query

         c:\pdoxwin\sample\biolife.db | Common Name  | Species
Name   |
                                      | check ~cName | check
|

       endQuery

  executeQBE(qs, "myFish.db")    ; run query, write contents
to myFish table
  tv.open("myFish")              ; view myFish view
endif
endmethod
```

See Also vChar

data

Method	Returns information about a MenuEvent.
Type	MenuEvent
Syntax	**data** ( ) LongInt
Description	This method 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.
See Also	id setData setId

id

Beginner

Method Returns the ID of a MenuEvent.

Type MenuEvent

Syntax **id** ( ) SmallInt

Description **id** returns the ID number of a MenuEvent. ObjectPAL provides constants (like MenuFileOpen) for many common menu choices. Constants are listed online. To display the list, open an ObjectPAL Editor window, choose Language | Constants, then, from the Types of Constants column, choose MenuCommands. The constants appear in the Constants column.

Example 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)
var
  theFile String
  tv TableView
endVar
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
```

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
```

See Also setId

isFromUI

Method	Reports whether an event was generated by the user interacting with Paradox.
Type	MenuEvent
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.
See Also	Event::isPreFilter

menuChoice

Beginner

Method Returns a string containing an item chosen from a menu.

Type MenuEvent

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
```

Example This 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 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
```

See Also

id

setId

reason

Method Reports the type of menu chosen.

Type MenuEvent

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 the following constants for testing the value returned by **reason**:

MenuNormal means the MenuEvent was caused by a menu command or SpeedBar button that changes depending on which window you're using.

MenuControl means the MenuEvent was caused by a System menu command or by the maximize or minimize button.

MenuDesktop means the means the MenuEvent was caused by one of the basic desktop menu commands.

Example In this 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
```

See Also setReason

setData

Method	Specifies information about a MenuEvent.
Type	MenuEvent
Syntax	**setData** (const ***menuData*** LongInt)
Description	This method 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.
See Also	data id setId

setId

Method Specifies the ID of a MenuEvent.

Type MenuEvent

Syntax **setId** (const ***actionId*** SmallInt)

Description **setId** specifies in *actionId* an action to take as the result of a menu choice. ObjectPAL defines constants for *actionId* so you don't have to remember numeric values.

Constants are listed online. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose MenuCommands. The constants appear in the Constants column.

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 that MenuEvent with **setReason**.

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.

Example See the example for id.

See Also id
Event::errorCode
Event::setErrorCode
Form::menuAction
UIObject::menuAction

setReason

Method Specifies a reason for generating a MenuEvent.

Type MenuEvent

Syntax **setReason** (const ***reasonId*** SmallInt)

Description **setReason** specifies a reason for generating a MenuEvent. This method takes one of the following MoveReason constants as an argument:

MenuNormal means the MenuEvent was caused by a menu command or SpeedBar button that changes depending on which window you're using.

MenuControl means the MenuEvent was caused by a System menu command or by the maximize or minimize button.

MenuDesktop means the MenuEvent was caused by a one of the basic desktop menu commands.

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.

Example See the example for id.

See Also reason

id

Form::menuAction

UIObject::menuAction

getMousePosition

Method Returns the mouse position as a Point.

Type MouseEvent

Syntax **1. getMousePosition** (var ***p*** Point)

2. getMousePosition (var ***xPosition*** LongInt, ***yPosition*** LongInt)

Description **getMousePosition** returns the mouse position. Syntax 1 stores the value in a Point variable, *p*. Syntax 2 stores the value in *xPosition* and *yPosition*, two LongInt variables representing the x- and y-coordinates of the mouse pointer. When you use Syntax 1, you can use Point type methods (for example, **isLeft** and **isRight**) to get more information.

This method gets the mouse position at the time of the MouseEvent. The current mouse position may be different.

Example This 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, 100, 100)
  crObj.Color = DarkBlue
  crObj.Visible = True
endif
endmethod
```

See Also setMousePosition

getObjectHit

Event::getTarget

getObjectHit

Method Creates a handle to the UIObject that received the event.

Type MouseEvent

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 **mouseExit** and **mouseEnter.getObjectHit** can return a different object than **getTarget** during a **mouseExit** or **mouseEnter** method.

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
```

See Also getMousePosition

Event::getTarget

isControlKeyDown

Method Reports whether the user has held (or is holding) down Ctrl during a MouseEvent.

Type MouseEvent

Syntax **isControlKeyDown** ( ) Logical

Description **isControlKeyDown** returns True if Ctrl is held down during a MouseEvent; otherwise, it returns False.

Example This 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
```

See Also setControlKeyDown
isShiftKeyDown

isFromUI

Method	Reports whether an event was generated by the user interacting with Paradox.
Type	MouseEvent
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.
See Also	Event**::**isPreFilter

isInside

Method Reports whether the mouse is inside the border of the target object.

Type MouseEvent

Syntax **isInside** ( ) Logical

Description **isInside** reports whether the mouse is inside the border of the target object at the time of the event.

Example In this 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
```

See Also getObjectHit
setInside

isLeftDown

Method Reports whether the left (or primary) mouse button is held down during a MouseEvent.

Type 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.

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
```

See Also isRightDown
isMiddleDown
setLeftDown

isMiddleDown

Method Reports whether the middle mouse button is held down during a MouseEvent.

Type 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.

Example This 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
```

See Also setMiddleDown
isLeftDown
isRightDown

isRightDown

Method Reports whether the right mouse button is pressed during a MouseEvent.

Type 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.

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
```

See Also setRightDown
isLeftDown
isMiddleDown

isShiftKeyDown

Method Reports whether Shift is held down during a MouseEvent.

Type MouseEvent

Syntax **isShiftKeyDown** ( ) Logical

Description **isShiftKeyDown** returns True if Shift is held down during a MouseEvent; otherwise, it returns False.

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
```

See Also isControlKeyDown
setShiftKeyDown

setControlKeyDown

Method Simulates pressing and holding Ctrl during a MouseEvent.

Type 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.

Example The following example creates a MouseEvent and sets Ctrl to Yes. The event is then sent to the **mouseUp** built-in 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
```

See Also isControlKeyDown

setShiftKeyDown

setInside

Method Sets the mouse to be inside the current object.

Type MouseEvent

Syntax **setInside** (const ***TrueFalse*** Logical) Logical

Description **setInside** sets the MouseEvent to be inside the current object.

Example In this 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
```

See Also isInside
getObjectHit

setLeftDown

Method Simulates pressing the left mouse button.

Type MouseEvent

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.

Example This 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
```

See Also isLeftDown
setMiddleDown
setRightDown

setMiddleDown

Method Simulates pressing the middle mouse button.

Type MouseEvent

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.

Example This 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
```

See Also isMiddleDown
setLeftDown
setRightDown

setMousePosition

Method Sets the position of the mouse for an event.

Type MouseEvent

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.

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
```

See Also getMousePosition

setRightDown

Method Simulates pressing the right mouse button.

Type MouseEvent

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.

Example This 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
```

See Also isRightDown
setMiddleDown
setLeftDown

setShiftKeyDown

Method Simulates pressing and holding *Shift*.

Type MouseEvent

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.

Example The following example creates a MouseEvent and sets Shift to Yes. The event is then sent to the **mouseUp** built-in 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
```

See Also isShiftKeyDown
setControlKeyDown

setX

Method Specifies the horizontal coordinate of the mouse pointer position. Coordinates must be specified relative to the upper-left corner of the current object.

Type MouseEvent

Syntax **setX** (const ***xPosition*** LongInt)

Description **setX** sets the horizontal coordinate (in twips) of the mouse pointer position to xPosition.

Example This 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
```

See Also x

y
setY
UIObject::convertPointWithRespectTo

setY

Method	Specifies the vertical coordinate of the mouse pointer position. Coordinates must be specified relative to the upper-left corner of the current object.
Type	MouseEvent
Syntax	**setY** (const ***yPosition*** LongInt)
Description	**setY** sets the vertical coordinate (in twips) of the mouse pointer position to *yPosition*.
Example	See the example for setX.
See Also	x y setX UIObject**::**convertPointWithRespectTo

x

Method	Returns the horizontal coordinate of the mouse pointer position.
Type	MouseEvent
Syntax	**x** ( ) LongInt
Description	**x** returns (in twips) the horizontal coordinate of the mouse pointer position.
Example	See the example for setX.
See Also	y setX UIObject**::**convertPointWithRespectTo

y

Method	Returns the vertical coordinate of the mouse pointer position.
Type	MouseEvent
Syntax	**y** ( ) LongInt
Description	**y** returns (in twips) the vertical coordinate of the mouse pointer position.
Example	See the example for setX.
See Also	x setY setX UIObject**::**convertPointWithRespectTo

getDestination

Method Reports which object is the destination of a move.

Type MoveEvent

Syntax **getDestination** (var ***dest*** UIObject)

Description **getDestination** returns in *dest* the object that Paradox is trying to move to in a form.

Example In this 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
```

See Also reason

Event::getTarget

reason

Method Reports why a move occurred.

Type MoveEvent

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 the following constants for testing the value returned by **reason**:

PalMove means the move was generated by an ObjectPAL statement.

RefreshMove means the move was generated by table values being refreshed across a network.

ShutDownMove means the move was generated as the form closed.

StartupMove means the move was generated as the form opened.

UserMove means the move was caused by the user interacting with the form.

Example In this example, assume a form contains a field object named *fieldOne*, and a button named *moveToFieldOne*, as well as at least one other field object. A movement 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
```

See Also setReason

setReason

Method Specifies a Reason for a Move Event.

Type MoveEvent

Syntax **setReason** (const ***reasonId*** SmallInt)

Description **setReason** specifies a reason for generating a MoveEvent. This method takes one of the following MoveReason constants as an argument:

PalMove means the move was generated by an ObjectPAL statement.

RefreshMove means the move was generated by table values being refreshed across a network.

ShutDownMove means the move was generated as the form closed.

StartupMove means the move was generated as the form opened.

UserMove means the move was caused by the user interacting with the form.

Example In this 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
```

See Also reason

reason

Method Reports why a move occurred.

Type StatusEvent

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 the following constants for testing the value returned by **reason**:

StatusWindow means the message was sent to the Status area (the largest area on the status bar, it occupies the left two-thirds or so of the status bar).

ModeWindow1 means the message was sent to the first small window to the right of the Status area.

ModeWindow2 means the message was sent to the second small window to the right of the Status area.

ModeWindow3 means the message was sent to the third small window to the right of the Status area (the right-most window).

Example In the following example, assume that a form contains two fields, *fieldOne* and *fieldTwo*. The **status** method for *fieldTwo* examines the event packet for an error code; if it finds one, the method displays a message in the status line. The **status** method for the form sets an error code if the event's target is *fieldTwo*. The following code is attached to the status method for *fieldTwo*.

```
; fieldTwo::status
method status(var eventInfo StatusEvent)
if eventInfo.errorCode() <> 0 then
  ; display a different message if there is an error
  eventInfo.setStatusValue("There was an error here.")
endif
endmethod
```

This code is attached to the **status** method for the form:

```
; thisForm::status
method status(var eventInfo StatusEvent)
var
  targObj   UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    eventInfo.getTarget(targObj)
    ; set an arbitrary error for fieldTwo
    if targObj.Name = "fieldTwo" AND
       eventInfo.reason() = StatusWindow then
      eventInfo.setErrorCode(1)
    endif
  else
    ; code here executes just for form itself

endif
endmethod
```

See Also setReason

setReason

Method Specifies a reason for a move.

Type 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 the following constants for setting the reason for a StatusEvent:

StatusWindow means the message was sent to the Status area (the largest area on the status bar, it occupies the left two-thirds or so of the status bar).

ModeWindow1 means the message was sent to the first small window to the right of the Status area.

ModeWindow2 means the message was sent to the second small window to the right of the Status area.

ModeWindow3 means the message was sent to the third small window to the right of the Status area (the right-most window).

Example In this 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
```

See Also reason

Event::errorCode

Event::setErrorCode

setStatusValue

Method	Specifies the text of a status message.
Type	StatusEvent
Syntax	**setStatusValue** (const ***statusValue*** AnyType)
Description	**setStatusValue** specifies the text of a status message in *messageText*.
Example	See the example for setReason.
See Also	setReason statusValue

statusValue

Method Returns the text of a status message.

Type StatusEvent

Syntax **statusValue** ( ) AnyType

Description **statusValue** returns the text of a status message.

Example This 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
```

See Also reason

setStatusValue

newValue

Beginner

Method Returns the unposted new value of a ValueEvent.

Type 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()
```

Example In this 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
```

See Also setNewValue

setNewValue

Method Specifies a value to set for a ValueEvent.

Type 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.

Example In this 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 String
  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
```

See Also newValue

executeSQL

Method/ Procedure Executes a SQL statement.

Type SQL

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 Executes a passthrough 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. If you use 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.

A SQL query in ObjectPAL code begins with a 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 **executeSQLString**.)

Note: executeSQL is a passthrough 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.

Example This 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 a 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
```

See also executeSQLFile

executeSQLString

executeSQLFile

Method Executes a SQL statement contained in a file.

Type SQL

Syntax

1. executeSQLFile(const ***db*** Database, const ***fileName*** SQL) Logical

2. executeSQLFile(const ***db*** Database, const ***fileName*** SQL, ***ansTbl*** String) Logical

3. executeSQLFile(const ***db*** Database, const ***fileName*** SQL, ***ansTbl*** Table) Logical

4. executeSQLFile(const ***db*** Database, const ***fileName*** SQL, ***ansTbl*** TCursor) Logical

Description Executes the SQL statements stored in the file specified in *fileName*. In syntax 1, where the answer table is not specified, **executeSQLFile** 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. If you use syntax 4, a TCursor is opened onto the answer set, which may be an in-memory table or a cursor onto the answer set.

executeSQLFile returns True if the query is executed on the server (even if the resulting table is empty); otherwise, it returns False.

Note: executeSQLFile is a passthrough 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.

Example This example creates a pop-up menu listing the SQL file in the user's private directory. When the user chooses a file from the menu, this code calls **executeSQLFile** to execute the query and store the results in a TCursor. Then it passes the TCursor to a customer 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
   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

   ; execute the SQL file chosen by the user
      if executeSQLFile(db, sqlFileName, answerTC) then
         doSomething(answerTC) ; call custom proc to process
data
      else
         errorShow("executeSQLFile failed")
      endIf
   else
      msgStop("File not found:", sqlFileSpec)
   endIf

endMethod
```

See also executeSQL

executeSQLString

executeSQLString

Method Executes SQL statements contained in a string.

Type SQL

Syntax

1. executeSQLString(const ***db*** Database, const ***sqlString*** SQL) Logical

2. executeSQLString(const ***db*** Database, const ***sqlString*** SQL, ***ansTbl*** String) Logical

3. executeSQLString(const ***db*** Database, const ***sqlString*** SQL, ***ansTbl*** Table) Logical

4. executeSQLString(const ***db*** Database, const ***sqlString*** SQL, ***ansTbl*** TCursor) Logical

Description Executes SQL statements represented by a string and writes the results to an answer table. In syntax 1, where the answer table is not specified, **executeSQLString** 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. If you use syntax 4, a TCursor is opened onto the answer set, which may be an in-memory table or a cursor onto the answer set.

executeSQLString is useful when you're building a query from smaller strings. Because it's a SQL string (not a SQL statement), you cannot use tilde variables. However, you can use String variables to get the same effect. If you want to use tilde variables, use **executeSQL**.

Note: executeSQLString is a passthrough 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.

Example This example prompts the user to enter a SQL keyword, then uses that keyword as part of a 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
   endVar

   ; initialize variables
   myAlias = "itchy"
   aliasTableName = ":PRIV:aliasNam.db"
   promptString = "Enter an SQL keyword (e.g. SELECT):"

   enumAliasNames(aliasTableName)

   ; prompt user to enter a 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 a SQL string
   ;notice how \n characters are used
   ; to represent newline characters
   bigOrderString = "SQL + \n\n" +

                    sqlKeyword + "CustName, Order_no,
Sale_date, Qty
                    FROM     Customer
                    WHERE    Qty > 1000


                    endSQL"

   ; execute the query and process the results
   if executeSQLString(db, bigOrderString, bigOrderTC) then
      doSomething(bigOrderTC) ; call custom proc to process
data
   else
      errorShow("executeSQLString failed")
   endIf

endMethod
```

See also executeSQL

executeSQLFile

writeSQL

Method/ Procedure Writes a SQL statement or a SQL string to a file.

Type SQL

Syntax Method:

1. writeSQL(const ***fileName*** String) Logical

Procedure:

2. writeSQL(const ***sqlString*** String, const ***fileName*** String) Logical

Description 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 a 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")**.

Example This 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 a 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
      sqlVar          SQL
      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

      sqlVar = SQL

               SELECT *
               FROM ~userTableName

               endSQL
```

```
      sqlVar.writeSQL(sqlFileName) ; write user's query to a
file

   endIf

endMethod
```

See also executeSQL
executeSQLString
executeSQLFile

ObjectPAL Glossary

ObjectPAL terms

active

An ObjectPAL variable that represents the object that has focus (described below).

ANSI

An acronym for American National Standards Institute; a sequence of eight-bit codes that defines 256 standard characters, letters, numbers, and symbols. The ASCII character set (see below) is the same as the first 128 ANSI characters.

application

An ObjectPAL type that provides a handle to the Paradox. Also a group of forms, methods, queries, and procedures forming a single unit, where users can enter view, maintain, and report their data.

argument

Information passed to a method or procedure

array

A special kind of object that consists of several elements. Items in an array are designated by subscripts enclosed in square brackets, so that ar[1] and ar[2] are the first two items of an array named ar.

array element

One component of an array. For example, the array created with the following declaration has seven string elements, ar[1] through ar[7].

ar Array [7] String

Elements are also called items.

ASCII

An acronym for American Standard Code for Information Interchange; a sequence of seven-bit codes that define 128 standard characters, letters, numbers, and symbols. The IBM PC extends this code to 8 bits to include some special graphic characters. Windows products use the ANSI character set.

blank

A field or variable that has no value.

braces

The symbols { and } . Braces mark comments in code.

branching commands

Control structures that perform specific commands depending on whether certain conditions are met. Examples: if, while.

breakpoint

A flag set in source code that causes execution to suspend. Used in debugging.

bubbling

A process by which events pass from the target object up through the containership hierarchy.

built-in method

Pre-defined code that comes with every object you can place in a form. Built-in methods define an object's default response to events.

column

A vertical component of a Paradox table that contains one field. In a Paradox report, a vertical area containing one or more fields.

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.

concatenation

The joining of two or more strings to form a single string. The concatenation operator is a plus (+) sign.

constant

A constant represents a value that cannot be changed. For example, DataNextRecord is an ObjectPAL constant that specifies a move to the next record in a table.

containership

One object contains another object if the other object is completely within the borders of the first object. Containership affects the availability of variables, methods, and procedures.

control structure

A sequence of branching statements, such as if...then...endIF or while ...endWhile, that affects the order in which statements execute.

Ctrl+Break

A key sequence that halts program execution. You can configure Paradox to respond to Ctrl+Break by opening Debugger.

data

The information Paradox stores in a table.

Database

An object type that contains information about relationships between tables.

data type

The type of data that a field, variable, or array element contains.

DDE

Acronym for Dynamic Data Exchange. 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 test and trace execution of commands in your methods.

Desktop

The main window in Paradox.

Display manager

A category of object types that includes Application, Form, Report, and Table View.

DLL

Acronym for Dynamic Link Library. A DLL is a program that allows Windows programs to share code that performs common tasks.

dynamic array

ff A special kind of array where each item has a string for an index. For example, ["Product"], [" Paradox for Windows"], [" Type"] [" Relational database"], ["]Version"] [1.0]

Editor

The component of the Paradox 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.

event

The action that triggers a method (that is, causes code to execute). Also an object type (Event)

event-driven application

An application where code executes in response to events, as opposed to a procedural application, where code executes in a linear sequence.

event model

The rules that specify how events are processed by objects in a form.

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 that can include data values, variables, arrays, operators, or functions that represent 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 of information in a table. A collection of related fields makes up one record.

field assignment

Use of dot notation to assign the value of an expression to a field.

field object

A UIObject that may or may not be associated with a field in a table.

field type

The kind of information that can be entered into a field of a table. 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. Fields objects have a Value property.

field 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

An ObjectPAL type.FileSystem variables contain information about disk files.

focus

An attribute of an object. An object that has focus (also called the active object) is ready to handle keyboard input. Typically the active object is highlighted.

format specification

The way in which a field value is displayed onscreen or output to a printer.

form

A window for displaying data and objects. Also an ObjectPAL type (Form). The form is the highest-level container object.

function keys

The 12 keys across the top of the keyboard. (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 you can use in code to manipulate objects.

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.

incremental development

A process of application development in which small parts or the general structure of the application 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. 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 view or change an object's properties. To inspect an object, either right-click it or select it with the keyboard and press F6. The object's menu appears. Choose from the menu the property you want to change.

key

A field or group of fields in a used to order records. See also key field.

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.

key field

A field designated as all or part of an identifier of the records in a Paradox table. Establishing a key has three effects: the table is prevented from containing duplicate records, the records are maintained in sorted order based on the key fields, and a primary index is created for the table. See also index.

keyword

A word reserved by ObjectPAL. A keyword must not be used as the name of a variable, array, method, or procedure.

library

A collection of ObjectPAL code that can be used by objects in one or more forms.

lifetime

The length of time an item 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, and 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 value

A value (True or False) assigned to an expression when it is evaluated. Also known as Boolean value.

loops

Control structures that repeat a series of commands until a certain condition is met. See also control structures.

menu

A display of the choices or options available. Using ObjectPAL, you can create and edit both application menus and pop-up menus.

menu choice

A command chosen from a menu.

message

A string expression displayed in the status line.

method

ObjectPAL code attached to an object that defines the object's response to an event.

normalized data structure

An arrangement of data in tables where each record includes the fewest number of fields necessary to establish unique categories. Rather than using a few records to provide all possible information, a normalized table spreads out information over many records using fewer fields. A normalized table provides more flexibility in terms of analysis.

object

An encapsulation of code and data. All entities that can be manipulated in Paradox are objects.

Object Tree

A diagram that shows how objects in a form are related in terms of containership.

parameter

The variable into which an argument is passed. Used in defining procedures.

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 name comes from picture element.

point

An ordered pair of numbers that represents a location on screen.

pointer

A visual marker that indicates the mouse location on screen.

post

Accept changes to a record and put the data into the table. Also called commit.

primary index

An index on the key fields 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. See also key, secondary index.

procedure

Code bracketed by the keywords PROC and ENDPROC. Unlike a method, it has no object to give it context.

prompt

Instructions displayed on the screen, usually in the status bar. Prompts ask for information or guide the user through an operation.

properties

The attributes of an object. You right-click an object to view or change properties. See also inspect.

QBE

See Query by example

query

A question you ask about information in a Paradox table, formulated in a query form. Also an ObjectPAL type (Query).

query by example (QBE)

The method of asking questions about data by providing examples of the answers you're looking for.

quoted string

Text enclosed in double quotation marks.

raster operation

An operation that specifies how colors are blended on the screen.

record

A horizontal row in a Paradox table that contains a group of related fields of data. Also an

ObjectPAL type (Record).

record number

A unique number that identifies each record in a table.

relational database

A database design in accordance with a set of principles called the relational mode1. Data in a relational database must be organized into tables.

reserved words

The names of commands, keywords, functions, system variables, and operators. These words 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 table, called a record, in Paradox.

run-time error

An error that occurs when a syntactically valid statement cannot be carried out in the current context.

run-time library

A collection of pre-defined methods and procedures that operate on objects in specific types.

scope

The accessibility or availability of a variable, method, or procedure to other objects.

script

A collection of ObjectPAL code that executes without opening a window.

secondary index

An index used for linking, querying, and changing the view order of tables.

Self

An ObjectPAL variable. Self refers to the object to which the currently executing code is attached.

session

A channel to the database engine. Also an OjectPAL type (Session).

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.

string

An alphanumeric value, or an expression consisting of alphanumeric characters. Also an ObjectPAL type (String).

structure

The arrangement of fields in a table.

subject

The object used to call a custom method. For example, in the following statement, theBox is the subject.

theBOX.do Something()

substring

Any part of a string.

syntax error

An error that occurs due to an incorrectly expressed statement.

Tableview

The representation of a table in Paradox table format in rows and columns. Also an ObjectPAL type.

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. A pointer to the data in a table. Using TCursors, you can manipulate data without having to display the actual table.

tilde variable

A variable used in a query form, which must be preceded by a tilde(~).

transaction

A group of related changes to a database.

twip

A unit of measurement equal to 1/1440 of an inch (1/20 of a printer's point).

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 place in memory to store data temporarily.

active

An ObjectPAL variable that represents the object that has focus (described below).

ANSI

An acronym for American National Standards Institute; a sequence of eight-bit codes that defines 256 standard characters, letters, numbers, and symbols. The ASCII character set (see below) is the same as the first 128 ANSI characters.

application

An ObjectPAL type that provides a handle to the Paradox. Also a group of forms, methods, queries, and procedures forming a single unit, where users can enter view, maintain, and report their data.

argument

Information passed to a method or procedure

array

A special kind of object that consists of several elements. Items in an array are designated by subscripts enclosed in square brackets, so that ar[1] and ar[2] are the first two items of an array named ar.

array element

One component of an array. For example, the array created with the following declaration has seven string elements, ar[1] through ar[7].

ar Array [7] String

Elements are also called items.

ASCII

An acronym for American Standard Code for Information Interchange; a sequence of seven-bit codes that define 128 standard characters, letters, numbers, and symbols. The IBM PC extends this code to 8 bits to include some special graphic characters. Windows products use the ANSI character set.

blank

A field or variable that has no value.

braces

The symbols { and } . Braces mark comments in code.

branching commands

Control structures that perform specific commands depending on whether certain conditions are met. Examples: if, while.

breakpoint

A flag set in source code that causes execution to suspend. Used in debugging.

bubbling

A process by which events pass from the target object up through the containership hierarchy.

built-in method

Pre-defined code that comes with every object you can place in a form. Built-in methods define an object's default response to events.

column

A vertical component of a Paradox table that contains one field. In a Paradox report, a vertical area containing one or more fields.

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. For example, DataNextRecord is an ObjectPAL constant that specifies a move to the next record in a table.

containership

One object contains another object if the other object is completely within the borders of the first object. Containership affects the availability of variables, methods, and procedures.

control structure

A sequence of branching statements, such as if...then...endIF or while ...endWhile, that affects the order in which statements execute.

Ctrl+Break

A key sequence that halts program execution. You can configure Paradox to respond to Ctrl+Break by opening Debugger.

data

The information Paradox stores in a table.

Database

An object type that contains information about relationships between tables.

data type

The type of data that a field, variable,ar array element contains.

DDE

Acronym for Dynamic Data Exchange. 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 test and trace execution of commands in your methods.

Desktop

The main window in Paradox.

Display manager

A category of object types that includes Application, Form, Report, and Table View.

DLL

Acronym for Dynamic Link Library. A DLL is a program that allows Windows programs to share code that performs common tasks.

dynamic array

A special kind of array where each item has a string for an index. For example, ["Product"], [" Paradox for Windows"], [" Type"] [" Relational database"], ["]Version"] [1.0]

Editor

The component of the Paradox 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.

event

The action that triggers a method (that is, causes code to execute). Also an object type (Event)

event-driven application

An application where code executes in response to events, as opposed to a procedural application, where code executes in a linear sequence.

event model

The rules that specify how events are processed by objects in a form.

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 that can include data values, variables, arrays, operators, or functions that represent 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 of information in a table. A collection of related fields makes up one record.

field assignment

Use of dot notation to assign the value of an expression to a field.

field object

A UIObjects that may or may not be associated with a field in a table.

field type

The kind of information that can be entered into a field of a table. 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.

field 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

An ObjectPAL type.FileSystem variables contain information about disk files.

focus

An attribute of an object. An object that has focus (also called the active object) is ready to handle keyboard input. Typically the active object is highlighted.

format specification

The way in which a field value is displayed onscreen or output to a printer.

form

A window for displaying data and objects. Also an ObjectPAL type (Form). The form is the highest-level container object.

function keys

The 12 keys across the top of the keyboard. (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 you can use in code to manipulate objects.

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.

incremental development

A process of application development in which small parts or the general structure of the application 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. 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 view or change an object's properties. To inspect an object, either right-click it or select it with the keyboard and press F6. The object's menu appears. Choose from the menu the property you want to change.

key

A field or group of fields in a used to order records. See also key field.

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.

key field

A field designated as all or part of an identifier of the records in a Paradox table. Establishing a key has three effects: the table is prevented from containing duplicate records, the records are maintained in sorted order based on the key fields, and a primary index is created for the table. See also index.

keyword

A word reserved by ObjectPAL. A keyword must not be used as the name of a variable, array, method, or procedure.

library

A collection of ObjectPAL code that can be used by objects in one or more forms.

lifetime

The length of time an item 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, and 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 value

A value (True or False) assigned to an expression when it is evaluated. Also known as Boolean value.

loops

Control structures that repeat a series of commands until a certain condition is met. See also control structures.

menu

A display of the choices or options available. Using ObjectPAL, you can create and edit both application menus and pop-up menus.

menu choice

A command chosen from a menu.

message

A string expression displayed in the status line.

method

ObjectPAL code attached to an object that defines the object's response to an event.

normalized data structure

An arrangement of data in tables where each record includes the fewest number of fields necessary to establish unique categories. Rather than using a few records to provide all possible information, a normalized table spreads out information over many records using fewer fields. A normalized table provides more flexibility in terms of analysis.

object

An encapsulation of code attached to an object that defines that object's response to an event.

Object Tree

A diagram that shows how objects in a form are related in terms of containership.

parameter

The variable into which an argument is passed. Used in defining procedures.

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 name comes from picture element.

point

An ordered pair of numbers that represents a location on screen.

pointer

A visual marker that indicates the mouse location on screen.

post

Accept changes to a record and put the data into the table. Also called commit.

primary index

An index on the key fields 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. See also key, secondary index.

procedure

Code bracketed by the keywords PROC and ENDPROC. Unlike a method, it has no object to give it context.

prompt

Instructions displayed on the screen, usually in the status bar. Prompts ask for information or guide the user through an operation.

properties

The attributes of an object. You right-click an object to view or change properties. See also inspect.

QBE

See Query by example

query

A question you ask about information in a Paradox table, formulated in a query form. Also an ObjectPAL type (Query).

query by example (QBE)

The method of asking questions about data by providing examples of the answers you're looking for.

quoted string

Text enclosed in double quotation marks.

raster operation

An operation that specifies how colors are blended on the screen.

record

A horizontal row in a Paradox table that contains a group of related fields of data. Also an ObjectPAL type (Record).

record number

A unique number that identifies each record in a table.

relational database

A database design in accordance with a set of principles called the relational mode1. Data in a relational database must be organized into tables.

reserved words

The names of commands, keywords, functions, system variables, and operators. These words 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 table, called a record, in Paradox.

run-time error

An error that occurs when a syntactically valid statement cannot be carried out in the current context.

run-time library

A collection of pre-defined methods and procedures that operate on objects in specific types.

scope

The accessibility or availability of a variable, method, or procedure to other objects.

script

A collection of ObjectPAL code that executes without opening a window.

secondary index

An index used for linking, querying, and changing the view order of tables.

Self

An ObjectPAL variable. Self refers to the object to which the currently executing code is attached.

session

A channel to the database engine. Also an OjectPAL type (Session).

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.

string

An alphanumeric value, or an expression consisting of alphanumeric characters. Also an ObjectPAL type (String).

structure

The arrangement of fields in a table.

subject

The object used to call a custom method. For example, in the following statement, theBox is the subject.

theBOX.do Something()

substring

Any part of a string.

syntax error

An error that occurs due to an incorrectly expressed statement.

Tableview

The representation of a table in Paradox table format in rows and columns. Also an ObjectPAL type.

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. A pointer to the data in a table. Using TCursors, you can manipulate data without having to display the actual table.

tilde variable

A variable used in a query form, which must be preceded by a tilde(~).

transaction

A group of related changes to a database.

twip

A unit of measurement equal to 1/1440 of an inch (1/20 of a printer's point).

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 place in memory to store data temporarily.