# Using Object Pascal with the VCL

This section of the Help system discusses how to use Object Pascal and the object and component library in Delphi applications.

## Object Pascal and the VCL

Object Pascal, a set of object-oriented extensions to standard Pascal, is the language of Delphi. The Visual Component Library (VCL) is a hierarchy of classes—written in Object Pascal and tied to the Delphi IDE—that allows you to develop applications quickly. Using Delphi's Component palette and Object Inspector, you can place VCL components on forms and manipulate their properties without writing code.

All VCL objects descend from *TObject*, an abstract class whose methods encapsulate fundamental behavior like construction, destruction, and message handling. *TObject* is the immediate ancestor of many simple classes.

*Components* in the VCL descend from the abstract class *TComponent*. Components are objects that you can manipulate on forms at design time. Visual components—that is, components like *TForm* and *TSpeedButton* that appear on the screen at runtime—are called *controls*, and they descend from *TControl*.

Despite its name, the VCL consists mostly of nonvisual objects. The Delphi IDE allows you to add many nonvisual components to your programs by dropping them onto forms. For example, if you were writing an application that connects to a database, you might place a *TDataSource* component on a form. Although *TDataSource* is nonvisual, it is represented on the form by an icon (which doesn't appear at runtime). You can manipulate the properties and events of *TDataSource* in the Object Inspector just as you would those of a visual control.

When you write classes of your own in Object Pascal, they should descend from *TObject*. By deriving new classes from the VCL's base class (or one of its descendants), you provide your classes with essential functionality and ensure that they work with the VCL.

# Using the object model

Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you create an object (or, more formally, a class), you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

If you want to create new components and put them on the Delphi Component palette, see Overview of component creation.

The following topics discuss how to use objects in your applications:

- What is an object?
- Inheriting data and code from an object
- Scope and qualifiers
- Using object variables
- Creating, instantiating, and destroying objects

# What is an object?

An object, or *class*, is a data type that encapsulates *data* and *operations on data* in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements.

You can begin to understand objects if you understand Object Pascal *records*. Records (analogous to *structures* in C) are made of up fields that contain data, where each field has its own type. Records make it easy to refer to a collection of varied data elements.

Objects are also collections of data elements. But objects—unlike records—contain procedures and functions that operate on their data. These procedures and functions are called *methods*.

An object's data elements are accessed through *properties*. The properties of Delphi objects have values that you can change at design time without writing code. If you want a property value to change at runtime, you need to write only a small amount of code.

The combination of data and functionality in a single unit is called *encapsulation*. In addition to encapsulation, object-oriented programming is characterized by *inheritance* and *polymorphism*. Inheritance means that objects derive functionality from other objects (called *ancestors*); objects can modify their inherited behavior. Polymorphism means that different objects derived from the same ancestor support the same method and property interfaces, which often can be called interchangeably.

## Examining a Delphi object

When you create a new project, Delphi displays a new form for you to customize. In the Code editor, Delphi declares a new class type for the form and produces the code that creates the new form instance. The generated code looks like this:

```
unit Unit1;
interface
uses Windows, Classes, Graphics, Forms, Controls, Dialogs;
type
  TForm1 = class(TForm) { The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end; { The type declaration of the form ends here }
var
  Form1: TForm1;
implementation { Beginning of implementation part }
{$R *.DFM}
end.   { End of implementation part and unit}
```

The new class type is *TForm1*, and it is derived from type *TForm*, which is also a class.

A class is like a record in that they both contain data fields, but a class also contains methods—code that acts on the object's data. So far, *TForm1* appears to contain no fields or methods, because you haven't added to the form any components (the fields of the new object) and you haven't created any event handlers (the methods of the new object). *TForm1* does contain inherited fields and methods, even though you don't see them in the type declaration.

This variable declaration declares a variable named *Form1* of the new type *TForm1*.

```
var
  Form1: TForm1;
```

*Form1* represents an instance, or object, of the class type *TForm1*. You can declare more than one instance of a class type; you might want to do this, for example, to create multiple child windows in a Multiple Document Interface (MDI) application. Each instance maintains its own data, but all instances use the same code to execute methods.

Although you haven't added any components to the form or written any code, you already have a complete Delphi application that you can compile and run. All it does is display a blank form.

Suppose you add a button component to this form and write an *OnClick* event handler that changes the color of the form when the user clicks the button. The result might look like this:
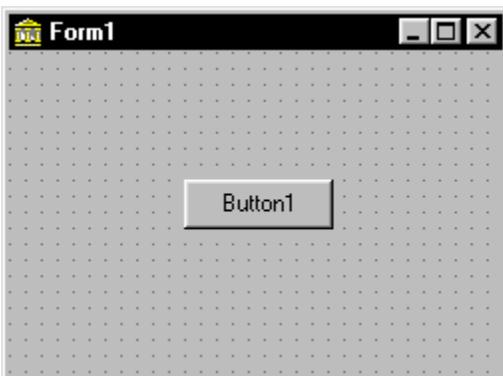


A simple form

When the user clicks the button, the form's color changes to green. This is the event-handler code for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
   Form1.Color := clGreen;
end;
```

Objects can contain other objects as data fields. Each time you place a component on a form, a new field appears in the form's type declaration. If you create the application described above and look at the code in the Code editor, this is what you see:

```
unit Unit1;
interface
uses Windows, Classes, Graphics, Forms, Controls;
type
  TForm1 = class(TForm)
    Button1: TButton; { New data field }
    procedure Button1Click(Sender: TObject);  { New method declaration }
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject); { The code of the new method }
begin
   Form1.Color := clGreen;
end;
end.
```

*TForm1* has a *Button1* field that corresponds to the button you added to the form. *TButton* is a class type, so *Button1* refers to an object.

All the event handlers you write in Delphi are methods of the form object. Each time you create an event handler, a method is declared in the form object type. The *TForm1* type now contains a new method, the *Button1Click* procedure, declared within the *TForm1* type declaration. The code that implements the *Button1Click* method appears in the **implementation** part of the unit.

## Changing the name of a component

You should always use the Object Inspector to change the name of a component. For example, suppose you want to change a form's name from the default *Form1* to a more descriptive name, such as *ColorBox*. When you change the form's *Name* property in the Object Inspector, the new name is automatically reflected in the form's .DFM file (which you usually don't edit manually) and in the Object Pascal source code that Delphi generates:

```
unit Unit1;
interface
uses Windows, Classes, Graphics, Forms, Controls;
type
  TColorBox = class(TForm){ Changed from TForm1 to TColorBox }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  ColorBox: TColorBox;  { Changed from Form1 to ColorBox }
implementation
{$R *.DFM}
procedure TColorBox.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen; { The reference to Form1 didn't change! }
end;
end.
```

Note that the code in the *OnClick* event handler for the button hasn't changed. Because you wrote the code, you have to update it yourself and correct any references to the form:

```
procedure TColorBox.Button1Click(Sender: TObject);
begin
  ColorBox.Color := clGreen;
end;
```

## Inheriting data and code from an object

The *TForm1* object described in <u>Examining a Delphi object</u> seems simple. *TForm1* appears to contain one field (*Button1*), one method (*Button1Click*), and no properties. Yet you can show, hide, or resize of the form, add or delete standard border icons, and set up the form to become part of a Multiple Document Interface (MDI) application. You can do these things because the form has *inherited* all the properties and methods of the VCL component *TForm*. When you add a new form to your project, you start with *TForm* and customize it by adding components, changing property values, and writing event handlers. To customize any object, you first derive a new object from the existing one; when you add a new form to your project, Delphi automatically derives a new form from the *TForm* type:

```
TForm1 = class(TForm)
```

A derived object inherits all the properties, events, and methods of the object it derives from. The derived object is called a *descendant* and the object it derives from is called an *ancestor*. If you look up *TForm* in the online Help, you'll see lists of its properties, events, and methods, including the ones that *TForm* inherits from *its* ancestors. An object can have only one immediate ancestor, but it can have many direct descendants.

## Objects, components, and controls
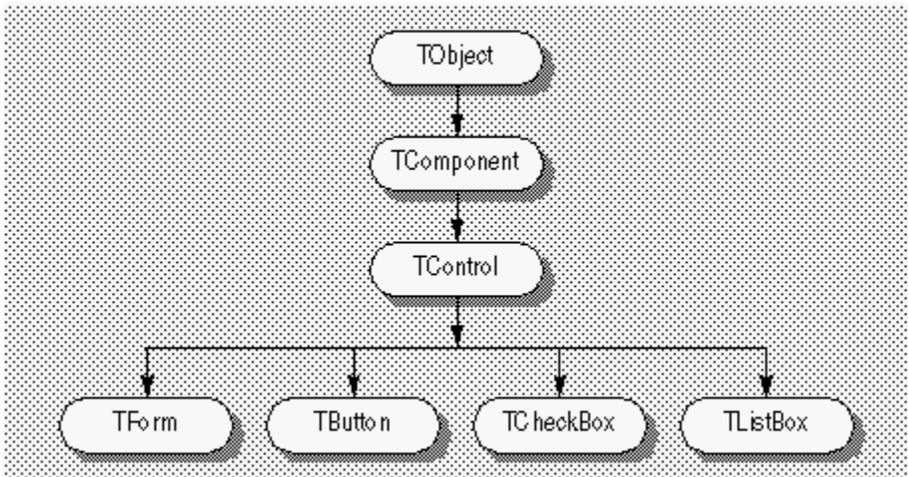
*Simplified VCL hierarchy*

The diagram above is a greatly simplified view of the inheritance hierarchy of the Visual Component Library. Every object inherits from *TObject*, and many objects inherit from *TComponent*. Controls, which inherit from *TControl*, have the ability to display themselves at runtime. A control like *TCheckBox* inherits all the functionality of *TObject*, *TComponent*, and *TControl*, and adds specialized capabilities of its own.

## Scope and qualifiers

*Scope* determines the accessibility of an object's fields, properties, and methods. All members declared within an object are available to that object and its descendants. Although a method's implementation code appears outside of the object declaration, the method is still within the scope of the object because it is declared within the object's declaration.

When you write code to implement a method that refers to properties, methods, or fields of the object where the method is declared, you don't need to preface those identifiers with the name of the object. For example, if you put a button on a new form, you could write this event handler for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Color := clFuchsia;
  Button1.Color := clLime;
end;
```

The first statement is equivalent to

```
Form1.Color := clFuchsia
```

You don't need to qualify *Color* with *Form1* because the *Button1Click* method is part of *TForm1*; identifiers in the method body therefore fall within the scope of the *TForm1* instance where the method is called. The second statement, in contrast, refers to the color of the button object (not of the form where the event handler is declared), so it requires qualification.

Delphi creates a separate unit (source code) file for each form. If you want to access one form's components from another form's unit file, you need to qualify the component names, like this:

```
Form2.Edit1.Color := clLime;
```

In the same way, you can access a component's methods from another form. For example,

```
Form2.Edit1.Clear;
```

To access *Form2*'s components from *Form1*'s unit file, you must also add *Form2*'s unit to the **uses** clause of *Form1*'s unit.

The scope of an object extends to the object's descendants. You can, however, redeclare a field, property, or method within a descendant object. Such redeclarations either hide or override the inherited member.

For more information about scope, see Blocks and scope. For more information about the uses clause, see Unit references and the uses clause. For more information about hiding and overriding inherited members, see Classes and objects.

## Private, protected, public, and published declarations

When you declare a field, property, or method, the new member has a *visibility* indicated by one of the keywords **private**, **protected**, **public**, or **published**. The visibility of a member determines its accessibility to other objects and units.

- A private member is accessible only within the unit where it is declared. Private members are often used within a class to implement other (public or published) methods and properties.
- A protected member is accessible within the unit where its class is declared and within any descendant class, regardless of the descendant class's unit.
- A public member is accessible from wherever the object it belongs to is accessible—that is, from the unit where the class is declared and from any unit that uses that unit.
- A published member has the same visibility as a public member, but the compiler generates runtime type information for published members. Published properties appear in the Object Inspector at design time.

For more information about visibility, see Visibility of class members.

## Using object variables

You can assign one object variable to another object variable if the variables are of the same type or assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable you are assigning to is an ancestor of the type of the variable being assigned. For example, here is a *TDataForm* type declaration and a variable declaration section declaring two variables, *AForm* and *DataForm*:

```
type
  TDataForm = class(TForm)
  Button1: TButton;
    Edit1: TEdit;
    DataGrid1: TDataGrid;
    Database1: TDatabase;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  AForm: TForm;
  DataForm: TDataForm;
```

*AForm* is of type *TForm*, and *DataForm* is of type *TDataForm*. Because *TDataForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm := DataForm;
```

Suppose you write an event handler for the *OnClick* event of a button. When the button is clicked, the event handler for the *OnClick* event is called. Each event handler has a *Sender* parameter of type *TObject*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
...
end;
```

Because *Sender* is of type *TObject*, any object can be assigned to *Sender*. The value of *Sender* is always the control or component that responds to the event. You can test *Sender* to find the type of component or control that called the event handler using the reserved word **is**. For example,

```
if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;
```

## Creating, instantiating, and destroying objects

Many of the objects you use in Delphi, such as buttons and edit boxes, are visible at both design time and runtime. Some, such as common dialog boxes, appear only at runtime. Still others, such as timers and datasource components, have no visual representation at runtime.

You may want to create your own objects. For example, you could create a *TEmployee* object that contains *Name*, *Title*, and *HourlyPayRate* properties. You could then add a *CalculatePay* method that uses the data in *HourlyPayRate* to compute a paycheck amount. The *TEmployee* type declaration might look like this:

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
    function CalculatePay: Double;
  end;
```

In addition to the fields, properties, and methods you've defined, *TEmployee* inherits all the methods of *TObject*. You can place a type declaration like this one in either the **interface** or **implementation** part of a unit, and then create instances of the new class by calling the *Create* method that *TEmployee* inherits from *TObject*:

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

The *Create* method is called a *constructor*. It allocates memory for a new instance object and returns a reference to the object.

Components on a form are created and destroyed automatically by Delphi. But if you write your own code to instantiate objects, you are responsible for disposing of them as well. Every object inherits a *Destroy* method (called a *destructor*) from *TObject*. To destroy an object, however, you should call the *Free* method (also inherited from *TObject*), because *Free* checks for a **nil** reference before calling *Destroy*. For example,

```
  Employee.Free
```

destroys the *Employee* object and deallocates its memory.

## Components and ownership

Delphi has a built-in memory-management mechanism that allows one component to assume responsibility for freeing another. The former component is said to *own* the latter. The memory for an owned component is automatically freed when its owner's memory is freed. The owner of a component —the value of its *Owner* property—is determined by a parameter passed to the constructor when the component is created. By default, a form owns all components on it and is in turn owned by the application. Thus, when the application shuts down, the memory for all forms and the components on them is freed.

Ownership applies only to *TComponent* and its descendants. If you create, for example, a *TStringList* or *TCollection* object (even if it is associated with a form), you are responsible for freeing the object.

**Note:** Don't confuse a component's *owner* with its *parent*.

## Using components

All components share features inherited from *TComponent*. By placing components on forms, you build the interface and functionality of your application. The standard components included with Delphi are sufficient for most application development, but you can extend the VCL by creating components of your own. For more information about creating custom components, see Overview of component creation..

## Delphi's standard components

The Component palette contains a selection of components that handle a wide variety of programming tasks. You can add, remove, and rearrange components on the palette, and you can create component *templates* and *frames* that group several components.

The components on the palette are arranged in pages according to their purpose and functionality. Which pages appear in the default configuration depends on the version of Delphi you are running. The following table lists typical default pages and the kinds of components they contain.

| Page name | Contents |
|---|---|
| Standard | Standard Windows controls, menus |
| Additional | Additional controls |
| Win32 | Windows 9x/NT 4.0 common controls |
| System | Components and controls for system-level access, including timers, multimedia, and DDE |
| Internet | Components for internet communication protocols and Web applications |
| Data Access | Nonvisual components for accessing databases tables, queries, and reports |
| Data Controls | Visual, data-aware controls |
| Decision Cube | Controls that let you summarize information from databases and view it from a variety of perspectives |
| QReport | Quick Report components for creating embedded reports |
| Dialogs | Windows common dialog boxes |
| Win 3.1 | Components for compatibility with Delphi 1.0 projects |
| Samples | Sample custom components |
| ActiveX | Sample ActiveX controls |
| Midas | Components used for creating multi-tiered database applications |

The online Help provides information about the components on the default palette. The components on the ActiveX and Samples pages, however, are provided as examples only and are not documented.

# Properties common to visual components

All visual components (descendants of *TControl*) share certain properties including

- Position and size properties
- Display properties
- Parent properties
- Navigation properties
- Drag-and-drop properties
- Drag-and-dock properties

While these properties are inherited from *TControl*, they are published—and hence appear in the Object Inspector—only for components to which they are applicable. For example, *TImage* does not publish the *Color* property, since its color is determined by the graphic it displays.

## Position and size properties

Four properties define the position and size of a control on a form:

- *Height* sets the vertical size
- *Width* sets the horizontal size
- *Top* positions the top edge
- *Left* positions the left edge

These properties aren't accessible in nonvisual components, but Delphi does keep track of where you place the component icons on your forms. Most of the time you'll set and alter these properties by manipulating the control's image on the form or using the Alignment palette. You can, however, alter them at runtime.

## Display properties

The following properties govern the general appearance of a control:

- *BorderStyle* specifies whether a control has a border.
- *Color* changes the background color of a control.
- *BevelKind* specifies the type of bevel if the control has beveled edges.
- *Font* changes the color, type family, style, or size of text.

## Parent properties

To maintain a consistent appearance across your application, you can make any control look like its container—called its *parent*—by setting the *parent-* properties to *True*. For example, if you place a button on a form and set the button's *ParentFont* property to *True*, changes to the form's *Font* property will automatically propagate to the button (and to the form's other children). Later, if you change the button's *Font* property, your font choice will take effect and the *ParentFont* property will revert to *False*.

**Note:** Although parents are also responsible for freeing their children's memory, you should not confuse a component's *parent* with its *owner*.

## Navigation properties

Several properties determine how users navigate among the controls in a form:

- *Caption* contains the text string that labels a component. To underline a character in a string, include an ampersand (&) before the character. This type of character is called an accelerator character. The user can then select the control or menu item by pressing *Alt* while typing the underlined character.
- *TabOrder* indicates the position of the control in its parent's tab order, the order in which controls receive focus when the user presses the *Tab* key. Initially, tab order is the order in which the components are added to the form, but you can change this by changing *TabOrder*. *TabOrder* is meaningful only if *TabStop* is *True*.
- *TabStop* determines whether the user can tab to a control. If *TabStop* is *True*, the control is in the tab order.

## Drag-and-drop properties

Two component properties affect drag-and-drop behavior:

- *DragMode* determines how dragging starts. By default, *DragMode* is *dmManual*, and the application must call the *BeginDrag* method to start dragging. When *DragMode* is *dmAutomatic*, dragging starts as soon as the mouse button goes down.
- *DragCursor* determines the shape of the mouse pointer when it is over a draggable component.

# Drag-and-dock properties

The following properties control drag-and-dock behavior.

- *DockSite*
- *DragKind*
- *DragMode*
- *FloatingDockSiteClass*
- *AutoSize*

For more information, see <u>Implementing drag-and-dock in controls</u>.

## Text controls

Many applications present text to the user or allow the user to enter text. The type of control used for this purpose depends on the size and format of the information.

| Use this component: | When you want users to do this: |
| --- | --- |
| Edit | Edit a single line of text |
| Memo | Edit multiple lines of text |
| MaskEdit | Adhere to a particular format, such as a postal code or phone number |
| RichEdit | Edit multiple lines of text using rich text format |

## Properties common to all text controls

All of the text controls have these properties in common:

- *Text* determines the text that appears in the edit box or memo control.
- *CharCase* forces the case of the text being entered to lowercase or uppercase.
- *ReadOnly* specifies whether the user is allowed to change the text.
- *MaxLength* limits the number of characters in the control.
- *PasswordChar* hides the text by displaying a single character (usually an asterisk).
- *HideSelection* specifies whether selected text remains highlighted when the control does not have focus.

## Properties shared by memo and rich text controls

Memo and rich text controls, which handle multiple lines of text, have several properties in common:

- *Alignment* specifies how text is aligned (left, right, or center) in the component.
- The *Text* property contains the text in the control. Your application can tell if the text changes by checking the *Modified* property.
- *Lines* contains the text as a list of strings.
- *OEMConvert* determines whether the text is temporarily converted from ANSI to OEM as it is entered. This is useful for validating file names.
- *WordWrap* determines whether the text will wrap at the right margin.
- *WantReturns* determines whether the user can insert hard returns in the text.
- *WantTabs* determines whether the user can insert tabs in the text.
- *AutoSelect* determines whether the text is automatically selected (highlighted) when the control becomes active.
- *SelText* contains the currently selected (highlighted) part of the text.
- *SelStart* and *SelLength* indicate the position and length of the selected part of the text.

At runtime, you can select all the text in the memo with the *SelectAll* method.

# Rich text controls

The rich edit component is a memo control that supports rich text formatting, printing, searching, and drag-and-drop of text. It allows you to specify font properties, alignment, tabs, indentation, and numbering.

## Specialized input controls

The following components provide additional ways of capturing input.

| Use this component: | When you want users to do this: |
| --- | --- |
| ScrollBar | Select values on a continuous range |
| TrackBar | Select values on a continuous range (more visually effective than scroll bar) |
| UpDown | Select a value from a spinner attached to an edit component |
| HotKey | Enter *Ctrl*/*Shift*/*Alt* keyboard sequences |

## Scroll bars

The scroll bar component is a Windows scroll bar that you can use to scroll the contents of a window, form, or other control. In the *OnScroll* event handler, you write code that determines how the control behaves when the user moves the scroll bar.

The scroll bar component is not used very often, since many visual components provide scroll bars of their own that don't require additional coding. For example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBox*.

## Track bars

A track bar can set integer values on a continuous range. It is useful for adjusting properties like volume and brightness. The user moves the slide indicator by dragging it to a particular location or clicking within the bar.

▪        Use the *Max* and *Min* properties to set the upper and lower range of the track bar.

▪        Use *SelEnd* and *SelStart* to highlight a selection range. See the figure "Three views of the track bar component" below.

▪        The *Orientation* property determines whether the track bar is vertical or horizontal.

▪        By default, a track bar has one row of ticks along the bottom. Use the *TickMarks* property to change their location. To control the intervals between ticks, use the *TickStyle* property and *SetTicks* method.
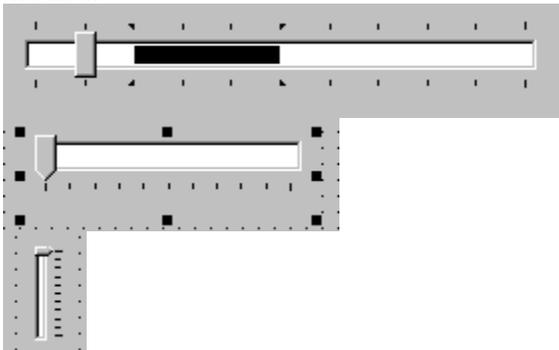


*Three views of the track bar component*

▪        *Position* sets a default position for the track bar and tracks the position at runtime.

▪        By default, users can move one tick up or down by pressing the up and down arrow keys. Set *LineSize* to change that increment.

▪        Set *PageSize* to determine the number of ticks moved when the user presses *Page Up* and *Page Down*.

## Up-down controls

An up-down control consists of a pair of arrow buttons that allow users to change an integer value in fixed increments. The current value is given by the *Position* property; the increment, which defaults to 1, is specified by the *Increment* property. Use the *Associate* property to attach another component (such as an edit control) to the up-down control.

## Hot key controls

Use the hot key component to assign a keyboard shortcut that transfers focus to any control. The *HotKey* property contains the current key combination and the *Modifiers* property determines which keys are available for *HotKey*.

# Splitter control

A splitter placed between aligned controls allows users to resize the controls. Used with components like panels and group boxes, splitters let you divide a form into several panes with multiple controls on each pane.

After placing a panel or other control on a form, add a splitter with the same alignment as the control. The last control should be client-aligned, so that it fills up the remaining space when the others are resized. For example, you can place a panel at the left edge of a form, set its *Alignment* to *alLeft*, then place a splitter (also aligned to *alLeft*) to the right of the panel, and finally place another panel (aligned to *alLeft* or *alClient*) to the right of the splitter.

Set *MinSize* to specify a minimum size the splitter must leave when resizing its neighboring control. Set *Beveled* to *True* to give the splitter's edge a 3D look.

## Buttons and similar controls

Aside from menus, buttons provide the most common way to invoke a command in an application. Delphi offers several button-like controls:

| Use this component: | To do this: |
| --- | --- |
| Button | Present command choices on buttons with text |
| BitBtn | Present command choices on buttons with text and glyphs |
| SpeedButton | Create grouped toolbar buttons |
| CheckBox | Present on/off options |
| RadioButton | Present a set of mutually exclusive choices |
| ToolBar | Arrange tool buttons and other controls in rows and automatically adjust their sizes and positions |
| CoolBar | Display a collection of windowed controls within movable, resizable bands |

## Button controls

Users click button controls to initiate actions. Double-clicking a button at design time takes you to the button's *OnClick* event handler in the Code editor.

- Set *Cancel* to *True* if you want the button to trigger its *OnClick* event when the user presses *Esc*.
- Set *Default* to *True* if you want the *Enter* key to trigger the button's *OnClick* event.

## Bitmap buttons

A bitmap button (*BitBtn*) is a button control that presents a bitmap image on its face.

- To choose a bitmap for your button, set the *Glyph* property.
- Use *Kind* to automatically configure a button with a glyph and default behavior.
- By default, the glyph is to the left of any text. To move it, use the *Layout* property.
- The glyph and text are automatically centered in the button. To move their position, use the *Margin* property. *Margin* determines the number of pixels between the edge of the image and the edge of the button.
- By default, the image and the text are separated by 4 pixels. Use *Spacing* to increase or decrease the distance.
- Bitmap buttons can have 3 states: up, down, and held down. Set the *NumGlyphs* property to 3 to show a different bitmap for each state.

## Speed buttons

Speed buttons, which usually have images on their faces, can function in groups. They are commonly used with panels to create toolbars.

▪        To make speed buttons act as a group, give the *GroupIndex* property of all the buttons the same nonzero value.

▪        By default, speed buttons appear in an up (unselected) state. To initially display a speed button as selected, set the *Down* property to *True*.

▪        If *AllowAllUp* is *True*, all of the speed buttons in a group can be unselected. Set *AllowAllUp* to *False* if you want a group of buttons to act like a radio group.

## Check boxes

A check box is a toggle that presents the user with two, or sometimes three, choices.

▪        Set *Checked* to *True* to make the box appear checked by default.

▪        Set *AllowGrayed* to *True* to give the check box three possible states: checked, unchecked, and grayed.

▪        The *State* property indicates whether the check box is checked (*cbChecked*), unchecked (*cbUnchecked*), or grayed (*cbGrayed*).

## Radio buttons

Radio buttons present a set of mutually exclusive choices. You can use individual radio buttons or the *radio group* component, which arranges groups of radio buttons automatically. See Grouping components for more information.

## Toolbars

Toolbars provide an easy way to arrange and manage visual controls. You can create a toolbar out of a panel component and speed buttons, or you can use the *ToolBar* component, then right-click and choose New Button to add buttons to the toolbar. The *ToolBar* component has several advantages: Buttons on a toolbar automatically maintain uniform dimensions and spacing; other controls maintain their relative position and height; controls can automatically wrap around to start a new row when they do not fit horizontally; and the *ToolBar* offers display options like transparency, pop-up borders, and spaces and dividers to group controls.

## Cool bars

A cool bar (or rebar) contains child controls that can be moved and resized independently. Each control resides on an individual band. The user positions the controls by dragging the sizing grip to the left of each band.

The cool bar requires version 4.70 or later of COMCTL32.DLL (usually located in the WINDOWS\SYSTEM or WINDOWS\SYSTEM32 directory) at both design time and runtime.

- The *Bands* property holds a collection of *TCoolBand* objects. At design time, you can add, remove, or modify bands with the Bands editor. To open the Bands editor, select the *Bands* property in the Object Inspector, then double-click in the Value column to the right, or click the ellipsis (...) button. You can also create bands simply by adding new windowed controls from the palette.
- The *FixedOrder* property determines users can reorder the bands.
- The *FixedSize* property determines whether the bands maintain a uniform height.

## Handling lists

Lists present the user with a collection of items to select from. Several components display lists:

| Use this component: | To display: |
| --- | --- |
| ListBox | A list of text strings |
| CheckListBox | A list with a check box in front of each item |
| ComboBox | An edit box with a scrollable drop-down list |
| TreeView | A hierarchical list |
| ListView | A list of (draggable) items with optional icons, columns, and headings |
| DateTimePicker | A list box for entering dates or times |
| MonthCalendar | A calendar for selecting dates |

Use the nonvisual *TStringList* and *TImageList* components to manage sets of strings and images. For more information about string lists, see Working with string lists.

## List boxes and check-list boxes

List boxes and check-list boxes display lists from which users can select items.

- *Items* uses a *TStrings* object to fill the control with values.
- *ItemIndex* indicates which item in the list is selected.
- *MultiSelect* specifies whether a user can select more than one item at a time.
- *Sorted* determines whether the list is arranged alphabetically.
- *Columns* specifies the number of columns in the list control.
- *IntegralHeight* specifies whether the list box shows only entries that fit completely in the vertical space.
- *ItemHeight* specifies the height of each item in pixels. The *Style* property can cause *ItemHeight* to be ignored.
- The *Style* property determines how a list control displays its items. By default, items are displayed as strings. By changing the value of *Style*, you can create *owner-draw* list boxes that display items graphically or in varying heights. For information on owner-draw controls, see Adding graphics to controls.

## Combo boxes

A combo box combines an edit box with a scrollable list. When users enter data into the control—by typing or selecting from the list—the value of the *Text* property changes.

Use the *Style* property to select the type of combo box you need:

▪ Use *csDropdown* if you want an edit box with a drop-down list. Use *csDropDownList* to make the edit box read-only (forcing users to choose from the list). Set the *DropDownCount* property to change the number of items displayed in the list.

▪ Use *csSimple* to create a combo box with a fixed list that does not close. Be sure to resize the combo box so that the list items are displayed.

▪ Use *csOwnerDrawFixed* or *csOwnerDrawVariable* to create *owner-draw* combo boxes that display items graphically or in varying heights. For information on owner-draw controls, see Adding graphics to controls.

## Tree views

A tree view displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. You can include icons with items' text labels and display different icons to indicate whether a node is expanded or collapsed. You can also include graphics, such as a check boxes, that reflect state information about the items.

- *Indent* sets the number of pixels horizontally separating items from their parents.
- *ShowButtons* enables the display of '+' and '–' buttons to indicate whether an item can be expanded.
- *ShowLines* enables display of connecting lines to show hierarchical relationships.
- *ShowRoot* determines whether lines connecting the top-level items are displayed.

## List views

List views display lists in various formats. Use the *ViewStyle* property to choose the kind of list you want:

- *vsIcon* and *vsSmallIcon* display each item as an icon with a label. Users can drag items within the list view window.
- *vsList* displays items as labeled icons that cannot be dragged.
- *vsReport* displays items on separate lines with information arranged in columns. The leftmost column contains a small icon and label, and subsequent columns contain subitems specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.

# Date-time pickers and month calendars

The DateTimePicker component displays a list box for entering dates or times, while the MonthCalendar component presents a calendar for entering dates or ranges of dates. To use these components, you must have version 4.70 or later of COMCTL32.DLL (usually located in the WINDOWS\SYSTEM or WINDOWS\SYSTEM32 directory) at both design time and runtime.

## Grouping components

A graphical interface is easier to use when related controls and information are presented in groups. Delphi provides several components for grouping components:

| Use this component: | When you want this: |
| --- | --- |
| GroupBox | A standard group box with a title |
| RadioGroup | A simple group of radio buttons |
| Panel | A more visually flexible group of controls |
| ScrollBox | A scrollable region containing controls |
| TabControl | A set of mutually exclusive notebook-style tabs |
| PageControl | A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls |
| HeaderControl | Resizable column headers |

## Group boxes and radio groups

A group box is a standard Windows component that arranges related controls on a form. The most commonly grouped controls are radio buttons. After placing a group box on a form, select components from the Component palette and place them in the group box. The *Caption* property contains text that labels the group box at runtime.

The radio group component simplifies the task of assembling radio buttons and making them work together. To add radio buttons to a radio group, edit the *Items* property in the Object Inspector; each string in *Items* makes a radio button appear in the group box with the string as its caption. The value of the *ItemIndex* property determines which radio button is currently selected. Display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property. To respace the buttons, resize the radio group component.

## Panels

The panel component provides a generic container for other controls. Panels can be aligned with the form to maintain the same relative position when the form is resized. The *BorderWidth* property determines the width, in pixels, of the border around a panel.

## Header controls

Scroll boxes create scrolling areas within a form. Applications often need to display more information than will fit in a particular area. Some controls—such as list boxes, memos, and forms themselves—can automatically scroll their contents. Scroll boxes give you the additional flexibility to define arbitrary scrolling subregions of a form.

Like panels and group boxes, scroll boxes contain other controls. But a scroll box is normally invisible. If the controls in the scroll box cannot fit in its visible area, the scroll box automatically displays scroll bars.

## Header controls

The tab control component looks like notebook dividers. You can create tabs by editing the *Tabs* property in the Object Inspector; each string in *Tabs* represents a tab. The tab control is a single panel with one set of components on it. To change the appearance of the control when the tabs are clicked, you need to write an *OnChange* event handler. To create a multipage dialog box, use a page control instead.

## Page controls

The page control component is a page set suitable for multipage dialog boxes. To create a new page in a page control, right-click the control and choose New Page.

## Header controls

A header control is a is a set of column headers that the user can select or resize at runtime. Edit the control's *Sections* property to add or modify headers.

## Visual feedback

There are many ways to provide users with information about the state of an application. For example, some components—including *TForm*—have a *Caption* property that can be set at runtime. You can also create dialog boxes to display messages. In addition, the following components are especially useful for providing visual feedback at runtime.

| Use this component or property: | To do this: |
|---|---|
| Label and StaticText | Display non-editable text |
| StatusBar | Display a status region (usually at the bottom of a window) |
| ProgressBar | Show the amount of work completed for a particular task |
| *Hint* and *ShowHint* | Activate fly-by or "tool-tip" help |
| *HelpContext* and *HelpFile* | Link context-sensitive online Help |

## Labels and static-text components

Labels display text and are usually placed next to other controls. The standard label component, *TLabel*, is a nonwindowed control, so it cannot receive focus; when you need a label with a window handle, use *TStaticText* instead. Label properties include the following:

- *Caption* contains the text string for the label.
- *FocusControl* links the label to another control on the form. If *Caption* includes an accelerator key, the control specified by *FocusControl* receives focus when the user presses the accelerator key.
- *ShowAccelChar* determines whether the label can display an underlined accelerator character. If *ShowAccelChar* is *True*, any character preceded by an ampersand (&) appears underlined and enables an accelerator key.
- *Transparent* determines whether items under the label (such as graphics) are visible.

## Status bars

Although you can use a panel to make a status bar, it is simpler to use the status-bar component. By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size.

You will usually divide a status bar into several text areas. To create text areas, edit the *Panels* property in the Object Inspector, setting each panel's *Width*, *Alignment*, and *Text* properties from the Panels editor. The *Text* property contains the text displayed in the panel.

## Progress bars

When your application performs a time-consuming operation, you can use a progress bar to show how much of the task is completed. A progress bar displays a dotted line that grows from left to right.



*A progress bar*

The *Position* property tracks the length of the dotted line. *Max* and *Min* determine the range of *Position*. To make the line grow, increment *Position* by calling the *StepBy* or *StepIt* method. The *Step* property determines the increment used by *StepIt*.

# Help and hint properties

Most visual controls can display context-sensitive Help as well as fly-by hints at runtime. The *HelpContext* and *HelpFile* properties establish a Help context number and Help file for the control.

The *Hint* property contains the text string that appears when the user moves the mouse pointer over a control or menu item. To enable hints, set *ShowHint* to *True*; setting *ParentShowHint* to *True* causes the control's *ShowHint* property to have the same value as its parent's.

## Grids

Grids display information in rows and columns. If you're writing a database application, use the *TDBGrid* or *TDBCtrlGrid* component described in "Using data controls". Otherwise, use a standard draw grid or string grid.

## Draw grids

A draw grid (*TDrawGrid*) displays arbitrary data in tabular format. Write an *OnDrawCell* event handler to fill in the cells of the grid.

▪        The *CellRect* method returns the screen coordinates of a specified cell, while the *MouseToCell* method returns the column and row of the cell at specified screen coordinates. The *Selection* property indicates the boundaries of the currently selected cells.

▪        The *TopRow* property determines which row is currently at the top of the grid. The *LeftCol* property determines the first visible column on the left. *VisibleColCount* and *VisibleRowCount* are the number of columns and rows visible in the grid.

▪        You can change the width or height of a column or row with the *ColWidths* and *RowHeights* properties. Set the width of the grid lines with the *GridLineWidth* property. Add scroll bars to the grid with the *ScrollBars* property.

▪        You can choose to have fixed or nonscrolling columns and rows with the *FixedCols* and *FixedRows* properties. Assign a color to the fixed columns and rows with the *FixedColor* property.

▪        The *Options*, *DefaultColWidth*, and *DefaultRowHeight* properties also affect the appearance and behavior of the grid.

## String grids

The string grid component is a descendant of *TDrawGrid* that adds specialized functionality to simplify the display of strings. The *Cells* property lists the strings for each cell in the grid; the *Objects* property lists objects associated with each string. All the strings and associated objects for a particular column or row can be accessed through the *Cols* or *Rows* property.

## Graphic display

The following components make it easy to incorporate graphics into an application.

| Use this component | To display: |
| --- | --- |
| Image | Graphics files |
| Shape | Geometric shapes |
| Bevel | 3D lines and frames |
| PaintBox | Graphics drawn by your program at runtime |
| Animate | AVI files |

## Images

The image component displays a graphical image, like a bitmap, icon, or metafile. The *Picture* property determines the graphic to be displayed. Use *Center*, *AutoSize*, *Stretch*, and *Transparent* to set display options.

## Shapes

The shape component displays a geometric shape. It is a nonwindowed control and cannot receive user input. The *Shape* property determines which shape the control assumes. To change the shape's color or add a pattern, use the *Brush* property, which holds a *TBrush* object. How the shape is painted depends on the *Color* and *Style* properties of *TBrush*.

## Bevels

The bevel component is a line that can appear raised or lowered. Some components, such as *TPanel*, have built-in properties to create beveled borders. When such properties are unavailable, use *TBevel* to create beveled outlines, boxes, or frames.

## Paint boxes

The paint box allows your application to draw on a form. Write an *OnPaint* event handler to render an image directly on the paint box's *Canvas*. Drawing outside the boundaries of the paint box is prevented. For more information, see <u>Overview of graphics programming</u>.

## Animation control

The animation component is a window that silently displays an Audio Video Interleaved (AVI) clip. An AVI clip is a series of bitmap frames, like a movie. Although AVI clips can have sound, animation controls work only with silent AVI clips. The files you use must be either uncompressed AVI or compressed using run-length encoding (RLE).

# Windows common dialog boxes

The components on the Dialogs page of the Component palette make the Windows common dialog boxes available in Delphi applications. These dialog boxes provide a consistent user interface for standard operations like finding and opening files, setting fonts and colors, and printing. The dialogs do not appear at runtime until activated by a call to their *Execute* method.

## Setting component properties

Published properties can be set at design time in the Object Inspector and, in some cases, with special property editors.

To set properties at runtime, assign them new values in your application source code.

For information about the properties of each component, see the VCL Help.

## Using the Object Inspector

When you select a component on a form, the Object Inspector displays its published properties and (when appropriate) allows you to edit them. Use the *Tab* key to toggle between the Value column and the Property column. When the cursor is in the Property column, you can navigate to any property by typing the first letters of its name. For properties of Boolean or enumerated types, you can choose values from a drop-down list or toggle their settings by double-clicking in Value column. If a plus (+) symbol appears next to a property name, clicking the plus symbol displays a list of subvalues for the property. Similarly, if a minus (-) symbol appears next to the property name, clicking the minus symbol hides the subvalues.

By default, properties in the Legacy category are not shown; to change the display filters, right-click in the Object Inspector and choose View. For more information, see Property categories in the Object Inspector.

When more than one component is selected, the Object Inspector displays all properties—except *Name* —that are shared by the selected components. If the value for a shared property differs among the selected components, the Object Inspector displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

## Using property editors

Some properties, such as *Font*, have special property editors. Such properties appear with ellipsis marks (...) next to their values when the property is selected in the Object Inspector. To open the property editor, double-click in the Value column, click the ellipsis mark, or type Ctrl+Enter when focus is on the property or its value. With some components, double-clicking the component on the form also opens a property editor.

Property editors let you set complex properties from a single dialog box. They provide input validation and often let you preview the results of an assignment.

## Setting properties at runtime

Any writable property can be set at runtime in your source code. For example, you can dynamically assign a caption to a form:

```
Form1.Caption := MyString;
```

## Calling methods

Methods are called just like ordinary procedures and functions. For example, visual controls have a
*Repaint* method that refreshes the control's image on the screen. You could call the *Repaint* method in a
draw-grid object like this:

```
DrawGrid1.Repaint;
```

As with properties, the scope of a method name determines the need for qualifiers. If you want, for
example, to repaint a form within an event handler of one of the form's child controls, you don't have to
prepend the name of the form to the method call:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Repaint;
end;
```

For more information about scope, see Scope and qualifiers.

## Working with events and event handlers

In Delphi, almost all the code you write is executed, directly or indirectly, in response to *events*. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event—called an *event handler*—is an Object Pascal procedure. The sections that follow show how to

- Generate a new event handler
- Generate a handler for a component's default event
- Locate event handlers
- Associate an event with an existing event handler
- Associate menu events with event handlers
- Delete event handlers

## Generating a new event handler

Delphi can generate skeleton event handlers for forms and other components. To create an event handler,

1   Select a component.
2   Click the Events tab in the Object Inspector. The Events page of the Object Inspector displays all events defined for the component.
3   Select the event you want, then double-click the Value column or press *Ctrl+Enter*. Delphi generates the event handler in the Code editor and places the cursor inside the **begin...end** block.
4   Inside the **begin...end** block, type the code that you want to execute when the event occurs.

## Generating a handler for a component's default event

Some components have a *default* event, which is the event the component most commonly needs to handle. For example, a button's default event is *OnClick*. To create a default event handler, double-click the component in the Form Designer; this generates a skeleton event-handling procedure and opens the Code editor with the cursor in the body of the procedure, where you can easily add code.

Not all components have a default event. Some components, such as the Bevel, don't respond to any events. Other components respond differently when you double-click on them in the Form Designer. For example, many components open a default property editor or other dialog when they are double-clicked at design time.

## Locating event handlers

If you generated a default event handler for a component by double-clicking it in the Form Designer, you can locate that event handler in the same way. Double-click the component, and the Code editor opens with the cursor at the beginning of the event-handler body.

To locate an event handler that's not the default,

1  In the form, select the component whose event handler you want to locate.
2  In the Object Inspector, click the Events tab.
3  Select the event whose handler you want to view and double-click in the Value column. The Code editor opens with the cursor at the beginning of the event-handler body.

## Associating an event with an existing event handler

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop-down menu commands. When a button initiates the same action as a menu command, you can write a single event handler and assign it to both the button's and the menu item's *OnClick* event.

To associate an event with an existing event handler,

 1   On the form, select the component whose event you want to handle.
 2   On the Events page of the Object Inspector, select the event to which you want to attach a handler.
 3   Click the down arrow in the Value column next to the event to open a list of previously written event handlers. (The list includes only event handlers written for events of the same name on the same form.) Select from the list by clicking an event-handler name.

The procedure above is an easy way to reuse event handlers. *Action lists*, however, provide a more powerful tool for centrally organizing the code that responds to user commands.

## Using the Sender parameter

In an event handler, the *Sender* parameter indicates which component received the event and therefore called the handler. Sometimes it is useful to have several components share an event handler that behaves differently depending on which component calls it. You can do this by using the *Sender* parameter in an **if...then...else** statement. For example, the following code displays the title of the application in the caption of a dialog box only if the *OnClick* event was received by *Button1*.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
if Sender = Button1 then
  AboutBox.Caption := 'About ' + Application.Title
else
  AboutBox.Caption := '';
AboutBox.ShowModal;
end;
```

## Displaying and coding shared events

When components share events, you can display their shared events in the Object Inspector. First, select the components by holding down the *Shift* key and clicking on them in the Form Designer; then choose the Events tab in the Object Inspector. From the Value column in the Object Inspector, you can now create a new event handler for, or assign an existing event handler to, any of the shared events.

## Associating menu events with event handlers

Delphi's Menu Designer, along with the *MainMenu* and *PopupMenu* components, make it easy to supply your application with drop-down and pop-up menus. For the menus to work, however, each menu item must respond to the *OnClick* event, which occurs whenever the user chooses the menu item or presses its accelerator or shortcut key. This section explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see <u>Creating and managing menus</u>.

To create an event handler for a menu item,

1   Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
2   Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item's *Name* property.
3   From the Menu Designer, double-click the menu item. Delphi generates an event handler in the Code editor and places the cursor inside the **begin...end** block.
4   Inside the **begin...end** block, type the code that you want to execute when the user selects the menu command.

To associate a menu item with an existing *OnClick* event handler,

1   Open the Menu Designer by double-clicking on a *MainMenu* or *PopupMenu* object.
2   Select a menu item in the Menu Designer. In the Object Inspector, make sure that a value is assigned to the item's *Name* property.
3   On the Events page of the Object Inspector, click the down arrow in the Value column next to *OnClick* to open a list of previously written event handlers. (The list includes only event handlers written for *OnClick* events on this form.) Select from the list by clicking an event handler name.

## Deleting event handlers

When you delete a component using the Form Designer, Delphi removes the component from the form's type declaration. It does not, however, delete any associated methods from the unit file, since these methods may still be called by other components on the form. You can manually delete a method—such as an event handler—but if you do so, be sure to delete both the method's forward declaration (in the **interface** section of the unit) and its implementation (in the **implementation** section); otherwise you'll get a compiler error when you build your project.

## Using helper objects

The VCL includes a variety of nonvisual objects that simplify common programming tasks. These topics describe a few Helper objects that facilitate

- Creating and managing lists
- Creating and managing string lists
- Editing the Windows registry and .INI files
- Streaming data to a hard disk or other storage device

## Working with lists

Several VCL objects provide functionality for creating and managing lists:

- *TList* maintains a list of pointers.
- *TObjectList* maintains a memory-managed list of instance objects.
- *TComponentList* maintains a memory-managed list of components (that is, instances of classes descended from *TComponent*).
- *TQueue* maintains a first-in first-out list of pointers.
- *TStack* maintains a last-in first-out list of pointers.
- *TObjectQueue* maintains a first-in first-out list of objects.
- *TObjectStack* maintains a last-in first-out list of objects.
- *TClassList* maintains a list of class types.
- *TCollection*, *TOwnedCollection*, and *TCollectionItem* maintain indexed collections of specially defined items.
- *TStringList* maintains a list of strings.

For more information about these objects, see the VCL Reference in the online Help.

# Working with string lists

Applications often need to manage lists of character strings. Examples include items in a combo box, lines in a memo, names of fonts, and names of rows and columns in a string grid. The VCL provides a common interface to any list of strings through an object called *TStrings* and its descendant *TStringList*. In addition to providing functionality for maintaining string lists, these objects allow easy interoperability; for example, you can edit the lines of a memo (which are an instance of *TStrings*) and then use these lines as items in a combo box (also an instance of *TStrings*).

A string-list property appears in the Object Inspector with *TStrings* in the Value column. Double-click *TStrings* to open the String List editor, where you can edit, add, or delete lines.

You can also work with string-list objects at runtime to perform such tasks as

- Loading and saving string lists
- Creating a new string list
- Manipulating strings in a list
- Associating objects with a string list

## Loading and saving string lists

String-list objects provide *SaveToFile* and *LoadFromFile* methods that let you store a string list in a text file and load a text file into a string list. Each line in the text file corresponds to a string in the list. Using these methods, you could, for example, create a simple text editor by loading a file into a memo component, or save lists of items for combo boxes.

The following example loads a copy of the WIN.INI file into a memo field and makes a backup copy called WIN.BAK.

```
procedure EditWinIni;
var
  FileName: string; { storage for file name }
begin
  FileName := 'C:\WINDOWS\WIN.INI';{ set the file name }
  with Form1.Memo1.Lines do
  begin
    LoadFromFile(FileName);  { load from file }
    SaveToFile(ChangeFileExt(FileName, '.BAK')); { save into backup file }
  end;
end;
```

## Creating a new string list

A string list is typically part of a component. There are times, however, when it is convenient to create independent string lists, for example to store strings for a lookup table. The way you create and manage a string list depends on whether the list is short-term (constructed, used, and destroyed in a single routine) or long-term (available until the application shuts down). Whichever type of string list you create, remember that you are responsible for freeing the list when you finish with it.

### Short-term string lists

If you use a string list only for the duration of a single routine, you can create it, use it, and destroy it all in one place. This is the safest way to work with string lists. Because the string-list object allocates memory for itself and its strings, you should use a **try...finally** block to ensure that the memory is freed even if an exception occurs.

1    Construct the string-list object.
2    In the **try** part of a **try...finally** block, use the string list.
3    In the **finally** part, free the string-list object.

The following event handler responds to a button click by constructing a string list, using it, and then destroying it.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TempList: TStrings; { declare the list }
begin
  TempList := TStringList.Create;  { construct the list object }
  try
    { use the string list }
  finally
    TempList.Free;  { destroy the list object }
  end;
end;
```

### Long-term string lists

If a string list must be available at any time while your application runs, construct the list at start-up and destroy it before the application terminates.

1    In the unit file for your application's main form, add a field of type *TStrings* to the form's declaration.
2    Write an event handler for the main form's *OnCreate* event. (*OnCreate* is the default event for a form, so just double-click on the form to generate a skeleton event handler.) The *OnCreate* event handler, which executes before the form appears, should create a string list and assign it to the field you declared in the first step.
3    Write an event handler that frees the string list for the form's *OnDestroy* event.

This example uses a long-term string list to record the user's mouse clicks on the main form, then saves the list to a file before the application terminates.

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList: TStrings;  { declare the field }
  end;
```

```pascal
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  ClickList := TStringList.Create; { construct the list }
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.LOG')); { save the list
}
  ClickList.Free;{ destroy the list object }
end;
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  ClickList.Add(Format('Click at (%d, %d)', [X, Y]));  { add a string to the list }
end;
end.
```

## Manipulating strings in a list

Operations commonly performed on string lists include

- Counting the strings in a list
- Accessing a particular string
- Finding the position of a string in the list
- Iterating through strings in a list
- Adding a string to a list
- Moving a string within a list
- Deleting a string from a list
- Copying a complete string list

## Counting the strings in a list

The read-only *Count* property returns the number of strings in the list. Since string lists use zero-based indexes, *Count* is one more than the index of the last string.

## Accessing a particular string

The array property *Strings* contains the strings in the list, referenced by a zero-based index. Since *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus

```
StringList1.Strings[0] := 'This is the first string.';
```

is equivalent to

```
StringList1[0] := 'This is the first string.';
```

## Finding the position of a string in the list

To locate a string in a string list, use the *IndexOf* method. *IndexOf* returns the index of the first string in the list that matches the parameter passed to it, and returns –1 if the parameter string is not found. *IndexOf* finds exact matches only; if you want to match partial strings, you must iterate through the string list yourself.

For example, you could use *IndexOf* to determine whether a given file name is found among the *Items* of a list box:

```
if FileListBox1.Items.IndexOf('WIN.INI') > -1 ...
```

## Iterating through strings in a list

To iterate through the strings in a list, use a **for** loop that runs from zero to *Count* – 1.

This example converts each string in a list box to uppercase characters.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Index: Integer;
begin
  for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
end;
```

## Adding a string to a list

To add a string to the end of a string list, call the *Add* method, passing the new string as the parameter. To insert a string into the list, call the *Insert* method, passing two parameters: the string and the index of the position where you want it placed. For example, to make the string "Three" the third string in a list, you would use

```
Insert(2, 'Three');
```

To append the strings from one list onto another, call *AddStrings*:

```
StringList1.AddStrings(StringList2);  { append the strings from StringList2 to
StringList1 }
```

## Moving a string within a list

To move a string in a string list, call the *Move* method, passing two parameters: the current index of the string and the index you want assigned to it. For example, to move the third string in a list to the fifth position, you would use

```
Move(2, 4)
```

## Deleting a string from a list

To delete a string from a string list, call the list's *Delete* method, passing the index of the string you want to delete. If you don't know the index of the string you want to delete, use the *IndexOf* method to locate it. To delete all the strings in a string list, use the *Clear* method.

This example uses *IndexOf* and *Delete* find and delete a string.

```
with ListBox1.Items do
begin
  BIndex:=IndexOf('bureaucracy');
  if BIndex > -1 then
    Delete(BIndex);
end;
```

## Copying a complete string list

You can use the *Assign* method to copy strings from a source list to a destination list, overwriting the contents of the destination list. To append strings without overwriting the destination list, use *AddStrings*. For example,

```
Memo1.Lines.Assign(ComboBox1.Items);    { overwrites original strings }
```

copies the lines from a combo box into a memo (overwriting the memo), while

```
Memo1.Lines.AddStrings(ComboBox1.Items);    { appends strings to end }
```

appends the lines from the combo box to the memo.

When making local copies of a string list, use the *Assign* method. If you simply assign one string-list variable to another—

```
StringList1 := StringList2;
```

—the original string-list object will be lost, often with unpredictable results.

## Associating objects with a string list

In addition to the strings stored in its *Strings* property, a string list can maintain references to *objects*, which it stores in its *Objects* property. Like *Strings*, *Objects* is an array with a zero-based index. The most common use for *Objects* is to associate bitmaps with strings for owner-draw controls.

Use the *AddObject* or *InsertObject* method to add a string and an associated object to the list in a single step. *IndexOfObject* returns the index of the first string in the list associated with a specified object. Methods like *Delete*, *Clear,* and *Move* operate on both strings and objects; for example, deleting a string removes the corresponding object (if there is one).

To associate an object with an existing string, assign the object to the *Objects* property at the same index. You cannot add an object without adding a corresponding string.

## The Windows registry and INI files

The Windows system registry is a hierarchical database where applications store configuration information. The VCL object *TRegistry* supplies methods that read and write to the registry.

Until Windows 95, most applications stored configuration information in initialization files, usually named with the extension .INI. The VCL provides objects that facilitate maintenance and migration of programs that use INI files. Use

- *TRegistry* to work with the registry.
- *TIniFile* or *TMemIniFile* to work with Windows 3.x INI files.
- *TRegistryIniFile* when you want to work with both the registry and INI files. *TRegistryIniFile* has properties and methods similar to those of *TIniFile*, but it reads and writes to the system registry. By using a variable of type *TCustomIniFile* (the common ancestor of *TIniFile*, *TMemIniFile*, and *TRegistryIniFile*), you can write generic code that accesses either the registry or an INI file, depending on where it is called.

## Using streams

Use specialized stream objects to read or write to storage media. Each descendant of *TStream* implements methods for accessing a particular medium, such as disk files, dynamic memory, and so on. *TStream* descendants include *TFileStream*, *TStringStream*, *TMemoryStream*, *TBlobStream*, and *TWinSocketStream*. In addition to methods for reading and writing, these objects permit applications to seek to an arbitrary position in the stream. Properties of *TStream* provide information about the stream, such as size and current position.

## Using data modules and remote data modules

A data module is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool.

There are two types of data module: standard and remote. To create a single- or two-tiered application, use a standard data module. If you have the Client/Server or Enterprise edition of Delphi and are creating a multi-tiered application, you can add a remote data module to your application server; see Adding a remote data module to an application server project.

# Creating and editing data modules

To create a data module, choose File|New and double-click on Data Module. Delphi opens an empty data module in the Data Module Designer, displays the unit file for the new module in the Code editor, and adds the module to the current project. When you reopen an existing data module, Delphi displays its components in the Data Module Designer.

The Data Module Designer is divided into two panes. The left pane displays a hierarchical tree view of the components in the module. The right pane has two tabs: Components and Data Diagram. The Components page shows the components as they would appear on a form. The Data Diagram page shows a graphical representation of internal relationships among the components, such as master-detail links and lookup fields.
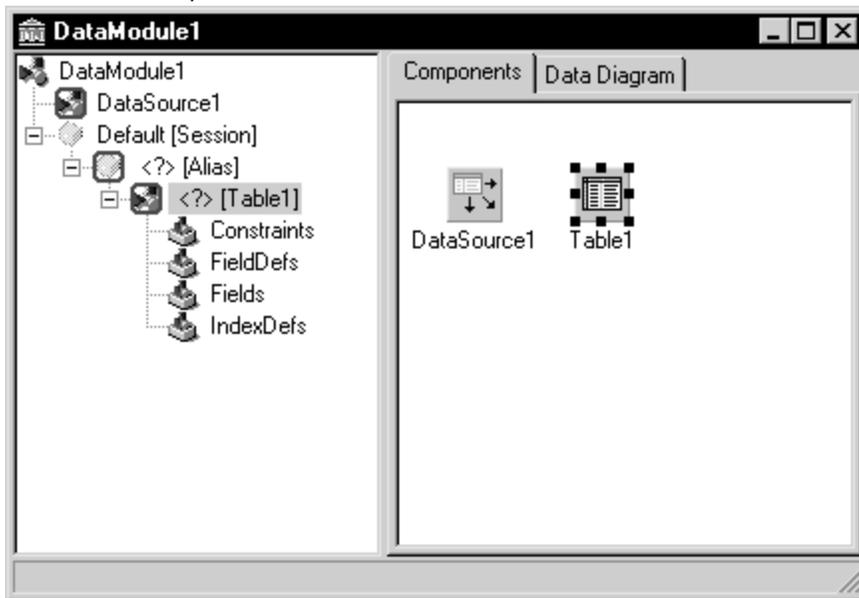


*A simple data module*

You can add components to a data module by selecting them on the Component palette and clicking in the Tree or Components view of the Data Module Designer. When a component is selected in the Data Module Designer, you can edit its properties in the Object Inspector just as you would if the component were on a form. For more information about the Data Module Designer, see About the Data Module Designer.

## Creating business rules in a data module

In a data module's unit file, you can write methods, including event handlers for the components in the module, as well as global routines that encapsulate business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping; you could call such a procedure from an event handler for a component in the module or from any unit that uses the module.

## Accessing a data module from a form

To associate visual controls on a form with a data module, you must first add the data module to the form's **uses** clause. You can do this in several ways:

▪        In the Code editor, open the form's unit file and add the name of the data module to the **uses** clause in the **interface** section.

▪        Choose File|Use Unit, then enter the name of the module or pick it from the list box in the Use Unit dialog.

▪        Double-click on a *TTable* or *TQuery* component in the data module to open the Fields editor. From the Fields editor, drag any fields onto your form. Delphi prompts you to confirm that you want to add the module to the form's **uses** clause, then creates controls (such as edit boxes) for the fields.

## Adding a remote data module to an application server project

Some versions of Delphi allow you to add *remote data modules* to application server projects. A remote data module has an interface that clients in a multi-tiered application can access across networks. To add a remote data module to a project, choose File|New, select the Multitier page in the New Items dialog box, and double-click the desired type of module (Remote Data Module, MTS Data Module, or CORBA Data Module) to open the Remote Data Module wizard. Once you add a remote data module to a project, you use it just like a standard data module.

For more information about multi-tiered database applications, see Creating multi-tiered applications.

## Using the Object Repository

The Object Repository (Tools|Repository) makes it easy share forms, dialog boxes, frames, and data modules. It also provides templates for new projects and wizards that guide the user through the creation of forms and projects. The repository is maintained in DELPHI32.DRO (by default in the BIN directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

## Sharing items within a project

You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (File|New), you'll see a page tab with the name of the current project. This page lists all the forms, dialog boxes, and data modules in the project. You can derive a new item from an existing item and customize it as needed.

## Adding items to the Object Repository

You can add your own projects, forms, frames, and data modules to those already available in the Object Repository. To add an item to the Object Repository,

1   If the item is a project or is in a project, open the project.
2   For a project, choose Project|Add To Repository. For a form or data module, right-click the item and choose Add To Repository.
3   Type a description, title, and author.
4   Decide which page you want the item to appear on in the New Items dialog box, then type the name of the page or select it from the Page combo box. If you type the name of a page that doesn't exist, Delphi creates a new page.
5   Choose Browse to select an icon to represent the object in the Object Repository.
6   Choose OK.

## Sharing objects in a team environment

You can share objects with your workgroup or development team by making a repository available over a network. To use a shared repository, all team members must select the same Shared Repository directory in the Environment Options dialog:

1   Choose Tools|Environment Options.
2   On the Preferences page, locate the Shared Repository panel. In the Directory edit box, enter the directory where you want to locate the shared repository. Be sure to specify a directory that's accessible to all team members.

The first time an item is added to the repository, Delphi creates a DELPHI32.DRO file in the Shared Repository directory if one doesn't exist already.

## Using an Object Repository item in a project

To access items in the Object Repository, choose File|New. The New Items dialog appears, showing all the items available. Depending on the type of item you want to use, you have up to three options for adding the item to your project:

- <u>Copy</u>
- <u>Inherit</u>
- <u>Use</u>

## Copying an item

Choose Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Copy is the only option available for project templates.

## Inheriting an item

Choose Inherit to derive a new class from the selected item in the Object Repository and add the new class to your project. When you recompile your project, any changes that have been made to the item in the Object Repository will be reflected in your derived class, in addition to changes you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository.

Inherit is available for forms, dialog boxes, and data modules, but not for project templates. It is the *only* option available for reusing items within the same project.

## Using an item

Choose Use when you want the selected item itself to become part of your project. Changes made to the item in your project will appear in all other projects that have added the item with the Inherit or Use option. Select this option with caution.

The Use option is available for forms, dialog boxes, and data modules.

## Using project templates

Templates are predesigned projects that you can use as starting points for your own work. To create a new project from a template,

1　Choose File|New to display the New Items dialog box.
2　Choose the Projects tab.
3　Select the project template you want and choose OK.
4　In the Select Directory dialog, specify a directory for the new project's files.

Delphi copies the template files to the specified directory, where you can modify them. The original project template is unaffected by your changes.

## Modifying shared items

If you modify an item in the Object Repository, your changes will affect all future projects that use the item as well as existing projects that have added the item with the Use or Inherit option. To avoid propagating changes to other projects, you have several alternatives:

- Copy the item and modify it in your current project only.
- Copy the item to the current project, modify it, then add it to the Repository under a different name.
- Create a component, DLL, component template, or frame from the item. If you create a component or DLL, you can share it with other developers.

## Specifying a default project, new form, and main form

By default, when you choose File|New Application or File|New Form, Delphi displays a blank form. You can change this behavior by reconfiguring the Repository:

1  Choose Tools|Repository

2  If you want to specify a default project, select the Projects page and choose an item under Objects. Then select the New Project check box.

3  If you want to specify a default form, select a Repository page (such as Forms), them choose a form under Objects. To specify the default new form (File|New Form), select the New Form check box. To specify the default main form for new projects, select the Main Form check box.

4  Click OK.

## Adding custom components to the IDE

You can install custom components—written by yourself or third parties—on the Component palette and use them in your applications. To write a component, see Overview of component creation. To install an existing component, see Installing component packages.

## Creating applications

The main use of Delphi is designing and building Windows applications. There are three basic kinds of Windows application:

- [Windows GUI applications](#)
- [Console applications](#)
- [Service applications](#)

## Windows applications

When you compile a project, an executable (.EXE) file is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an EXE. You can extend the application by calling DLLs, packages, and other support files from the executable.

Windows offers two application UI models:

- Single document interface (SDI)
- Multiple document interface (MDI)

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE.

# User interface models

Any form can be implemented as a multiple document interface (MDI) or single document interface (SDI) form. In an MDI application, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors. An SDI application, in contrast, normally contains a single document view. To make your form an SDI application, set the *FormStyle* property of your *Form* object to *fsNormal*.

For more information on developing the UI for an application, see Developing the application user interface.

## SDI Applications

To create a new SDI application,
1   Select File|New to bring up the New Items dialog.
2   Click on the Projects page and select SDI Application.
3   Click OK.

By default, the *FormStyle* property of your *Form* object is set to *fsNormal*, so Delphi assumes that all new applications are SDI applications.

## MDI applications

To create a new MDI application,

1   Select File|New to bring up the New Items dialog.
2   Click on the Projects page and select MDI Application.
3   Click OK.

MDI applications require more planning and are somewhat more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. Set the *FormStyle* property of the *TForm* object to specify whether a form is a child (*fsMDIForm*) or main form (*fsMDIChild*). It is a good idea to define a base class for your child forms and derive each child form from this class, to avoid having to reset the child form's properties.

## Setting IDE, project, and compilation options

Use Project|Project Options to specify various options for your project.

### Setting default project options

To change the default options that apply to all future projects, set the options in the Project Options dialog box and check the Default box at the bottom right of the window. All new projects will now have the current options selected by default.

## Programming templates

Programming templates are commonly used "skeleton" structures that you can add to your source code and then fill in. For example, if you want to use a **for** loop in your code, you could insert the following template:

```
for :=  to  do
begin
end;
```

To insert a code template in the Code editor, press *Ctrl-j* and select the template you want to use. You can also add your own templates to this collection. To add a template:

1  Select Tools|Environment Options.
2  Click the Code Insight tab.
3  In the templates section click Add.
4  Choose a shortcut name and enter a brief description of the new template.
5  Add the template code to the Code text box.
6  Click OK.

## Console applications

Console applications are 32-bit Windows programs that run without a graphical interface, usually in a console window. These applications typically don't require much user input and perform a limited set of functions.

To create a new console application, choose File|New and select Console Wizard from the New Items dialog box.

## Service applications

Service applications take requests from client applications, process those requests, and return information to the client applications. They typically run in the background, without much user input. A web, FTP, or e-mail server is an example of a service application.

To create an application that implements a Win32 service, Choose File|New, and select Service Application from the New Items page. This adds a global variable named *Application* to your project, which is of type *TServiceApplication*.

Once you have created a service application, you will see a window in the designer that corresponds to a service (*TService*). Implement the service by setting its properties and event handlers in the Object Inspector. You can add additional services to your service application by choosing Service from the new items dialog. Do not add services to an application that is not a service application. While a *TService* object can be added, the application will not generate the requisite events or make the appropriate Windows calls on behalf of the service.

Once your service application is built, you can install its services with the Service Control Manager (SCM). Other applications can then launch your services by sending requests to the SCM.

To install your application's services, run it using the /INSTALL option. The application installs its services and exits, giving a confirmation message if the services are successfully installed. You can suppress the confirmation message by running the service application using the /SILENT option.

To uninstall the services, run it from the command line using the /UNINTALL option. (You can also use the /SILENT option to suppress the confirmation message when uninstalling).

**Example:** This service has a *TServerSocket* whose port is set to 80. This is the default port for Web Browsers to make requests to Web Servers and for Web Servers to make responses to Web Browsers. This particular example produces a text document in the C:\Temp directory called WebLog*xxx*.log (where *xxx* is the ThreadID). There should be only one Server listening on any given port, so if you have a web server, you should make sure that it is not listening (the service is stopped).

To see the results: open up a web browser on the local machine and for the address, type 'localhost' (with no quotes). The Browser will time out eventually, but you should now have a file called weblog*xxx*.log in the C:\temp directory.

1   To create the example, choose File|New and select Service Application from the New Items dialog. You will see a window appear named Service1. From the Internet page of the component palette, add a ServerSocket component to the service window (Service1).

2   Next, add a private data member of type TMemoryStream to the TService1 class. The interface section of your unit should now look like this:

```
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;
type
  TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure Service1Execute(Sender: TService);
  private
    { Private declarations }
    Stream: TMemoryStream; // Add this line here
  public
    function GetServiceController: PServiceController; override;
    { Public declarations }
  end;
var
  Service1: TService1;
```

3   Next, select ServerSocket1, the component you added in step 1. In the Object Inspector, double click the *OnClientRead* event and add the following event handler:

```
procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buffer: PChar;
begin
  Buffer := nil;
while Socket.ReceiveLength > 0 do begin
    Buffer := AllocMem(Socket.ReceiveLength);
    try
      Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
      Stream.Write(Buffer^, StrLen(Buffer));
    finally
      FreeMem(Buffer);
    end;
  Stream.Seek(0, soFromBeginning);
  Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
  end;
end;
```

4   Finally, select Service1 by clicking in the window's client area (but not on the ServiceSocket). In the Object Inspector, double click the *OnExecute* event and add the following event handler:

```
procedure TService1.Service1Execute(Sender: TService);
begin
  Stream := TMemoryStream.Create;
  try
    ServerSocket1.Port := 80; // WWW port
    ServerSocket1.Active := True;
    while not Terminated do begin
      ServiceThread.ProcessRequests(False);
    end;
    ServerSocket1.Active := False;
  finally
    Stream.Free;
  end;
end;
```

When writing your service application, you should be aware of:

- Service threads
- Service name properties
- Debugging services

## Service threads

Each service has its own thread (*TServiceThread*), so if your service application implements more than one service you must ensure that the implementation of your services is thread-safe. *TServiceThread* is designed so that you can implement the service in the *TService OnExecute* event handler. The service thread has its own *Execute* method which contains a loop that calls the service's *OnStart* and *OnExecute* handlers before processing new requests. Because service requests can take a long time to process and the service application can receive simultaneous requests from more than one client, it is more efficient to spawn a new thread (derived from *TThread*, not *TServiceThread*) for each request and move the implementation of that service to the new thread's *Execute* method. This allows the service thread's *Execute* loop to process new requests continually without having to wait for the service's *OnExecute* handler to finish. The following example demonstrates.

**Example:** This service beeps every 500 milliseconds from within the standard thread. It handles pausing, continuing, and stopping of the thread when the service is told to pause, continue, or stop.

1. Choose File|New and select Service Application from the New Items dialog. You will see a window appear named Service1.

2. In the interface section of your unit, declare a new descendant of TThread named TSparkyThread. This is the thread that does the work for your service. The declaration should appear as follows:

```
TSparkyThread = class(TThread)
   public
      procedure Execute; override;
   end;
```

3. Next, in the implementation section of your unit, create a global variable for a TSparkyThread instance:

```
var
   SparkyThread: TSparkyThread;
```

4. Add the following code to the implementation section for the TSparkyThread Execute method (the thread function):

```
procedure TSparkyThread.Execute;
begin
   while not Terminated do
   begin
      Beep;
      Sleep(500);
   end;
end;
```

5. Select the Service window (Service1), and double-click the OnStart event in the Object Inspector. Add the following OnStart event handler:

```
procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
   SparkyThread := TSparkyThread.Create(False);
   Started := True;
end;
```

6. Double-click the OnContinue event in the Object Inspector. Add the following OnContinue event handler:

```
procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
begin
   SparkyThread.Resume;
   Continued := True;
end;
```

7. Double-click the OnPause event in the Object Inspector. Add the following OnPause event handler:

```
procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
   SparkyThread.Suspend;
   Paused := True;
end;
```

8    Finally, double-click the OnStop event in the Object Inspector and add the following OnStop event handler:

```
procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
  SparkyThread.Terminate;
  Stopped := True;
end;
```

When developing server applications, choosing to spawn a new thread depends on the nature of the service being provided, the anticipated number of connections, and the expected number of processors on the computer running the service.

## Service name properties

The VCL provides classes for creating service applications. These include *TService* and *TDependency*. When using these classes, the various name properties can be confusing. This section describes the differences.

Services have user names (called Service start names) that are associated with passwords, display names for display in manager and editor windows, and actual names (the name of the service). Dependencies can be services or they can be load ordering groups. They also have names and display names. And because service objects are derived from *TComponent*, they inherit the *Name* property. The following sections summarize the name properties:

### TDependency properties

The *TDependency DisplayName* is both a display name and the actual name of the service. It is nearly always the same as the *TDependency Name* property.

### TService name properties

The *TService Name* property is inherited from *TComponent*. It is the name of the component, and is also the name of the service. For dependencies that are services, this property is the same as the *TDependency Name* and *DisplayName* properties.

*TService*'s *DisplayName* is the name displayed in the Service Manager window. This often differs from the actual service name (*TService.Name*, *TDependency.DisplayName*, *TDependency.Name*). Note that the *DisplayName* for the Dependency and the *DisplayName* for the Service usually differ.

Service start names are distinct from both the service display names and the actual service names. A *ServiceStartName* is the user name input on the Start dialog selected from the Service Control Manager.

## Debugging services

Debugging service applications can be tricky, because it requires short time intervals:

1    First, launch the application in the debugger. Wait a few seconds until it has finished loading.
2    Quickly start the service from the control panel or from the command line:

```
net start MyServ
```

You must launch the service quickly (within 15-30 seconds of application startup) because the application will terminate if no service is launched.

Another approach is to attach to the service application process when it is already running. (That is, starting the service first, and then attaching to the debugger). To attach to the service application process, choose Run|Attach To Process, and select the service application in the resulting dialog.

In some cases, this second approach may fail, due to insufficient rights. If that happens, you can use the Service Control Manager to enable your service to work with the debugger:

1    First create a key called **Image File Execution Options** in the following registry location:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
```

2    Create a subkey with the same name as your service (for example, MYSERV.EXE). To this subkey, add a value of type REG_SZ, named Debugger. Use the full path to the debugger as the string value.
3    In the Services control panel applet, select your service, click Startup and check Allow Service to Interact with Desktop.

# Creating packages and DLLs

Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application.

Packages are special DLLs used by Delphi applications, the IDE, or both. There are two kinds of packages: runtime packages and design-time packages. Runtime packages provide functionality to a program while that program is running. Design-time packages extend the functionality of the IDE.

For more information on packages, see Working with packages and components.

## When to use packages and DLLs

For most applications written in Delphi, packages provide greater flexibility and are easier to create than DLLs. However, there are several situations where DLLs would be better suited to your projects than packages:

- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE container.

## Writing database applications

One of Delphi's strengths is its support for creating advanced database applications. Delphi includes built-in tools that allow you to connect to Oracle, Sybase, Informix, dBASE, Paradox, or other servers while providing transparent data sharing between applications. The Borland Database Engine (BDE) supports scaling from desktop to client/server applications.

Tools, such as the Database Explorer, simplify the task of writing database applications. The Database Explorer is a hierarchical browser for inspecting and modifying database server-specific schema objects including tables, fields, stored procedure definitions, triggers, references, and index descriptions.

Through a persistent connection to a database, Database Explorer lets you

- Create and maintain database aliases
- View schema data in a database, such as tables, stored procedures, and triggers
- View table objects, such as fields and indexes
- Create, view, and modify data in tables
- Enter SQL statements to directly query any database
- Create and maintain data dictionaries to store attribute sets

## Building distributed applications

Distributed applications are applications that are deployed to various machines and platforms and work together, typically over a network, to perform a set of related functions. For instance, an application for purchasing items and tracking those purchases for a nationwide company would require individual client applications for all the outlets, a main server that would process the requests of those clients, and an interface to a database that stores all the information regarding those transactions. By building a distributed client application (for instance, a web-based application), maintaining and updating the individual clients is vastly simplified.

Delphi provides several options for the implementation model of distributed applications:

- TCP/IP applications
- COM and DCOM applications
- CORBA applications
- Database applications

## Distributing applications using TCP/IP

TCP/IP is a communication protocol that allows you to write applications that communicate over networks. You can implement virtually any design in your applications. TCP/IP provides a transport layer, but does not impose any particular architecture for creating your distributed application.

The growth of the Internet has created an environment where most computers already have some form of TCP/IP access, which simplifies distributing and setting up the application.

Applications that use TCP/IP can be message-based distributed applications (such as Web server applications that service HTTP request messages) or distributed object applications (such as distributed database applications that communicate using Windows sockets).

The most basic method of adding TCP/IP functionality to your applications is to use client or server sockets. Delphi also provides support for applications that extend Web servers by creating CGI scripts or DLLs. In addition, Delphi provides support for TCP/IP-based database applications.

## Using sockets in applications

Two VCL classes, *TClientSocket* and *TServerSocket*, allow you to create TCP/IP socket connections to communicate with other remote applications. For more information on sockets, see Working with sockets.

## Creating Web server applications

To create a new Web server application, select File|New and select Web Server Application in the New Items dialog box. Then select the Web server application type:

- ISAPI and NSAPI
- CGI stand-alone
- Win-CGI stand-alone

CGI and Win-CGI applications use more system resources on the server, so complex applications are better created as ISAPI or NSAPI applications.

For more information on building Web server applications, see Creating Internet server applications.

### ISAPI and NSAPI Web server applications

Selecting this type of application sets up your project as a DLL. ISAPI or NSAPI Web server applications are DLLs loaded by the Web server. Information is passed to the DLL, processed, and returned to the client by the Web server.

### CGI stand-alone Web server applications

CGI Web server applications are console applications that receive requests from clients on standard input, processes those requests, and sends back the results to the server on standard output to be sent to the client.

### Win-CGI stand-alone Web server applications

Win-CGI Web server applications are Windows applications that receive requests from clients from an INI file written by the server and writes the results to a file that the server sends to the client.

## Distributing applications using COM and DCOM

COM is the Component Object Model, a Windows-based distributed object architecture designed to provide object interoperability using predefined routines called interfaces. COM applications use objects that are implemented by a different process or, if you use DCOM, on a separate machine.

### COM and DCOM

Delphi has classes and wizards that make it easy to create the essential elements of a COM, OLE, or ActiveX application. Using Delphi to create COM-based applications offers a wide range of possibilities, from improving software design by using interfaces internally in an application, to creating objects that can interact with other COM-based API objects on the system, such as the Win95 Shell extensions and DirectX multimedia support.

For more information on COM and Active X controls, see Overview of COM technologies, Creating an ActiveX control and Distributing a client application as an ActiveX control.

For more information on DCOM, see Using DCOM connections.

### MTS

The Microsoft Transaction Server (MTS) is a an extension to DCOM that provides transaction services, security, and resource pooling for distributed COM applications.

For more information on MTS, see Creating MTS objects and Using MTS.

## Distributing applications using CORBA

Common Object Request Broker Architecture (CORBA) is a method of using distributed objects in applications. The CORBA standard is used on many platforms, so writing CORBA applications allows you to make use of programs that are not running on a Windows machine.

Like COM, CORBA is a distributed object architecture, meaning that client applications can make use of objects that are implemented on a remote server.

For more information on CORBA, see Writing CORBA applications.

For instructions on distributing applications using CORBA, see Deploying CORBA applications.

## Distributing database applications

Delphi provides support for creating distributed database applications using the MIDAS technology. This powerful technology includes a coordinated set of components that allow you to build a wide variety of multi-tiered database applications. Distributed database applications can be built on a variety of communications protocols, including DCOM, CORBA, TCP/IP, and OLEnterprise.

For more information about building distributed database applications, see Creating multi-tiered applications..

Distributing database applications often requires you to distribute the Borland Database Engine (BDE) in addition to the application files. For information on deploying the BDE, see Deploying database applications.

## Common programming tasks

This section of the Help system discusses the fundamentals for some of the common programming tasks in Delphi:

- Handling exceptions
- Using interfaces
- Working with strings
- Working with files
- Defining new data types

# Handling exceptions

Delphi provides a mechanism to ensure that applications are robust, meaning that they handle errors in a consistent manner. Exception handling allows the application to recover from errors if possible and to shut down if need be, without losing data or resources. Error conditions in Delphi are indicated by exceptions. This topic describes the following tasks for using exceptions to create safe applications:

- Protecting blocks of code
- Protecting resource allocations
- Handling RTL exceptions
- Handling component exceptions
- Using TApplication.HandleException
- Silent exceptions
- Defining your own exceptions

# Protecting blocks of code

To make your applications robust, your code needs to recognize exceptions when they occur and respond to them. If you don't specify a response, the application will present a message box describing the error. Your job, then, is to recognize places where errors might happen, and define responses, particularly in areas where errors could cause the loss of data or system resources.

When you create a response to an exception, you do so on blocks of code. When you have a series of statements that all require the same kind of response to errors, you can group them into a block and define error responses that apply to the whole block.

Blocks with specific responses to exceptions are called protected blocks because they can guard against errors that might otherwise either terminate the application or damage data.

To protect blocks of code you need to understand
- Responding to exceptions
- Exceptions and the flow of control
- Nesting exception responses

## Responding to exceptions

When an error condition occurs, the application raises an exception, meaning it creates an exception object. Once an exception is raised, your application can execute cleanup code, handle the exception, or both.

- **Executing cleanup code:** The simplest way to respond to an exception is to guarantee that some cleanup code is executed. This kind of response doesn't correct the condition that caused the error but lets you ensure that your application doesn't leave its environment in an unstable state. You typically use this kind of response to ensure that the application frees allocated resources, regardless of whether errors occur.
- **Handling an exception:** This is a specific response to a specific kind of exception. Handling an exception clears the error condition and destroys the exception object, which allows the application to continue execution. You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as "File not found," are easy to correct and retry, while others, such as running out of memory, might be more difficult for the application or the user to correct.

## Exceptions and the flow of control

Object Pascal makes it easy to incorporate error handling into your applications because exceptions don't get in the way of the normal flow of your code. In fact, by moving error checking and error handling out of the main flow of your algorithms, exceptions can simplify the code you write.

When you declare a protected block, you define specific responses to exceptions that might occur within that block. When an exception occurs in that block, execution immediately jumps to the response you defined, then leaves the block.

**Example:** The following code that includes a protected block. If any exception occurs in the protected block, execution jumps to the exception-handling part, which beeps. Execution resumes outside the block.

```
...
try { begin the protected block }
  Font.Name := 'Courier'; { if any exception occurs... }
  Font.Size := 24;  { ...in any of these statements... }
  Color := clBlue;
except{ ...execution jumps to here }
  on Exception do MessageBeep(0);  { this handles any exception by beeping }
end;
... { execution resumes here, outside the protected block}
```

## Nesting exception responses

Your code defines responses to exceptions that occur within blocks. Because Pascal allows you to nest blocks inside other blocks, you can customize responses even within blocks that already customize responses.

In the simplest case, for example, you can protect a resource allocation, and within that protected block, define blocks that allocate and protect other resources. Conceptually, that might look something like this:

```
                  { allocate first resource }
              ┌──────try
              │          { allocate second resource }
              │      ┌──────try
              │      │          { code that uses both resources }
              │      │      finally
              │      │          { release second resource }
              │      └──────end;
              │      finally
              │          { release first resource }
              └──────end;
```
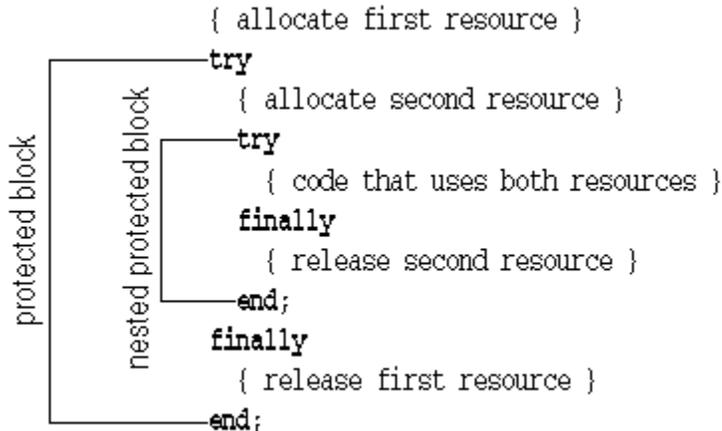
You can also use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. Conceptually, that looks something like this:

```
              ┌──────try
              │          { protected code }
              │      ┌──────try
              │      │          { specially protected code }
              │      │      except
              │      │          { local exception handling }
              │      └──────end;
              │      except
              │          { global exception handling }
              └──────end;
```
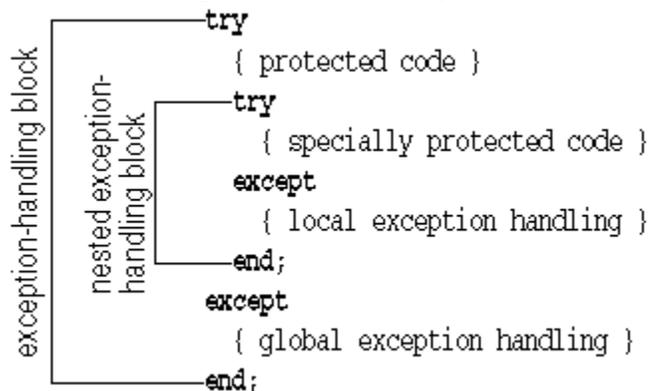
You can also mix different kinds of exception-response blocks, nesting resource protections within exception handling blocks and vice versa.

# Protecting resource allocations

One key to having a robust application is ensuring that if it allocates resources, it also releases them, even if an exception occurs. For example, if your application allocates memory, you need to make sure it eventually releases the memory, too. If it opens a file, you need to make sure it closes the file later.

Keep in mind that exceptions don't come just from your code. A call to an RTL routine, for example, or another component in your application might raise an exception. Your code needs to ensure that if these conditions occur, you release allocated resources.

To protect resources effectively, you need to understand the following:

- What kind of resources need protection?
- Creating a resource protection block

## What kind of resources need protection?

Under normal circumstances, you can ensure that an application frees allocated resources by including code for both allocating and freeing. When exceptions occur, however, you need to ensure that the application still executes the resource-freeing code.

Some common resources that you should always be sure to release are:
- Files
- Memory
- Windows resources
- Objects

**Example:** The following event handler allocates memory, then generates an error, so it never executes the code to free the memory:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024); { allocate 1K of memory }
  AnInteger := 10 div ADividend; { this generates an error }
  FreeMem(APointer, 1024);{ it never gets here }
end;
```

Although most errors are not that obvious, the example illustrates an important point: When the division-by-zero error occurs, execution jumps out of the block, so the *FreeMem* statement never gets to free the memory.

In order to guarantee that the *FreeMem* gets to free the memory allocated by *GetMem*, you need to put the code in a resource-protection block.

## Creating a resource protection block

To ensure that you free allocated resources, even in case of an exception, you embed the resource-using code in a protected block, with the resource-freeing code in a special part of the block. Here's an outline of a typical protected resource allocation:

```
{ allocate the resource }
try
  { statements that use the resource }
finally
  { free the resource }
end;
```

The key to the **try..finally** block is that the application always executes any statements in the **finally** part of the block, even if an exception occurs in the protected block. When any code in the **try** part of the block (or any routine called by code in the **try** part) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the **finally** part, which is called the cleanup code. After the finally part is executed, the exception handler is called. If no exception occurs, the cleanup code is executed in the normal order, after all the statements in the **try** part.

**Example:** The following code illustrates an event handler that allocates memory and generates an error, but still frees the allocated memory:

```
procedure TForm1.Button1Click(Sender: TComponent);
var
  APointer: Pointer;
  AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024); { allocate 1K of memory }
  try
    AnInteger := 10 div ADividend; { this generates an error }
  finally
    FreeMem(APointer, 1024); { execution resumes here, despite the error }
  end;
end;
```

The statements in the termination code do not depend on an exception occurring. If no statement in the **try** part raises an exception, execution continues through the termination code.

## Handling RTL exceptions

When you write code that calls routines in the runtime library (RTL), such as mathematical functions or file-handling procedures, the RTL reports errors back to your application in the form of exceptions. By default, RTL exceptions generate a message that the application displays to the user. You can define your own exception handlers to handle RTL exceptions in other ways.

There are also silent exceptions that do not, by default, display a message.

To handle RTL exceptions effectively, you need to understand the following:

- What are the RTL exceptions?
- Creating an exception handler
- Exception handling statements
- Using the exception instance
- Scope of exception handlers
- Providing default exception handlers
- Handling classes of exceptions
- Reraising the exception

# What are the RTL exceptions?

The runtime library's exceptions are defined in the *SysUtils* unit, and they all descend from a generic exception-object type called Exception. Exception provides the string for the message that RTL exceptions display by default.

There are several kinds of exceptions raised by the RTL, as described in the following table.

| Error type | Cause | Meaning |
| --- | --- | --- |
| Input/output | Error accessing a file or I/O device | Most I/O exceptions are related to error codes returned by Windows when accessing a file. |
| Heap | Error using dynamic memory | Heap errors can occur when there is insufficient memory available, or when an application disposes of a pointer that points outside the heap. |
| Integer math | Illegal operation on integer-type expressions | Errors include division by zero, numbers or expressions out of range, and overflows. |
| Floating-point math | Illegal operation on real-type expressions | Floating-point errors can come from either a hardware coprocessor or the software emulator. Errors include invalid instructions, division by zero, and overflow or underflow. |
| Typecast | Invalid typecasting with the **as** operator | Objects can only be typecast to compatible types. |
| Conversion | Invalid type conversion | Type-conversion functions such as IntToStr, StrToInt, and StrToFloat raise conversion exceptions when the parameter cannot be converted to the desired type. |
| Hardware | System condition | Hardware exceptions indicate that either the processor or the user generated some kind of error condition or interruption, such as an access violation, stack overflow, or keyboard interrupt. |
| Variant | Illegal type coercion | Errors can occur when referring to variants in expressions where the variant cannot be coerced into a compatible type. |

For a list of the RTL exception types, see the *SysUtils* unit.

# Creating an exception handler

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code.

To define an exception handler, embed the code you want to protect in an exception-handling block and specify the exception handling statements in the **except** part of the block. Here is an outline of a typical exception-handling block:

```
try
   { statements you want to protect }
except
   { exception-handling statements }
end;
```

The application executes the statements in the **except** part only if an exception occurs during execution of the statements in the **try** part. Execution of the **try** part statements includes routines called by code in the **try** part. That is, if code in the **try** part calls a routine that doesn't define its own exception handler, execution returns to the exception-handling block, which handles the exception.

When a statement in the **try** part raises an exception, execution immediately jumps to the **except** part, where it steps through the specified exception-handling statements, or exception handlers, until it finds a handler that applies to the current exception.

Once the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Execution continues at the end of the current block.

## Exception handling statements

Each **on** statement in the **except** part of a **try..except** block defines code for handling a particular kind of exception. The form of these exception-handling statements is as follows:

```
on <type of exception> do <statement>;
```

**Example:** You can define an exception handler for division by zero to provide a default result:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  try
    Result := Sum div NumberOfItems; { handle the normal case }
  except
    on EDivByZero do Result := 0;  { handle the exception only if needed }
  end;
end;
```

Note that this is clearer than having to test for zero every time you call the function. Here's an equivalent function that doesn't take advantage of exceptions:

```
function GetAverage(Sum, NumberOfItems: Integer): Integer;
begin
  if NumberOfItems <> 0 then { always test }
    Result := Sum div NumberOfItems{ use normal calculation }
  else Result := 0; { handle exceptional case }
end;
```

The difference between these two functions really defines the difference between programming with exceptions and programming without them. This example is quite simple, but you can imagine a more complex calculation involving hundreds of steps, any one of which could fail if one of dozens of inputs were invalid.

By using exceptions, you can spell out the "normal" expression of your algorithm, then provide for those exceptional cases when it doesn't apply. Without exceptions, you have to test every single time to make sure you're allowed to proceed with each step in the calculation.

## Using the exception instance

Most of the time, an exception handler doesn't need any information about an exception other than its type, so the statements following on..do are specific only to the type of exception. In some cases, however, you might need some of the information contained in the exception instance.

To read specific information about an exception instance in an exception handler, you use a special variation of on..do that gives you access to the exception instance. The special form requires that you provide a temporary variable to hold the instance.

**Example:** If you create a new project that contains a single form, you can add a scroll bar and a command button to the form. Double-click the button and add the following line to its click-event handler:

```
ScrollBar1.Max := ScrollBar1.Min - 1;
```

That line raises an exception because the maximum value of a scroll bar must always exceed the minimum value. The default exception handler for the application opens a dialog box containing the message in the exception object. You can override the exception handling in this handler and create your own message box containing the exception's message string:

```
try
  ScrollBar1.Max := ScrollBar1.Min - 1;
except
  on E: EInvalidOperation do
    MessageDlg('Ignoring exception: ' + E.Message, mtInformation, [mbOK], 0);
end;
```

The temporary variable (E in this example) is of the type specified after the colon (*EInvalidOperation* in this example). You can use the as operator to typecast the exception into a more specific type if needed.

**Note:** Never destroy the temporary exception object. Handling an exception automatically destroys the exception object. If you destroy the object yourself, the application attempts to destroy the object again, generating an access violation.

# Scope of exception handlers

You do not need to provide handlers for every kind of exception in every block. In fact, you need to provide handlers only for those exceptions that you want to handle specially within a particular block.

If a block does not handle a particular exception, execution leaves that block and returns to the block that contains the block (or to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception.

## Providing default exception handlers

You can provide a single default exception handler to handle any exceptions you haven't provided specific handlers for. To do that, you add an else part to the **except** part of the exception-handling block:

```
try
  { statements }
except
  on ESomething do
    { specific exception-handling code };
  else
    { default exception-handling code };
end;
```

Adding default exception handling to a block guarantees that the block handles every exception in some way, thereby overriding all handling from the containing block.

**Caution:** It is not advisable to use this all-encompassing default exception handler. The **else** clause handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. If you want to handle cleanup and leave the exception handling to code that has more information about the exception and how to handle it, then you can do so use an enclosing **try..finally** block:

```
try
  try
  { statements }
  except
    on ESomething do { specific exception-handling code };
  end;
finally
  {cleanup code };
end;
```

For another approach to augmenting exception handling, see Reraising the exception.

## Handling classes of exceptions

Topic groups

Because exception objects are part of a hierarchy, you can specify handlers for entire parts of the hierarchy by providing a handler for the exception class from which that part of the hierarchy descends.

**Example:** The following block outlines an example that handles all integer math exceptions specially:

```
try
  { statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

You can still specify specific handlers for more specific exceptions, but you need to place those handlers above the generic handler, because the application searches the handlers in the order they appear in, and executes the first applicable handler it finds. For example, this block provides special handling for range errors, and other handling for all other integer math errors:

```
try
  { statements performing integer math }
except
  on ERangeError do { out-of-range handling };
 on EIntError do { handling for other integer math errors };
end;
```

Note that if the handler for *EIntError* came before the handler for *ERangeError*, execution would never reach the specific handler for *ERangeError*.

## Reraising the exception

Sometimes when you handle an exception locally, you actually want to augment the handling in the enclosing block, rather than replacing it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond.

**Example:** When an exception occurs, you might want to display some sort of message to the user, then proceed with the standard handling. To do that, you declare a local exception handler that displays the message then calls the reserved word raise. This is called reraising the exception, as shown in this example:

```
try
  { statements }
  try
    { special statements }
  except
    on ESomething do
    begin
      { handling for only the special statements }
      raise; { reraise the exception }
    end;
  end;
except
  on ESomething do ...; { handling you want in all cases }
end;
```

If code in the { statements } part raises an *ESomething* exception, only the handler in the outer **except** part executes. However, if code in the { special statements } part raises *ESomething*, the handling in the inner **except** part is executed, followed by the more general handling in the outer **except** part.

By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

## Handling component exceptions

Delphi's components raise exceptions to indicate error conditions. Most component exceptions indicate programming errors that would otherwise generate a runtime error. The mechanics of handling component exceptions are no different than handling RTL exceptions.

**Example:** A common source of errors in components is range errors in indexed properties. For example, if a list box has three items in its list (0..2) and your application attempts to access item number 3, the list box raises a "List index out of bounds" exception.

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('a string');  { add a string to list box }
  ListBox1.Items.Add('another string'); { add another string... }
  ListBox1.Items.Add('still another string'); { ...and a third string }
  try
    Caption := ListBox1.Items[3];  { set form caption to fourth string in list
box }
  except
    on EStringListError do
      MessageDlg('List box contains fewer than four strings', mtWarning, [mbOK], 0);
  end;
end;
```

If you click the button once, the list box has only three strings, so accessing the fourth string (Items[3]) raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

## Using TApplication.HandleException

*HandleException* provides default handling of exceptions for the application. If an exception passes through all the **try** blocks in the application code, the application automatically calls the *HandleException* method, which displays a dialog box indicating that an error has occurred. You can use *HandleException* in this fashion:

```
try
  { statements }
except
  Application.HandleException(Self);
end;
```

For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. Therefore, if you want to both handle the exception, and provide this default behavior as the VCL does, you can add a call to *HandleException* to your code:

```
try
  { special statements }
except
  on ESomething do
  begin
    { handling for only the special statements }
    Application.HandleException(Self);  { call HandleException }
  end;
end;
```

**Note:** Do not call HandleException from within a thread's exception handling code.

For more information, search for exception handling routines in the Help index.

## Silent exceptions

Delphi applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message.

Silent exceptions are useful when you don't intend to handle an exception, but you want to abort an operation. Aborting an operation is similar to using the *Break* or *Exit* procedures to break out of a block, but can break out of several nested levels of blocks.

Silent exceptions all descend from the standard exception type *EAbort*. The default exception handler for Delphi VCL applications displays the error-message dialog box for all exceptions that reach it except those descended from *EAbort*.

**Note:** For console applications, an error-message dialog is displayed on any unhandled *EAbort* exceptions.

There is a shortcut for raising silent exceptions. Instead of manually constructing the object, you can call the *Abort* procedure. *Abort* automatically raises an *EAbort* exception, which will break out of the current operation without displaying an error message.

**Example:** The following code shows a simple example of aborting an operation. On a form containing an empty list box and a button, attach the following code to the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 10 do { loop ten times }
  begin
   ListBox1.Items.Add(IntToStr(I));{ add a numeral to the list }
   if I = 7 then Abort; { abort after the seventh one }
  end;
end;
```

## Defining your own exceptions

In addition to protecting your code from exceptions generated by the runtime library and various components, you can use the same mechanism to manage exceptional conditions in your own code.

To use exceptions in your code, you need to understand these steps:

- Declaring an exception object type
- Raising an exception

## Declaring an exception object type

Because exceptions are objects, defining a new kind of exception is as simple as declaring a new object type. Although you can raise any object instance as an exception, the standard exception handlers handle only exceptions descended from Exception.

It's therefore a good idea to derive any new exception types from *Exception* or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by a specific exception handler for that exception, one of the standard handlers will handle it instead.

**Example:** For example, consider the following declaration:

```
type
  EMyException = class(Exception);
```

If you raise *EMyException* but don't provide a specific handler for *EMyException*, a handler for *Exception* (or a default exception handler) will still handle it. Because the standard handling for *Exception* displays the name of the exception raised, you can see that it is your new exception that is raised.

## Raising an exception

To indicate an error condition in an application, you can raise an exception which involves constructing an instance of that type and calling the reserved word **raise**.

To raise an exception, call the reserved word raise, followed by an instance of an exception object. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

Setting the exception address is done through the *ErrorAddr* variable in the *System* unit. Raising an exception sets this variable to the address where the application raised the exception. You can refer to *ErrorAddr* in your exception handlers, for example, to notify the user of where the error occurred. You can also specify a value for *ErrorAddr* when you raise an exception.

**Warning:** Do not assign a value to *ErrorAddr* yourself.

To specify an error address for an exception, add the reserved word **at** after the exception instance, followed by an address expression such as an identifier.

For example, given the following declaration,

```
type
   EPasswordInvalid = class(Exception);
```

you can raise a "password invalid" exception at any time by calling raise with an instance of *EPasswordInvalid*, like this:

```
if Password <> CorrectPassword then
   raise EPasswordInvalid.Create('Incorrect password entered');
```

## Using interfaces

Delphi's **interface** keyword allows you to create and use interfaces in your application. Interfaces are a way extending the single-inheritance model of the VCL by allowing a single class to implement more than one interface, and by allowing several classes descended from different bases to share the same interface. Interfaces are useful when sets of operations, such as streaming, are used across a broad range of objects. Interfaces are also a fundamental aspect of the COM (the Component Object Model) and CORBA (Common Object Request Broker Architecture) distributed object models.

# Interfaces as a language feature

An interface is like a class that contains only abstract methods and a clear definition of their functionality. Strictly speaking, interface method definitions include the number and types of their parameters, their return type, and their expected behavior. Interface methods are semantically or logically related to indicate the purpose of the interface. It is the convention for interfaces to be named according to their behavior and to be prefaced with a capital I. For example, an *IMalloc* interface would allocate, free, and manage memory. Similarly, an *IPersist* interface could be used as a general base interface for descendants, each of which defines specific method prototypes for loading and saving the state of an object to a storage, stream, or file. A simple example of declaring an interface is:

```
type
IEdit = interface
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

Like abstract classes, interfaces themselves can never be instantiated. To use an interface, you need to obtain it from an implementing class.

To implement an interface, you must define a class that declares the interface in its ancestor list, indicating that it will implement all of the methods of that interface:

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy; stdcall;
  procedure Cut; stdcall;
  procedure Paste; stdcall;
  function Undo: Boolean; stdcall;
end;
```

While interfaces define the behavior and signature of their methods, they do not define the implementations. As long as the class's implementation conforms to the interface definition, the interface is fully polymorphic, meaning that accessing and using the interface is the same for any implementation of it.

## Sharing interfaces between classes

Using interfaces offers a design approach to separating the way a class is used from the way it is implemented. Two classes can share the same interface without requiring that they descend from the same base class. This polymorphic invocation of the same methods on unrelated objects is possible as long as the objects implement the same interface. For example, consider the interface,

```
IPaint = interface
  procedure Paint;
end;
```

and the two classes,

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;
TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Whether or not the two classes share a common ancestor, they are still assignment compatible with a variable of *IPaint* as in

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

This could have been accomplished by having *TCircle* and *TSquare* descend from say, *TFigure* which implemented a virtual method *Paint*. Both *TCircle* and *TSquare* would then have overridden the *Paint* method. The above *IPaint* would be replaced by *TFigure*. However, consider the following interface:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

which makes sense for the rectangle to support but not the circle. The classes would look like

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;
TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Later, you could create a class *TFilledCircle* that implements the *IRotate* interface to allow rotation of a pattern used to fill the circle without having to add rotation to the simple circle.

**Note:** For these examples, the immediate base class or an ancestor class is assumed to have implemented the methods of *IUnknown* that manage reference counting. For more information, see Implementing IUnknown and Memory management of interface objects.

## Using interfaces with procedures

Interfaces also allow you to write generic procedures that can handle objects without requiring the objects to descend from a particular base class. Using the above *IPaint* and *IRotate* interfaces you can write the following procedures,

```
procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;
procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

*RotateObjects* does not require that the objects know how to paint themselves and *PaintObjects* does not require the objects know how to rotate.   This allows the above objects to be used more often than if they where written to only work against a *TFigure* class.

Form details about the syntax, language definitions and rules for interfaces, see the *Object Pascal Language Guide* online Help section on Object interfaces.

## Implementing IUnknown

All interfaces derive either directly or indirectly from the *IUnknown* interface. This interface provides the essential functionality of an interface, that is, dynamic querying and lifetime management. This functionality is established in the three *IUnknown* methods:

- *QueryInterface* provides a method for dynamically querying a given object and obtaining interface references for the interfaces the object supports.
- *AddRef* is a reference counting method that increments the count each time the call to *QueryInterface* succeeds. While the reference count is nonzero the object must remain in memory.
- *Release* is used in conjunction with *AddRef* to enable an object to track its own lifetime and to determine when it is safe to delete itself. Once the reference count reaches zero the interface implementation releases the underlying object(s).

Every class that implements interfaces must implement the three *IUnknown* methods, as well as all of the methods declared by any other ancestor interfaces, and all of the methods declared by the interface itself. You can, however, inherit the implementations of methods of interfaces declared in your class.

By implementing these methods yourself, you can provide an alternative means of life-time management, disabling reference-counting. This is a powerful technique that lets you decouple interfaces from reference-counting.

# TInterfacedObject

The VCL defines a simple class, *TInterfacedObject*, that serves as a convenient base because it implements the methods of *IUnknown*. *TInterfacedObject* class is declared in the *System* unit as follows:

```
type
  TInterfacedObject = class(TObject, IUnknown)
  private
    FRefCount: Integer;
  protected
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    property RefCount: Integer read FRefCount;
  end;
```

Deriving directly from *TInterfacedObject* is straightforward. In the following example declaration, *TDerived* is a direct descendant of *TInterfacedObject* and implements a hypothetical *IPaint* interface.

```
type
  TDerived = class(TInterfacedObject, IPaint)
    ...
  end;
```

Because it implements the methods of *IUnknown*, *TInterfacedObject* automatically handles reference counting and memory management of interfaced objects. For more information, see Memory management of interface objects, which also discusses writing your own classes that implement interfaces but that do not follow the reference-counting mechanism inherent in *TInterfacedObject*.

## Using the as operator

Classes that implement interfaces can use the **as** operator for dynamic binding on the interface. In the following example:

```
procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;
begin
  X := P as IPaint;
{ statements }
end;
```

the variable P of type *TInterfacedObject*, can be assigned to the variable X, which is an *IPaint* interface reference. Dynamic binding makes this assignment possible. For this assignment, the compiler generates code to call the *QueryInterface* method of P's *IUnknown* interface since the compiler cannot tell from P's declared type whether P's instance actually supports *IPaint*. At runtime, P either resolves to an *IPaint* reference or an exception is raised. In either case, assigning P to X will not generate a compile-time error, as it would if P was of a class type that did not implement *IUnknown*.

When you use the **as** operator for dynamic binding on an interface, you should be aware of the following requirements:

▪        Explicitly declaring *IUnknown*: Although all interfaces derive from *IUnknown*, it is not sufficient, if you want to use the **as** operator, for a class to simply implement the methods of *IUnknown*. This is true even if it also implements the interfaces it explicitly declares. The class must explicitly declare *IUnknown* in its ancestor list.

▪        Using an IID: Interfaces can use an identifier that is based on a GUID (globally unique identifier). GUIDs that are used to identify interfaces are referred to as interface identifiers (IIDs). If you are using the **as** operator with an interface, it must have an associated IID. To create a new GUID in your source code you can use the *Ctrl+Shift+G* editor shortcut key.

# Reusing code and delegation

One approach to reusing code with interfaces is to have an object contain, or be contained by another. The VCL uses properties that are object types as an approach to containment and code reuse. To support this design for interfaces Delphi has a keyword **implements**, that makes if easy to write code to delegate all or part of the implementation of an interface to a sub-object. Aggregation is another way of reusing code through containment and delegation. In aggregation, an outer object contains an inner object that implements interfaces which are exposed only by the outer object. The VCL has classes that support aggregation.

## Using implements for delegation

Many classes in the VCL have properties that are sub-objects. You can also use interfaces as property types. When a property is of an interface type (or a class type that implements the methods of an interface) you can use the keyword **implements** to specify that the methods of that interface are delegated to the object or interface reference which is the property instance. The delegate only needs to provide implementation for the methods, it does not have to declare the interface support. The class containing the property must include the interface in its ancestor list. By default using the keyword **implements** delegates all interface methods. However, you can use methods resolution clauses or declare methods in your class that implement some of the interface methods as a way of overriding this default behavior.

The following example uses the implements keyword in the design of a color adapter object that converts an 8-bit RGB color value to a *Color* reference:

```
unit cadapt;

type
IRGB8bit = interface
    ['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;
IColorRef = interface
    ['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;
{ TRGB8ColorRefAdapter   map an IRGB8bit to an IColorRef }
  TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
  end;
implementation

constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit := rgb;
end;

function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
end;
end.
```

For more information about the syntax, implementation details, and language rules of the **implements** keyword, see the *Object Pascal Language Guide* online Help section on Object interfaces.

## Aggregation

Aggregation offers a modular approach to code reuse through sub-objects that define the functionality of a containing object, but that hide the implementation details from that object. In aggregation, an outer object implements one or more interfaces. The only requirement is that it implement *IUnknown*. The inner object, or objects, can implement one or more interfaces, however only the outer object exposes the interfaces. These include both the interfaces it implements and the ones implemented by its contained objects. Clients know nothing about inner objects. While the outer object provides access to the inner object interfaces, their implementation is completely transparent. Therefore, the outer object class can exchange the inner object class type for any class that implements the same interface. Correspondingly, the code for the inner object classes can be shared by other classes that want to use it.

The implementation model for aggregation defines explicit rules for implementing *IUnknown* using delegation. The inner object must implement an *IUnknown* on itself, that controls the inner object's reference count. This implementation of *IUnknown* tracks the relationship between the outer and the inner object. For example, when an object of its type (the inner object) is created, the creation succeeds only for a requested interface of type *IUnknown*. The inner object also implements a second *IUnknown* for all the interfaces it implements. These are the interfaces exposed by the outer object. This second *IUnknown* delegates calls to *QueryInterface*, *AddRef*, and *Release* to the outer object. The outer *IUnknown* is referred to as the "controlling Unknown."

Refer to the MS online help for the rules about creating an aggregation. When writing your own aggregation classes, you can also refer to the implementation details of *IUnknown* in *TComObject*. *TComObject* is a COM class that supports aggregation. If you are writing COM applications, you can also use *TComObject* directly as a base class.

# Memory management of interface objects

One of the concepts behind the design of interfaces is ensuring the lifetime management of the objects that implement them. The *AddRef* and *Release* methods of *IUnknown* provide a way of implementing this functionality. Their defined behavior states that they will track the lifetime of an object by incrementing the reference count on the object when an interface reference is passed to a client, and will destroy the object when that reference count is zero.

If you are creating COM objects for distributed applications, then you should strictly adhere to the reference counting rules. However, if you are using interfaces only internally in your application, then you have a choice that depends upon the nature of your object and how you decide to use it.

## Using reference counting

Delphi provides most of the *IUnknown* memory management for you by its implementation of interface querying and reference counting. Therefore, if you have an object that lives and dies by its interfaces, you can easily use reference counting by deriving from these classes. *TInterfacedObject* is the non-CoClass that provides this behavior. If you decide to use reference counting, then you must be careful to only hold the object as an interface reference, and to be consistent in your reference counting. For example:

```
procedure beep(x: ITest);
function test_func()
var
  y: ITest;
begin
  y := TTest.Create; // because y is of type ITest, the reference count is one
beep(y); // the act of calling the beep function increments the reference count
// and then decrements it when it returns
  y.something; // object is still here with a reference count of one
end;
```

This is the cleanest and safest approach to memory management; and if you use *TInterfacedObject* it is handled automatically. If you do not follow this rule, your object can unexpectedly disappear, as demonstrated in the following code:

```
function test_func()
var
  x: TTest;
begin
  x := TTest.Create; // no count on the object yet
  beep(x as ITest); // count is incremented by the act of calling beep
// and decremented when it returns
  x.something; // surprise, the object is gone
end;
```

**Note:** In the examples above, the *beep* procedure, as it is declared, will increment the reference count (call *AddRef*) on the parameter, whereas either of the following declarations:

```
procedure beep(const x: ITest);
```

or

```
procedure beep(var x: ITest);
```

will not. These declarations generate smaller, faster code.

One case where you cannot use reference counting, because it cannot be consistently applied, is if your object is a component or a control owned by another component. In that case, you can still use interfaces, but you should not use reference counting because the lifetime of the object is not dictated by its interfaces.

## Not using reference counting

If your object is a VCL component or a control that is owned by another component, then your object is part of a different memory management system that is based in *TComponent*. You should not mix the object lifetime management approaches of VCL components and COM reference counting. If you want to create a component that supports interfaces, you can implement the *IUnknown AddRef* and *Release* methods as empty functions to bypass the COM reference counting mechanism:

```
function TMyObject.AddRef: Integer;
begin
Result := -1;
end;
function TMyObject.Release: Integer;
begin
Result := -1;
end;
```

You would still implement *QueryInterface* as usual to provide dynamic querying on your object.

Note that, because you do implement *QueryInterface*, you can still use the **as** operator for interfaces on components, as long as you create an interface identifier (IID). You can also use aggregation. If the outer object is a component, the inner object implements reference counting as usual, by delegating to the "controlling Unknown." It is at the level of the outer, component object that the decision is made to circumvent the *AddRef* and *Release* methods, and to handle memory management via the VCL approach. In fact, you can use *TInterfacedObject* as a base class for an inner object of an aggregation that has a component as its containing outer object.

**Note:** The "controlling Unknown" is the *IUnknown* implemented by the outer object and the one for which the reference count of the entire object is maintained. For more information distinguishing the various implementations of the *IUnknown* interface by the inner and outer objects, see Aggregation and the Microsoft online Help topics on the "controlling Unknown."

## Using interfaces in distributed applications

Interfaces are a fundamental element in the COM and CORBA distributed object models. Delphi provides base classes for these technologies that extend the basic interface functionality in *TInterfacedObject*, which simply implements the *IUnknown* interface methods.

COM classes add functionality for using class factories and class identifiers (CLSIDs). Class factories are responsible for creating class instances via CLSIDs. The CLSIDs are used to register and manipulate COM classes. COM classes that have class factories and class identifiers are called CoClasses. CoClasses take advantage of the versioning capabilities of *QueryInterface*, so that when a software module is updated *QueryInterface* can be invoked at runtime to query the current capabilities of an object.

New versions of old interfaces, as well as any new interfaces or features of an object, can become immediately available to new clients. At the same time, objects retain complete compatibility with existing client code; no recompilation is necessary because interface implementations are hidden (while the methods and parameters remain constant). In COM applications, developers can change the implementation to improve performance, or for any internal reason, without breaking any client code that relies on that interface. For more information about COM interfaces, see Overview of COM technologies.

The other distributed application technology is CORBA. The use of interfaces in CORBA applications is mediated by stub classes on the client and skeleton classes on the server. These stub and skeleton classes handle the details of marshaling interface calls so that parameter values and return values can be transmitted correctly. Applications must use either a stub or skeleton class, or employ the Dynamic Invocation Interface (DII) which converts all parameters to special variants (so that they carry their own type information.)

Although it is not a necessary feature of CORBA technology, Delphi implements CORBA using class factories, similar to the way in which COM uses class factories and CoClasses. By unifying the two distributed model architectures in this way, Delphi supports a combined COM/CORBA server that can service both COM and CORBA clients simultaneously. For more information about using interfaces with CORBA, see Writing CORBA applications.

# Working with strings

Delphi has a number of different character and string types that have been introduced throughout the development of the Object Pascal language. This section of the Help is an overview of these types, their purpose, and usage. For language details, see String types..

# Character types

Delphi has three character types: *Char*, *AnsiChar*, and *WideChar*.

The *Char* character type came from standard Pascal, and was used in Turbo Pascal and then in Object Pascal. Later Object Pascal added *AnsiChar* and *WideChar* as specific character types that were used to support standards for character representation on the Windows operating system. *AnsiChar* was introduced to support an 8-bit character ANSI standard, and *WideChar* was introduced to support a 16-bit Unicode standard. The name *WideChar* is used because Unicode characters are also known as wide characters. Wide characters are two bytes instead of one, so that the character set can represent many more different characters. When *AnsiChar* and *WideChar* were implemented, *Char* became the default character type representing the currently recommended implementation. If you use *Char* in your application, remember that its implementation is subject to change in future versions of Delphi.

The following table summarizes these character types:

| Type | Bytes | Contents | Purpose |
| --- | --- | --- | --- |
| Char | 1 | Holds a single ANSI character. | Default character type. |
| AnsiChar | 1 | Holds a single ANSI character. | 8-bit Ansi character standard on Windows. |
| WideChar | 2 | Holds a single Unicode character. | 16-bit Unicode standard on Windows. |

For more information about using these character types, see Character types. For more information about Unicode characters, see About extended character sets.

# String types

Delphi has three categories of types that you can use when working with strings. These are character pointers, string types, and VCL string classes. This topic summarizes string types, and discusses using them with character pointers. For information about using VCL string classes, see TStrings.

There are currently three string implementations in Delphi: short strings, long strings, and wide strings. There are several different string types that represent these implementations. In addition, there is a reserved word **string** that defaults to the currently recommended string implementation.

## Short strings

**String** was the first string type used in Turbo Pascal. **String** was originally implemented as a short string. Short strings are an allocation of between 1 and 256 bytes, of which the first byte contains the length of the string and the remaining bytes contain the characters in the string:

```
S: string[0..n]// the original string type
```

When long strings were implemented, **string** was changed to a long string implementation by default and *ShortString* was introduced as a backward compatibility type. *ShortString* is a predefined type for a maximum length string:

```
S: string[255] // the ShortString type
```

The size of the memory allocated for a *ShortString* is static, meaning that it is determined at compile time. However, the location of the memory for the *ShortString* can be dynamically allocated, for example if you use a *PShortString*, which is a pointer to a *ShortString*. The number of bytes of storage occupied by a short string type variable is the maximum length of the short string type plus one. For the *ShortString* predefined type the size is 256 bytes.

Both short strings, declared using the syntax **string**[0..n], and the *ShortString* predefined type exist primarily for backward compatibility with earlier versions of Delphi and Borland Pascal.

A compiler directive, $H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {$H-} directive. The {$H-} state is mostly useful for using code from versions of Object Pascal that used short strings by default. However, short strings can be useful in data structures where you need a fixed-size component or in DLLs when you don't want to use the *ShareMem* unit (see also Memory Management). You can locally override the meaning of string-type definitions to ensure generation of short strings. You can also change declarations of short string types to string[255] or *ShortString*, which are unambiguous and independent of the $H setting.

For details about short strings and the *ShortString* type, see Short strings..

## Long strings

Long strings are dynamically-allocated strings with a maximum length limited only by available memory. Like short strings, long strings use 8-bit Ansi characters and have a length indicator. Unlike short strings, long strings have no zeroth element that contains the dynamic string length. To find the length of a long string you must use the *Length* standard function, and to set the length of a long string you must use the *SetLength* standard procedure. Long strings are also reference-counted and, like *PChars*, long strings are null-terminated. For details about the implementation of longs strings, see Long strings.

Long strings are denoted by the reserved word **string** and by the predefined identifier *AnsiString*. For new applications, it is recommended that you use the long string type. All components in the Visual Component Library are compiled in this state, typically using **string**. If you write components, they should also use long strings, as should any code that receives data from VCL string-type properties. If you want to write specific code that always uses a long string, then you should use *AnsiString*. If you want to write flexible code that allows you to easily change the type as new string implementations become standard, then you should use **string**.

# WideString

The *WideChar* type allows wide character strings to be represented as arrays of *WideChars*. Wide strings are strings composed of 16-bit Unicode characters. As with long strings, WideStrings are dynamically allocated with a maximum length limited only by available memory. However, wide strings are not reference counted. The dynamically allocated memory that contains the string is deallocated when the wide string goes out of scope. In all other respects wide strings possess the same attributes as long strings. The *WideString* type is denoted by the predefined identifier *WideString*.

Since the 32-bit version of OLE uses Unicode for all strings, strings must be of wide string type in any OLE automated properties and method parameters. Also, most OLE API functions use null-terminated wide strings.

For more information about WideStrings, see <u>WideString</u>.

## PChar types

A *PChar* is a pointer to a null-terminated string of characters of the type *Char*. Each of the three character types also has a built-in pointer type:

- A *PChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PAnsiChar* is a pointer to a null-terminated string of 8-bit characters.
- A *PWideChar* is a pointer to a null-terminated string of 16-bit characters.

*PChars* are, with short strings, one of the original Object Pascal string types. They were created primarily as a C language and Windows API compatibility type.

## OpenString

An *OpenString* is obsolete, but you may see it in older code. It is for 16-bit compatibility and is allowed only in parameters. *OpenString* was used, before long strings were implemented, to allow a short string of an unspecified length string to be passed as a parameter. For example, this declaration:

```
procedure a(v : openstring);
```

will allow any length string to be passed as a parameter, where normally the string length of the formal and actual parameters must match exactly. You should not have to use *OpenString* in any new applications you write.

Refer also to the {$P+/-} switch in Compiler directives for strings.

## Runtime library string handling routines

The runtime library provides many specialized string handling routines specific to a string type. These are routines for wide strings, longs strings, and null-terminated strings (meaning *PChars*). Routines that deal with *PChar* types use the null-termination to determine the length of the string. For more details about null-terminated strings, see Working with null-terminated strings.

The runtime library also includes a category of string formatting routines. There are no categories of routines listed for *ShortString* types. However, some built-in compiler routines deal with the *ShortString* type. These include, for example, the *Low* and *High* standard functions.

Because wide strings and long strings are the commonly used types, the remaining sections discuss these routines.

# Wide character routines

When working with strings you should make sure that the code in your application can handle the strings it will encounter in the various target locales. Sometimes you will need to use wide characters and wide strings. In fact, one approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. The runtime library includes the following wide character string functions for converting between standard single-byte character strings (or MBCS strings) and Unicode strings:

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second half of a character for the start of a different character.

A disadvantage of working with wide characters is that Windows 95 does not support wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require tremendous amounts of extra code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

# Commonly used long string routines

The long string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether or not they use a particular criteria in their calculations. The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string

Where appropriate, the tables also provide columns indicating whether or not a routine satisfies the following criteria.

- Uses case sensitivity: If the Windows locale is used, it determines the definition of case. If the routine does not use the Windows locale, analysis are based upon the ordinal values of the characters. If the routine is case-insensitive, there is a logical merging of upper and lower case characters that is determined by a predefined pattern.
- Uses the Windows locale: Windows locale enablement allows you to add extra features to your application for specific locales. In particular, for Asian language environments. Most Windows locales consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters. Routines that use the Windows locale are typically prefaced with Ansi (that is, AnsiXXX).
- Supports the multi-byte character set (MBCS): MBCSs are used when writing code for far eastern locales. Multi-byte characters are represented as a mix of one and two byte character codes, so the length in bytes does not necessarily correspond to the length of the string. The routines that support MBCS are written parse one- and two-byte characters. The *ByteType* and *StrByteType* determine whether a particular byte is the lead byte of a two-byte character. Be careful when using multi-byte characters not to truncate a string by cutting a two-byte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character cannot be predetermined. Pass, instead, a pointer to a to a character or string. For more information about MBCS, see Enabling application code.

TABLE comparison

| Routine | Case-sensitive | Uses Windows locale | Supports MBCS |
|---|---|---|---|
| AnsiCompareStr | yes | yes | yes |
| AnsiCompareText | no | yes | yes |
| AnsiCompareFileName | no | yes | yes |
| CompareStr | yes | no | no |
| CompareText | no | no | no |

TABLE Case conversion

| Routine | Uses Windows locale | Supports MBCS |
|---|---|---|
| AnsiLowerCase | yes | yes |
| AnsiLowerCaseFileName | yes | yes |
| AnsiUpperCaseFileName | yes | yes |
| AnsiUpperCase | yes | yes |
| LowerCase | no | no |
| UpperCase | no | no |

TABLE Modification

| Routine | Case-sensitive | Supports MBCS |
|---|---|---|
| AdjustLineBreaks | NA | yes |
| AnsiQuotedStr | NA | yes |
| StringReplace | optional by flag | yes |

| | | |
|---|---|---|
| Trim | NA | yes |
| TrimLeft | NA | yes |
| TrimRight | NA | yes |
| WrapText | NA | yes |

TABLE Sub-string

| Routine | Case-sensitive | Supports MBCS |
|---|---|---|
| AnsiExtractQuotedStr | NA | yes |
| AnsiPos | yes | yes |
| IsDelimiter | yes | yes |
| IsPathDelimiter | yes | yes |
| LastDelimiter | yes | yes |
| QuotedStr | no | no |

The routines used for string filenames: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, and *AnsiUpperCaseFileName* all use the Windows locale. You should always use filenames that are perfectly portable because the locale (character set) used for filenames can and might differ from the default user interface.

## Declaring and initializing strings

When you declare a long string:

```
S: string;
```

you do not need to initialize it. Long strings are automatically initialized to empty. To test a string for empty you can either use the *EmptyStr* variable:

```
S = EmptyStr;
```

or test against an empty string:

```
S = '';
```

An empty string has no valid data. Therefore, trying to index an empty string is like trying to access **nil** and will result in an access violation:

```
var
  S: string;
begin
  S[i];  // this will cause an access violation
  // statements
end;
```

Similarly, if you cast an empty string to a *PChar*, the result is a **nil** pointer. So, if you are passing such a *PChar* to a routine that needs to read or write to it, be sure that the routine can handle **nil**:

```
var
  S: string; // empty string
begin
  proc(PChar(S)); // be sure that proc can handle nil
  // statements
end;
```

If it cannot, then you can either initialize the string:

```
S := 'No longer nil';
proc(PChar(S)); // proc does not need to handle nil now
```

or set the length, using the *SetLength* procedure:

```
SetLength(S, 100);  //sets the dynamic length of S to 100
proc(PChar(S)); // proc does not need to handle nil now
```

When you use *SetLength*, existing characters in the string are preserved, but the contents of any newly allocated space is undefined. Following a call to *SetLength*, S is guaranteed to reference a unique string, that is a string with a reference count of one. To obtain the length of a string, use the *Length* function.

Remember when declaring a **string** that:

```
S: string[n];
```

implicitly declares a short string, not a long string of *n* length. To declare a long string of specifically *n* length, declare a variable of type **string** and use the *SetLength* procedure.

```
S: string;
SetLength(S, n);
```

# Mixing and converting string types

Short strings, long strings and wide strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions. However, when assigning a string value to a short string variable, be aware that the string value is truncated if it is longer than the declared maximum length of the short string variable.

Long strings are already dynamically allocated. If you use one of the built-in pointer types, such as *PAnsiString*, *PString*, or *PWideString*, remember that you are introducing another level of indirection. Be sure this is what you intend.

## String to PChar conversions

Long string to *PChar* conversions are not automatic. Some of the differences between strings and *PChars* can make conversions problematic:

- Long strings are reference-counted, while *PChars* are not.
- Assigning to a string copies the data, while a *PChar* is a pointer to memory.
- Long strings are null-terminated and also contain the length of the string, while *PChars* are simply null-terminated.

Situations in which these differences can cause subtle errors are discussed in this section.

## String dependencies

Sometimes you will need convert a long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. However, because long strings are reference counted, typecasting a string to a *PChar* increases the dependency on the string by one, without actually incrementing the reference count. When the reference count hits zero, the string will be destroyed, even though there is an extra dependency on it. The cast *PChar* will also disappear, while the routine you passed it to may still be using it. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. For example:

```
procedure my_func(x: string);
begin
  // do something with x
  some_proc(PChar(x)); // cast the string to a PChar
  // you now need to guarantee that the string remains
  // as long as the some_proc procedure needs to use it
end;
```

## Returning a PChar local variable

A common error when working with *PChars* is to store in a data structure, or return as a value, a local variable. When your routine ends, the *PChar* will disappear because it is simply a pointer to memory, and is not a reference counted copy of the string. For example:

```
function title(n: Integer): PChar;
var
  s: string;
begin
  s := Format('title - %d', [n]);
  Result := PChar(s); // DON'T DO THIS
end;
```

This example returns a pointer to string data that is freed when the *title* function returns.

## Passing a local variable as a PChar

Consider that you have a local string variable that you need to initialize by calling a function that takes a *PChar*. One approach is to create a local **array of char** and pass it to the function, then assign that variable to the string:

```
// assume MAXSIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := GetModuleFilename(0, @buf, SizeOf(buf));  // treats @buf as a PChar
  S := buf;
  //statements
end;
```

This approach is useful if the size of the buffer is relatively small, since it is allocated on the stack. It is also safe, since the conversion between an **array of char** and a **string** is automatic. When *GetModuleFilename* returns, the *Length* of the string correctly indicates the number of bytes written to *buf*.

To eliminate the overhead of copying the buffer, you can cast the string to a *PChar* (if you are certain that the routine does not need the *PChar* to remain in memory). However, synchronizing the length of the string does not happen automatically, as it does when you assign an **array of char** to a **string**. You should reset the string *Length* so that it reflects the actual width of the string. If you are using a function that returns the number of bytes copied, you can do this safely with one line of code:

```
var
  S: string;
begin
  SetLength(S, 100);  // when casting to a PChar, be sure the string is not empty
  SetLength(S, GetModuleFilename( 0, PChar(S), Length(S) ) );
  // statements
end;
```

## Compiler directives for strings

The following compiler directives affect character and string types.

▪ {$H+/-}: A compiler directive, $H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {$H-} directive.

▪ {$P+/-}: The $P directive is meaningful only for code compiled in the {$H-} state, and is provided for backwards compatibility with earlier versions of Delphi and Borland Pascal. $P controls the meaning of variable parameters declared using the string keyword in the {$H-} state. In the {$P-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {$P+} state, they are open string parameters. Regardless of the setting of the $P directive, the OpenString identifier can always be used to declare open string parameters. Make a link to the compiler directives, since this is a direct quote.

▪ {$V+/-}: The $V directive controls type checking on short strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types. In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Be aware that this could lead to memory corruption. For example:

```
var S: string[3];
procedure Test(var T: string);
begin
  T := '1234';
end;
begin
  Test(S);
end.
```

▪ {$X+/-}: The {$X+} compiler directive enables Delphi's support for null-terminated strings by activating the special rules that apply to the built-in *PChar* type and zero-based character arrays. (These rules allow zero-based arrays and character pointers to be used with *Write*, *Writeln*, *Val*, *Assign*, and *Rename* from the *System* unit.)

## Strings and characters: related topics

The following *Object Pascal Language Guide* topics discuss strings and character sets. Also see Creating international applications.

- "About extended character sets" (Discusses international character sets.)
- "Working with null-terminated strings" (Contains information about character arrays.)
- "Character strings"
- "Character pointers"
- "String operators."

# Working with files

This section describes working with files and distinguishes between manipulating files on disk, and input/output operations such as reading and writing to files. The first topic discusses the runtime library and Windows API routines you would use for common programming tasks that involve manipulating files on disk. The next topic is an overview of file types used with file I/O. The last topic focuses on the recommended approach to working with file I/O, which is to use file streams.

**Note:** Previous versions of the Object Pascal language performed operations on files themselves, rather than on the filename parameters commonly used now. With these older file types you had to locate a file and assign it to a file variable before you could, for example, rename the file.

## Manipulating files

There are several common file operations built into Object Pascal's runtime library. The procedures and functions for working with files operate at a high level. For most routines, you specify the name of the file and the routine makes the necessary calls to the operating system for you. In some cases, you use file handles instead. Object Pascal provides routines for most file manipulation. When it does not, alternative routines are discussed.

## Deleting a file

Deleting a file erases the file from the disk and removes the entry from the disk's directory. There is no corresponding operation to restore a deleted file, so applications should generally allow users to confirm deletions of files. To delete a file, pass the name of the file to the *DeleteFile* function:

```
DeleteFile(FileName);
```

*DeleteFile* returns *True* if it deleted the file and *False* if it did not (for example, if the file did not exist or if it was read-only). *DeleteFile* erases the file named by *FileName* from the disk.

## Finding a file

There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a filename with a given set of attributes in a specified directory. *FindNext* returns the next entry matching the name and attributes specified in a previous call to *FindFirst*. *FindClose* releases memory allocated by *FindFirst*. In 32-bit Windows you should always use *FindClose* to terminates a *FindFirst/FindNext* sequence. If you want to know if a file exists, there is a *FileExists* function that returns *True* if the file exists, *False* otherwise.

The three file find routines take a *TSearchRec* as one of the parameters. *TSearchRec* defines the file information searched for by *FindFirst* or *FindNext*. The declaration for *TSearchRec* is:

```
type
TFileName = string;
TSearchRec = record
   Time: Integer;  //Time contains the time stamp of the file.
   Size: Integer;  //Size contains the size of the file in bytes.
   Attr: Integer;  //Attr represents the file attributes of the file.
   Name: TFileName;  //Name contains the DOS filename and extension.
   ExcludeAttr: Integer;
   FindHandle: THandle;
   FindData: TWin32FindData;  //FindData contains additional information such as
 //file creation time, last access time, long and short filenames.
end;
```

If a file is found, the fields of the *TSearchRec* type parameter are modified to specify the found file. You can test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

| Constant | Value | Description |
| --- | --- | --- |
| faReadOnly | $00000001 | Read-only files |
| faHidden | $00000002 | Hidden files |
| faSysFile | $00000004 | System files |
| faVolumeID | $00000008 | Volume ID files |
| faDirectory | $00000010 | Directory files |
| faArchive | $00000020 | Archive files |
| faAnyFile | $0000003F | Any file |

To test for an attribute, combine the value of the *Attr* field with the attribute constant with the **and** operator. If the file has that attribute, the result will be greater than 0. For example, if the found file is a hidden file, the following expression will evaluate to *True*: (*SearchRec.Attr* and *faHidden* > 0). Attributes can be combined by adding their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass (*faReadOnly* + *faHidden*) the *Attr* parameter.

**Example:** This example uses a label, a button named *Search*, and a button named *Again* on a form. When the user clicks the *Search* button, the first file in the specified path is found, and the name and the number of bytes in the file appear in the label's caption. Each time the user clicks the *Again* button, the next matching filename and size is displayed in the label:

```
var
   SearchRec: TSearchRec;
procedure TForm1.SearchClick(Sender: TObject);
begin
   FindFirst('c:\Program Files\delphi5\bin\*.*', faAnyFile, SearchRec);
   Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in
size';
end;
procedure TForm1.AgainClick(Sender: TObject);
begin
   if (FindNext(SearchRec) = 0)
```

```
        Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + '
bytes in size';
  else
    FindClose(SearchRec);
end;
```

## Changing file attributes

Every file has various attributes stored by the operating system as bitmapped flags. File attributes include such items as whether a file is read-only or a hidden file. Changing a file's attributes requires three steps: reading, changing, and setting.

**Reading file attributes:** Operating systems store file attributes in various ways, generally as bitmapped flags. To read a file's attributes, pass the filename to the *FileGetAttr* function, which returns the file attributes of a file. The return value is a group of bitmapped file attributes, of type *Word*. The attributes can be examined by AND-ing the attributes with the constants defined in *TSearchRec*. A return value of -1 indicates that an error occurred.

**Changing individual file attributes:** Because Delphi represents file attributes in a set, you can use normal logical operators to manipulate the individual attributes. Each attribute has a mnemonic name defined in the *SysUtils* unit. For example, to set a file's read-only attribute, you would do the following:

```
Attributes := Attributes or faReadOnly;
```

You can also set or clear several attributes at once. For example, the clear both the system-file and hidden attributes:

```
Attributes := Attributes and not (faSysFile or faHidden);
```

**Setting file attributes:** Delphi enables you to set the attributes for any file at any time. To set a file's attributes, pass the name of the file and the attributes you want to the *FileSetAttr* function. *FileSetAttr* sets the file attributes of a specified file.

You can use the reading and setting operations independently, if you only want to determine a file's attributes, or if you want to set an attribute regardless of previous settings. To change attributes based on their previous settings, however, you need to read the existing attributes, modify them, and write the modified attributes.

## Renaming a file

To change a filename, simply use the *RenameFile* function:

```
function RenameFile(const OldFileName, NewFileName: string): Boolean;
```

which changes a filename, identified by *OldFileName*, to the name specified by *NewFileName*. If the operation succeeds, *RenameFile* returns *True*. If it cannot rename the file, for example, if a file called *NewFileName* already exists, it returns *False*. For example:

```
if not RenameFile('OLDNAME.TXT','NEWNAME.TXT') then
   ErrorMsg('Error renaming file!');
```

You cannot rename (move) a file across drives using *RenameFile*. You would need to first copy the file and then delete the old one.

**Note:** *RenameFile* is a wrapper around the Windows API *MoveFile* function, so *MoveFile* will not work across drives either.

## File date-time routines

The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on operating system date-time values. *FileAge* returns the date-and-time stamp of a file, or -1 if the file does not exist. *FileSetDate* sets the date-and-time stamp for a specified file, and returns zero on success or a Windows error code on failure. *FileGetDate* returns a date-and-time stamp for the specified file or -1 if the handle is invalid.

As with most of the file manipulating routines, *FileAge* uses a string filename. *FileGetDate* and *FileSetDate*, however, take a Windows *Handle* type as a parameter. To get access to a Windows file *Handle* either

▪ Call the Windows API *CreateFile* function. *CreateFile* is a 32-bit only function that creates or opens a file and returns a *Handle* that can be used to access the file.
▪ Instantiate *TFileStream* to create or open a file. Then use the *Handle* property as you would a Windows' file *Handle*. See Using file streams for more information.

## Copying a file

The runtime library does not provide any routines for copying a file. However, you can directly call the Windows API *CopyFile* function to copy a file. Like most of the Delphi runtime library file routines, *CopyFile* takes a filename as a parameter, not a Window *Handle*. When copying a file, be aware that the file attributes for the existing file are copied to the new file, but the security attributes are not. *CopyFile* is also useful when moving files across drives because neither the Delphi *RenameFile* function nor the Windows API *MoveFile* function can rename/move files across drives.

# File types with file I/O

There are three file types you can use when working with file I/O: Old style Pascal files, Windows file handles, and file stream objects. This section describes these types.

**Old style Pascal files:** These are the types used with the old file variables, usually of the format "F: Text: or "F: File". There are three classes of these files: typed, text, and untyped and a number of Delphi file-handling routines, such as *AssignPrn* and *writeln,* use them. These file types are obsolete and are incompatible with Windows file handles. If you need to work with the old file types, see Untyped files and File types.

**Windows file handles:** The Object Pascal file handles are wrappers for the Windows file handle type. The runtime library file-handling routines that use Windows file *Handles* are typically wrappers around Windows API functions. For example, the *FileRead* calls the Windows *ReadFile* function. Because the Delphi functions use Object Pascal syntax, and occasionally provide default parameter values, they are a convenient interface to the Windows API. Using these routines is straightforward, and if you are familiar and comfortable with the Windows API file routines, you may want to use them when working with file I/O.

**File streams:** File streams are object instances of the VCL *TFileStream* class used to access the information in disk files. File streams are a portable and high level approach to file I/O. *TFileStream* has a *Handle* property that gives you access to the Windows file handle. The Using file streams discusses *TFileStream*.

## Using file streams

*TFileStream* is a VCL class used for high level object representations of file streams. *TFileStream* offers multiple functionality: persistence, interaction with other streams, and file I/O.

▪ *TFileStream* is a descendant of the stream classes. As such, one advantage of using file streams is that you automatically inherit support for persistence. The stream classes are enabled to work with the *TFiler* classes, *TReader* and *TWriter*, to stream objects out to disk. Therefore, when you have a file stream, you can use that same code for the VCL streaming mechanism. For more information about using the VCL streaming system, see the *VCL Reference* online Help on the *TStream*, *TFiler*, *TReader*, *TWriter*, and *TComponent* classes.

▪ *TFileStream* can interact easily with other stream classes. For example, if you want to dump a dynamic memory block to disk, you can do so using a *TFileStream* and a *TMemoryStream*.

▪ *TFileStream* provides the basic methods and properties for file I/O. The following topics focus on this aspect of file streams:

> Creating and opening files
> Using the file handle
> Reading and writing to files
> Reading and writing strings
> File position and size
> Seeking a file
> Copying

## Creating and opening files

To create or open a file and get access to a handle for the file, you simply instantiate a *TFileStream*. This opens or creates a named file and provides methods to read from or write to it. If the file can not be opened, *TFileStream* raises an exception.

```
constructor Create(const filename: string; Mode: Word);
```

The *Mode* parameter specifies how the file should be opened when creating the file stream. The *Mode* parameter consists of an open mode and a share mode or'ed together. The open mode must be one of the following values:

| Value | Meaning |
| --- | --- |
| fmCreate | TFileStream a file with the given name. If a file with the given name exists, open the file in write mode. |
| fmOpenRead | Open the file for reading only. |
| fmOpenWrite | Open the file for writing only. Writing to the file completely replaces the current contents. |
| fmOpenReadWrite | Open the file to modify the current contents rather than replace them. |

The share mode must be one of the following values:

| Value | Meaning |
| --- | --- |
| fmShareCompat | Sharing is compatible with the way FCBs are opened. |
| fmShareExclusive | Other applications can not open the file for any reason. |
| fmShareDenyWrite | Other applications can open the file for reading but not for writing. |
| fmShareDenyRead | Other applications can open the file for writing but not for reading. |
| fmShareDenyNone | No attempt is made to prevent other applications from reading from or writing to the file. |

The file open and share mode constants are in the *SysUtils* unit.

# Using the file handle

When you instantiate *TFileStream* you get access to the file handle. The file handle is contained in the *Handle* property. *Handle* is read-only and indicates the mode in which the file was opened. If you want to change the attributes of the file *Handle*, you must create a new file stream object.

Some file manipulation routines take a Window's file handle as a parameter. Once you have a file stream, you can use the *Handle* property in any situation in which you would use a Window's file handle. Be aware that, unlike handle streams, file streams close file handles when the object is destroyed.

## Reading and writing to files

*TFileStream* has several different methods for reading from and writing to files. These are distinguished by whether they perform the following:

- Return the number of bytes read or written.
- Require the number of bytes is known.
- Raise an exception on error.

*Read* is a function that reads up to *Count* bytes from the file associated with the file stream, starting at the current *Position*, into *Buffer*. *Read* then advances the current position in the file by the number of bytes actually transferred. The prototype for *Read* is

```
function Read(var Buffer; Count: Longint): Longint; override;
```

*Read* is useful when the number of bytes in the file is not known. *Read* returns the number of bytes actually transferred, which may be less than *Count* if the end of file marker is encountered.

*Write* is a function that writes *Count* bytes from the *Buffer* to the file associated with the file stream, starting at the current *Position*. The prototype for *Write* is:

```
function Write(const Buffer; Count: Longint): Longint; override;
```

After writing to the file, *Write* advances the current position by the number bytes written, and returns the number of bytes actually written, which may be less than *Count* if the end of the buffer is encountered.

The counterpart procedures are *ReadBuffer* and *WriteBuffer* which, unlike *Read* and *Write*, do not return the number of bytes read or written. These procedures are useful in cases where the number of bytes is known and required, for example when reading in structures. *ReadBuffer* and *WriteBuffer* raise an exception on error (*EReadError* and *EWriteError*) while the *Read* and *Write* methods do not. The prototypes for *ReadBuffer* and *WriteBuffer* are:

```
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
```

These methods call the *Read* and *Write* methods, to perform the actual reading and writing.

## Reading and writing strings

If you are passing a string to a read or write function, you need to be aware of the correct syntax. The *Buffer* parameters for the read and write routines are **var** and **const** types, respectively. These are untyped parameters, so the routine takes the address of a variable.

The most commonly used type when working with strings is a long string. However, passing a long string as the *Buffer* parameter does not produce the correct result. Long strings contain a size, a reference count, and a pointer to the characters in the string. Consequently, dereferencing a long string does not result in only the pointer element. What you need to do is first cast the string to a *Pointer* or *PChar,* and then dereference it. For example:

```
procedure cast-string;
var
  fs: TFileStream;
  s: string = 'Hello';
begin
  fs := TFileStream.Create('Temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s)); // this will give you garbage
  fs.Write(PChar(s)^, Length(s));  // this is the correct way
end;
```

## Seeking a file

Most typical file I/O mechanisms have a process of seeking a file in order to read from or write to a particular location within it. For this purpose, *TFileStream* provides a *Seek* method. The prototype for *Seek* is:

```
function Seek(Offset: Longint; Origin: Word): Longint; override;
```

The *Origin* parameter indicates how to interpret the *Offset* parameter. *Origin* should be one of the following values:

| Value | Meaning |
| --- | --- |
| soFromBeginning | Offset is from the beginning of the resource. Seek moves to the position Offset. Offset must be >= 0. |
| soFromCurrent | Offset is from the current position in the resource. Seek moves to Position + Offset. |
| soFromEnd | Offset is from the end of the resource. Offset must be <= 0 to indicate a number of bytes before the end of the file. |

*Seek* resets the current *Position* of the stream, moving it by the indicated offset. *Seek* returns the new value of the *Position* property, the new current position in the resource.

## File position and size

*TFileStream* has properties that hold the current position and size of the file. These are used by the *Seek*, read, and write methods.

The *Position* property of *TFileStream* is used to indicate the current offset, in bytes, into the stream (from the beginning of the streamed data). The declaration for *Position* is:

```
property Position: Longint;
```

The *Size* property indicates the size in bytes of the stream. It is used as an end of file marker to truncate the file. The declaration for *Size* is:

```
property Size: Longint;
```

*Size* is used internally by routines that read and write to and from the stream.

Setting the *Size* property changes the size of the file. If the *Size* of the file can not be changed, an exception is raised. For example, trying to changes the *Size* for a file that was opened in *fmOpenRead* mode will raise an exception.

## Copying

*CopyFrom* copies a specified number of bytes from one (file) stream to another.

```
function CopyFrom(Source: TStream; Count: Longint): Longint;
```

Using *CopyFrom* eliminates the need for the user to create, read into, write from, and free a buffer when copying data.

*CopyFrom* copies *Count* bytes from *Source* into the stream. *CopyFrom* then moves the current position by *Count* bytes, and returns the number of bytes copied. If *Count* is 0, *CopyFrom* sets *Source* position to 0 before reading and then copies the entire contents of *Source* into the stream. If *Count* is greater than or less than 0, *CopyFrom* reads from the current position in *Source*.

## Defining new data types

Object Pascal has many predefined data types. You can use these predefined types to create new types that meet the specific needs of your application. For an overview of types, see About types. The syntax for declaring new types is described in Declaring types.

## Developing the application user interface: Overview

With Delphi, you create a user interface (UI) by selecting components from the Component palette and dropping them onto forms.

## Understanding TApplication, TScreen, and TForm

*TApplication*, *TScreen*, and *TForm* are VCL classes that form the backbone of all Delphi applications by controlling the behavior of your project. The *TApplication* class forms the foundation of a Windows application by providing properties and methods that encapsulate the behavior of a standard Windows program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as system specific information such as screen resolution and what fonts are available for display. Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes of your application are based on *TForm*.

## Using the main form

*TForm* is the key class for creating Windows GUI applications.

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

To change the project main form,
 1   Choose Project|Options and select the Forms page.
 2   In the Main Form combo box, select the form you want as the project main form and choose OK.

Now if you run the application, your new main form choice is displayed.

## Adding additional forms

To add an additional form to your project, select File|New Form. You can see all your project's forms and their associated units listed in the Project Manager (View|Project Manager).

## Linking forms

Adding a form to a project adds a reference to it in the project file, but not to any other units in the project. Before you can write code that references the new form, you need to add a reference to it in the referencing forms' unit files. This is called *form linking*.

A common reason to link forms is to provide access to the components in that form. For example, you'll often use form linking to enable a form that contains data-aware components to connect to the data-access components in a data module.

To link a form to another form,

 1   Select the form that needs to refer to another.
 2   Choose File|Use Unit.
 3   Select the name of the form unit for the form to be referenced.
 4   Choose OK.

Linking a form to another just means that the **uses** clauses of one form unit contains a reference to the other's form unit, meaning that the linked form and its components are now in scope for the linking form.

## Avoiding circular unit references

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one of the following:

- Place both **uses** clauses, with the unit identifiers, in the **implementation** parts of the respective unit files. (This is what the File|Use Unit command does.)
- Place one **uses** clause in an **interface** part and the other in an **implementation** part. (You rarely need to place another form's unit identifier in this unit's **interface** part.)

Do not place both **uses** clauses in the **interface** parts of their respective unit files. This will generate the "Circular reference" error at compile time.

## Working at the application level

The global variable *Application*, of type *TApplication*, is in every Delphi Windows application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* would handle how you would call a help file from the menu of your program. Understanding how *TApplication* works is more important to a component writer than to developers of stand-alone applications, but you should set the options that *Application* handles in the Project|Options Application page when you create a project.

In addition, *Application* receives many events that apply to the application as a whole. For example, the *OnActivate* event lets you perform actions when the application first starts up, the *OnIdle* event lets you perform background processes when the application is not busy, the *OnMessage* event lets you intercept Windows messages, and so on. Although you can't use the IDE to examine the properties and events of the global *Application* variable, another component, *TApplicationEvents*, intercepts the events and lets you supply event-handlers using the IDE.

## Handling the screen

An global variable of type *TScreen* called *Screen* is created when you create a project. Screen encapsulates the state of the screen on which your application is running. Common tasks performed by Screen include specifying the look of the cursor, the size of the window in which your application is running, the list of fonts available to the screen device, and multiple screen behavior. If your application runs on multiple monitors, *Screen* maintains a list of monitors and their dimensions so that you can effectively manage the layout of your user interface.

## Managing layout

At its simplest, you control the layout of your user interface by how you place controls in your forms. The placement choices you make are reflected in the control's *Top*, *Left*, *Width*, and *Height* properties. You can change these values at runtime to change the position and size of the controls in your forms.

Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole.

Two properties affect how a control is positioned and sized in relation to its parent. The *Align* property lets you force a control to fit perfectly within its parent along a specific edge or filling up the entire client area after any other controls have been aligned. When the parent is resized, the controls aligned to it are automatically resized and remain positioned so that they fit against a particular edge.

If you want to keep a control positioned relative to a particular edge of its parent, but don't want it to necessarily touch that edge or be resized so that it always runs along the entire edge, you can use the *Anchors* property.

If you want to ensure that a control does not grow too big or too small, you can use the *Constraints* property. *Constraints* lets you specify the control's maximum height, minimum height, maximum width, and minimum width. Set these to limit the size (in pixels) of the control's height and width. For example, by setting the *MinWidth* and *MinHeight* of the constraints on a container object, you can ensure that child objects are always visible.

The value of *Constraints* propagates through the parent/child hierarchy so that an object's size can be constrained because it contains aligned children that have size constraints. *Constraints* can also prevent a control from being scaled in a particular dimension when its *ChangeScale* method is called.

*TControl* introduces a protected event, *OnConstrainedResize*, of type *TConstrainedResizeEvent*:

```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight,
MaxWidth, MaxHeight: Integer) of object;
```

This event allows you to override the size constraints when an attempt is made to resize the control. The values of the constraints are passed as var parameters which can be changed inside the event handler. *OnConstrainedResize* is published for container objects (*TForm*, *TScrollBox*, *TControlBar*, and *TPanel*). In addition, component writers can use or publish this event for any descendant of *TControl*.

Controls that have contents that can change in size have an *AutoSize* property that causes the control to adjust its size to its font or contained objects.

## Working with messages

A message is a notification that some event has occurred that is sent by Windows to an application. The message itself is a record passed to a control by Windows. For instance, when you click a mouse button on a dialog box, Windows sends a message to the active control and the application containing that control reacts to this new event. If the click occurs over a button, the *OnClick* event could be activated upon receipt of the message. If the click occurs just in the form, the application can ignore the message.

The record type passed to the application by Windows is called a *TMsg*. Windows predefines a constant for each message, and these values are stored in the message field of the *TMsg* record. Each of these constants begin with the letters wm.

The VCL automatically handles messages unless you override the message handling system and create your own message handlers. For more information on messages and message handling, see Understanding the message-handling system, Changing message handling, and Creating new message handlers.

## More details on forms

When you create a form in Delphi from the IDE, Delphi automatically creates the form in memory by including code in the *WinMain()* function. Usually, this is the desired behavior and you don't have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default Delphi behavior when creating the form for your main window.

However, you may not want all your application's forms in memory for the duration of the program execution. That is, if you do not want all your application's dialogs in memory at once, you can create the dialogs dynamically when you want them to appear.

Forms can be modal or modeless. Modal forms are forms with which the user must interact before switching to another form (for example, a dialog box requiring user input). Modeless forms, though, are windows that are displayed until they are either obscured by another window or until they are closed or minimized by the user.

## Controlling when forms reside in memory

By default, Delphi automatically creates the application's main form in memory by including the following code in the application's project source unit:

```
Application.CreateForm(TForm1, Form1);
```

This function creates a global variable with the same name as the form. So, every form in an application has an associated global variable. This variable is a pointer to an instance of the form's class and is used to reference the form while the application is running. Any unit that includes the form's unit in its **uses** clause can access the form via this variable.

All forms created in this way in the project unit appear when the program is invoked and exist in memory for the duration of the application.

## Displaying an auto-created form

If you choose to create a form at startup, and do not want it displayed until sometime later during program execution, the form's event handler uses the *ShowModal* method to display the form that is already loaded in memory:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm.ShowModal;
end;
```

In this case, since the form is already in memory, there is no need to create another instance or destroy that instance.

## Creating forms dynamically

You may not always want all your application's forms in memory at once. To reduce the amount of memory required at load time, you may want to create some forms only when you need to use them. For example, a dialog box needs to be in memory only during the time a user interacts with it.

To create a form at a different stage during execution using the IDE, you:
 1   Select the File|New Form from the Component bar to display the new form.
 2   Remove the form from the Auto-create forms list of the Project Options|Forms page.

   This removes the form's invocation at startup. As an alternative, you can manually remove the following line from the project source:
```
Application.CreateForm(TResultsForm, ResultsForm);
```
 3   Invoke the form when desired by using the form's *Show* method, if the form is modeless, or *ShowModal* method, if the form is modal.

An event handler for the main form must create an instance of the result form and destroy it. One way to invoke the result form is to use the global variable as follows. Note that *ResultsForm* is a modal form so the handler uses the *ShowModal* method.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm:=TResultForm.Create(self)
  ResultsForm.ShowModal;
  ResultsForm.Free;
end;
```

The event handler in the example deletes the form after it is closed, so the form would need to be recreated if you needed to use *ResultsForm* elsewhere in the application. If the form were displayed using *Show* you could not delete the form within the event handler because *Show* returns while the form is still open.

**Note:** If you create a form using its constructor, be sure to check that the form is not in the Auto-create forms list on the Project Options|Forms page. Specifically, if you create the new form without deleting the form of the same name from the list, Delphi creates the form at startup and this event-handler creates a new instance of the form, overwriting the reference to the auto-created instance. The auto-created instance still exists, but the application can no longer access it. After the event-handler terminates, the global variable no longer points to a valid form. Any attempt to use the global variable will likely crash the application.

# Creating modeless forms such as windows

You must guarantee that reference variables for modeless forms exist for as long as the form is in use. This means that these variables should have global scope. In most cases, you use the global reference variable that was created when you made the form (the variable name that matches the name property of the form). If your application requires additional instances of the form, declare separate global variables for each instance.

## Using a local variable to create a form instance

A safer way to create a unique instance of a *modal form* is to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not matter whether *ResultsForm* is auto-created or not. The code in the event handler makes no reference to the global form variable. For example:

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

Notice how the global instance of the form is never used in this version of the event handler.

Typically, applications use the global instances of forms. However, if you need a new instance of a modal form, and you use that form in a limited, discrete section of the application, such as a single function, a local instance is usually the safest and most efficient way of working with the form.

Of course, you cannot use local variables in event handlers for modeless forms because they must have global scope to ensure that the forms exist for as long as the form is in use. *Show* returns as soon as the form opens, so if you used a local variable, the local variable would go out of scope immediately.

## Passing additional arguments to forms

Typically, you create forms for your application from within the IDE. When created this way, the forms have a constructor that takes one argument, *Owner*, which is the owner of the form being created. (The owner is the calling application object or form object.) *Owner* can be **nil**.

To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor. The example form class below shows an additional constructor, with the extra argument *whichButton*. This new constructor is added to the form class manually.

```
TResultsForm = class(TForm)
  ResultsLabel: TLabel;
  OKButton: TButton;
  procedure OKButtonClick(Sender: TObject);
private
public
  constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;
```

Here's the manually coded constructor that passes the additional argument, *whichButton*. This constructor uses the *whichButton* parameter to set the *Caption* property of a *Label* control on the form.

```
constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
  case whichButton of
    1: ResultsLabel.Caption := 'You picked the first button.';
    2: ResultsLabel.Caption := 'You picked the second button.';
    3: ResultsLabel.Caption := 'You picked the third button.';
  end;
end;
```

When creating an instance of a form with multiple constructors, you can select the constructor that best suits your purpose. For example, the following *OnClick* handler for a button on a form calls creates an instance of *TResultsForm* that uses the extra parameter:

```
procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
  rf.ShowModal;
  rf.Free;
end;
```

# Retrieving data from forms

Most real-world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. The way you get information from a form depends on whether the form is modal or modeless.

## Retrieving data from modeless forms

You can easily extract information from modeless forms by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modeless form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red", "Green", "Blue", and so on). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* each time a user selects a new color. The class declaration for the form is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;
```

The *OnClick* event handler for the listbox, *ColorListBoxClick*, sets the value of the *CurrentColor* property each time a new item in the listbox is selected. The event handler gets the string from the listbox containing the color name and assigns it to *CurrentColor*. The *CurrentColor* property uses the setter function, *SetColor*, to store the actual value for the property in the private data member *FColor*:

```
procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
    CurrentColor := '';
end;
```

Now suppose that another form within the application, called *ResultsForm*, needs to find out which color is currently selected on *ColorForm* whenever a button (called *UpdateButton)* on *ResultsForm* is clicked. The *OnClick* event handler for *UpdateButton* might look like this:

```
procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  if Assigned(ColorForm) then
  begin
    MainColor := ColorForm.CurrentColor;
    {do something with the string MainColor}
  end;
end;
```

The event handler first verifies that *ColorForm* exists using the *Assigned* function. It then gets the value of *ColorForm's CurrentColor* property.

Alternatively, if *ColorForm* had a public function named *GetColor*, another form could get the current color without using the *CurrentColor* property (for example, MainColor := ColorForm.GetColor;). In fact, there's nothing to prevent another form from getting the *ColorForm's* currently selected color by checking the listbox selection directly:

```
with ColorForm.ColorListBox do
  MainColor := Items[ItemIndex];
```

However, using a property makes the interface to *ColorForm* very straightforward and simple. All a form needs to know about *ColorForm* is to check the value of *CurrentColor*.

## Retrieving data from modal forms

Just like modeless forms, modal forms often contain information needed by other forms. The most common example is form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to decide how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But how do you handle situations where form B is deleted from memory upon closing? Since a form does not have an explicit return value, you must preserve important information from the form before it is destroyed.

To illustrate, consider a modified version of the *ColorForm* form that is designed to be a modal form. The class declaration is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

The form has a listbox called *ColorListBox* with a list of names of colors. When pressed, the button called *SelectButton* makes note of the currently selected color name in *ColorListBox* then closes the form. *CancelButton* is a button that simply closes the form.

Note that a user-defined constructor was added to the class that takes a *Pointer* argument. Presumably, this *Pointer* points to a string that the form launching *ColorForm* knows about. The implementation of this constructor is as follows:

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

The constructor saves the pointer to a private data member *FColor* and initializes the string to an empty string.

**Note:** To use the above user-defined constructor, the form must be explicitly created. It cannot be auto-created when the application is started. For details, see Controlling when forms reside in memory.

In the application, the user selects a color from the listbox and presses *SelectButton* to save the choice and close the form. The *OnClick* event handler for *SelectButton* might look like this:

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
  end;
  Close;
end;
```

Notice that the event handler stores the selected color name in the string referenced by the pointer that was passed to the constructor.

To use *ColorForm* effectively, the calling form must pass the constructor a pointer to an existing string. For example, assume *ColorForm* was instantiated by a form called *ResultsForm* in response to a button called *UpdateButton* on *ResultsForm* being clicked. The event handler would look as follows:

```
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
```

```
begin
  GetColor(Addr(MainColor));
  if MainColor <> '' then
    {do something with the MainColor string}
  else
    {do something else because no color was picked}
end;
procedure GetColor(PColor: Pointer);
begin
  ColorForm := TColorForm.CreateWithColor(PColor, Self);
  ColorForm.ShowModal;
  ColorForm.Free;
end;
```

*UpdateButtonClick* creates a String called MainColor. The address of MainColor is passed to the *GetColor* function which creates *ColorForm*, passing the pointer to MainColor as an argument to the constructor. As soon as *ColorForm* is closed it is deleted, but the color name that was selected is still preserved in MainColor, assuming that a color was selected. Otherwise, MainColor contains an empty string which is a clear indication that the user exited *ColorForm* without selecting a color.

This example uses one string variable to hold information from the modal form. Of course, more complex objects can be used depending on the need. Keep in mind that you should always provide a way to let the calling form know if the modal form was closed without making any changes or selections (such as having MainColor default to an empty string).

## Reusing components and groups of components

Delphi offers several ways to save and reuse work you've done with VCL components:

▪          *Component templates* provide a simple, quick way of configuring and saving groups of components.

▪          You can save forms, data modules, and projects in the *Repository*. This gives you a central database of reusable elements and lets you use form inheritance to propagate changes.

▪          You can save *frames* on the Component palette or in the repository. Frames use form inheritance and can be embedded into forms or other frames.

▪          Creating a *custom component* is the most complicated way of reusing code, but it offers the greatest flexibility.

# Creating and using component templates

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, save them as a *component template*. Later, by selecting the template from the Component palette, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time.

Once you place a template on a form, you can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

To create a component template,

1   Place and arrange components on a form. In the Object Inspector, set their properties and events as desired.

2   Select the components. The easiest way to select several components is to drag the mouse over all of them. Gray handles appear at the corners of each selected component.

3   Choose Component|Create Component Template.

4   Specify a name for the component template in the Component Name edit box. The default proposal is the component type of the first component selected in step 2 followed by the word "Template". For example, if you select a label and then an edit box, the proposed name will be "TLabelTemplate". You can change this name, but be careful not to duplicate existing component names.

5   In the Palette Page edit box, specify the Component palette page where you want the template to reside. If you specify a page that does not exist, a new page is created when you save the template.

6   Under Palette Icon, select a bitmap to represent the template on the palette. The default proposal will be the bitmap used by the component type of the first component selected in step 2. To browse for other bitmaps, click Change. The bitmap you choose must be no larger than 24 pixels by 24 pixels.

7   Click OK.

To remove templates from the Component palette, choose Component|Configure Palette.

# Working with frames

A frame (*TFrame*), like a form, is a container for other components. It uses the same ownership mechanism as forms for automatic instantiation and destruction of the components on it, and the same parent-child relationships for synchronization of component properties. In some ways, however, a frame is more like a customized component than a form. Frames can be saved on the Component palette for easy reuse, and they can be nested within forms, other frames, or other container objects. After a frame is created and saved, it continues to function as a unit and to inherit changes from the components (including other frames) it contains. When a frame is embedded in another frame or form, it continues to inherit changes made to the frame from which it derives.

Creating frames

Using and modifying frames

Sharing frames

## Creating frames

To create an empty frame, choose File|New Frame, or choose File|New and double-click on Frame. You can now drop components (including other frames) onto your new frame.

It is usually best—though not necessary—to save frames as part of a project. If you want to create a project that contains only frames and no forms, choose File|New Application, close the new form and unit without saving them, then choose File|New Frame and save the project.

**Note:** When you save frames, avoid using the default names *Unit1*, *Project1*, and so forth, since these are likely to cause conflicts when you try to use the frames later.

At design time, you can display any frame included in the current project by choosing View|Forms and selecting a frame. As with forms and data modules, you can toggle between the Form Designer and the frame's .DFM file by right-clicking and choosing View as Form or View as Text.

### Adding frames to the Component palette

Frames are added to the Component palette as component templates. To add a frame to the Component palette, open the frame in the Form Designer (you cannot use a frame embedded in another component for this purpose), right-click on the frame, and choose Add to Palette. When the Component Template Information dialog opens, select a name, palette page, and icon for the new template.

## Using and modifying frames

To use a frame in an application, you must place it, directly or indirectly, on a form. You can add frames directly to forms, to other frames, or to other container objects such as panels and scroll boxes.

The Form Designer provides two ways to add a frame to an application:

▪       Select a frame from the Component palette and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame's unit file in your project.

▪       Select *Frames* from the Standard page of the Component palette and click on a form or another frame. A dialog appears with a list of frames that are already included in your project; select one and click OK.

When you drop a frame onto a form or other container, Delphi declares a new class that descends from the frame you selected. (Similarly, when you add a new form to a project, Delphi declares a new class that descends from *TForm*.) This means that changes made later to the original (ancestor) frame propagate to the embedded frame, but changes to the embedded frame do not propagate backward to the ancestor.

Suppose, for example, that you wanted to assemble a group of data-access components and data-aware controls for repeated use, perhaps in more than one application. One way to accomplish this would be to collect the components into a component template; but if you started to use the template and later changed your mind about the arrangement of the controls, you would have to go back and manually alter each project where the template was placed. If, on the other hand, you put your database components into a frame, later changes would need to be made in only one place; changes to an original frame automatically propagate to its embedded descendants when your projects are recompiled. At the same time, you are free to modify any embedded frame without affecting the original frame or other embedded descendants of it. The only limitation on modifying embedded frames is that you cannot add components to them.

▪

A Frame with data-aware controls and a data source component

In addition to simplifying maintenance, frames can help you to use resources more efficiently. For example, to use a bitmap or other graphic in an application, you might load the graphic into the *Picture* property of a *TImage* control. If, however, you use the same graphic repeatedly in one application, each *Imag*e object you place on a form will result in another copy of the graphic being added to the form's resource file. (This is true even if you set *TImage.Picture* once and save the *Image* control as a component template.) A better solution is to drop the *Image* object onto a frame, load your graphic into it, then use the frame where you want the graphic to appear. This results in smaller form files and has the added advantage of letting you change the graphic everywhere it occurs simply by modifying the *Image* on the original frame.

## Sharing frames

You can share a frame with other developers in two ways:

- Add the frame to the Object Repository.
- Distribute the frame's unit (.PAS) and form (.DFM) files.

To add a frame to the Repository, open any project that includes the frame, right-click in the Form Designer, and choose Add to Repository. For more information, see Using the Object Repository.

If you send a frame's unit and form files to other developers, they can open them and add them to the Component palette. If the frame has other frames embedded in it, they will have to open it as part of a project.

# Creating and managing menus

Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You simply add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time.

You don't even need to run your program to see the results—your design is immediately visible in the form, appearing just as it will during runtime. Your code can also change menus at runtime, to provide more information or options to the user.

This section explains how to use the Menu Designer to design menu bars and pop-up (local) menus. It discusses the following ways to work with menus at design time and runtime:

- Opening the Menu Designer
- Building menus
- Editing menu items in the Object Inspector
- Using the Menu Designer context menu
- Using menu templates
- Saving a menu as a template
- Adding images to menu items

For information about hooking up menu items to the code that executes when they are selected, see Associating menu events with event handlers.

## Opening the Menu Designer

To start using the Menu Designer, first add either a MainMenu or PopupMenu component to your form. Both menu components are located on the Standard page of the Component palette.



A MainMenu component creates a menu that's attached to the form's title bar. A PopupMenu component creates a menu that appears when the user right-clicks in the form. Pop-up menus do not have a menu bar.

To open the Menu Designer, select a menu component on the form, and then choose from one of the following methods:

▪        Double-click the menu component.
▪        From the Properties page of the Object Inspector, select the *Items* property, and then either double-click [Menu] in the Value column, or click the ellipsis (...) button.

The Menu Designer appears, with the first (blank) menu item highlighted in the Designer, and the *Caption* property highlighted in the Object Inspector.

## Building menus

You add a menu component to your form, or forms, for every menu you want to include in your application. You can build each menu structure entirely from scratch, or you can start from one of the predesigned menu templates.

This section discusses the basics of creating a menu at design time.For more information about menu templates, see Using menu templates.

## Naming menus

As with all components, when you add a menu component to the form, Delphi gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows Object Pascal naming conventions.

Delphi adds the menu name to the form's type declaration, and the menu name then appears in the Component list.

## Naming the menu items

In contrast to the menu component itself, you need to explicitly name menu items as you add them to the form. You can do this in one of two ways:

- Directly type in the value for the *Name* property.
- Type in the value for the *Caption* property first, and let Delphi derive the *Name* property from the caption.

> For example, if you give a menu item a *Caption* property value of *File*, Delphi assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Delphi leaves the *Caption* property blank until you type in a value.

**Note:** If you enter characters in the *Caption* property that are not valid for Object Pascal identifiers, Delphi modifies the *Name* property accordingly. For example, if you want the caption to start with a number, Delphi precedes the number with a character to derive the *Name* property.

> The following table demonstrates some examples of this, assuming all menu items shown appear in the same menu bar.

| Component caption | Derived name | Explanation |
| --- | --- | --- |
| &File | File1 | Removes ampersand |
| &File (2nd occurrence) | File2 | Numerically orders duplicate items |
| 1234 | N12341 | Adds a preceding letter and numerical order |
| 1234 (2nd occurrence) | N12342 | Adds a number to disambiguate the derived name |
| $@@@# | N1 | Removes all non-standard characters, adding preceding letter and numerical order |
| - (hyphen) | N2 | Numerical ordering of second occurrence of caption with no standard characters |

As with the menu component, Delphi adds any menu item names to the form's type declaration, and those names then appear in the Component list.

## Adding, inserting, and deleting menu items

The following procedures describe how to perform the basic tasks involved in building your menu structure. Each procedure assumes you have the Menu Designer window open.

To add menu items at design time,

1   Select the position where you want to create the menu item.

    If you've just opened the Menu Designer, the first position on the menu bar is already selected.

2   Begin typing to enter the caption. Or enter the *Name* property first by specifically placing your cursor in the Object Inspector and entering a value. In this case, you then need to reselect the *Caption* property and enter a value.

3   Press *Enter*.

    The next placeholder for a menu item is selected.

    If you entered the *Caption* property first, use the arrow keys to return to the menu item you just entered. You'll see that Delphi has filled in the *Name* property based on the value you entered for the caption. (See Naming the menu items.)

4   Continue entering values for the *Name* and *Caption* properties for each new item you want to create, or press *Esc* to return to the menu bar.

    Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press *Enter* to complete an action. To return to the menu bar, press *Esc*.

To insert a new, blank menu item,

1   Place the cursor on a menu item.

2   Press *Ins*.

    Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

To delete a menu item or command,

1   Place the cursor on the menu item you want to delete.

2   Press *Del*.

**Note:** You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in your menu at runtime.

## Adding separator bars

Separator bars insert a line between menu items. You can use separator bars to indicate groupings within the menu list, or simply to provide a visual break in a list.

To make the menu item a separator bar, type a hyphen (-) for the caption.

## Specifying accelerator keys and keyboard shortcuts

Accelerator keys enable the user to access a menu command from the keyboard by pressing *Alt+* the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Delphi automatically checks for duplicate accelerators and adjusts them at runtime. This ensures that menus built dynamically at runtime contain no duplicate accelerators and that all menu items have an accelerator. You can turn off this automatic checking by setting the *AutoHotkeys* property of a menu item to *maManual***.**

To specify an accelerator,

▪     Add an ampersand in front of the appropriate letter.

For example, to add a Save menu command with the S as an accelerator key, type &Save.

Keyboard shortcuts enable the user to perform the action without accessing the menu directly, by typing in the shortcut key combination.

To specify a keyboard shortcut,

▪     Use the Object Inspector to enter a value for the *ShortCut* property, or select a key combination from the drop-down list.

This list is only a subset of the valid combinations you can type in.

When you add a shortcut, it appears next to the menu item caption.

**Caution:** Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

## Creating submenus

Many application menus contain drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. Delphi supports as many levels of such submenus as you want to build into your menu.

Organizing your menu structure this way can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one submenu, if any.)

To create a submenu,

1   Select the menu item under which you want to create a submenu.
2   Press *Ctrl®* to create the first placeholder, or right-click and choose Create Submenu.
3   Type a name for the submenu item, or drag an existing menu item into this placeholder.
4   Press *Enter*, or ¯, to create the next placeholder.
5   Repeat steps 3 and 4 for each item you want to create in the submenu.
6   Press *Esc* to return to the previous menu level.

## Creating submenus by demoting existing menus

You can create a submenu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact submenu. This pertains to submenus as well—moving a menu item into an existing submenu just creates one more level of nesting.

## Moving menu items

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a different menu entirely.

The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own submenu. However, you can move any item into a *different* menu, no matter what its original position is.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar,

1   Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.

2   Release the mouse button to drop the menu item at the new location.

To move a menu item into a menu list,

1   Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu.

   This causes the menu to open, enabling you to drag the item to its new location.

2   Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

## Adding images to menu items

Images can help users navigate in menus by matching glyphs and images to menu item action, similar to toolbar images. To add an image to a menu item:

1   Drop a *TMainMenu* or *TPopupMenu* object on a form.
2   Drop a *TImageList* object on the form.
3   Open the ImageList editor by double clicking on the *TImageList* object.
4   Click Add to select the bitmap or bitmap group you want to use in the menu. Click OK.
5   Set the *TMainMenu* or *TPopupMenu* object's *Images* property to the ImageList you just created.
6   Create your menu items and submenu items as described in this topic group.
7   Select the menu item you want to have an image in the Object Inspector and set the *ImageIndex* property to the corresponding number of the image in the *ImageList* (the default value for *ImageIndex* is -1, which doesn't display an image).

**Note:** Use images that are 16 by 16 pixels for proper display in the menu. Although you can use other sizes for the menu images, alignment and consistency problems may result when using images greater than or smaller than 16 by 16 pixels.

## Viewing the menu

You can view your menu in the form at design time without first running your program code. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu Designer to view a pop-up menu at design time.)

To view the menu,

1  If the form is visible, click the form, or from the View menu, choose the form whose menu you want to view.

2  If the form has more than one menu, select the menu you want to view from the form's *Menu* property drop-down list.

The menu appears in the form exactly as it will when you run the program.

## Editing menu items in the Object Inspector

This section has discussed how to set several properties for menu items—for example, the *Name* and *Caption* properties—by using the Menu Designer.

The section has also described how to set menu item properties, such as the *ShortCut* property, directly in the Object Inspector, just as you would for any component selected in the form.

When you edit a menu item by using the Menu Designer, its properties are still displayed in the Object Inspector. You can switch focus to the Object Inspector and continue editing the menu item properties there. Or you can select the menu item from the Component list in the Object Inspector and edit its properties without ever opening the Menu Designer.

To close the Menu Designer window and continue editing menu items,

1　Switch focus from the Menu Designer window to the Object Inspector by clicking the properties page of the Object Inspector.

2　Close the Menu Designer as you normally would.

　　The focus remains in the Object Inspector, where you can continue editing properties for the selected menu item. To edit another menu item, select it from the Component list.

For information about assigning event handlers to menus, see <u>Associating menu events with event handlers</u>"Associating menu events with event handlers" on page 2-26.

## Using the Menu Designer context menu

The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options. (For more information about menu templates, refer to Using menu templates.)

To display the context menu, right-click the Menu Designer window, or press *Alt*+F10 when the cursor is in the Menu Designer window.

# Commands on the context menu

The following table summarizes the commands on the Menu Designer context menu.

| Menu command | Action |
| --- | --- |
| Insert | Inserts a placeholder above or to the left of the cursor. |
| Delete | Deletes the selected menu item (and all its sub-items, if any). |
| Create Submenu | Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item. |
| Select Menu | Opens a list of menus in the current form. Double-clicking a menu name opens the designer window for the menu. |
| Save As Template | Opens the Save Template dialog box, where you can save a menu for future reuse. |
| Insert From Template | Opens the Insert Template dialog box, where you can select a template to reuse. |
| Delete Templates | Opens the Delete Templates dialog box, where you can choose to delete any existing templates. |
| Insert From Resource | Opens the Insert Menu from Resource file dialog box, where you can choose an .MNU file to open in the current form. |

## Switching between menus at design time

If you're designing several menus for your form, you can use the Menu Designer context menu or the Object Inspector to easily select and move among them.

To use the context menu to switch between menus in a form,

1   Right-click in the Menu Designer and choose Select Menu.

The Select Menu dialog box appears.

This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.

2   From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch between menus in a form,

1   Give focus to the form whose menus you want to choose from.

2   From the Component list, select the menu you want to edit.

3   On the Properties page of the Object Inspector, select the *Items* property for this menu, and then either click the ellipsis button, or double-click [Menu].

## Using menu templates

Delphi provides several predesigned menus, or menu templates, that contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

The menu templates shipped with Delphi are stored in the BIN subdirectory in a default installation. These files have a .DMT (Delphi menu template) extension.

You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

To add a menu template to your application,

1  Right-click the Menu Designer and choose Insert From Template.

   (If there are no templates, the Insert From Template option appears dimmed in the context menu.)

   The Insert Template dialog box opens, displaying a list of available menu templates.

2  Select the menu template you want to insert, then press *Enter* or choose OK.

   This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template,

1  Right-click the Menu Designer and choose Delete Templates.

   (If there are no templates, the Delete Templates option appears dimmed in the context menu.)

   The Delete Templates dialog box opens, displaying a list of available templates.

2  Select the menu template you want to delete, and press *Del*.

   Delphi deletes the template from the templates list and from your hard disk.

## Saving a menu as a template

Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

The menu templates you save are stored in your BIN subdirectory as .DMT files.

To save a menu as a template,

1  Design the menu you want to be able to reuse.

   This menu can contain as many items, commands, and submenus as you like; everything in the active Menu Designer window will be saved as one reusable menu.

2  Right-click in the Menu Designer and choose Save As Template.

   The Save Template dialog box appears.

3  In the Template Description edit box, type a brief description for this menu, and then choose OK.

   The Save Template dialog box closes, saving your menu design and returning you to the Menu Designer window.

**Note:** The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the *Name* or *Caption* property for the menu.

## Naming conventions for template menu items and event handlers

When you save a menu as a template, Delphi does not save its *Name* property, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Delphi then generates new names for it and all of its items.

For example, suppose you save a File menu as a template. In the original menu, you name it *MyFile*. If you insert it as a template into a new menu, Delphi names it *File1*. If you insert it into a menu with an existing menu item named *File1*, Delphi names it *File2*.

Delphi also does not save any *OnClick* event handlers associated with a menu saved as a template, since there is no way to test whether the code would be applicable in the new form. When you generate a new event handler for the menu template item, Delphi still generates the event handler name.

You can easily associate items in the menu template with existing *OnClick* event handlers in the form. For more information, see Associating an event with an existing event handler.

## Manipulating menu items at runtime

Sometimes you want to add menu items to an existing menu structure while the application is running, to provide more information or options to the user. You can insert a menu item by using the menu item's *Add* or *Insert* method, or you can alternately hide and show the items in a menu by changing their *Visible* property. The *Visible* property determines whether the menu item is displayed in the menu. To dim a menu item without hiding it, use the *Enabled* property.

For examples that use the menu item's *Visible* and *Enabled* properties, see <u>Disabling menu items</u>.

In multiple document interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. See <u>Merging menus</u> for more information.

## Merging menus

For MDI applications, such as the text editor sample application, and for OLE client applications, your application's main menu needs to be able to receive menu items either from another form or from the OLE server object. This is often called *merging menus*.

You prepare menus for merging by specifying values for two properties:

- *Menu*, a property of the form
- *GroupIndex*, a property of menu items in the menu

## Specifying the active menu: Menu property

The *Menu* property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the *Menu* property in code. For example,

```
Form1.Menu := SecondMenu;
```

## Determining the order of merged menu items: GroupIndex property

The *GroupIndex* property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar, or can be inserted.

The default value for *GroupIndex* is 0. Several rules apply when specifying a value for *GroupIndex*:

- Lower numbers appear first (farther left) in the menu.

  For instance, set the *GroupIndex* property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn't be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.

- To replace items in the main menu, give items on the child menu the same *GroupIndex* value.

  This can apply to groupings or to single items. For example, if your main form has an Edit menu item with a *GroupIndex* value of 1, you can replace it with one or more items from the child form's menu by giving them a *GroupIndex* value of 1 as well.

  Giving multiple items in the child menu the same *GroupIndex* value keeps their order intact when they merge into the main menu.

- To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.

  For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3 and 4.

## Importing resource files

Delphi supports menus built with other applications, so long as they are in the standard Windows resource (.RC) file format. You can import such menus directly into your Delphi project, saving you the time and effort of rebuilding menus that you created elsewhere.

To load existing .RC menu files,

  1   In the Menu Designer, place your cursor where you want the menu to appear.

      The imported menu can be part of a menu you are designing, or an entire menu in itself.
  2   Right-click and choose Insert From Resource.

      The Insert Menu From Resource dialog box appears.
  3   In the dialog box, select the resource file you want to load, and choose OK.

      The menu appears in the Menu Designer window.

**Note:** If your resource file contains more than one menu, you first need to save each menu as a separate resource file before importing it.

## Designing toolbars and cool bars

A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. A *cool bar* (also called a rebar) is a kind of toolbar that displays controls on movable, resizable bands. If you have multiple panels aligned to the top of the form, they stack vertically in the order added.

You can put controls of any sort on a toolbar. In addition to buttons, you may want to put use color grids, scroll bars, labels, and so on.

There are several ways to add a toolbar to a form:

- Place a panel (*TPanel*) on the form and add controls (typically speed buttons) to it.
- Use a toolbar component (*TToolBar*) instead of *TPanel*, and add controls to it. *TToolBar* manages buttons and other controls, arranging them in rows and automatically adjusting their sizes and positions. If you use tool button (*TToolButton*) controls on the toolbar, *TToolBar* makes it easy to group the buttons functionally and provides other display options.
- Use a cool bar (*TCoolBar*) component and add controls to it. The cool bar displays controls on independently movable and resizable bands.

How you implement your toolbar depends on your application. The advantage of using the Panel component is that you have total control over the look and feel of the toolbar.

By using the toolbar and cool bar components, you are ensuring that your application has the look and feel of a Windows application because you are using the native Windows controls. If these operating system controls change in the future, your application could change as well. Also, since the toolbar and cool bar rely on common components in Windows, your application requires the COMCTL32.DLL. Toolbars and cool bars are not supported in WinNT 3.51 applications.

The following sections describe how to

- Adding a toolbar using a panel component
- Adding a toolbar using the toolbar component
- Adding a cool bar component
- Responding to clicks
- Adding hidden toolbars
- Hiding and showing toolbars

## Adding a toolbar using a panel component

To add a toolbar to a form using the panel component,

1  Add a panel component to the form (from the Standard page of the Component palette).
2  Set the panel's *Align* property to *alTop*. When aligned to the top of the form, the panel maintains its height, but matches its width to the full width of the form's client area, even if the window changes size.
3  Add speed buttons or other controls to the panel.

Speed buttons are designed to work on toolbar panels. A speed button usually has no caption, only a small graphic (called a *glyph*), which represents the button's function.

Speed buttons have three possible modes of operation. They can

- Act like regular pushbuttons
- Toggle on and off when clicked
- Act like a set of radio buttons

To implement speed buttons on toolbars, do the following:

- Adding a speed button to a panel
- Assigning a speed button's glyph
- Setting the initial condition of a speed button
- Creating a group of speed buttons
- Allowing toggle buttons

## Adding a speed button to a panel

To add a speed button to a toolbar panel, place the speed button component (from the Additional page of the Component palette) on the panel.

The panel, rather than the form, "owns" the speed button, so moving or hiding the panel also moves or hides the speed button.

The default height of the panel is 41, and the default height of speed buttons is 25. If you set the *Top* property of each button to 8, they'll be vertically centered. The default grid setting snaps the speed button to that vertical position for you.

## Assigning a speed button's glyph

Each speed button needs a graphic image called a *glyph* to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer.

You normally assign glyphs to speed buttons at design time, although you can assign different glyphs at runtime.

To assign a glyph to a speed button at design time,

 1   Select the speed button.
 2   In the Object Inspector, select the *Glyph* property.
 3   Double-click the Value column beside *Glyph* to open the Picture Editor and select the desired bitmap.

## Setting the initial condition of a speed button

Speed buttons use their appearance to give the user clues as to their state and purpose. Because they have no caption, it's important that you use the right visual cues to assist users.

The table below lists some actions you can set to change a speed button's appearance:

| To make a speed button: | Set the toolbar's: |
| --- | --- |
| Appear pressed | *GroupIndex* property to a value other than zero and its *Down* property to *True*. |
| Appear disabled | *Enabled* property to *False*. |
| Have a left margin | *Indent* property to a value greater than 0. |

If your application has a default drawing tool, ensure that its button on the toolbar is pressed when the application starts. To do so, set its *GroupIndex* property to a value other than zero and its *Down* property to *True*.

## Creating a group of speed buttons

A series of speed buttons often represents a set of mutually exclusive choices. In that case, you need to associate the buttons into a group, so that clicking any button in the group causes the others in the group to pop up.

To associate any number of speed buttons into a group, assign the same number to each speed button's *GroupIndex* property.

The easiest way to do this is to select all the buttons you want in the group, and, with the whole group selected, set *GroupIndex* to a unique value.

## Allowing toggle buttons

Sometimes you want to be able to click a button in a group that's already pressed and have it pop up, leaving no button in the group pressed. Such a button is called a *toggle*. Use *AllowAllUp* to create a grouped button that acts as a toggle: click it once, it's down; click it again, it pops up.

To make a grouped speed button a toggle, set its *AllowAllUp* property to *True*.

Setting *AllowAllUp* to *True* for any speed button in a group automatically sets the same property value for all buttons in the group. This enables the group to act as a normal group, with only one button pressed at a time, but also allows every button to be up at the same time.

## Adding a toolbar using the toolbar component

The toolbar component (*TToolBar*) offers button management and display features that panel components do not. To add a toolbar to a form using the toolbar component,

1  Add a toolbar component to the form (from the Win32 page of the Component palette). The toolbar automatically aligns to the top of the form.
2  Add tool buttons or other controls to the bar.

Tool buttons are designed to work on toolbar components. Like speed buttons, tool buttons can

- Act like regular pushbuttons
- Toggle on and off when clicked
- Act like a set of radio buttons

To implement tool buttons on a toolbar, do the following:

- Adding a tool button
- Assigning images to tool buttons
- Setting tool button appearance and initial conditions
- Creating groups of tool buttons
- Allowing toggled tool buttons

## Adding a tool button

To add a tool button to a toolbar, right-click on the toolbar and choose New Button.

The toolbar "owns" the tool button, so moving or hiding the toolbar also moves or hides the button. In addition, all tool buttons on the toolbar automatically maintain the same height and width. You can drop other controls from the Component palette onto the toolbar, and they will automatically maintain a uniform height. Controls will also wrap around and start a new row when they do not fit horizontally on the toolbar.

## Assigning images to tool buttons

Each tool button has an *ImageIndex* property that determines what image appears on it at runtime. If you supply the tool button only one image, the button manipulates that image to indicate whether the button is disabled. To assign images to tool buttons at design time,

1  Select the toolbar on which the buttons appear.
2  In the Object Inspector, assign a *TImageList* object to the toolbar's *Images* property. An image list is a collection of same-sized icons or bitmaps.
3  Select a tool button.
4  In the Object Inspector, assign an integer to the tool button's *ImageIndex* property that corresponds to the image in the image list that you want to assign to the button.

You can also specify separate images to appear on the tool buttons when they are disabled and when they are under the mouse pointer. To do so, assign separate image lists to the toolbar's *DisabledImages* and *HotImages* properties.

## Setting tool button appearance and initial conditions

The table below lists some actions you can set to change a tool button's appearance:

| To make a tool button: | Set the toolbar's: |
| --- | --- |
| Appear pressed | *GroupIndex* property to a nonzero value and its *Down* property to *True*. |
| Appear disabled | *Enabled* property to *False*. |
| Have a left margin | *Indent* property to a value greater than 0. |
| Appear to have "pop-up" borders, thus making the toolbar appear transparent | *Flat* property to *True*. |

**Note:** Using the *Flat* property of *TToolBar* requires version 4.70 or later of COMCTL32.DLL.

To force a new row of controls after a specific tool button, Select the tool button that you want to appear last in the row and set its *Wrap* property to *True*.

To turn off the auto-wrap feature of the toolbar, set the toolbar's *Wrapable* property to *False*.

## Creating groups of tool buttons

To create a group of tool buttons, select the buttons you want to associate and set their *Style* property to *tbsCheck*; then set their *Grouped* property to *True*. Selecting a grouped tool button causes other buttons in the group to pop up, which is helpful to represent a set of mutually exclusive choices.

Any unbroken sequence of adjacent tool buttons with *Style* set to *tbsCheck* and *Grouped* set to *True* forms a single group. To break up a group of tool buttons, separate the buttons with any of the following:

- A tool button whose *Grouped* property is *False*.
- A tool button whose *Style* property is not set to *tbsCheck*. To create spaces or dividers on the toolbar, add a tool button whose *Style* is *tbsSeparator* or *tbsDivider*.
- Another control besides a tool button.

## Allowing toggled tool buttons

Use *AllowAllUp* to create a grouped tool button that acts as a toggle: click it once, it is down; click it again, it pops up. To make a grouped tool button a toggle, set its *AllowAllUp* property to *True*.

As with speed buttons, setting *AllowAllUp* to *True* for any tool button in a group automatically sets the same property value for all buttons in the group.

## Adding a cool bar component

The cool bar component—also called a *rebar*—displays windowed controls on independently movable, resizable bands. The user can position the bands by dragging the resizing grips on the left side of each band.

To add a cool bar to a form,

1   Add a cool bar component to the form (from the Win32 page of the Component palette). The cool bar automatically aligns to the top of the form.
2   Add windowed controls from the Component palette to the bar.

Only components that descend from *TWinControl* are windowed controls. You can add graphic controls —such as labels or speed buttons—to the cool bar, but they will not appear on separate bands.

**Note:** The cool bar component requires version 4.70 or later of COMCTL.DLL.

## Setting the appearance of the cool bar

The cool bar component offers several useful configuration options. The table below lists some actions you can set to change a tool button's appearance:

| To make the cool bar: | Set the toolbar's: |
|---|---|
| Resize automatically to accommodate the bands it contains | *AutoSize* property to *True*. |
| Bands maintain a uniform height | *FixedSize* property to *True*. |
| Reorient to vertical rather than horizontal | *Vertical* property to *True*. This changes the effect of the *FixedSize* property. |
| Prevent the *Text* properties of the bands from displaying at runtime | *ShowText* property to *False*. Each band in a cool bar has its own *Text* property. |
| Remove the border around the bar | *BandBorderStyle* to *bsNone.* |
| Keep users from changing the bands' order at runtime. (The user can still move and resize the bands.) | *FixedOrder* to *True*. |
| Create a background image for the cool bar | *Bitmap* property to *TBitmap* object. |
| Choose a list of images to appear on the left of any band | *Images* property to *TImageList* object. |

To assign images to individual bands, select the cool bar and double-click on the *Bands* property in the Object Inspector. Then select a band and assign a value to its *ImageIndex* property.

## Responding to clicks

When the user clicks a control, such as a button on a toolbar, the application generates an *OnClick* event which you can respond to with an event handler. Since *OnClick* is the default event for buttons, you can generate a skeleton handler for the event by double-clicking the button at design time. For more information, see Working with events and event handlers and Generating a handler for a component's default event.

## Assigning a menu to a tool button

If you are using a toolbar (*TToolBar*) with tool buttons (*TToolButton*), you can associate menu with a specific button:

1   Select the tool button.
2   In the Object Inspector, assign a pop-up menu (*TPopupMenu*) to the tool button's *DropDownMenu* property.

If the menu's *AutoPopup* property is set to *True*, it will appear automatically when the button is pressed.

## Adding hidden toolbars

Toolbars do not have to be visible all the time. In fact, it is often convenient to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.

To create a hidden toolbar,

1 Add a toolbar, cool bar, or panel component to the form.
2 Set the component's *Visible* property to *False*.

Although the toolbar remains visible at design time so you can modify it, it remains hidden at runtime until the application specifically makes it visible.

## Hiding and showing toolbars

Often, you want an application to have multiple toolbars, but you do not want to clutter the form with them all at once. Or you may want to let users decide whether to display toolbars. As with all components, toolbars can be shown or hidden at runtime as needed.

To hide or show a toolbar at runtime, set its *Visible* property to *False* or *True*, respectively. Usually you do this in response to particular user events or changes in the operating mode of the application. To do this, you typically have a close button on each toolbar. When the user clicks that button, the application hides the corresponding toolbar.

You can also provide a means of toggling the toolbar. In the following example, a toolbar of pens is toggled from a button on the main toolbar. Since each click presses or releases the button, an *OnClick* event handler can show or hide the Pen toolbar depending on whether the button is up or down.

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
   PenBar.Visible := PenButton.Down;
end;
```

## Using action lists

Action lists let you centralize the response to user commands (actions) for objects such as menus and buttons that respond to those commands. This section is an overview of actions and action lists, describing how to use them and how they interact with their clients and targets.

The following help topics are discussed in this section:

- Action objects
- Using Actions
- Pre-defined action classes
- Writing action components
- Demo programs

## Action objects

Actions are user commands that operate on target objects. You create actions in the action list component editor. These actions are later connected to client controls via their action links. Following are descriptions of each type of component in the action/action list mechanism:

▪       An action (*TAction*) is the implementation of an action, such as copying highlighted text, on a target, such as an edit control. An action is triggered by a client in response to a user command (such as a mouse click). Clients are typically menu items or buttons. The *StdActns* unit contains classes derived from *TAction* that implement the basic Edit and Window menu commands (actions) found in most Window applications.

▪       An action list (*TActionList*) is a component that maintains a list of actions (*TAction*). Action lists are the design-time user interface for working with actions.

▪       An action link (*TActionLink*) is an object that maintains the connection between actions and clients. Action links determine whether an action, or which action, is currently applicable for a given client.

▪       A client of an action is typically a menu item or a button (*TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox, TRadioButton*, and so on). An action is initiated by a corresponding command in the client. Typically a client *Click* is associated with an action *Execute*.

▪       An action target is usually a control, such as a rich edit, a memo, or a data control. The *DBActns* unit, for example, contains classes that implement actions specific to data set controls. Component writers can create their own actions specific to the needs of the controls they design and use, and then package those units to create more modular applications.

The following figure shows the relationship of these objects. In this diagram, *Cut1* is the action, *ActionList1* is the action list containing *Cut1*, *SpeedButton1* is the client of *Cut1*, and *Memo1* is the target. Unlike actions, action lists, action clients, and action targets, action links are non-visual objects. The action link in this diagram is therefore indicated by a white rectangle. The action link associates the *SpeedButton1* client to the *Cut1* action contained in *ActionList1.*
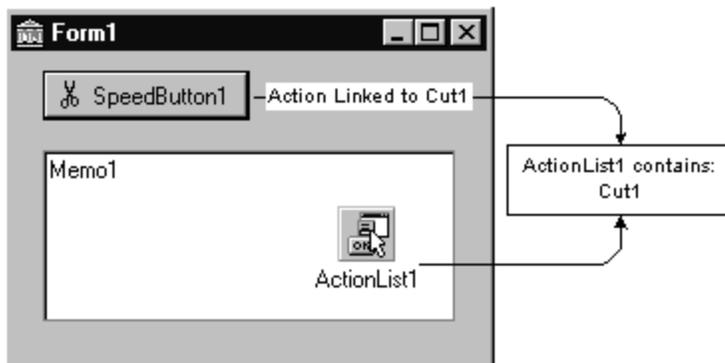


The VCL includes *TAction*, *TActionList*, and *TActionLink* type classes for working with Action lists. By unit, these are

▪       ActnList.pas: *TAction, TActionLink, TActionList, TContainedAction, TCustomAction*, and *TCustomActionList*

▪       Classes.pas: *TBasicAction* and *TBasicActionLink*

▪       Controls.pas: *TControlActionLink* and *TWinControlActionLink*

▪       ComCtrls.pas: *TToolButtonActionLink*

▪       Menus.pas: *TMenuActionLink*

▪       StdCtrls.pas: *TButtonActionLink*

There are also two units, *StdActns* and *DBActns*, that contain auxiliary classes that implement specific, commonly used standard Windows and data set actions. These are described in Pre-defined action classes.. Many of the VCL controls include properties (such as *Action*) and methods (such as *ExecuteAction*) that enable them to be used as action clients and targets.

## Using Actions

You can add an action list to your forms or data modules from the standard page of the Component Palette. Double-click the action list to display the Action List editor, which lets you add, delete, and rearrange actions in much the same way you use the collection editor.

In the Object Inspector, set the properties for each action. The *Name* property identifies the action, and the other properties and events (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, and *Visible*) correspond to the properties of client controls. These are typically, but not necessarily, the same name as the client property. For example, an action's *Checked* property corresponds to a *TToolButton*'s *Down* property.

How to use actions is discussed in the following help topics:
- Centralizing code
- Linking properties
- Executing actions
- Updating actions

## Centralizing code

A number of controls such as *TToolButton*, *TSpeedButton*, *TMenuItem*, and *TButton* have a published property called *Action*. When you set the *Action* property to one of the actions in your action list, the values of the corresponding properties in the action are copied to those of the control. All properties and events in common with the action object (except *Name* and *Tag*) are dynamically linked to the control. Thus, for example, instead of duplicating the code that disables buttons and menu items, you can centralize this code in an action object, and when the action is disabled, all corresponding buttons and menu items are disabled.

## Linking properties

The client's action link is the mechanism through which its properties are associated with (linked to) the properties of an action. When an action changes, the action link is responsible for updating the client's properties. For details about which properties a particular action link class handles, refer to the individual action link classes in the VCL reference.

You can selectively override the values of the properties controlled by an associated action object by setting the property's value in the client component or control. This does not change the property in the action, so only the client is affected.

## Executing actions

When a client component or control is clicked, the *OnExecute* event occurs for it's associated action. For example, the following code illustrates the *OnExecute* event handler for an action that toggles the visibility of a toolbar when the action is executed:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
  { Toggle Toolbar1's visibility }
  ToolBar1.Visible := not ToolBar1.Visible;
end;
```

**Note:** If you are using a tool button or a menu item, you must manually set the *Images* property of the corresponding toolbar or menu component to the *Images* property of the action list. This is true even though the *ImageIndex* property is dynamically linked to the client.

For general information about events and event handlers, see Working with events and event handlers.

The following figure illustrates the dispatching sequence for the execution cycle of an action called *Cut1*. This diagram assumes the relationship of the components in the figure in Action objects, meaning that the *Speedbutton1* client is linked to the *Cut1* action via its action link. *Speedbutton1*'s *Action* property is therefore *Cut1*. Consequently, *Speedbutton1*'s *Click* method invokes *Cut1*'s *Execute* method.
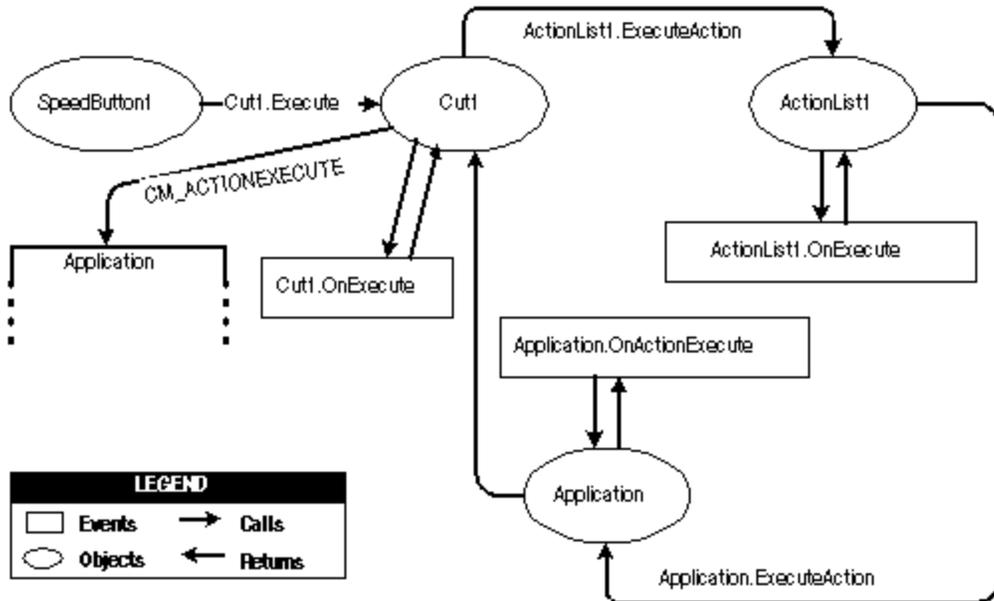


**Note:** In the description of this sequence, one method invoking another does not necessarily mean that the invocation is explicit in the code for that method.

Clicking on *Speedbutton1* initiates the following execution cycle:

▪       *Speedbutton1*'s *Click* method invokes *Cut1.Execute.*

▪       The *Cut1* action defers to its action list (*ActionList1*) for the processing of its *Execute*. This is done by calling the Action list's *ExecuteAction* method, passing itself as a parameter.

▪       *ActionList1* calls its event handler (*OnExecute*) for *ExecuteAction*. (An action list's *ExecuteAction* method applies to all actions contained by the action list.) This handler has a parameter *Handled*, that returns *False* by default. If the handler is assigned and handles the event, it should return *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled:
Boolean);
begin
  { Prevent execution of actions contained by ActionList1 }
  Handled := True;
end;
```

If execution is not handled, at this point, in the action list event handler, then processing continues:

- The *Cut1* action is routed to the *Application* object's *ExecuteAction* method, which invokes the *OnActionExecute* event handler. (The application's *ExecuteAction* method applies to all of the actions in that application.) The sequence is the same as for the action list *ExecuteAction*: The handler has a parameter *Handled* that returns *False* by default. If the handler is assigned and handles the event, it should return *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled:
Boolean);
begin
  { Prevent execution of all actions in Application }
  Handled := True;
end;
```

If execution is not handled in the application's event handler, then *Cut1* send the CM_ACTIONEXECUTE message to the application's *WndProc*, passing itself as a parameter. The application then tries to find a target on which to execute the action (see the figure "Action targets").

## Updating actions

When the application is idle, the *OnUpdate* event occurs for every action that is linked to a visible control or menu item that is showing. This provides an opportunity for applications to execute centralized code for enabling and disabling, checking and unchecking, and so on. For example, the following code illustrates the *OnUpdate* event handler for an action that is "checked" when the toolbar is visible:

```
procedure TForm1.Action1Update(Sender: TObject);
begin
  { Indicate whether ToolBar1 is currently visible }
  (Sender as TAction).Checked := ToolBar1.Visible;
end;
```

See also the RichEdit demo(Demos\RichEdit).

The dispatching cycle for updating actions follows the same sequence as the execution cycle in The figure in Executing actions.

**Note:** Do not add time-intensive code to the *OnUpdate* event handler. This executes whenever the application is idle. If the event handler takes too much time, it will adversely affect performance of the entire application.

## Pre-defined action classes

Component writers can use the classes in the *StdActns* and *DBActns* units as examples for deriving their own action classes that implement behaviors specific to certain controls or components. The base classes for these specialized actions (*TEditAction*, *TWindowAction*) generally override *HandlesTarget, UpdateTarget,* and other methods to limit the target for the action to a specific class of objects. The descendant classes typically override *ExecuteTarget* to perform a specialized task.

The pre-defined action classes are grouped into the following categories:

- Standard edit actions
- Standard Window actions
- Standard Help actions
- DataSet actions

## Standard edit actions

The standard edit actions are designed to be used with an edit control target. *TEditAction* is the base class for descendants that each override the *ExecuteTarget* method to implement copy, cut, and paste tasks by using the Windows Clipboard.

- *TEditAction* ensures that the target control is a *TCustomEdit* class (or descendant).
- *TEditCopy* copies highlighted text to the Clipboard.
- *TEditCut* cuts highlighted text from the target to the Clipboard.
- *TEditPaste* pastes text from the Clipboard to the target and ensures that the Clipboard is enabled for the text format.
- *TEditDelete* deletes the highlighted text.
- *TEditSelectAll* selects all the text in the target edit control.
- *TEditUndo* undoes the last edit made to the target edit control.

## Standard Window actions

The standard Window actions are designed to be used with forms as targets in an MDI application. *TWindowAction* is the base class for descendants that each override the *ExecuteTarget* method to implement arranging, cascading, closing, tiling, and minimizing MDI child forms.

- *TWindowAction* ensures that the target control is a *TForm* class and checks whether the form has MDI child forms.
- *TWindowArrange* arranges the icons of minimized MDI child forms.
- *TWindowCascade* cascades the MDI child forms.
- *TWindowClose* closes the active MDI child form.
- *TWindowMinimizeAll* minimizes all of the MDI child forms.
- *TWindowTileHorizontal* arranges MDI child forms so that they are all the same size, tiled horizontally.
- *TWindowTileVertical* arranges MDI child forms so that they are all the same size, tiled vertically.

## Standard Help actions

The standard Help actions are designed to be used with any target. *THelpAction* is the base class for descendants that each override the *ExecuteTarget* method to pass the command on to WinHelp.

- *THelpAction* ensures that the global *Application* variable is available, so that commands can be handled using its *HelpCommand* method.
- *THelpContents* brings up the Help Topics dialog on the tab (Contents, Index or Find) that was last used.
- *THelpTopicSearch* brings up the Help Topics dialog on the Index tab.
- *THelpOnHelp* brings up the Microsoft help file on how to use Help. Note that this file is an HTML help file on recent versions of Windows, and does not describe the WinHelp system.

# DataSet actions

The standard dataset actions are designed to be used with a dataset component target. *TDataSetAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement navigation and editing of the target.

▪        The *TDataSetAction* introduces a *DataSource* property which ensures actions are performed on that dataset. If *DataSource* is **nil**, the currently focused data-aware control is used. *TDataSetAction* ensures that the target is a *TDataSource* class and has an associated data set.

▪        *TDataSetCancel* cancels the edits to the current record, restores the record display to its condition prior to editing, and turns off Insert and Edit states if they are active.

▪        *TDataSetDelete* deletes the current record and makes the next record the current record.

▪        *TDataSetEdit* puts the dataset into *Edit* state so that the current record can be modified.

▪        *TDataSetFirst* sets the current record to the first record in the dataset.

▪        *TDataSetInsert* inserts a new record before the current record, and sets the dataset into Insert and Edit states.

▪        *TDataSetLast* sets the current record to the last record in the dataset.

▪        *TDataSetNext* sets the current record to the next record.

▪        *TDataSetPost* writes changes in the current record to the dataset.

▪        *TDataSetPrior* sets the current record to the previous record.

▪        *TDataSetRefresh* refreshes the buffered data in the associated dataset.

## Writing action components

The pre-defined actions are examples of extending the VCL action classes. The following topics are useful if you are writing your own action classes:

- How actions find their targets
- Registering actions
- Writing action list editors

## How actions find their targets

The figure in Executing actions illustrates the execution cycle for the standard VCL action classes. If execution is not handled by the action list, the application, or the default action event handlers, then the CM_ACTIONEXECUTE message is sent to the application's *WndProc*. The following figure continues the execution sequence at this point. The pre-defined action classes as well as any action class that you create, use this path of execution:
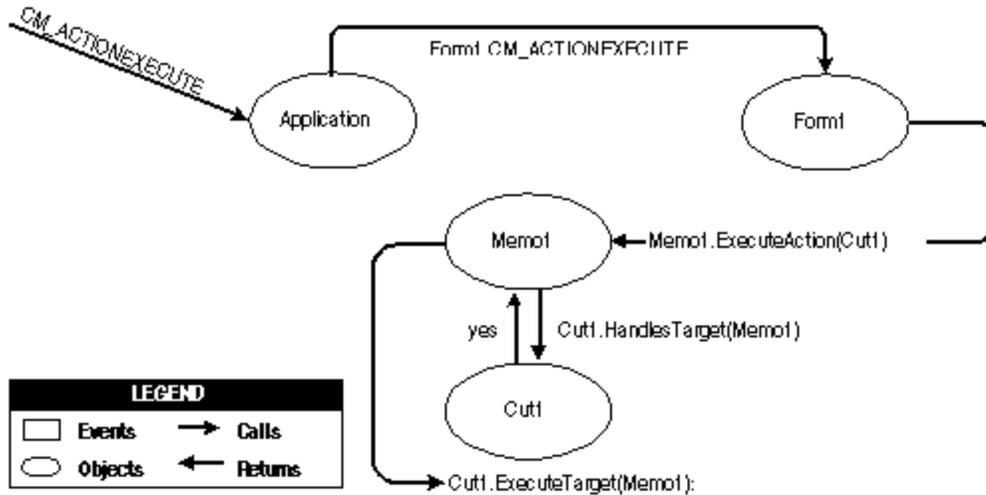


- ▪       Upon receiving the CM_ACTIONEXECUTE message the application first dispatches it to the Screen's *ActiveForm*. If there is no active form, the application sends the message to it's *MainForm*.
- ▪       *Form1* (in this example, the active form) first looks for the active control (*Memo1*) and calls that control's *ExecuteAction* method passing *Cut1* as a parameter.
- ▪       *Memo1* calls *Cut1*'s *HandlesTarget* method, passing itself to determine whether it is an appropriate target for the action. If *Memo1* is not an appropriate target, *HandlesTarget* returns *False* and *Memo1*'s *ExecuteAction* handler returns *False*.
- ▪       In this case, *Memo1* is an appropriate target for *Cut1*, so *HandlesTarget* returns *True*. *Memo1* then calls *Cut1.ExecuteTarget* passing itself as a parameter. Finally, since *Cut1* is an instance of a *TEditCut* action, the *action* calls *Memo1*'s *CutToClipoard* method:

```
procedure TEditCut.ExecuteTarget(Target: TObject);
begin
  GetControl(Target).CutToClipboard;
end;
```

If the control were not an appropriate target, processing would continue as follows:

- ▪       *Form1* calls its *ExecuteAction* method. If Form1 is an appropriate target (for example, a form would be a target for the *TWindowCascade* action) then it calls *Cut1*'s *ExecuteTarget* method, passing itself as a parameter.
- ▪       If *Form1* is not an appropriate target, it invokes *ExecuteAction* on every visible control it owns until a target is found.

**Note:** If the action involved is a *TCustomAction* type, then the action is automatically disabled for you if, the action is not handled and, its *DisableIfNoHandler* property is *True*.

## Registering actions

You can register and unregister your own actions with the IDE by using the global routines in the
*ActnList* unit:

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);
procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
```

Use these routines the same way you would when registering components *RegisterComponents*). For
example, the following code registers the standard actions with the IDE:

```
{ Standard action registration }
RegisterActions('', [TAction], nil);
RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);
RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
```

## Writing action list editors

You may want to write your own component editor for action lists. If you do, you can assign your own procedures to the four global procedure variables in the *ActnList* unit:

```
CreateActionProc: function (AOwner: TComponent; ActionClass: TBasicActionClass):
TBasicAction = nil;

EnumRegisteredActionsProc: procedure(Proc: TEnumActionProc; Info: Pointer) = nil;

RegisterActionsProc:procedure (const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass) = nil;

UnRegisterActionsProc: procedure (const AClasses: array of TBasicActionClass) = nil;
```

You only need to reassign these if you want to manage the registration, unregistration, creation, and enumeration procedures of actions differently from the default behavior. If you do, write your own handlers and assign them to these variables within the initialization section of your design-time unit.

## Demo programs

For examples of programs that use actions and action lists, refer to Demos\RichEdit. In addition, the Application wizard (File|New Project page) demos, MDI Application, SDI Application, and Win95 Logo Application can use the action and action list objects.

# Implementing drag-and-drop in controls

Drag-and-drop is often a convenient way for users to manipulate objects. You can let users drag an entire control, or let them drag items from one control—such as a list box or tree view—into another.

- [Starting a drag operation](#)
- [Accepting dragged items](#)
- [Dropping items](#)
- [Ending a drag operation](#)
- [Customizing drag and drop with a drag object](#)
- [Changing the drag mouse pointer](#)

## Starting a drag operation

Every control has a property called *DragMode* that determines how drag operations are initiated. If *DragMode* is *dmAutomatic*, dragging begins automatically when the user presses a mouse button with the cursor on the control. Because *dmAutomatic* can interfere with normal mouse activity, you may want to set *DragMode* to *dmManual* (the default) and start the dragging by handling mouse-down events.

To start dragging a control manually, call the control's *BeginDrag* method. *BeginDrag* takes a Boolean parameter called *Immediate*. If you pass *True*, dragging begins immediately. If you pass *False*, dragging does not begin until the user moves the mouse a short distance. Calling BeginDrag(False) allows the control to accept mouse clicks without beginning a drag operation.

You can place other conditions on whether to begin dragging, such as checking which mouse button the user pressed, by testing the parameters of the mouse-down event before calling *BeginDrag*. The following code, for example, handles a mouse-down event in a file list box by initiating a drag operation only if the left mouse button was pressed.

```
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then  { drag only if left button pressed }
    with Sender as TFileListBox do  { treat Sender as TFileListBox }
      begin
        if ItemAtPos(Point(X, Y), True) >= 0 then  { is there an item here? }
          BeginDrag(False);  { if so, drag it }
      end;
end;
```

## Accepting dragged items

When the user drags something over a control, that control receives an *OnDragOver* event, at which time it must indicate whether it can accept the item if the user drops it there. The drag cursor changes to indicate whether the control can accept the dragged item. To accept items dragged over a control, attach an event handler to the control's *OnDragOver* event.

The drag-over event has a parameter called *Accept* that the event handler can set to *True* if it will accept the item. If *Accept* is *True*, the application sends a drag-drop event to the control.

The drag-over event has other parameters, including the source of the dragging and the current location of the mouse cursor, that the event handler can use to determine whether to accept the drop. In the following example, a directory tree view accepts dragged items only if they come from a file list box.

```
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TFileListBox then
    Accept := True;
  else
    Accept := False;
end;
```

## Dropping items

If a control indicates that it can accept a dragged item, it needs to handle the item should it be dropped. To handle dropped items, attach an event handler to the *OnDragDrop* event of the control accepting the drop. Like the drag-over event, the drag-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. The latter parameter allows you to monitor the path an item takes while being dragged; you might, for example, want to use this information to change the color of components as they are passed over.

In the following example, a directory tree view, accepting items dragged from a file list box, responds by moving files to the directory on which they are dropped.

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileList.FileName, Items[GetItem(X, Y)].FullPath);
end;
```

## Ending a drag operation

A drag operation ends when the item is either successfully dropped or released over a control that cannot accept it. At this point an end-drag event is sent to the control from which the item was dragged. To enable a control to respond when items have been dragged from it, attach an event handler to the control's *OnEndDrag* event.

The most important parameter in an *OnEndDrag* event is called *Target*, which indicates which control, if any, accepts the drop. If *Target* is **nil**, it means no control accepts the dragged item. The *OnEndDrag* event also includes the coordinates on the receiving control.

In this example, a file list box handles an end-drag event by refreshing its file list.

```
procedure TFMForm.FileList1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileList1.Update;
end;
```

## Customizing drag and drop with a drag object

You can use a *TDragObject* descendant to customize an object's drag-and-drop behavior. The standard drag-over and drag-drop events indicate the source of the dragged item and the coordinates of the mouse cursor over the accepting control. To get additional state information, derive a custom drag object from *TDragObject* and override its virtual methods. Create the custom drag object in the *OnStartDrag* event.

Normally, the source parameter of the drag-over and drag-drop events is the control that starts the drag operation. If different kinds of control can start an operation involving the same kind of data, the source needs to support each kind of control. When you use a descendant of *TDragObject*, however, the source is the drag object itself; if each control creates the same kind of drag object in its *OnStartDrag* event, the target needs to handle only one kind of object. The drag-over and drag-drop events can tell if the source is a drag object, as opposed to the control, by calling the *IsDragObject* function.

Drag objects let you drag items between a form implemented in the application's main EXE file and a form implemented in a DLL, or between forms that are implemented in different DLLs.

## Changing the drag mouse pointer

You can customize the appearance of the mouse pointer during drag operations by setting the source component's *DragCursor* property.

## Implementing drag-and-dock in controls

Descendants of *TWinControl* can act as docking sites and descendants of *TControl* can act as child windows that are docked into docking sites. For example, to provide a docking site at the left edge of a form window, align a panel to the left edge of the form and make the panel a docking site. When dockable controls are dragged to the panel and released, they become child controls of the panel.

- Making a windowed control a docking site
- Making a control a dockable child
- Controlling how child controls are docked
- Controlling how child controls are undocked
- Controlling how child controls respond to drag-and-dock operations

## Making a windowed control a docking site

To make a windowed control a docking site,

1   Set the *DockSite* property to *True*.

2   If the dock site object should not appear except when it contains a docked client, set its *AutoSize* property to *True*. When *AutoSize* is *True*, the dock site is sized to 0 until it accepts a child control for docking. Then it resizes to fit around the child control.

## Making a control a dockable child

To make a control a dockable child,

1  Set its *DragKind* property to *dkDock*. When *DragKind* is *dkDock*, dragging the control moves the control to a new docking site or undocks the control so that it becomes a floating window. When *DragKind* is *dkDrag* (the default), dragging the control starts a drag-and-drop operation which must be implemented using the *OnDragOver*, *OnEndDrag*, and *OnDragDrop* events.

2  Set its *DragMode* to *dmAutomatic*. When *DragMode* is *dmAutomatic*, dragging (for drag-and-drop or docking, depending on *DragKind*) is initiated automatically when the user starts dragging the control with the mouse. When *DragMode* is *dmManual*, you can still begin a drag-and-dock (or drag-and-drop) operation by calling the *BeginDrag* method.

3  Set its *FloatingDockSiteClass* property to indicate the *TWinControl* descendant that should host the control when it is undocked and left as a floating window. When the control is released and not over a docking site, a windowed control of this class is created dynamically, and becomes the parent of the dockable child. If the dockable child control is a descendant of *TWinControl*, it is not necessary to create a separate floating dock site to host the control, although you may want to specify a form in order to get a border and title bar. To omit a dynamic container window, set *FloatingDockSiteClass* to the same class as the control, and it will become a floating window with no parent.

## Controlling how child controls are docked

A docking site automatically accepts child controls when they are released over the docking site. For most controls, the first child is docked to fill the client area, the second splits that into separate regions, and so on. Page controls dock children into new tab sheets (or merge in the tab sheets if the child is another page control).

Three events allow docking sites to further constrain how child controls are docked:

```
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var
InfluenceRect: TRect; var CanDock: Boolean) of object;
```

*OnGetSiteInfo* occurs on the docking site when the user drags a dockable child over the control. It allows the site to indicate whether it will accept the control specified by the *DockClient* parameter as a child, and if so, where the child must be to be considered for docking. When *OnGetSiteInfo* occurs, *InfluenceRect* is initialized to the screen coordinates of the docking site, and *CanDock* is intialized to *True*. A more limited docking region can be created by changing *InfluenceRect* and the child can be rejected by setting *CanDock* to *False*.

```
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer;
State: TDragState; var Accept: Boolean) of object;
```

*OnDockOver* occurs on the docking site when the user drags a dockable child over the control. It is analogous to the *OnDragOver* event in a drag-and-drop operation. Use it to signal that the child can be released for docking, by setting the *Accept* parameter. If the dockable control is rejected by the *OnGetSiteInfo* event handler (perhaps because it is the wrong type of control), *OnDockOver* does not occur.

```
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer)
of object;
```

*OnDockDrop* occurs on the docking site when the user releases the dockable child over the control. It is analogous to the *OnDragDrop* event in a normal drag-and-drop operation. Use this event to perform any necessary accommodations to accepting the control as a child control. Access to the child control can be obtained using the *Control* property of the *TDockObject* specified by the *Source* parameter.

## Controlling how child controls are undocked

A docking site automatically allows child controls to be undocked when they are dragged and have a *DragMode* property of *dmAutomatic*. Docking sites can respond when child controls are dragged off, and even prevent the undocking, in an *OnUnDock* event handler:

```
property OnUnDock: TUnDockEvent;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean)    of
object;
```

The *Client* parameter indicates the child control that is trying to undock, and the *Allow* parameter lets the docking site (*Sender*) reject the undocking. When implementing an *OnUnDock* event handler, it can be useful to know what other children (if any) are currently docked. This information is available in the read-only *DockClients* property, which is an indexed array of *TControl*. The number of dock clients is given by the read-only *DockClientCount* property.

## Controlling how child controls respond to drag-and-dock operations

Dockable child controls have two events that occur during drag-and-dock operations: *OnStartDock*, analogous to the *OnStartDrag* event of a drag-and-drop operation, allows the dockable child control to create a custom drag object. *OnEndDock*, like *OnEndDrag*, occurs when the dragging terminates.

## Working with text in controls

The following topics how to use various features of rich edit and memo controls. Some of these features work with edit controls as well.

- Setting text alignment
- Adding scrollbars at runtime
- Adding the Clipboard object
- Selecting text
- Selecting all text
- Cutting, copying, and pasting text
- Deleting selected text
- Disabling menu items
- Providing a pop-up menu
- Handling the OnPopup event

## Setting text alignment

In a rich edit or memo component, text can be left- or right-aligned or centered. To change text alignment, set the edit component's *Alignment* property. Alignment takes effect only if the *WordWrap* property is *True*; if word wrapping is turned off, there is no margin to align to.

For example, the following code attaches an OnClick event handler to the Character|Left menu item, then attaches the same event handler to both the Right and Center menu items on the Character menu.

```
procedure TEditForm.AlignClick(Sender: TObject);
begin
  Left1.Checked := False;  { clear all three checks }
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True;  { check the item clicked }
  with Editor do  { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;
```

## Adding scroll bars at runtime

Rich edit and memo components can contain horizontal or vertical scroll bars, or both, as needed. When word-wrapping is enabled, the component needs only a vertical scroll bar. If the user turns off word-wrapping, the component might also need a horizontal scroll bar, since text is not limited by the right side of the editor.

To add scroll bars at runtime,

1   Determine whether the text might exceed the right margin. In most cases, this means checking whether word wrapping is enabled. You might also check whether any text lines actually exceed the width of the control.

2   Set the rich edit or memo component's *ScrollBars* property to include or exclude scroll bars.

The following example attaches an *OnClick* event handler to a Character|WordWrap menu item.

```
procedure TEditForm.WordWrap1Click(Sender: TObject);
begin
  with Editor do
  begin
    WordWrap := not WordWrap;  { toggle word-wrapping }
    if WordWrap then
      ScrollBars := ssVertical  { wrapped requires only vertical }
    else
      ScrollBars := ssBoth;  { unwrapped might need both }
      WordWrap1.Checked := WordWrap;  { check menu item to match property }
  end;
end;
```

The rich edit and memo components handle their scroll bars in a slightly different way. The rich edit component can hide its scroll bars if the text fits inside the bounds of the component. The memo always shows scroll bars if they are enabled.

## Adding the Clipboard object

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. The *Clipboard* object in Delphi encapsulates the Windows Clipboard and includes methods for cutting, copying, and pasting text (and other formats, including graphics). The *Clipboard* object is declared in the Clipbrd unit.

To add the Clipboard object to an application,

 1    Select the unit that will use the Clipboard.
 2    Search for the **implementation** reserved word.
 3    Add *Clipbrd* to the **uses** clause below **implementation**.

▪         If there is already a **uses** clause in the **implementation** part, add *Clipbrd* to the end of it.
▪         If there is not already a **uses** clause, add one that says

     **uses** Clipbrd;

For example, in an application with a child window, the uses clause in the unit's implementation part might look like this:

```
uses
   MDIFrame, Clipbrd;
```

# Selecting text

Before you can send any text to the Clipboard, that text must be selected. Highlighting of selected text is built into the edit components. When the user selects text, it appears highlighted.

The table below lists properties commonly used to handle selected text.

| Property | Description |
| --- | --- |
| SelText | Contains a string representing the selected text in the component. |
| SelLength | Contains the length of a selected string. |
| SelStart | Contains the starting position of a string. |

## Selecting all text

The *SelectAll* method selects the entire contents of the rich edit or memo component. This is especially useful when the component's contents exceed the visible area of the component. In most other cases, users select text with either keystrokes or mouse dragging.

To select the entire contents of a rich edit or memo control, call the *RichEdit1* control's *SelectAll* method.

For example,

```
procedure TMainForm.SelectAll(Sender: TObject);
begin
  RichEdit1.SelectAll;  { select all text in RichEdit }
end;
```

## Cutting, copying, and pasting text

Applications that use the *Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the Windows Clipboard. The edit components that encapsulate the standard Windows text-handling controls all have methods built into them for interacting with the Clipboard.

To cut, copy, or paste text with the Clipboard, call the edit component's *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods, respectively.

For example, the following code attaches event handlers to the *OnClick* events of the Edit|Cut, Edit|Copy, and Edit|Paste commands, respectively:

```
procedure TEditForm.CutToClipboard(Sender: TObject);
begin
  Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender: TObject);
begin
  Editor.PasteFromClipboard;
end;
```

# Deleting selected text

You can delete the selected text in an edit component without cutting it to the Clipboard. To do so, call the *ClearSelection* method. For example, if you have a Delete item on the Edit menu, your code could look like this:

```
procedure TEditForm.Delete(Sender: TObject);
begin
  RichEdit1.ClearSelection;
end;
```

## Disabling menu items

It is often useful to disable menu commands without removing them from the menu. For example, in a text editor, if there is no text currently selected, the Cut, Copy, and Delete commands are inapplicable. An appropriate time to enable or disable menu items is when the user selects the menu. To disable a menu item, set its Enabled property to *False*.

In the following example, an event handler is attached to the *OnClick* event for the Edit item on a child form's menu bar. It sets *Enabled* for the Cut, Copy, and Delete menu items on the Edit menu based on whether *RichEdit1* has selected text. The Paste command is enabled or disabled based on whether any text exists on the Clipboard.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;  { declare a temporary variable }
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);  {enable or disable the Paste
menu item}
  HasSelection := Editor.SelLength > 0;  { True if text is selected }
  Cut1.Enabled := HasSelection;  { enable menu items if HasSelection is True }
  Copy1.Enabled := HasSelection;
  Delete1.Enabled := HasSelection;
end;
```

The *HasFormat* method of the Clipboard returns a Boolean value based on whether the Clipboard contains objects, text, or images of a particular format. By calling *HasFormat* with the parameter CF_TEXT, you can determine whether the Clipboard contains any text, and enable or disable the Paste item as appropriate.

## Providing a pop-up menu

Pop-up, or local, menus are a common ease-of-use feature for any application. They enable users to minimize mouse movement by clicking the right mouse button in the application workspace to access a list of frequently used commands.

In a text editor application, for example, you can add a pop-up menu that repeats the Cut, Copy, and Paste editing commands. These pop-up menu items can use the same event handlers as the corresponding items on the Edit menu. You don't need to create accelerator or shortcut keys for pop-up menus because the corresponding regular menu items generally already have shortcuts.

A form's PopupMenu property specifies what pop-up menu to display when a user right-clicks any item on the form. Individual controls also have *PopupMenu* properties that can override the form's property, allowing customized menus for particular controls.

To add a pop-up menu to a form,

1  Place a pop-up menu component on the form.
2  Use the Menu Designer to define the items for the pop-up menu.
3  Set the *PopupMenu* property of the form or control that displays the menu to the name of the pop-up menu component.
4  Attach handlers to the *OnClick* events of the pop-up menu items.

## Handling the OnPopup event

You may want to adjust pop-up menu items before displaying the menu, just as you may want to enable or disable items on a regular menu. With a regular menu, you can handle the *OnClick* event for the item at the top of the menu.

With a pop-up menu, however, there is no top-level menu bar, so to prepare the pop-up menu commands, you handle the event in the menu component itself. The pop-up menu component provides an event just for this purpose, called OnPopup.

To adjust menu items on a pop-up menu before displaying them,

1   Select the pop-up menu component.
2   Attach an event handler to its *OnPopup* event.
3   Write code in the event handler to enable, disable, hide, or show menu items.

In the following code, the *EditEditClick* event handler described previously in <u>Disabling menu items</u> is attached to the pop-up menu component's *OnPopup* event. A line of code is added to *EditEditClick* for each item in the pop-up menu.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
  Paste2.Enabled := Paste1.Enabled;{Add this line}
  HasSelection := Editor.SelLength <> 0;
  Cut1.Enabled := HasSelection;
  Cut2.Enabled := HasSelection;  {Add this line}
  Copy1.Enabled := HasSelection;
  Copy2.Enabled := HasSelection; {Add this line}
  Delete1.Enabled := HasSelection;
end;
```

## Adding graphics to controls

Windows list-box, combo-box, and menu controls have a style available called "owner draw," which means that instead of using Windows' standard method of drawing text for each item in the control, the control's owner (generally, the form) draws each item at runtime. The most common use for owner-draw controls is to provide graphics instead of, or in addition to, text for items. For information on using owner-draw to add images to menus, see .Adding images to menu items.

All owner-draw controls contain lists of items. By default, those lists are lists of strings, which Windows displays as text. You can associate an object with each item in a list to make it easy to use that object when drawing items.

In general, creating an owner-draw control in Delphi involves these steps:

1  Setting the owner-draw style
2  Adding graphical objects to a string list
3  Drawing owner-drawn items

## Setting the owner-draw style

Both list boxes and combo boxes have a property called *Style. Style* determines whether the control uses the default drawing (called the "standard" style) or owner drawing. Grids use a property called *DefaultDrawing* to enable or disable the default drawing.

List boxes and combo boxes have additional owner-draw styles, called *fixed* and *variable*, as the following table describes. Owner-draw grids are always fixed: although the size of each row and column might vary, the size of each cell is determined before drawing the grid.

| Owner-draw style | Meaning | Examples |
|---|---|---|
| Fixed | Each item is the same height, with that height determined by the *ItemHeight* property. | *lbOwnerDrawFixed, csOwnerDrawFixed* |
| Variable | Each item might have a different height, determined by the data at runtime. | *lbOwnerDrawVariable, csOwnerDrawVariable* |

## Adding graphical objects to a string list

Every string list has the ability to hold a list of objects in addition to its list of strings.

For example, in a file manager application, you may want to add bitmaps indicating the type of drive along with the letter of the drive. To do that, you need to add the bitmap images to the application, then copy those images into the proper places in the string list as described in the following sections.

# Adding images to an application

An image control is a nonvisual control that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form. You can also use them to hold hidden images that you'll use in your application. For example, you can store bitmaps for owner-draw controls in hidden image controls, like this:

1  Add image controls to the main form.
2  Set their *Name* properties.
3  Set the *Visible* property for each image control to *False*.
4  Set the *Picture* property of each image to the desired bitmap using the Picture editor from the Object Inspector.

The image controls are invisible when you run the application.

# Adding images to a string list

Once you have graphical images in an application, you can associate them with the strings in a string list. You can either add the objects at the same time as the strings, or associate objects with existing strings. The preferred method is to add objects and strings at the same time, if all the needed data is available.

The following example shows how you might want to add images to a string list. This is part of a file manager application where, along with a letter for each valid drive, it adds a bitmap indicating each drive's type. The *OnCreate* event handler looks like this:

```
procedure TFMForm.FormCreate(Sender: TObject);
var
  Drive: Char;
  AddedIndex: Integer;
begin
  for Drive := 'A' to 'Z' do  { iterate through all possible drives }
  begin
    case GetDriveType(Drive + ':/') of  { positive values mean valid drives }
    DRIVE_REMOVABLE:  { add a tab }
      AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
    DRIVE_FIXED:  { add a tab }
      AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
    DRIVE_REMOTE:  { add a tab }
      AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
    end;
    if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then  { current drive? }
      DriveTabSet.TabIndex := AddedIndex;  { then make that current tab }
  end;
end;
```

## Drawing owner-drawn items

When you set a control's style to owner draw, Windows no longer draws the control on the screen. Instead, it generates events for each visible item in the control. Your application handles the events to draw the items.

To draw the items in an owner-draw control, do the following for each visible item in the control. Use a single event handler for all items.

1   Size the item, if needed.

Items of the same size (for example, with a list box style of *lsOwnerDrawFixed*), do not require sizing.

2   Draw the item.

## Sizing owner-draw items

Before giving your application the chance to draw each item in a variable owner-draw control, Windows generates a measure-item event. The measure-item event tells the application where the item appears on the control.

Windows determines the size the item (generally, it is just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle Windows chose. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to be the size of the bitmap. If you want a bitmap and text, adjust the rectangle to be big enough for both.

To change the size of an owner-draw item, attach an event handler to the measure-item event in the owner-draw control. Depending on the control, the name of the event can vary. List boxes and combo boxes use *OnMeasureItem*. Grids have no measure-item event.

The sizing event has two important parameters: the index number of the item and the size of that item. The size is variable: the application can make it either smaller or larger. The positions of subsequent items depend on the size of preceding items.

For example, in a variable owner-draw list box, if the application sets the height of the first item to five pixels, the second item starts at the sixth pixel down from the top, and so on. In list boxes and combo boxes, the only aspect of the item the application can alter is the height of the item. The width of the item is always the width of the control.

Owner-draw grids cannot change the sizes of their cells as they draw. The size of each row and column is set before drawing by the *ColWidths* and *RowHeights* properties.

The following code, attached to the *OnMeasureItem* event of an owner-draw list box, increases the height of each list item to accommodate its associated bitmap.

```
procedure TFMForm.DriveTabSetMeasureTab(Sender: TObject; Index: Integer;
  var TabWidth: Integer);  { note that TabWidth is a var parameter}
var
  BitmapWidth: Integer;
begin
  BitmapWidth := TBitmap(DriveTabSet.Tabs.Objects[Index]).Width;
  { increase tab width by the width of the associated bitmap plus two }
  Inc(TabWidth, 2 + BitmapWidth);
end;
```

**Note:** You must typecast the items from the *Objects* property in the string list. *Objects* is a property of type *TObject* so that it can hold any kind of object. When you retrieve objects from the array, you need to typecast them back to the actual type of the items.

# Drawing each owner-draw item

When an application needs to draw or redraw an owner-draw control, Windows generates draw-item events for each visible item in the control.

To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control.

The names of events for owner drawing always start with *OnDraw*, such as *OnDrawItem* or *OnDrawCell*.

The draw-item event contains parameters indicating the index of the item to draw, the rectangle in which to draw, and usually some information about the state of the item (such as whether the item has focus). The application handles each event by rendering the appropriate item in the given rectangle.

For example, the following code shows how to draw items in a list box that has bitmaps associated with each string. It attaches this handler to the *OnDrawItem* event for the list box:

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
  Draw(R.Left, R.Top + 4, Bitmap);  { draw bitmap }
  TextOut(R.Left + 2 + Bitmap.Width,  { position text }
    R.Top + 2, DriveTabSet.Tabs[Index]);  { and draw it to the right of the
bitmap }
  end;
end;
```

## Working with graphics and multimedia

Graphics and multimedia elements can add polish to your applications. Delphi offers a variety of ways to introduce these features into your application. To add graphical elements, you can insert pre-drawn pictures at design time, create them using graphical controls at design time, or draw them dynamically at runtime. To add multimedia capabilities, Delphi includes special components that can play audio and video clips.

This following topics describe how to enhance your applications by introducing graphics or multimedia elements:

- Overview of graphics programming
- Working with multimedia

## Overview of graphics programming

The VCL graphics components encapsulate the Windows Graphics Device Interface (GDI), making it very easy to add graphics to your Windows programming.

To draw graphics in a Delphi application, you draw on an object's *canvas*, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and it takes care of device context, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or metafiles. Canvases are available only at runtime, so you do all your work with canvases by writing code.

**Note:** Since *TCanvas* is a wrapper resource manager around the Windows device context, you can also use all Windows GDI functions on the canvas. The *Handle* property of the canvas *is* the device context Handle.

How graphic images appear in your application depends on the type of object whose canvas you draw on. If you are drawing directly onto the canvas of a control, the picture is displayed immediately. However, if you draw on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from the bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its *OnPaint* message.

When working with graphics, you often encounter the terms *drawing* and *painting*:

▪        Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.

▪        Painting is the creation of the entire appearance of an object. Painting usually involves drawing. That is, in response to *OnPaint* events, an object generally draws some graphics. An edit box, for example, paints itself by drawing a rectangle and then drawing some text inside. A shape control, on the other hand, paints itself by drawing a single graphic.

 The following topics describe how to use the VCL graphics components to simplify your coding.

▪        Refreshing the screen
▪        Types of graphic objects
▪        Common properties and methods of canvases
▪        Handling multiple drawing objects in an application
▪        Drawing on a bitmap
▪        Loading and saving graphics files
▪        Using the Clipboard with graphics
▪        Rubber banding example

# Refreshing the screen

At certain times, Windows determines that objects onscreen need to refresh their appearance, so it generates WM_PAINT messages, which the VCL routes to *OnPaint* events. The VCL calls any *OnPaint* event handler that you have written for that object when you use the *Refresh* method. The default name generated for the *OnPaint* event handler in a form is *FormPaint*. You may want to use the *Refresh* method at times to refresh a component or form. For example, you might call *Refresh* in the form's *OnResize* event handler to redisplay any graphics or if you want to paint a background on a form.

While some operating systems automatically handle the redrawing of the client area of a window that has been invalidated, Windows does not. In the Windows operating system anything drawn on the screen is permanent. When a form or control is temporarily obscured, for example during window dragging, the form or control must repaint the obscured area when it is re-exposed. For more information about the WM_PAINT message, see the Windows online Help.

If you use the *TImage* control, the painting and refreshing of the graphic contained in the *TImage* is handled automatically by the VCL. Drawing on a *TImage* creates a persistent image. Consequently, you do not need to do anything to redraw the contained image. In contrast, *TPaintBox*'s canvas maps directly onto the screen device, so that anything drawn to the *PaintBox*'s canvas is transitory. This is true of nearly all controls, including the form itself. Therefore, if you draw or paint on a *TPaintBox* in its constructor, you will need to add that code to your *OnPaint* event handler in order for image to be repainted each time the client area is invalidated.

## Types of graphic objects

The VCL provides the following graphic objects. These objects have methods to draw on the canvas, which are described in Using Canvas methods to draw graphic objects and to load and save to graphics files, as described in Loading and saving graphics files

| Object | Description |
| --- | --- |
| Picture | Used to hold any graphic image. To add additional graphic file formats, use the Picture *Register* method. Use this to handle arbitrary files such as displaying images in an image control. |
| Bitmap | A powerful graphics object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. Creating copies of a bitmap is fast since the *handle* is copied, not the image. |
| Clipboard | Represents the container for any text or graphics that are cut, copied, or pasted from or to an application. With the clipboard, you can get and retrieve data according to the appropriate format; handle reference counting, and opening and closing the Clipboard; manage and manipulate formats for objects in the Clipboard. |
| Icon | Represents the value loaded from a Windows icon file (::ICO file). |
| Metafile | Contains a metafile, which records the operations required to construct an image, rather than contain the actual bitmap pixels of the image. Metafiles are extremely scalable without the loss of image detail and often require much less memory than bitmaps, particularly for high-resolution devices, such as printers. However, metafiles do not draw as fast as bitmaps. Use a metafile when versatility or precision is more important than performance. |

## Common Properties and Methods of Canvas

The following table lists the commonly used properties of the Canvas object.

| Properties | Descriptions |
| --- | --- |
| Font | Specifies the font to use when writing text on the image. Set the properties of the TFont object to specify the font face, color, size, and style of the font. |
| Brush | Determines the color and pattern the canvas uses for filling graphical shapes and backgrounds. Set the properties of the TBrush object to specify the color and pattern or bitmap to use when filling in spaces on the canvas. |
| Pen | Specifies the kind of pen the canvas uses for drawing lines and outlining shapes. Set the properties of the TPen object to specify the color, style, width, and mode of the pen. |
| PenPos | Specifies the current drawing position of the pen. |
| Pixels | Specifies the color of the area of pixels within the current ClipRect. |

These properties are described in more detail in Using the properties of the Canvas object.

Here is a list of several methods you can use:

| Method | Descriptions |
| --- | --- |
| Arc | Draws an arc on the image along the perimeter of the ellipse bounded by the specified rectangle. |
| Chord | Draws a closed figure represented by the intersection of a line and an ellipse. |
| CopyRect | Copies part of an image from another canvas into the canvas. |
| Draw | Renders the graphic object specified by the Graphic parameter on the canvas at the location given by the coordinates (X, Y). |
| Ellipse | Draws the ellipse defined by a bounding rectangle on the canvas. |
| FillRect | Fills the specified rectangle on the canvas using the current brush. |
| FloodFill | Fills an area of the canvas using the current brush. |
| FrameRect | Draws a rectangle using the Brush of the canvas to draw the border. |
| LineTo | Draws a line on the canvas from PenPos to the point specified by X and Y, and sets the pen position to (X, Y). |
| MoveTo | Changes the current drawing position to the point (X,Y). |
| Pie | Draws a pie-shaped the section of the ellipse bounded by the rectangle (X1, Y1) and (X2, Y2) on the canvas. |
| Polygon | Draws a series of lines on the canvas connecting the points passed in and closing the shape by drawing a line from the last point to the first point. |
| PolyLine | Draws a series of lines on the canvas with the current pen, connecting each of the points passed to it in Points. |
| Rectangle | Draws a rectangle on the canvas with its upper left corner at the point (X1, Y1) and its lower right corner at the point (X2, Y2). Use *Rectangle* to draw a box using Pen and fill it using Brush. |

| | |
|---|---|
| RoundRect | Draws a rectangle with rounded corners on the canvas. |
| StretchDraw | Draws a graphic on the canvas so that the image fits in the specified rectangle. The graphic image may need to change its magnitude or aspect ratio to fit. |
| TextHeight, TextWidth | Returns the height and width, respectively, of a string in the current font. Height includes leading between lines. |
| TextOut | Writes a string on the canvas, starting at the point (X,Y), and then updates the PenPos to the end of the string. |
| TextRect | Writes a string inside a region; any portions of the string that fall outside the region do not appear. |

These methods are described in more detail in <u>Using Canvas methods to draw graphic objects</u>.

## Using the properties of the Canvas object

With the Canvas object, you can set the properties of a pen for drawing lines, a brush for filling shapes, a font for writing text, and an array of pixels to represent the image.

This section describes

- Using pens
- Using brushes
- Reading and setting pixels

## Using pens

The *Pen* property of a canvas controls the way lines appear, including lines drawn as the outlines of shapes. Drawing a straight line is really just changing a group of pixels that lie between two points.

The pen itself has four properties you can change: *Color*, *Width*, *Style*, and *Mode*.

- Color property: Changes the pen color
- Width property: Changes the pen width
- Style property: Changes the pen style
- Mode property: Changes the pen mode

The values of these properties determine how the pen changes the pixels in the line. By default, every pen starts out black, with a width of 1 pixel, a solid style, and a mode called copy that overwrites anything already on the canvas.

## Changing the pen color

You can set the color of a pen as you would any other *Color* property at runtime. A pen's color determines the color of the lines the pen draws, including lines drawn as the boundaries of shapes, as well as other lines and polylines. To change the pen color, assign a value to the *Color* property of the pen.

To let the user choose a new color for the pen, put a color grid on the pen's toolbar. A color grid can set both foreground and background colors. For a non-grid pen style, you must consider the background color, which is drawn in the gaps between line segments. Background color comes from the Brush color property.

Since the user chooses a new color by clicking the grid, this code changes the pen's color in response to the *OnClick* event:

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
  Canvas.Pen.Color := PenColor.ForegroundColor;
end;
```

## Changing the pen width

A pen's width determines the thickness, in pixels, of the lines it draws.

**Note:** When the thickness is greater than 1, Windows 95 always draw solid lines, no matter what the value of the pen's *Style* property.

To change the pen width, assign a numeric value to the pen's *Width* property.

Suppose you have a scroll bar on the pen's toolbar to set width values for the pen. And suppose you want to update the label next to the scroll bar to provide feedback to the user. Using the scroll bar's position to determine the pen width, you update the pen width every time the position changes.

This is how to handle the scroll bar's *OnChange* event:

```
procedure TForm1.PenWidthChange(Sender: TObject);
begin
  Canvas.Pen.Width := PenWidth.Position;  { set the pen width directly }
  PenSize.Caption := IntToStr(PenWidth.Position); { convert to string for caption }
end;
```

## Changing the pen style

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

**Note:** Windows 95 does not support dashed or dotted line styles for pens wider than one pixel and makes all larger pens solid, no matter what style you specify.

The task of setting the properties of pen is an ideal case for having different controls share same event handler to handle events. To determine which control actually got the event, you check the *Sender* parameter.

To create one click-event handler for six pen-style buttons on a pen's toolbar, do the following:

1   Select all six pen-style buttons and select the Object Inspector|Events|*OnClick* event and in the Handler column, type SetPenStyle.

   Delphi generates an empty click-event handler called *SetPenStyle* and attaches it to the *OnClick* events of all six buttons.

2   Fill in the click-event handler by setting the pen's style depending on the value of *Sender*, which is the control that sent the click event:

```
procedure TForm1.SetPenStyle(Sender: TObject);
begin
  with Canvas.Pen do
  begin
    if Sender = SolidPen then Style := psSolid
   else if Sender = DashPen then Style := psDash
   else if Sender = DotPen then Style := psDot
   else if Sender = DashDotPen then Style := psDashDot
   else if Sender = DashDotDotPen then Style := psDashDotDot
   else if Sender = ClearPen then Style := psClear;
  end;
end;
```

## Changing the pen mode

A pen's *Mode* property lets you specify various ways to combine the pen's color with the color on the canvas. For example, the pen could always be black, be an inverse of the canvas background color, inverse of the pen color, and so on.

## Getting the pen position

The current drawing position—the position from which the pen begins drawing its next line—is called the pen position. The canvas stores its pen position in its *PenPos* property. Pen position affects the drawing of lines only; for shapes and text, you specify all the coordinates you need.

To set the pen position, call the *MoveTo* method of the canvas. For example, the following code moves the pen position to the upper left corner of the canvas:

```
Canvas.MoveTo(0, 0);
```

**Note:** Drawing a line with the *LineTo* method also moves the current position to the endpoint of the line.

## Using brushes

The *Brush* property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way.

The brush has three properties you can manipulate:

- Color property: Changes the fill color
- Style property: Changes the brush style
- Bitmap property: Uses a bitmap as a brush pattern

The values of these properties determine the way the canvas fills shapes or other areas. By default, every brush starts out white, with a solid style and no pattern bitmap.

## Changing the brush color

A brush's color determines what color the canvas uses to fill shapes. To change the fill color, assign a value to the brush's *Color* property. Brush is used for background color in text and line drawing so you typically set the background color property.

You can set the brush color just as you do the pen color, in response to a click on a color grid on the brush's toolbar :

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
  Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
```

## Changing the brush style

A brush style determines what pattern the canvas uses to fill shapes. It lets you specify various ways to combine the brush's color with any colors already on the canvas. The predefined styles include solid color, no color, and various line and hatch patterns.

To change the style of a brush, set its *Style* property to one of the predefined values: *bsSolid*, *bsClear*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsBDiagonal*, *bsCross*, or *bsDiagCross*.

This example sets brush styles by sharing a click-event handler for a set of eight brush-style buttons. All eight buttons are selected, the Object Inspector|Events|*OnClick is set*, and the *OnClick* handler is named *SetBrushStyle*. Here is the handler code:

```
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
  with Canvas.Brush do
  begin
    if Sender = SolidBrush then Style := bsSolid
   else if Sender = ClearBrush then Style := bsClear
   else if Sender = HorizontalBrush then Style := bsHorizontal
   else if Sender = VerticalBrush then Style := bsVertical
   else if Sender = FDiagonalBrush then Style := bsFDiagonal
   else if Sender = BDiagonalBrush then Style := bsBDiagonal
   else if Sender = CrossBrush then Style := bsCross
   else if Sender = DiagCrossBrush then Style := bsDiagCross;
  end;
end;
```

## Setting the Brush Bitmap property

A brush's *Bitmap* property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas.

The following example loads a bitmap from a file and assigns it to the Brush of the Canvas of Form1:

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('MyBitmap.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end;
```

**Note:** The brush does not assume ownership of a bitmap object assigned to its *Bitmap* property. You must ensure that the Bitmap object remain valid for the lifetime of the Brush, and you must free the Bitmap object yourself afterwards.

## Reading and setting pixels

You will notice that every canvas has an indexed *Pixels* property that represents the individual colored points that make up the image on the canvas. You rarely need to access *Pixels* directly, it is available only for convenience to perform small actions such as finding or setting a pixel's color.

**Note:** Setting and getting individual pixels is thousands of times slower than performing graphics operations on regions. Do not use the Pixel array property to access the image pixels of a general array. For high-performance access to image pixels, see the *TBitmap.ScanLine* property.

## Using Canvas methods to draw graphic objects

This section shows how to use some common methods to draw graphic objects. It covers:

- Drawing lines and polylines
- Drawing shapes
- Drawing rounded rectangles
- Drawing polygons

## Drawing lines and polylines

A canvas can draw straight lines and polylines. A straight line is just a line of pixels connecting two points. A polyline is a series of straight lines, connected end-to-end. The canvas draws all lines using its pen.

## Drawing lines

To draw a straight line on a canvas, use the _LineTo_ method of the canvas.

_LineTo_ draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen.

For example, the following method draws crossed diagonal lines across a form whenever the form is painted:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
 begin
   MoveTo(0, 0);
   LineTo(ClientWidth, ClientHeight);
   MoveTo(0, ClientHeight);
   LineTo(ClientWidth, 0);
 end;
end;
```

## Drawing polylines

In addition to individual lines, the canvas can also draw polylines, which are groups of any number of connected line segments.

To draw a polyline on a canvas, call the *Polyline* method of the canvas.

The parameter passed to the *PolyLine* method is an array of points. You can think of a polyline as performing a *MoveTo* on the first point and *LineTo* on each successive point. For drawing multiple lines, *Polyline* is faster than using the *MoveTo* method and the *LineTo* method because it eliminates a lot of call overhead.

**Example: Drawing polylines**

The following method draws a rhombus in a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do
    PolyLine([Point(0, 0), Point(50, 0), Point(75, 50), Point(25, 50), Point(0, 0)]);
end;
```

This example takes advantage of Delphi's ability to create an open-array parameter on-the-fly. You can pass any array of points, but an easy way to construct an array quickly is to put its elements in brackets and pass the whole thing as a parameter.

## Drawing shapes

Canvases have methods for drawing different kinds of shapes. The canvas draws the outline of a shape with its pen, then fills the interior with its brush. The line that forms the border for the shape is controlled by the current *Pen* object.

This section covers:

- Drawing rectangles and ellipses
- Drawing rounded rectangles
- Drawing polygons

## Drawing rectangles and ellipses

To draw a rectangle or ellipse on a canvas, call the canvas's _Rectangle_ method or _Ellipse_ method, passing the coordinates of a bounding rectangle.

The _Rectangle_ method draws the bounding rectangle; _Ellipse_ draws an ellipse that touches all sides of the rectangle.

**Example: Drawing rectangles and ellipses**

The following method draws a rectangle filling a form's upper left quadrant, then draws an ellipse in the same area:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Rectangle(0, 0, ClientWidth div 2, ClientHeight div 2);
 Canvas.Ellipse(0, 0, ClientWidth div 2, ClientHeight div 2);
end;
```

## Drawing rounded rectangles

To draw a rounded rectangle on a canvas, call the canvas's *RoundRect* method.

The first four parameters passed to *RoundRect* are a bounding rectangle, just as for the *Rectangle* method or the *Ellipse* method. *RoundRect* takes two more parameters that indicate how to draw the rounded corners.

**Example: Drawing rounded rectangles**

The following method draws a rounded rectangle in a form's upper left quadrant, rounding the corners as sections of a circle with a diameter of 10 pixels:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.RoundRect(0, 0, ClientWidth div 2, ClientHeight div 2, 10, 10);
end;
```

## Drawing polygons

To draw a polygon with any number of sides on a canvas, call the *Polygon* method of the canvas.

*Polygon* takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, *Polygon* uses the brush to fill the area inside the polygon.

**Example: Drawing polygons**

The following code draws a right triangle in the lower left half of a form:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Polygon([Point(0, 0), Point(0, ClientHeight),
   Point(ClientWidth, ClientHeight)]);
end;
```

## Handling multiple drawing objects in your application

Various drawing methods (rectangle, shape, line, and so on) are typically available on the toolbar and button panel. Applications can respond to clicks on speed buttons to set the desired drawing objects. This section describes how to:

- Keep track of which drawing tool to use
- Changing the tool with speed buttons
- Using drawing tools

## Keeping track of which drawing tool to use

A graphics program needs to keep track of what kind of drawing tool (such as a line, rectangle, ellipse, or rounded rectangle) a user might want to use at any given time. You could assign numbers to each kind of tool, but then you would have to remember what each number stands for. You can do that more easily by assigning mnemonic constant names to each number, but your code won't be able to distinguish which numbers are in the proper range and of the right type. Fortunately, Object Pascal provides a means to handle both of these shortcomings. You can declare an enumerated type.

An enumerated type is really just a shorthand way of assigning sequential values to constants. Since it's also a type declaration, you can use Object Pascal's type-checking to ensure that you assign only those specific values.

To declare an enumerated type, use the reserved work type, followed by an identifier for the type, then an equal sign, and the identifiers for the values in the type in parentheses, separated by commas.

For example, the following code declares an enumerated type for each drawing tool available in a graphics application:

```
type
   TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
```

By convention, type identifiers begin with the letter T, and groups of similar constants (such as those making up an enumerated type) begin with a 2-letter prefix (such as *dt* for "drawing tool").

The declaration of the TDrawingTool type is equivalent to declaring a group of constants:

```
const
   dtLine = 0;
   dtRectangle = 1;
   dtEllipse = 2;
   dtRoundRect = 3;
```

The main difference is that by declaring the enumerated type, you give the constants not just a value, but also a type, which enables you to use Object Pascal's type-checking to prevent many errors. A variable of type TDrawingTool can be assigned only one of the constants dtLine..dtRoundRect. Attempting to assign some other number (even one in the range 0..3) generates a compile-time error.

In the following code, a field added to a form keeps track of the form's drawing tool:

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
 TForm1 = class(TForm)
    ... { method declarations }
 public
   Drawing: Boolean;
  Origin, MovePt: TPoint;
  DrawingTool: TDrawingTool;  { field to hold current tool }
  end;
```

## Changing the tool with speed buttons

Each drawing tool needs an associated *OnClick* event handler. Suppose your application had a toolbar button for each of four drawing tools: line, rectangle, ellipse, and rounded rectangle. You would attach the following event handlers to the *OnClick* events of the four drawing-tool buttons, setting *DrawingTool* to the appropriate value for each:

```
procedure TForm1.LineButtonClick(Sender: TObject); { LineButton }
begin
  DrawingTool := dtLine;
end;
procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
  DrawingTool := dtRectangle;
end;
procedure TForm1.EllipseButtonClick(Sender: TObject);  { EllipseButton }
begin
  DrawingTool := dtEllipse;
end;
procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }
begin
  DrawingTool := dtRoundRect;
end;
```

## Using drawing tools

Now that you can tell what tool to use, you must indicate how to draw the different shapes. The only methods that perform any drawing are the mouse-move and mouse-up handlers, and the only drawing code draws lines, no matter what tool is selected.

To use different drawing tools, your code needs to specify how to draw, based on the selected tool. You add this instruction to each tool's event handler.

This section describes

- Drawing shapes
- Sharing code among event handlers

## Drawing shapes

Drawing shapes is just as easy as drawing lines: Each one takes a single statement; you just need the coordinates.

Here's a rewrite of the *OnMouseUp* event handler that draws shapes for all four tools:

```
procedure TForm1.FormMouseUp(Sender: TObject);
begin
  case DrawingTool of
    dtLine:
      begin
        Canvas.MoveTo(Origin.X, Origin.Y);
        Canvas.LineTo(X, Y)
      end;
    dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
    dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
    dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                    (Origin.X - X) div 2, (Origin.Y - Y) div 2);
  end;
  Drawing := False;
end;
```

Of course, you also need to update the *OnMouseMove* handler to draw shapes:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor;
    case DrawingTool of
      dtLine: begin
                Canvas.MoveTo(Origin.X, Origin.Y);
                Canvas.LineTo(MovePt.X, MovePt.Y);
                Canvas.MoveTo(Origin.X, Origin.Y);
                Canvas.LineTo(X, Y);
              end;
      dtRectangle: begin
                     Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
                     Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
                   end;
      dtEllipse: begin
                   Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                   Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
                 end;
      dtRoundRect: begin
                     Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                       (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                     Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
                       (Origin.X - X) div 2, (Origin.Y - Y) div 2);
                   end;
    end;
    MovePt := Point(X, Y);
  end;
  Canvas.Pen.Mode := pmCopy;
end;
```

Typically, all the repetitious code that is in the above example would be in a separate routine. The next section shows all the shape-drawing code in a single routine that all mouse-event handlers can call.

## Sharing code among event handlers

Any time you find that many your event handlers use the same code, you can make your application more efficient by moving the repeated code into a routine that all event handlers can share.

To add a method to a form,
 1   Add the method declaration to the form object.

   You can add the declaration in either the **public** or **private** parts at the end of the form object's declaration. If the code is just sharing the details of handling some events, it's probably safest to make the shared method **private**.
 2   Write the method implementation in the implementation part of the form unit.

The header for the method implementation must match the declaration exactly, with the same parameters in the same order.

**Example: Sharing code among event handlers**

The following code adds a method to the form called *DrawShape* and calls it from each of the handlers.
First, the declaration of *DrawShape* is added to the form object's declaration:

```
type
  TForm1 = class(TForm)
    ... { fields and methods declared here}
 public
    { Public declarations }
   procedure DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
  end;
```

Then, the implementation of *DrawShape* is written in the implementation part of the unit:

```
implementation
{$R *.FRM}
... { other method implementations omitted for brevity }
procedure TForm1.DrawShape(TopLeft, BottomRight: TPoint; AMode: TPenMode);
begin
  with Canvas do
  begin
    Pen.Mode := AMode;
   case DrawingTool of
     dtLine:
       begin
         MoveTo(TopLeft.X, TopLeft.Y);
         LineTo(BottomRight.X, BottomRight.Y);
       end;
     dtRectangle: Rectangle(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
     dtEllipse: Ellipse(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y);
     dtRoundRect: RoundRect(TopLeft.X, TopLeft.Y, BottomRight.X, BottomRight.Y,
         (TopLeft.X - BottomRight.X) div 2, (TopLeft.Y - BottomRight.Y) div 2);
    end;
  end;
end;
```

The other event handlers are modified to call *DrawShape*.

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DrawShape(Origin, Point(X, Y), pmCopy); { draw the final shape }
 Drawing := False;
end;
procedure TForm1.FormMouseMove(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    DrawShape(Origin, MovePt, pmNotXor);  { erase the previous shape }
   MovePt := Point(X, Y); { record the current point }
    DrawShape(Origin, MovePt, pmNotXor); { draw the current shape }
  end;
end;
```

# Drawing on a graphic

You don't need any components to manipulate your application's graphic objects. You can construct, draw on, save, and destroy graphic objects without ever drawing anything on screen. In fact, your applications rarely draw directly on a form. More often, an application operates on graphics and then uses a VCL image control component to display the graphic on a form.

Once you move the application's drawing to the graphic in the image control, it is easy to add printing, Clipboard, and loading and saving operations for any graphic objects. graphic objects can be bitmap files, metafiles, icons or whatever other graphics classes that have been installed such as JPEG graphics.

**Note:** Because you are drawing on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from a bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its paint message. But if you are drawing directly onto the canvas property of a control, the picture object is displayed immediately.

## Making scrollable graphics

The graphic need not be the same size as the form: it can be either smaller or larger. By adding a scroll box control to the form and placing the graphic image inside it, you can display graphics that are much larger than the form or even larger than the screen. To add a scrollable graphic first you add a *TScrollBox* component and then you add the image control.

## Adding an image control

An image control is a container component that allows you to display your bitmap objects. You use an image control to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures.

 **Note:** Adding graphics to controls shows how to use graphics in controls.

## Placing the control

You can place an image control anywhere on a form. If you take advantage of the image control's ability to size itself to its picture, you need to set the top left corner only. If the image control is a nonvisible holder for a bitmap, you can place it anywhere, just as you would a nonvisual component.

If you drop the image control on a scroll box already aligned to the form's client area, this assures that the scroll box adds any scroll bars necessary to access offscreen portions of the image's picture. Then set the image control's properties.

## Setting the initial bitmap size

When you place an image control, it is simply a container. However, you can set the image control's *Picture* property at design time to contain a static graphic. The control can also load its picture from a file at runtime, as described in .

To create a blank bitmap when the application starts,
  1   Attach a handler to the *OnCreate* event for the form that contains the image.
  2   Create a bitmap object, and assign it to the image control's *Picture.Graphic* property.

In this example, the image is in the application's main form, *Form1*, so the code attaches a handler to *Form1*'s *OnCreate* event:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bitmap: TBitmap;  { temporary variable to hold the bitmap }
begin
  Bitmap := TBitmap.Create; { construct the bitmap object }
 Bitmap.Width := 200; { assign the initial width... }
 Bitmap.Height := 200;  { ...and the initial height }
  Image.Picture.Graphic := Bitmap;  { assign the bitmap to the image control }
end;
```

Assigning the bitmap to the picture's *Graphic* property gives ownership of the bitmap to the picture object. The picture object destroys the bitmap when it finishes with it, so you should not destroy the bitmap object. You can assign a different bitmap to the picture at which point the picture disposes of the old bitmap and assumes ownership of the new one.

If you run the application now, you see that client area of the form has a white region, representing the bitmap. If you size the window so that the client area cannot display the entire image, you'll see that the scroll box automatically shows scroll bars to allow display of the rest of the image. But if you try to draw on the image, you don't get any graphics, because the application is still drawing on the form, which is now behind the image and the scroll box.

## Drawing on the bitmap

To draw on a bitmap, use the image control's canvas and attach the mouse-event handlers to the appropriate events in the image control. Typically you would use region operations (fills, rectangles, polylines, and so on). These are fast and efficient methods of drawing.

An efficient way to draw images when you need to access individual pixels is to use the bitmap *ScanLine* property. For general-purpose usage, you can set up the bitmap pixel format to 24 bits and then treat the pointer returned from *ScanLine* as an array of RGB. Otherwise, you will need to know the native format of the *ScanLine* property. This example shows how to use *ScanLine* to get pixels one line at a time.

```
procedure TForm1.Button1Click(Sender: TObject);
// This example shows drawing directly to the Bitmap
var
  x,y : integer;
  Bitmap : TBitmap;
  P : PByteArray;
begin
  Bitmap := TBitmap.create;
  try
    Bitmap.LoadFromFile('C:\Program Files\Borland\Delphi
4\Images\Splash\256color\factory.bmp');
    for y := 0 to Bitmap.height -1 do
    begin
      P := Bitmap.ScanLine[y];
      for x := 0 to Bitmap.width -1 do
        P[x] := y;
    end;
  canvas.draw(0,0,Bitmap);
  finally
    Bitmap.free;
  end;
end;
```

## Loading and saving graphics files

Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. The VCL's image control makes it easy to load pictures from a file and save them again.

The VCL components you use to load, save, and replace graphic images support many graphic formats including bitmap files, metafiles, glyphs, and so on. They also support installable graphic classes.

The way to load and save graphics files is the similar to any other files and is described in these topics:

- Loading a picture from a file
- Saving a picture to a file
- Replacing the picture

## Loading a picture from a file

Your application should provide the ability to load a picture from a file if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can modify the picture.

To load a graphics file into an image control, call the *LoadFromFile* method of the image control's *Picture* object.

The following code gets a file name from an open-file dialog box, and then loads that file into an image control named *Image*:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    CurrentFile := OpenDialog1.FileName;
    Image.Picture.LoadFromFile(CurrentFile);
  end;
end;
```

## Saving a picture to a file

The VCL picture object can load and save graphics in several formats, and you can create and register your own graphic-file formats so that picture objects can load and store them as well.

To save the contents of an image control in a file, call the *SaveToFile* method of the image control's *Picture* object.

The *SaveToFile* method requires the name of a file in which to save. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to get a file name from the user before saving, as shown in the next section.

The following pair of event handlers, attached to the File|Save and File|Save As menu items, respectively, handle the resaving of named files, saving of unnamed files, and saving existing files under new names.

```
procedure TForm1.Save1Click(Sender: TObject);
begin
  if CurrentFile <> '' then
    Image.Picture.SaveToFile(CurrentFile) { save if already named }
 else SaveAs1Click(Sender); { otherwise get a name }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
  if SaveDialog1.Execute then  { get a file name }
  begin
    CurrentFile := SaveDialog1.FileName;  { save the user-specified name }
   Save1Click(Sender);  { then save normally }
  end;
end;
```

## Replacing the picture

You can replace the picture in an image control at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one.

To replace the picture in an image control, assign a new graphic to the image control's *Picture* object.

Creating the new graphic is the same process you used to create the initial graphic , but you should also provide a way for the user to choose a size other than the default size used for the initial graphic. An easy way to provide that option is to present a dialog box.

With such a dialog box in your project, add it to the uses clause in the unit for your main form. You can then attach an event handler to the File|New menu item's *OnClick* event. Here's an example:

```
procedure TForm1.New1Click(Sender: TObject);
var
  Bitmap: TBitmap;  { temporary variable for the new bitmap }
begin
  with NewBMPForm do
  begin
    ActiveControl := WidthEdit;  { make sure focus is on width field }
    WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width); { use current
dimensions... }
    HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height); { ...as default }
    if ShowModal <> idCancel then { continue if user doesn't cancel dialog box }
     begin
       Bitmap := TBitmap.Create;  { create fresh bitmap object }
      Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
      Bitmap.Height := StrToInt(HeightEdit.Text); { use specified height }
      Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
      CurrentFile := ''; { indicate unnamed file }
     end;
  end;
end;
```

**Note:** Assigning a new bitmap to the picture object's *Graphic* property causes the picture object to destroy the existing bitmap and take ownership of the new one. The VCL handles the details of freeing the resources associated with the previous bitmap automatically.

## Using the Clipboard with graphics

You can use the Windows Clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The VCL's Clipboard object makes it easy to handle different kinds of information, including graphics.

Before you can use the Clipboard object in your application, you must add the Clipbrd unit to the uses clause of any unit that needs to access Clipboard data.

## Copying graphics to the Clipboard

You can copy any picture, including the contents of image controls, to the Clipboard. Once on the Clipboard, the picture is available to all Windows applications.

To copy a picture to the Clipboard, assign the picture to the Clipboard object using the *Assign* method.

This code shows how to copy the picture from an image control named *Image* to the Clipboard in response to a click on an Edit|Copy menu item:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
```

## Cutting graphics to the Clipboard

Cutting a graphic to the Clipboard is exactly like copying it, but you also erase the graphic from the source.

To cut a graphic from a picture to the Clipboard, first copy it to the Clipboard, then erase the original.

In most cases, the only issue with cutting is how to show that the original image is erased. Setting the area to white is a common solution, as shown in the following code that attaches an event handler to the *OnClick* event of the Edit|Cut menu item:

```
procedure TForm1.Cut1Click(Sender: TObject);
var
  ARect: TRect;
begin
  Copy1Click(Sender); { copy picture to Clipboard }
 with Image.Canvas do
  begin
   CopyMode := cmWhiteness; { copy everything as white }
   ARect := Rect(0, 0, Image.Width, Image.Height); { get bitmap rectangle }
   CopyRect(ARect, Image.Canvas, ARect);  { copy bitmap over itself }
   CopyMode := cmSrcCopy; { restore normal mode }
  end;
end;
```

## Pasting graphics from the Clipboard

If the Windows Clipboard contains a bitmapped graphic, you can paste it into any image object, including image controls and the surface of a form.

To paste a graphic from the Clipboard,

1  Call the Clipboard's _HasFormat_ method to see whether the Clipboard contains a graphic.

   _HasFormat_ is a Boolean function. It returns _True_ if the Clipboard contains an item of the type specified in the parameter. To test for graphics, you pass CF_BITMAP.

2  Assign the Clipboard to the destination.

This code shows how to paste a picture from the Clipboard into an image control in response to a click on an Edit|Paste menu item:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
  Bitmap: TBitmap;
begin
 if Clipboard.HasFormat(CF_BITMAP) then { is there a bitmap on the Clipboard? }
 begin
   Image.Picture.Bitmap.Assign(Clipboard);
 end;
end;
```

The graphic on the Clipboard could come from this application, or it could have been copied from another application, such as Windows Paintbrush. You do not need to check the clipboard format in this case because the paste menu should be disabled when the clipboard does not contain a supported format.

## Rubber banding example

This section walks you through the details of implementing the "rubber banding" effect in an graphics application that tracks mouse movements as the user draws a graphic at runtime. The example code in this section is taken from a sample application located in the EXAMPLES\DOC\GRAPHEX directory. The application draws lines and shapes on a window's canvas in response to clicks and drags: pressing a mouse button starts drawing, and releasing the button ends the drawing.

To start with, the example code shows how to draw on the surface of the main form. Later examples demonstrate drawing on a bitmap.

This section covers:

- Responding to the mouse
- Adding a field to a form object to track mouse actions
- Refining line drawing

## Responding to the mouse

Your application can respond to the mouse actions: mouse-button down, mouse moved, and mouse-button up. It can also respond to a click (a complete press-and-release, all in one place) that can be generated by some kinds of keystrokes (such as pressing *Enter* in a modal dialog box).

This section covers:

- What's in a mouse event
- Responding to a mouse-down action
- Responding to a mouse-up action
- Responding to a mouse move

## What's in a mouse event

The VCL has three mouse events: _OnMouseDown_ event, _OnMouseMove_ event, and _OnMouseUp_ event.

When a VCL application detects a mouse action, it calls whatever event handler you've defined for the corresponding event, passing five parameters. Use the information in those parameters to customize your responses to the events. The five parameters are as follows:

| Parameter | Meaning |
| --- | --- |
| _Sender_ | The object that detected the mouse action |
| _Button_ | Indicates which mouse button was involved: _mbLeft_, _mbMiddle_, or _mbRight_ |
| _Shift_ | Indicates the state of the _Alt_, _Ctrl_, and _Shift_ keys at the time of the mouse action |
| X, Y | The coordinates where the event occurred |

Most of the time, you need the coordinates returned in a mouse-event handler, but sometimes you also need to check _Button_ to determine which mouse button caused the event.

**Note:** Delphi uses the same criteria as Microsoft Windows in determining which mouse button has been pressed. Thus, if you have switched the default "primary" and "secondary" mouse buttons (so that the right mouse button is now the primary button), clicking the primary (right) button will record _mbLeft_ as the value of the _Button_ parameter.

## Responding to a mouse-down action

Whenever the user presses a button on the mouse, an _OnMouseDown_ event goes to the object the pointer is over. The object can then respond to the event.

To respond to a mouse-down action, attach an event handler to the *OnMouseDown* event.

The VCL generates an empty handler for a mouse-down event on the form:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
end;
```

Here's code that displays some text at the point where the mouse button is pressed. It uses the X and Y parameters sent to the method, and calls the *TextOut* method of the canvas to display text there:

**Example: Responding to a mouse-down action**

The following code displays the string 'Here!' at the location on a form clicked with the mouse:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.TextOut(X, Y, 'Here!');  { write text at (X, Y) }
end;
```

When the application runs, you can press the mouse button down with the mouse cursor on the form and have the string, "Here!" appear at the point clicked. This code sets the current drawing position to the coordinates where the user presses the button:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(X, Y); { set pen position }
end;
```

Pressing the mouse button now sets the pen position, setting the line's starting point. To draw a line to the point where the user releases the button, you need to respond to a mouse-up event.

## Responding to a mouse-up action

An *OnMouseUp* event occurs whenever the user releases a mouse button. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form.

To respond to mouse-up actions, define a handler for the *OnMouseUp* event.

**Example: Responding to a mouse-up action**

Here's a simple *OnMouseUp* event handler that draws a line to the point of the mouse-button release:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);  { draw line from PenPos to (X, Y) }
end;
```

This code lets a user draw lines by clicking, dragging, and releasing. In this case, the user cannot see the line until the mouse button is released.

## Responding to a mouse move

An *OnMouseMove* event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button. This allows you to give the user some intermediate feedback by drawing temporary lines while the mouse moves.

To respond to mouse movements, define an event handler for the *OnMouseMove* event. This example uses mouse-move events to draw intermediate shapes on a form while the user holds down the mouse button, thus providing some feedback to the user. The *OnMouseMove* event handler draws a line on a form to the location of the *OnMouseMove* event:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);  { draw line to current position }
end;
```

With this code, moving the mouse over the form causes drawing to follow the mouse, even before the mouse button is pressed.

Mouse-move events occur even when you haven't pressed the mouse button.

If you want to track whether there is a mouse button pressed, you need to add an object field to the form object.

## Adding a field to a form object to track mouse actions

To track whether a mouse button was pressed, you must add an object field to the form object. When you add a component to a form, Delphi adds a field that represents that component to the form object, so that you can refer to the component by the name of its field. You can also add your own fields to forms by editing the type declaration in the form unit's header file.

In the following example, the form needs to track whether the user has pressed a mouse button. To do that, it adds a Boolean field and sets its value when the user presses the mouse button.

To add a field to an object, edit the object's type definition, specifying the field identifier and type after the **public** directive at the bottom of the declaration.

Delphi "owns" any declarations before the **public** directive: that's where it puts the fields that represent controls and the methods that respond to events.

**Example: Adding a field to a form object to track mouse actions**

The following code gives a form a field called *Drawing* of type Boolean, in the form object's declaration. It also adds two fields to store points *Origin* and *MovePt* of typeTPoint.

```
type
  TForm1 = class(TForm)
   procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
   procedure FormMouseMove(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  public
     Drawing: Boolean; { field to track whether button was pressed }
     Origin, MovePt: TPoint; { fields to store points }
   end;
```

When you have a *Drawing* field to track whether to draw, set it to *True* when the user presses the mouse button, and *False* when the user releases it:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;  { set the Drawing flag }
 Canvas.MoveTo(X, Y);
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.LineTo(X, Y);
 Drawing := False;  { clear the Drawing flag }
end;
```

Then you can modify the *OnMouseMove* event handler to draw only when *Drawing* is *True*:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then{ only draw if Drawing flag is set }
   Canvas.LineTo(X, Y);
end;
```

This results in drawing only between the mouse-down and mouse-up events, but you still get a scribbled line that tracks the mouse movements instead of a straight line.

The problem is that each time you move the mouse, the mouse-move event handler calls *LineTo*, which moves the pen position, so by the time you release the button, you've lost the point where the straight line was supposed to start.

## Refining line drawing

With fields in place to track various points, you can refine an application's line drawing.

## Tracking the origin point

When drawing lines, track the point where the line starts with the *Origin* field.

*Origin* must be set to the point where the mouse-down event occurs, so the mouse-up event handler can use *Origin* to place the beginning of the line, as in this code:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
 Canvas.MoveTo(X, Y);
 Origin := Point(X, Y); { record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(Origin.X, Origin.Y); { move pen to starting point }
 Canvas.LineTo(X, Y);
 Drawing := False;
end;
```

Those changes get the application to draw the final line again, but they do not draw any intermediate actions--the application does not yet support "rubber banding."

## Tracking movement

The problem with this example as the *OnMouseMove* event handler is currently written is that it draws the line to the current mouse position from the last *mouse position,* not from the original position. You can correct this by moving the drawing position to the origin point, then drawing to the current point:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.MoveTo(Origin.X, Origin.Y);  { move pen to starting point }
    Canvas.LineTo(X, Y);
  end;
end;
```

The above tracks the current mouse position, but the intermediate lines do not go away, so you can hardly see the final line. The example needs to erase each line before drawing the next one, by keeping track of where the previous one was. The *MovePt* field allows you to do this.

*MovePt* must be set to the endpoint of each intermediate line, so you can use *MovePt* and *Origin* to erase that line the next time a line is drawn:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Drawing := True;
  Canvas.MoveTo(X, Y);
  Origin := Point(X, Y);
  MovePt := Point(X, Y); { keep track of where this move was }
end;
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Drawing then
  begin
    Canvas.Pen.Mode := pmNotXor; { use XOR mode to draw/erase }
    Canvas.MoveTo(Origin.X, Origin.Y);{ move pen back to origin }
    Canvas.LineTo(MovePt.X, MovePt.Y);{ erase the old line }
    Canvas.MoveTo(Origin.X, Origin.Y);{ start at origin again }
    Canvas.LineTo(X, Y); { draw the new line }
  end;
  MovePt := Point(X, Y); { record point for next move }
  Canvas.Pen.Mode := pmCopy;
end;
```

Now you get a "rubber band" effect when you draw the line. By changing the pen's mode to *pmNotXor*, you have it combine your line with the background pixels. When you go to erase the line, you're actually setting the pixels back to the way they were. By changing the pen mode back to *pmCopy* (its default value) after drawing the lines, you ensure that the pen is ready to do its final drawing when you release the mouse button.

## Working with multimedia

Delphi allows you to add multimedia components to your applications. To do this, you can use either the *TAnimate* component on the Win32 page or the *TMediaPlayer* component on the System page of the Component palette. Use the animate component when you want to add silent video clips to your application. Use the media player component when you want to add audio and/or video clips to an application.


The following topics are discussed in this section:

- Adding silent video clips to an application
- Adding audio and/or video clips to an application

## Adding silent video clips to an application

The animation control in Delphi allows you to add silent video clips to your application.

To add a silent video clip to an application:

1  Double-click the animate icon on the Win32 page of the Component palette. This automatically puts an animation control on the form window in which you want to display the video clip.

2  Using the Object Inspector, select the *Name* property and enter a new*name* for your animation control. You will use this name when you call the animation control. (Follow the standard rules for naming Delphi identifiers).

   Always work directly with the Object Inspector when setting design time properties and creating event handlers.

3   Do one of the following:

▪        Select the *Common* AVI property and choose one of the AVIs available from the drop down list; or

▪        Select the *FileName* property and click the ellipsis (...) button, choose an AVI file from any available local or network directories and click Open in the Open AVI dialog; or

▪        Select the resource of an AVI using the *ResName* or *ResID* properties. Use *ResHandle* to indicate the module that contains the resource identified by *ResName* or *ResID*.

   This loads the AVI file into memory. If you want to display the first frame of the AVI clip on-screen until it is played using the *Active* property or the *Play* method, then set the *Open* property to *True*.

4  Set the *Repetitions* property to the number of times you want to the AVI clip to play. If this value is 0, then the sequence is repeated until the *Stop* method is called.

5  Make any other changes to the animation control settings. For example, *if* you want to change the first frame displayed when animation control opens, then set the *StartFrame*property to the desired frame value.

6  Set the *Active* property to *True* using the drop down list or write an event handler to run the AVI clip when a specific event takes place at runtime. For example, to activate the AVI clip when a button object is clicked, write the button's *OnClick* event specifying that. You may also call the *Play* method to specify when to play the AVI.

**Note:** If you make any changes to the form or any of the components on the form after setting *Active* to *True*, the *Active* property becomes *False* and you have to reset it to *True*. Do this either just before runtime or at runtime.

For more information on using the animation control, see the topic called Example of adding silent video clips.

## Example of adding silent video clips

Suppose you want to display an animated logo as the first screen that appears when your application starts. After the logo finishes playing the screen disappears.

To run this example, create a new project and save the Unit1.pas file as Frmlogo.pas and save the Project1.dpr file as Logo.dpr. Then:

1  Double-click the animate icon from the Win32 page of the Component palette.
2  Using the Object Inspector, set its Name property to *Logo1*.
3  Select its FileName property, click the ellipsis (...) button, choose the cool.avi file from your .. \Demos\Coolstuf directory. Then click Open in the Open AVI dialog.

   This loads the cool.avi file into memory.
4  Position the animation control box on the form by clicking and dragging it to the top right hand side of the form.
5  Set its Repetitions property to 5.
6  Click the form to bring focus to it and set its Name property to *LogoForm1* and its Caption property to *Logo Window*. Now decrease the height of the form to right- center the animation control on it.
7  Double-click the form's *OnActivate* event and write the following code to run the AVI clip when the form is in focus at runtime:

```
Logo1.Active := True;
```
8  Double-click the Label icon on the Standard page of the Component palette. Select its Caption property and enter *Welcome to Cool Images 4.0*. Now select its Font property, click the ellipsis (...) button and choose Font Style: Bold, Size: 18, Color: Navy from the Font dialog and click OK. Click and drag the label control to center it on the form.
9   Click the animation control to bring focus back to it. Double-click its *OnStop* event and write the following code to close the form when the AVI file stops:

```
LogoForm1.Close;
```
10  Select Run|Run to execute the animated logo window.

# Adding audio and/or video clips to an application

The media player component in Delphi allows you to add audio and/or video clips to your application. It opens a media device and plays, stops, pauses, records, etc., the audio and/or video clips used by the media device. The media device may be hardware or software.

To add an audio and/or video clip to an application:

1  Double-click the media player icon on the System page of the Component palette. This automatically put a media player control on the form window in which you want the media feature.

2  Using the Object Inspector, select the _Name_ pr*operty* and enter a new name for your media player control. You will use this when you call the media player control. (Follow the standard rules for naming Delphi identifiers.)

    Always work directly with the Object Inspector when setting design time properties and creating event handlers.

3  Select the _DeviceType_ property and choose the appropriate device type to open using the _AutoOpen_ property or the _Open_ method. (If _DeviceType_ is dtAutoSelect the device type is selected based on the file extension of the media file specified by the _FileName_ property.) For more information on device types and their functions, see the table below.

4   If the device stores its media in a file, specify the name of the media file using the _FileName_ property. Select the _FileName_ property, click the ellipsis (...) button, and choose a media file from any available local or network directories and click Open in the Open dialog. Otherwise, insert the hardware the media is stored in (disk, cassette, and so on) for the selected media device, at runtime.

5  Set the _AutoOpen_ property to _True_. This way the media player automatically opens the specified device when the form containing the media player control is created at runtime. If _AutoOpen_ is _False_, the device must be opened with a call to the _Open_ method.

6  Set the _AutoEnable_ property to _True_ to automatically enable or disable the media player buttons as required at runtime; or, double-click the _EnabledButtons_ property to set each button to _True_ or _False_ depending on which ones you want to enable or disable.

    The multimedia device is played, paused, stopped, and so on when the user clicks the corresponding button on the media player component. The device can also be controlled by the methods that correspond to the buttons (Play, Pause, Stop, Next, Previous, and so on).

7  Position the media player control bar on the form by either clicking and dragging it to the appropriate place on the form or by selecting the _Align_ property and choosing the appropriate align position from the drop down list.

    If you want the media player to be invisible at runtime, set the _Visible_ property to _False_ and control the device by calling the appropriate methods (_Play_, _Pause_, _Stop_, _Next_, _Previous_, _Step_, _Back_, _Start Recording_, _Eject_).

8  Make any other changes to the media player control settings. For example, if the media requires a display window, set the _Display_ property to the control that displays the media. If the device uses multiple tracks, set the _Tracks_ property to the desired track.

| Device Type | Software/Hardware used | Plays | Uses Tracks |
|---|---|---|---|
| dtAVIVideo | AVI Video Player for Windows | AVI Video files | No |
| dtCDAudio | CD Audio Player for Windows or a CD Audio Player | CD Audio Disks | Yes |
| dtDAT | Digital Audio Tape Player | Digital Audio Tapes | Yes |
| dtDigitalVideo | Digital Video Player for Windows | AVI, MPG, MOV files | No |
| dtMMMovie | MM Movie Player | MM film | No |
| dtOverlay | Overlay device | Analog Video | No |
| dtScanner | Image Scanner | N/A for Play (scans images on Record) | No |

| dtSequencer | MIDI Sequencer for Windows | MIDI files | Yes |
| dtVCR | Video Cassette Recorder | Video Cassettes | No |
| dtWaveAudio | Wave Audio Player for Windows | WAV files | No |

For more information on using the media player control, see the topic called <u>Example of adding audio and/or video clips</u>.

## Example of adding audio and/or video clips

This example runs an AVI video clip of a multimedia advertisement for Delphi. To run this example, create a new project and save the Unit1.pas file to FrmAd.pas and save the Project1.dpr file to DelphiAd.dpr. Then:

1 Double-click the media player icon on the System page of the Component palette.
2 Using the Object Inspector, set the Name property of the media player to *VideoPlayer1*.
3 Select its DeviceType property and choose dtAVIVideo from the drop down list.
4 Select its FileName property, click the ellipsis (...) button,   choose the speedis.avi file from your .. \Demos\Coolstuf directory. Click Open in the Open dialog.
5 Set its AutoOpen property to *True* and its Visible property to *False*.
6 Double-click the Animate icon from the Win32 page of the Component palette. Set its AutoSize property to *False*, its Height property to *175* and Width property to *200*. Click and drag the animation control to the top left corner of the form.
7 Click the media player to bring back focus to it. Select its Display property and choose Animate1 from the drop down list.
8 Click the form to bring focus to it and select its Name property and enter *Delphi_Ad*. Now resize the form to the size of the animation control.
9 Double-click the form's *OnActivate* event and write the following code to run the AVI video when the form is in focus:

```
Videoplayer1.Play;
```

10 Choose Run|Run to execute the AVI video.

## Using threads

The VCL provides several objects that make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by

▪        **Avoiding bottlenecks.** With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.

▪        **Organizing program behavior.** Often, a program's behavior can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases. Use threads to assign priorities to various program tasks so that you can give more CPU time to more critical tasks.

▪        **Multiprocessing.** If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

**Note:** Not all operating systems implement true multi-processing, even when it is supported by the underlying hardware. For example Windows 95 only simulates multiprocessing, even if the underlying hardware supports it.

The following topics discuss support for threads in Delphi:

▪        Defining thread objects
▪        Coordinating threads
▪        Executing thread objects
▪        Using threads in distributed applications
▪        Debugging multi-threaded applications

## Defining thread objects

For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads.

**Note:** Thread objects do not allow you to control the security attributes or stack size of your threads. If you need to control these, you must use the _BeginThread_ function. Even when using _BeginThread_, you can still benefit from some of the thread synchronization objects and methods described in Coordinating threads.

To use a thread object in your application, you must create a new descendant of _TThread_. To create a descendant of _TThread_, choose File|New from the main menu. In the new objects dialog box, select Thread Object. You are prompted to provide a class name for your new thread object. After you provide the name, Delphi creates a new unit file to implement the thread.

**Note:** Unlike most dialog boxes in the IDE that require a class name, the New Thread Object dialog does not automatically prepend a 'T' to the front of the class name you provide.

The automatically generated file contains the skeleton code for your new thread object. If you named your thread _TMyThread_, it would look like the following:

```
unit Unit2;
interface
uses
  Classes;
type
  TMyThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
  { Place thread code here }
end;
end.
```

In the automatically generated unit file, you

- Optionally, Initialize the thread.
- Write the thread function by filling in the Execute method.
- Optionally, Write clean-up code

## Initializing the thread

If you want to write initialization code for your new thread class, you must override the Create method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation. This is where you can assign a default priority for your thread and indicate whether it should be freed automatically when it finishes executing.

### Assigning a default priority

Priority indicates how much preference the thread gets when the operating system schedules CPU time among all the threads in your application. Use a high priority thread to handle time critical tasks, and a low priority thread to perform other tasks. To indicate the priority of your thread object, set the! *Priority* property. *Priority* values fall along a seven point scale, as described in the following table:

| Value | Priority |
|---|---|
| tpIdle | The thread executes only when the system is idle. Windows won't interrupt other threads to execute a thread with *tpIdle* priority. |
| tpLowest | The thread's priority is two points below normal. |
| tpLower | The thread's priority is one point below normal. |
| tpNormal | The thread has normal priority. |
| tpHigher | The thread's priority is one point above normal. |
| tpHighest | The thread's priority is two points above normal. |
| tpTimeCritical | The thread gets highest priority. |

**Warning:** Boosting the thread priority of a CPU intensive operation may "starve" other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

The following code shows the constructor of a low-priority thread that performs background tasks which should not interfere with the rest of the application's performance:

```
constructor TMyThread.Create(CreateSuspended: Boolean);
{
inheritedCreate(CreateSuspended);
  Priority := tpIdle;
}
```

### Indicating when threads are freed

Usually, when threads finish their operation, they can simply be freed. In this case, it is easiest to let the thread object free itself. To do this, set the *FreeOnTerminate* property to *True*.

There are times, however, when the termination of a thread must be coordinated with other threads. For example, you may be waiting for one thread to return a value before performing an action in another thread. To do this, you do not want to free the first thread until the second has received the return value. You can handle this situation by setting *FreeOnTerminate* to *False* and then explicitly freeing the first thread from the second.

## Writing the thread function

The *Execute* method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program because you must make sure that you don't overwrite memory that is used by other threads in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

When implementing the Execute method, you can manage these issues by

- Using the main VCL thread.
- Using thread-local variables.
- Avoiding simultaneous access.
- Waiting for other threads.
- Checking for termination by other threads.

## Using the main VCL thread

When you use objects from the VCL object hierarchy, their properties and methods are not guaranteed to be thread-safe. That is, accessing properties or executing methods may perform some actions that use memory which is not protected from the actions of other threads. Because of this, a main VCL thread is set aside for access of VCL objects. This is the thread that handles all Windows messages received by components in your application.

If all objects access their properties and execute their methods within this single thread, you need not worry about your objects interfering with each other. To use the main VCL thread, create a separate routine that performs the required actions. Call this separate routine from within your thread's *Synchronize* method. For example:

```
procedure TMyThread.PushTheButton;
begin
  Button1.Click;
end;
procedure TMyThread.Execute;
begin
  ...
  Synchronize(PushTheButton);
  ...
end;
```

*Synchronize* waits for the main VCL thread to enter the message loop and then executes the passed method.

**Note:** Because *Synchronize* uses the message loop, it does not work in console applications. You must use other mechanisms, such as critical sections, to protect access to VCL objects in console applications.

You do not always need to use the main VCL thread. Some objects are thread-aware. Omitting the use of the *Synchronize* method when you know an object's methods are thread-safe will improve performance because you don't need to wait for the VCL thread to enter its message loop. You do not need to use the *Synchronize* method in the following situations:

▪    Data access components are thread-safe as long as each thread has its own database session component. The one exception to this is when you are using Access drivers. Access drivers are built using the Microsoft ADO library, which is not thread-safe.

When using data access components, you must still wrap all calls that involve data-aware controls in the *Synchronize* method. Thus, for example, you need to synchronize calls that link a data control to a dataset by setting the *DataSet* property of the data source object, but you don't need to synchronize to access the data in a field of the dataset.

For more information about using database sessions with threads, see Managing multiple sessions.

▪    Graphics objects are thread-safe. You do not need to use the main VCL thread to access *TFont*, *TPen*, *TBrush*, *TBitmap*, *TMetafile*, or *TIcon*. Canvas objects can be used outside the *Synchronize* method by locking them.

▪    While list objects are not thread-safe, you can use a thread-safe version, *TThreadList*, instead of *TList*.

## Using thread-local variables

 The thread function and any of the routines it calls have their own local variables, just like any other Object Pascal routines. These routines also can access any global variables. In fact, global variables provide a powerful mechanism for communicating between threads.

Sometimes, however, you may want to use variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Make a variable thread-local by declaring it in a **threadvar** section. For example,

```
threadvar
     x : integer;
```

declares an integer type variable that is private to each thread in the application, but global within each thread.

The threadvar section can only be used for global variables. Pointer and Function variables can't be thread variables. Types that use copy-on-write semantics, such as long strings don't work as thread variables either.

## Checking for termination by other threads

Your thread object begins running when the *Execute* method is called (see <u>Executing thread objects</u>) and continues until *Execute* finishes. This reflects the model that the thread performs a specific task, and then stops when it is finished. Sometimes, however, an application needs a thread to execute until some external criterion is satisfied.

You can allow other threads to signal that it is time for your thread to finish executing by checking the *Terminated* property. When another thread tries to terminate your thread, it calls the *Terminate* method. *Terminate* sets your thread's *Terminated* property to *True*. It is up to your *Execute* method to implement the *Terminate* method by checking and responding to the *Terminated* property. The following example shows one way to do this:

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
```

## Writing clean-up code

You can centralize the code that cleans up when your thread finishes executing. Just before a thread shuts down, an *OnTerminate* event occurs. Put any clean-up code in the *OnTerminate* event handler to ensure that it is always executed, no matter what execution path the *Execute* method follows.

The *OnTerminate* event handler is not run as part of your thread. Instead, it is run in the context of the main VCL thread of your application. This has two implications:

- You can't use any thread-local variables in an *OnTerminate* event handler (unless you want the main VCL thread values).
- You can safely access any components and VCL objects from the *OnTerminate* event handler without worrying about clashing with other threads.

## Coordinating threads

When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads.

Whether using thread objects or generating threads using BeginThread, the following topics describe techniques for coordinating threads:

- Avoiding simultaneous access.
- Waiting for other threads.
- Using the main VCL thread.

When global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads.

## Avoiding simultaneous access

To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads.

The VCL includes support for three techniques that prevent other threads from accessing the same memory as your thread:

- Locking objects.
- Using critical sections.
- Using a multi-read exclusive-write synchronizer.

## Locking objects

Some objects have built-in locking that prevents the execution of other threads from using that object instance.

For example, canvas objects (*TCanvas* and descendants) have a *Lock* method that prevents other threads from accessing the canvas until the *Unlock* method is called.

The VCL also includes a thread-safe list object, *TThreadList*. Calling *TThreadList.LockList* returns the list object while also blocking other execution threads from using the list until the *UnlockList* method is called. Calls to *TCanvas.Lock* or *TThreadList.LockList* can be safely nested. The lock is not released until the last locking call is matched with a corresponding unlock call in the same thread.

## Using critical sections

If objects do not provide built-in locking, you can use a critical section. Critical sections work like gates that allow only a single thread to enter at a time. To use a critical section, create a global instance of *TCriticalSection*. *TCriticalSection* has two methods, *Acquire*(which blocks other threads from executing the section) and *Release*(which removes the block).

Each critical section is associated with the global memory you want to protect. Every thread that accesses that global memory should first use the *Acquire* method to ensure that no other thread is using it. When finished, threads call the *Release* method so that other threads can access the global memory by calling *Acquire*.

**Warning:** Critical sections only work if every thread uses them to access the associated global memory. Threads that ignore the critical section and access the global memory without calling *Acquire* can introduce problems of simultaneous access.

For example, consider an application that has a global critical section variable, *LockXY*, that blocks access to global variables X and Y. Any thread that uses X or Y must surround that use with calls to the critical section such as the following:

```
LockXY.Acquire; { lock out other threads }
try
  Y := sin(X);
finally
  LockXY.Release;
end;
```

## Using the multi-read exclusive-write synchronizer

When you use critical sections to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. There is no danger in multiple threads reading the same memory simultaneously, as long as no thread is writing to it.

When you have some global memory that is read often, but to which threads occasionally write, you can protect it using *TMultiReadExclusiveWriteSynchronizer*. This object acts like a critical section, but one which allows multiple threads to read the memory it protects as long as no thread is writing to it. Threads must have exclusive access to write to memory protected by *TMultiReadExclusiveWriteSynchronizer*.

To use a multi-read exclusive-write synchronizer, create a global instance of *TMultiReadExclusiveWriteSynchronizer* that is associated with the global memory you want to protect. Every thread that reads from this memory must first call the *BeginRead* method. *BeginRead* ensures that no other thread is currently writing to the memory. When a thread finishes reading the protected memory, it calls the *EndRead* method. Any thread that writes to the protected memory must call *BeginWrite* first. *BeginWrite* ensures that no other thread is currently reading or writing to the memory. When a thread finishes writing to the protected memory, it calls the *EndWrite* method, so that threads waiting to read the memory can begin.

**Warning:** Like critical sections, the multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Threads that ignore the synchronizer and access the global memory without calling *BeginRead* or *BeginWrite* introduce problems of simultaneous access.

## Waiting for other threads

If your thread must wait for another thread to finish some task, you can tell your thread to temporarily suspend execution. You can either

- Wait for another thread to completely finish executing, or
- Wait for a task to be completed.

## Waiting for a thread to finish executing

To wait for another thread to finish executing, use the _WaitFor_ method of that other thread. _WaitFor_ doesn't return until the other thread terminates, either by finishing its own _Execute_ method or by terminating due to an exception. For example, the following code waits until another thread fills a thread list object before accessing the objects in the list:

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
    for I := 0 to Count - 1 do
      ProcessItem(Items[I]);
  end;
  ThreadList1.UnlockList;
end;
```

In the previous example, the list items were only accessed when the _WaitFor_ method indicated that the list was successfully filled. This return value must be assigned by the _Execute_ method of the thread that was waited for. However, because threads that call _WaitFor_ want to know the result of thread execution, not code that calls _Execute_, the _Execute_ method does not return any value. Instead, the _Execute_ method sets the _ReturnValue_ property. _ReturnValue_ is then returned by the _WaitFor_ method when it is called by other threads. Return values are integers. Your application determines their meaning.

## Waiting for a task to be completed

Sometimes, you need to wait for a thread to finish some operation rather than waiting for a particular thread to complete execution. To do this, use an event object. Event objects (*TEvent*) should be created with global scope so that they can act like signals that are visible to all threads.

When a thread completes an operation that other threads depend on, it calls *TEvent.SetEvent*. *SetEvent* turns on the signal, so any other thread that checks will know that the operation has completed. To turn off the signal, use the *ResetEvent* method.

For example, consider a situation where you must wait for several threads to complete their execution rather than a single thread. Because you don't know which thread will finish last, you can't simply use the *WaitFor* method of one of the threads. Instead, you can have each thread increment a counter when it is finished, and have the last thread signal that they are all done by setting an event.

The following code shows the end of the *OnTerminate* event handler for all of the threads that must complete. *CounterGuard* is a global critical section object that prevents multiple threads from using the counter at the same time. *Counter* is a global variable that counts the number of threads that have completed.

```
procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
  ...
  CounterGuard.Acquire; { obtain a lock on the counter }
  Dec(Counter);    { decrement the global counter variable }
  if Counter = 0 then
    Event1.SetEvent; { signal if this is the last thread }
  CounterGuard.Release; { release the lock on the counter }
  ...
end;
```

The main thread initializes the Counter variable, launches the task threads, and waits for the signal that they are all done by calling the *WaitFor* method. *WaitFor* waits for a specified time period for the signal to be set, and returns one of the values from the following table:

| Value | Meaning |
| --- | --- |
| wrSignaled | The signal of the event was set. |
| wrTimeout | The specified time elapsed without the signal being set. |
| wrAbandoned | The event object was destroyed before the timeout period elapsed. |
| wrError | An error occurred while waiting. |

The following shows how the main thread launches the task threads and then resumes when they have all completed:

```
Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
  TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) != wrSignaled then
  raise Exception;
{ now continue with the main thread. All task threads have finished }
```

**Note:** If you do not want to stop waiting for an event after a specified time period, pass the *WaitFor* method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

## Executing thread objects

Once you have implemented a thread class by giving it an *Execute* method, you can use it in your application to launch the code in the *Execute* method. To use a thread, first create an instance of the thread class. You can create a thread instance that starts running immediately, or you can create your thread in a suspended state so that it only begins when you call the _Resume_ method. To create a thread so that it starts up immediately, set the constructor's *CreateSuspended* parameter to *False.* For example, the following line creates a thread and starts its execution:

```
SecondProcess := TMyThread.Create(false); {create and run the thread }
```

**Warning:** Do not create too many threads in your application. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.

You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.

The following topics discuss how to use the threads in your application:

- Overriding the default priority.
- Starting and stopping threads

## Overriding the default priority

When the amount of CPU time the thread should receive is implicit in the thread's task, its priority is set in the constructor. This is described in Initializing the thread. However, if the thread priority varies depending on when the thread is executed, create the thread in a suspended state, set the priority, and then start the thread running:

```
SecondProcess :=  TMyThread.Create(True); { create but don't run }
SecondProcess.Priority := tpLower; { set the priority lower than normal }
SecondProcess.Resume; { now run the thread }
```

## Starting and stopping threads

A thread can be started and stopped any number of times before it finishes executing. To stop a thread temporarily, call its *Suspend* method. When it is safe for the thread to resume, call its *Resume* method. *Suspend* increases an internal counter, so you can nest calls to *Suspend* and *Resume*. The thread does not resume execution until all suspensions have been matched by a call to *Resume*.

You can request that a thread end execution prematurely by calling the *Terminate* method. *Terminate* sets the thread's *Terminated* property to *True*. If you have implemented the *Execute* method properly, it checks the *Terminated* property periodically, and stops execution when *Terminated* is *True*.

## Using threads in distributed applications

Distributed applications introduce additional challenges for writing multi-threaded applications. When considering how to underline coordinate threads, you must also keep in mind how other processes affect the threads in your application.

Usually, handling distributed threading issues is the responsibility of the server application. When writing servers, you must consider how requests from clients are serviced.

If each client request has its own thread, you must ensure that different client threads do not interfere with each other. In addition to the usual issues that arise when coordinating multiple threads, you may need to ensure that each client has a consistent view of your application. For example, you can't use thread variables to store information that must persist over multiple client requests if each time the client calls your application it uses a different thread. When clients change the values of object properties or global variables, they are influencing not only their own view of that object or variable, but the view of any other clients.

The following topics describe some of the issues for using threads with

- Message-based servers.
- Distributed objects.

## Using threads in message-based servers

Message-based servers receive client request messages, perform some action in response to that message, and return messages to the client. Examples include internet server applications and simple services that you can write using sockets.

Usually, when writing message-based servers, each client message gets its own thread. When client messages are received, the application spawns a thread to handle the message. This thread runs until it sends a response to the client, and then terminates. You must be careful when using global objects and variables, but it is fairly easy to control how threads are created and run because client messages are all received and dispatched by the main application thread.

# Using threads with distributed objects

When writing servers for distributed objects, the threading issues are complicated. Unlike message-based servers, where there is a point in the code where messages are received and dispatched, clients call into server objects by calling any of their methods or by accessing any of their properties. Because of this, there is no easy way for server applications to spawn separate threads for each client request.

## Writing applications (.EXEs)

When writing an .EXE that implements an object or objects for remote clients, client requests come in as threads. How this works depends on whether clients access your object using COM or CORBA.

▪ **Under COM**, client requests come in as part of the application's message loop. This means that any code which executes after the application's main message loop starts up must be prepared to protect access to objects and global memory from other threads. When running in an environment that supports DCOM, Delphi ensures that no client requests occur until all code in the initialization part of your units has executed. If you are not running in an environment that supports DCOM, you must ensure that any code in the initialization part of your units is thread-safe.

▪ **Under CORBA**, you can choose a threading model in the wizard that starts a new CORBA server. You can choose either single-threading or multi-threading. Under both models, each client connection has its own thread. You can use thread variables for information that persists across client calls because all calls for a given client use the same thread. With single-threading, only one client thread has access to an object instance at a time. While you must protect access to global memory, you are assured of no conflicts when accessing the object's instance data (such as property values). With multi-threading, multiple clients may access your application at the same time. If you are sharing object instances over clients, you must protect both global data and instance data.

## Writing libraries

When an Active Library implements the distributed object, threading is usually controlled by the technology (COM, DCOM, or MTS) that supports distributed object calls. When you first create your server library with the appropriate wizard, you are prompted to specify a threading model that dictates how client requests are assigned threads. These models include the following:

▪ **Single-threaded model.** Client requests are serialized by the calling mechanism. Your .DLL does not need to be concerned with threading issues because it receives one client request at a time.

▪ **Single-threaded apartment model**. (Also called Apartment model.) Each object instantiated by a client is accessed by one thread at a time. You must protect against multiple threads accessing global memory, but instance data (such as object properties) is thread-safe. Further, each client always accesses the object instance using the same thread, so that you can use thread variables.

▪ **Activity model.** (Called both Apartment-threaded and Free-threaded under MTS.) Each object instance is accessed by one thread at a time, but clients do not always use the same thread for every call. Instance data is safe, but you must guard global memory, and thread variables will not be consistent across client calls.

▪ **Multi-threaded apartment model.** (Also called Free-threading.) Each object instance may be called by multiple threads simultaneously. You must protect instance data as well as global memory. Thread variables are not consistent across client calls.

▪ **Single/Multi-threaded apartment model.** (Also called Both.) This is the same as the Multi-threaded apartment model, except that all callbacks supplied by clients are guaranteed to execute in the same thread. This means you do not need protect values supplied as parameters to callback functions.

Note: Typically, a wizard assigns a threading model to your object. When you add multiple COM objects to an EXE, the application initializes COM with the highest level of thread support indicated (where single-threaded is the lowest and Both is highest). You can manually override the way your application initializes COM threading support by changing the global *CoInitFlags* variable in the program's main source file before the call to *Application.Intitialize*.

COM-based systems use the application's message loop to synchronize threads in all but the Multi-threaded apartment model (which is only available under DCOM). Because of this, you must ensure that any lengthy call made through a COM interface calls the application object's *ProcessMessages* method.

Failure to do so prevents other clients from gaining access to your application, effectively making your library single-threaded.

# Debugging multi-threaded applications

When debugging multi-threaded applications, it can be confusing trying to keep track of the status of all the threads that are executing simultaneously, or even to determine which thread is executing when you stop at a breakpoint. You can use the Thread Status box to help you keep track of and manipulate all the threads in your application. To display the Thread status box, choose View|Threads from the main menu.

When a debug event occurs (breakpoint, exception, paused), the thread status view indicates the status of each thread. Right-click the Thread Status box to access commands that locate the corresponding source location or make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

The Thread Status box lists all your application's execution threads by their thread ID. If you are using thread objects, the thread ID is the value of the *ThreadID* property. If you are not using thread objects, the thread ID for each thread is returned by the call to *BeginThread*.

## About packages

A *package* is a special dynamic-link library used by Delphi applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by calling runtime packages. To distinguish them from other DLLs, package libraries are stored in files that end with the .BPL (Borland package library) extension.

Like other runtime libraries, packages contain code that can be shared among applications. For example, the most frequently used Delphi components reside in a package called VCL50. Each time you create an application, you can specify that it uses VCL50. When you compile an application created this way, the application's executable image contains only the code and data unique to it; the common code is in VCL50.BPL. A computer with several package-enabled applications installed on it needs only a single copy of VCL50.BPL, which is shared by all the applications and the IDE itself.

Delphi ships with several precompiled runtime packages, including VCL50, that encapsulate VCL components. Delphi also uses design-time packages to manipulate components in the IDE.

You can build applications with or without packages. However, if you want to add custom components to the IDE, you must install them as design-time packages.

You can create your own runtime packages to share among applications. If you write Delphi components, you can compile your components into design-time packages before installing them.

Package topics:
- Why use packages?
- Packages and standard DLLs
- Runtime packages
- Using runtime packages in an application
- Deciding which runtime packages to use
- Custom packages
- Design-time packages
- Installing component packages
- Creating and editing packages
- Creating a package
- Editing an existing package
- Editing package source files manually
- Understanding the structure of a package
  - Naming packages
  - The Requires clause
    - Avoiding circular package references
    - Handling duplicate package references
  - The Contains clause
    - Avoiding redundant source code uses
- Compiling packages
  - Package-specific compiler directives
    - Weak packaging
  - Using the command-line compiler and linker
  - Package files created by a successful compilation
- Deploying packages
- Deploying applications that use packages
- Distributing packages to other developers
- Package collection files

## Why use packages?

Design-time packages simplify the tasks of distributing and installing custom components. Runtime packages, which are optional, offer several advantages over conventional programming. By compiling reused code into a runtime library, you can share it among applications. For example, all of your applications—including Delphi itself—can access standard components through packages. Since the applications don't have separate copies of the component library bound into their executables, the executables are much smaller—saving both system resources and hard disk storage. Moreover, packages allow faster compilation because only code unique to the application is compiled with each build.

Packages and standard DLLs

## Packages and standard DLLs

Create a package when you want to make a custom component that's available through the IDE. Create a standard DLL when you want to build a library that can be called from any Windows application, regardless of the development tool used to build the application.

The following table lists the file types associated with packages

| File extension | Contents |
| --- | --- |
| .DPK | The source file listing the units contained in the package. |
| DCP | A binary image containing a package header and the concatenation of all DCU files in the package, including all symbol information required by the compiler. A single DCP file is created for each package. The base name for the DCP is the base name of the DPK source file. You must have a .DCP file to build an application with packages. |
| DCU | A binary image for a unit file contained in a package. One DCU is created, when necessary, for each unit file. |
| BPL | The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the BPL is the base name of the DPK source file. |

**Note:** Packages share their global data with other modules in an application.

## Runtime packages

Runtime packages are deployed with Delphi applications. They provide functionality when a user runs the application.

To run an application that uses packages, a computer must have both the application's .EXE file and all the packages (.BPL files) that the application uses. The .BPL files must be on the system path for an application to use them. When you deploy an application, you must make sure that users have correct versions of any required .BPLs.

- Using runtime packages in an application
- Deciding which runtime packages to use
- Custom packages

## Using runtime packages in an application

To use packages in an application,

1   Load or create a project in the IDE.
2   Choose Project|Options.
3   Choose the Packages tab.
4   Select the "Build with Runtime Packages" check box, and enter one or more package names in the edit box underneath. (Runtime packages associated with installed design-time packages are already listed in the edit box.) To add a package to an existing list, click the Add button and enter the name of the new package in the Add Runtime Package dialog. To browse from a list of available packages, click the Add button, then click the Browse button next to the Package Name edit box in the Add Runtime Package dialog.

    If you edit the Search Path edit box in the Add Runtime Package dialog, you will be changing Delphi's global Library Path.

    You do not need to include file extensions with package names. If you type directly into the Runtime Packages edit box, be sure to separate multiple names with semicolons. For example:

```
VCL50;VCLDB50;VCLDBX50
```

Packages listed in the Runtime Packages edit box are automatically linked to your application when you compile. Duplicate package names are ignored, and if the edit box is empty the application is compiled without packages.

Runtime packages are selected for the current project only. To make the current choices into automatic defaults for new projects, select the "Defaults" check box at the bottom of the dialog.

**Note:** When you create an application with packages, you still need to include the names of the original Delphi units in the **uses** clause of your source files. For example, the source file for your main form might begin like this:

```
unit MainForm;
interface
uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

Each of the units referenced in this example is contained in the VCL50 package. Nonetheless, you must keep these references in the **uses** clause, even if you use VCL50 in your application, or you will get compiler errors. In generated source files, Delphi adds these units to the **uses** clause automatically.

▪       Deciding which runtime packages to use

## Dynamically loading packages

To load a package at runtime, call the *LoadPackage* function. For example, the following code could be executed when a file is chosen in a file-selection dialog.

```
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(Filename)));
```

To unload a package dynamically, call *UnloadPackage*. Be careful to destroy any instances of classes defined in the package and to unregister classes that were registered by it.

## Deciding which runtime packages to use

Delphi ships with several precompiled runtime packages, including VCL50, which supply basic language and component support.

The VCL50 package contains the most commonly used components, system functions, and Windows interface elements. It does not include database or Windows 3.1 components, which are available in separate packages. The following table lists some runtime packages shipped with Delphi and the units they contain.

| Package | Units |
| --- | --- |
| VCL50.BPL | Ax, Buttons, Classes, Clipbrd, Comctrls, Commctrl, Commdlg, Comobj, Comstrs, Consts, Controls, Ddeml, Dialogs, Dlgs, Dsgnintf, Dsgnwnds, Editintf, Exptintf, Extctrls, Extdlgs, Fileintf, Forms, Graphics, Grids, Imm, IniFiles, Isapi, Isapi2, Istreams, Libhelp, Libintf, Lzexpand, Mapi, Mask, Math, Menu, Messages, Mmsystem, Nsapi, Ole2l, Oleconst, Olectnrs, Olectrls, Oledlg, Penwin, Printers, Proxies, Registry, Regstr, Richedit, Shellapi, Shlobj, Stdctrls, Stdvcl, Sysutils, Tlhelp32, Toolintf, Toolwin, Typinfo, Vclcom, Virtintf, Windows, Wininet, Winsock, Winspool, Winsvc |
| VCLX50.BPL | Checklst, Colorgrd, Ddeman, Filectrl, Mplayer, Outline, Tabnotbk, Tabs |
| VCLDB50.BPL | Bde, Bdeconst, Bdeprov, Db, Dbcgrids, Dbclient, Dbcommon, Dbconsts, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables, Dsintf, Provider, SMintf |
| VCLDBX50.BPL | Dblookup, Report |
| DSS50.BPL | Mxarrays, Mxbutton, Mxcommon, Mxconsts, Mxdb, Mxdcube, Mxdssqry, Mxgraph, Mxgrid, Mxpivsrc, Mxqedcom, Mxqparse, Mxqryedt, Mxstore, Mxtables, Mxqvb |
| QRPT50.BPL | Qr2const, Qrabout, Qralias, Qrctrls, Qrdatasu, Qrexpbld, Qrextra, Qrprev, Qrprgres, Qrprntr, Qrqred32, Quickrpt |
| TEE50.BPL | Arrowcha, Bubblech, Chart, Ganttch, Series, Teeconst, Teefunci, Teengine, Teeprocs, Teeshape |
| TEEDB50.BPL | Dbchart, Qrtee |
| TEEUI50.BPL | Areaedit, Arrowedi, Axisincr, Axmaxmin, Baredit, Brushdlg, Bubbledi, Custedit, Dbeditch, Editchar, Flineedi, Ganttedi, Ieditcha, Pendlg, Pieedit, Shapeedi, Teeabout, Teegally, Teelisb, Teeprevi, Teexport |
| VCLSMP50.BPL | Sampreg, Smpconst |

To create a client/server database application that uses packages, you need at least two runtime packages: VCL50 and VCLDB50. If you want to use Outline components in your application, you also need VCLX50. To use these packages, choose Project|Options, select the Packages tab, and enter the following list in the Runtime Packages edit box.

```
VCL50;VCLDB50;VCLX50
```

Actually, you don't have to include VCL50, because VCL50 is referenced in the Requires clause of VCLDB50. Your application will compile just the same whether or not VCL50 is included in the Runtime Packages edit box.

## Custom packages

A custom package is either a BPL you code and compile yourself, or a precompiled package from a third-party vendor. To use a custom runtime package with an application, choose Project|Options and add the name of the package to the Runtime Packages edit box on the Packages page. For example, suppose you have a statistical package called STATS.BPL. To use it in an application, the line you enter in the Runtime Packages edit box might look like this:

```
VCL50;VCLDB50;STATS
```

If you create your own packages, you can add them to the list as needed.

## Design-time packages

Design-time packages are used to install components on the IDE's Component palette and to create special property editors for custom components.

Delphi ships with the following design-time component packages preinstalled in the IDE.

| Package | Component palette pages |
|---------|------------------------|
| DCLSTD50.BPL | Standard, Additional, System, Win32, Dialogs |
| DCLTEE50.BPL | Additional (*TChart* component) |
| DCLDB50.BPL | Data Access, Data Controls |
| DCLMID50.BPL | Data Access (MIDAS) |
| DCL31W50.BPL | Win 3.1 |
| DCLNET50.BPL | Internet |
| NMFAST50.BPL | |
| DCLSMP50.BPL | Samples |
| DCLOCX50.BPL | ActiveX |
| DCLQRT50.BPL | QReport |
| DCLDSS50.BPL | Decision Cube |
| IBSMP50.BPL | Samples (*IBEventAlerter* component) |
| DCLINT50.BPL | (International Tools—Resource DLL wizard) |
| RCEXPERT.BPL | (Resource Expert) |
| DBWEBXPRT.BPL | (Web Wizard) |

These design-time packages work by calling runtime packages, which they reference in their Requires clause. For example, DCLSTD50 references VCL50. DCLSTD50 itself contains additional functionality that makes most of the standard components available on the Component palette.

In addition to preinstalled packages, you can install your own component packages, or component packages from third-party developers, in the IDE. The DCLUSR50 design-time package is provided as a default container for new components.

- Installing component packages

## Installing component packages

All components are installed in the IDE as packages. If you've written your own components, <u>create and compile</u> a package that contains them. Your component source code must follow the model described in <u>Overview of component creation</u>.

To install or uninstall your own components, or components from a third-party vendor, follow these steps:

1  If you are installing a new package, copy or move the package files to a local directory. If the package is shipped with .BPL, .DCP, and .DCU files be sure to copy all of them.

   The directory where you store the .DCP file—and the .DCU files, if they are included with the distribution—must be in the Delphi Library Path.

   If the package is shipped as a .DPC (package collection) file, only the one file need be copied; the .DPC file contains the other files. (For more information about package collection files, see <u>Package collection files</u>.)

2  Choose Component|Install Packages from the IDE menu, or choose Project|Options and click the Packages tab.

3  A list of available packages appears under "Design packages".

- To install a package in the IDE, select the check box next to it.
- To uninstall a package, deselect its check box.
- To see a list of components included in an installed package, select the package and click Components.
- To add a package to the list, click Add and browse in the Open Package dialog box for the directory where the .BPL or .DPC file resides (see step 1). Select the .BPL or .DPC file and click Open. If you select a .DPC file, a new dialog box appears to handle the extraction of the .BPL and other files from the package collection.
- To remove a package from the list, select the package and click Remove.

4  Click OK.

The components in the package are installed on the Component palette pages specified in the components' *RegisterComponents* procedure, with the names they were assigned in the same procedure.

New projects are created with all available packages installed, unless you change the default settings. To make the current installation choices into the automatic default for new projects, check the Default check box at the bottom of the dialog box.

To remove components from the Component palette without uninstalling a package, select Component| Configure Palette, or select Tools|Environment Options and click the Palette tab. The Palette options tab lists each installed component along with the name of the Component palette page where it appears. Selecting any component and clicking Hide removes the component from the palette.

## Creating and editing packages

Creating a package involves specifying

- A *name* for the package.
- A list of other packages to be *required* by, or linked to, the new package.
- A list of unit files to be *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which contain the functionality of the compiled .BPL. The Contains clause is where you put the source-code units for custom components that you want to compile into a package.

Package source files, which end with the .DPK extension, are generated by the Package editor.

- Creating a package
- Editing an existing package
- Editing package source files manually
- Understanding the structure of a package
- Compiling packages

## Creating a package

To create a package, follow the procedure below. Refer to Understanding the structure of a package for more information about the steps outlined here.

1    Choose File|New, select the Package icon, and click OK.
2    The generated package is displayed in the Package editor.
3    The Package editor shows a *Requires* node and a *Contains* node for the new package.
4    To add a unit to the **contains** clause, click the Add to package speed button. In the Add unit page, type a .PAS file name in the Unit file name edit box, or click Browse to browse for the file, and then click OK. The unit you've selected appears under the Contains node in the Package editor. You can add additional units by repeating this step.
5    To add a package to the **requires** clause, click the Add to package speed button. In the Requires page, type a .DCP file name in the Package name edit box, or click Browse to browse for the file, and then click OK. The package you've selected appears under the Requires node in the Package editor. You can add additional packages by repeating this step.
6    Click the Options speed button, and decide what kind of package you want to build.

▪        To create a design-time–only package (a package that cannot be used at runtime), select the Designtime only radio button. (Or add the {$DESIGNONLY} compiler directive to the DPK file.)
▪        To create a runtime-only package (a package that cannot be installed), select the Runtime only radio button. (Or add the {$RUNONLY} compiler directive to the DPK file.)
▪        To create a package that is available at both design time and runtime, select the Designtime and runtime radio button.

7    In the Package editor, click the Compile package speed button to compile your package.

## Editing an existing package

There are several ways to open an existing package for editing.

- Choose File|Open (or File|Reopen) and select a DPK file.
- Choose Component|Install Packages, select a package from the Design Packages list, and click the Edit button.
- When the Package editor is open, select one of the packages in the Requires node, right-click, and choose Open.

To edit a package's description or set usage options, click the Options speed button in the Package editor and select the Description tab.

The Project Options dialog has a Default check box in the lower left corner. If you click OK when this box is checked, the options you've chosen are saved as default settings for new projects. To restore the original defaults, delete or rename the DEFPROJ.DOF file.

## Editing Package source files manually

Package source files, like project files, are generated by Delphi from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .DPK (Delphi package) extension to avoid confusion with other files containing Object Pascal source code.

To open a package source file in the Code editor,
 1   Open the package in the Package editor.
 2   Right-click in the Package editor and select View Source.
▪        The **package** heading specifies the name for the package.
▪        The **requires** clause lists other, external packages used by the current package. If a package does not contain any units that use units in another package, then it doesn't need a **requires** clause.
▪        The **contains** clause identifies the unit files to be compiled and bound into the package. All units used by contained units which do not exist in required packages will also be bound into the package, although they won't be listed in the contains clause (the compiler will give a warning).
     For example, the following code declares the VCLDB50 package.

```
package VCLDB50;
  requires VCL50;
  contains Db, Dbcgrids, Dbctrls, Dbgrids, Dbinpreq, Dblogdlg, Dbpwdlg, Dbtables,
mycomponent in 'C:\components\mycomponent.pas';
end.
```

## Understanding the structure of a package

**Naming packages**

Package names must be unique within a project. If you name a package STATS, the Package editor generates a source file for it called STATS.DPK; the compiler generates an executable and a binary image called STATS.BPL and STATS.DCP, respectively. Use STATS to refer to the package in the **requires** clause of another package, or when using the package in an application.

**The Requires clause**

The **requires** clause specifies other, external packages that are used by the current package. An external package included in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in your package make references to other packaged units, the other packages should appear in your package's **requires** clause or you should add them. If the other packages are omitted from the **requires** clause, the compiler will import them into your package 'implicitly contained units'.

**Note:** Most packages that you create will require VCL50. Any package that depends on VCL units (including SysUtils) must list VCL50, or another package that requires VCL50, in its **requires** clause.

**Avoiding circular package references**

Packages cannot contain circular references in their **requires** clause. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

**Handling duplicate package references**

Duplicate references in a package's **requires** clause—or in the Runtime Packages edit box—are ignored by the compiler. For programming clarity and readability, however, you should catch and remove duplicate package references.

**The Contains clause**

The **contains** clause identifies the unit files to be bound into the package. If you are writing your own package, put your source code in PAS files and include them in the **contains** clause.

**Avoiding redundant source code uses**

A package cannot appear in the **contains** clause of another package.

All units included directly in a package's **contains** clause, or included indirectly in any of those units, are bound into the package at compile time.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application, *including the Delphi IDE*. This means that if you create a package that contains one of the units in VCL50, you won't be able to install your package in the IDE. To use an already-packaged unit file in another package, put the first package in the second package's **requires** clause.

## Compiling packages

You can compile a package from the IDE or from the command line. To recompile a package by itself from the IDE,

1   Choose File|Open.
2   Select Delphi Package Source (*.DPK) from the Files Of Type drop-down list.
3   Select a .DPK file in the dialog.
4   When the Package editor opens, click the Compile speed button.

You can insert compiler directives into your package source code. For more information, see "Package-specific compiler directives", below.

If you compile from the command line, several package-specific switches are available. For more information, see "Using the command-line compiler and linker" on page 9-21.

- Package-specific compiler directives
- Weak packaging
- Using the command-line compiler and linker
- Package files created by a successful compilation

## Package-specific compiler directives

The following table lists package-specific compiler directives that you can insert into your source code.

| Directive | Purpose |
| --- | --- |
| {$IMPLICITBUILD OFF} | Prevents a package from being implicitly recompiled later. Use in .DPK files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed. |
| {$G-} or {IMPORTEDDATA OFF} | Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages. |
| {$WEAKPACKAGEUNIT ON} | Packages unit "weakly." See Weak packaging. |
| {$DENYPACKAGEUNIT ON} | Prevents unit from being placed in a package. |
| {$DESIGNONLY ON} | Compiles the package for installation in the IDE. (Put in .DPK file.) |
| {$RUNONLY ON} | Compiles the package as runtime only. (Put in .DPK file.) |

**Note:** Including **{$DENYPACKAGEUNIT ON}** in your source code prevents the unit file from being packaged. Including **{$G-}** or {**IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages. Packages compiled with the **{$DESIGNONLY ON}** directive should not ordinarily be used in applications, since they contain extra code required by the IDE. Other compiler directives may be included, if appropriate, in package source code. See Compiler directives for information on compiler directives not discussed here.

## Weak packaging

The **$WEAKPACKAGEUNIT** directive affects the way a .DCU file is stored in a package's .DCP and .BPL files. (For information about files generated by the compiler, see Package files created by a successful compilation .) If **{$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from BPLs when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be "weakly packaged."

For example, suppose you've created a package called PACK that contains only one unit, UNIT1. Suppose UNIT1 does not use any further units, but it makes calls to RARE.DLL. If you put **{$WEAKPACKAGEUNIT ON}** in UNIT1.PAS when you compile your package, UNIT1 will not be included in PACK.BPL; you will not have to distribute copies of RARE.DLL with PACK. However, UNIT1 will still be included in PACK.DCP. If UNIT1 is referenced by another package or application that uses PACK, it will be copied from PACK.DCP and compiled directly into the project.

Now suppose you add a second unit, UNIT2, to PACK. Suppose that UNIT2 uses UNIT1. This time, even if you compile PACK with **{$WEAKPACKAGEUNIT ON}** in UNIT1.PAS, the compiler will include UNIT1 in PACK.BPL. But other packages or applications that reference UNIT1 will use the (non-packaged) copy taken from PACK.DCP.

**Note:** Unit files containing the **{$WEAKPACKAGEUNIT ON}** directive must not have global variables, initialization sections, or finalization sections.

The **$WEAKPACKAGEUNIT** directive is an advanced feature intended for developers who distribute their BPLs to other Delphi programmers. It can help you to avoid distribution of infrequently used DLLs, and to eliminate conflicts among packages that may depend on the same external library.

For example, Delphi's PenWin unit references PENWIN.DLL. Most projects don't use PenWin, and most computers don't have PENWIN.DLL installed on them. For this reason, the PenWin unit is weakly packaged in VCL50. When you compile a project that uses PenWin and the VCL50 package, PenWin is copied from VCL50.DCP and bound directly into your project; the resulting executable is statically linked to PENWIN.DLL.

If PenWin were not weakly packaged, two problems would arise. First, VCL50 itself would be statically linked to PENWIN.DLL, and so you could not load it on any computer which didn't have PENWIN.DLL installed. Second, if you tried to create a package that contained PenWin, a compiler error would result because the PenWin unit would be contained in both VCL50 and your package. Thus, without weak packaging, PenWin could not be included in standard distributions of VCL50.

## Using the command-line compiler and linker

When you compile from the command line, you can use the package-specific switches listed in the following table.

| Switch | Purpose |
| --- | --- |
| -$G- | Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages. |
| -LE*path* | Specifies the directory where the package BPL file will be placed. |
| -LN*path* | Specifies the directory where the package DCP file will be placed. |
| -LU*package* | Use packages. |
| -Z | Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed. |

**Note:** Using the **-$G-** switch may prevent a package from being used in the same application with other packages. Other command-line options may be used, if appropriate, when compiling packages. See The Command-line compiler for information on command-line options not discussed here.

## Package files created by a successful compilation

To create a package, you compile a source file that has a .DPK extension. The base name of the .DPK file becomes the base name of the files generated by the compiler. For example, if you compile a package source file called TRAYPAK.DPK, the compiler creates a package called TRAYPAK.BPL.

The following table lists the files produced by the successful compilation of a package.

| File extension | Contents |
|---|---|
| DCP | A binary image containing a package header and the concatenation of all DCU files in the package. A single DCP file is created for each package. The base name for the DCP is the base name of the DPK source file. |
| DCU | A binary image for a unit file contained in a package. One DCU is created, when necessary, for each unit file. |
| BPL | The runtime package. This file is a Windows DLL with special Delphi-specific features. The base name for the BPL is the base name of the DPK source file. |

## Deploying packages

### Deploying applications that use packages

When distributing an application that uses runtime packages, make sure that your users have the application's .EXE file as well as all the library (.BPL or .DLL) files that the application calls. If the library files are in a different directory from the .EXE file, they must be accessible through the user's Path. You may want to follow the convention of putting library files in the Windows\System directory. If you use InstallShield Express, your installation script can check the user's system for any packages it requires before blindly reinstalling them.

### Distributing packages to other developers

If you distribute runtime or design-time packages to other Delphi developers, be sure to supply both .DCP and .BPL files. You will probably want to include .DCU files as well.

## Package collection files

Package collections (.DPC files) offer a convenient way to distribute packages to other developers. Each package collection contains one or more packages, including BPLs and any additional files you want to distribute with them. When a package collection is selected for IDE installation, its constituent files are automatically extracted from their .PCE container; the Installation dialog box offers a choice of installing all packages in the collection or installing packages selectively.

To create a package collection,

1   Choose Tools|Package Collection Editor to open the Package Collection editor.

2   Click the Add Package speed button, then select a BPL in the Select Package dialog and click Open. To add more BPLs to the collection, click the Add Package speed button again. A tree diagram on the left side of the Package editor displays the BPLs as you add them. To remove a package, select it and click the Remove Package speed button.

3   Select the Collection node at the top of the tree diagram. On the right side of the Package Collection editor, two fields will appear:

▪       In the Author/Vendor Name edit box, you can enter optional information about your package collection that will appear in the Installation dialog when users install packages.

▪       Under Directory List, list the default directories where you want the files in your package collection to be installed. Use the Add, Edit, and Delete buttons to edit this list. For example, suppose you want all source code files to be copied to the same directory. In this case, you might enter Source as a Directory Name with C:\MyPackage\Source as the Suggested Path. The Installation dialog box will display C:\MyPackage\Source as the suggested path for the directory.

4   In addition to BPLs, your package collection can contain .DCP, .DCU, and .PAS (unit) files, documentation, and any other files you want to include with the distribution. Ancillary files are placed in file groups associated with specific packages (BPLs); the files in a group are installed only when their associated BPL is installed. To place ancillary files in your package collection, select a BPL in the tree diagram and click the Add File Group speed button; type a name for the file group. Add more file groups, if desired, in the same way. When you select a file group, new fields will appear on the right in the Package Collection editor,

▪       In the Install Directory list box, select the directory where you want files in this group to be installed. The drop-down list includes the directories you entered under Directory List in step 3, above.

▪       Check the Optional Group check box if you want installation of the files in this group to be optional.

▪       Under Include Files, list the files you want to include in this group. Use the Add, Delete, and Auto buttons to edit the list. The Auto button allows you to select all files with specified extensions that are listed in the **contains** clause of the package; the Package Collection editor uses Delphi's global Library Path to search for these files.

5   You can select installation directories for the packages listed in the **requires** clause of any package in your collection. When you select a BPL in the tree diagram, four new fields appear on the right side of the Package Collection editor:

▪       In the Required Executables list box, select the directory where you want the .BPL files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory List in step 3, above.) The Package Collection Editor searches for these files using Delphi's global Library Path and lists them under Required Executable Files.

▪       In the Required Libraries list box, select the directory where you want the .DCP files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory List in step 3, above.) The Package Collection Editor searches for these files using Delphi's global Library Path and lists them under Required Library Files.

6   To save your package collection source file, choose File|Save. Package collection source files should be saved with the .PCE extension.

7   To build your package collection, click the Compile speed button. The Package Collection editor generates a .DPC file with the same name as your source (.PCE) file. If you have not yet saved the source file, the editor queries you for a file name before compiling.

To edit or recompile an existing .PCE file, select File|Open in the Package Collection editor.

## Creating international applications

This topic discusses guidelines for writing applications that you plan to distribute to an international market. By planning ahead, you can reduce the amount of time and code necessary to make your application function in its foreign market as well as in its domestic market.

The following topics are discussed in this section:

- Internationalization and localization
- Internationalizing applications
- Localizing applications

## Internationalization and localization

To create an application that you can distribute to foreign markets, there are two major steps that need to be performed:

- Internationalization
- Localization

## Internationalization

Internationalization is the process of enabling your program to work in multiple locales. A locale is the user's environment, which includes the cultural conventions of the target country as well as the language. Windows supports a large set of locales, each of which is described by a language and country pair.

# Localization

Localization is the process of translating an application to function in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified to be aware of the different tax laws in different countries.

## Internationalizing applications

It is not difficult to create internationalized applications. You need to complete the following steps:

1   You must <u>enable your code</u> to handle strings from international character sets.

2   You need to <u>design your user interface</u> so that it can accommodate the changes that result from localization.

3   You need to <u>isolate all resources</u> that need to be localized.

## Enabling application code

You must make sure that the code in your application can handle the strings it will encounter in the various target locales. To do this, you must consider the following:

- Character sets
- OEM and ANSI character sets
- Double byte character sets
- Wide characters
- Locale-specific features

## Character sets

The United States edition of Windows uses the ANSI Latin-1 (1252) character set. However, other editions of Windows use different character sets. For example, the Japanese version of Windows uses the Shift-Jis character set (code page 932), which represents Japanese characters as 1- or 2-byte character codes.

# OEM and ANSI character sets

It is sometimes necessary to convert between the Windows character set (ANSI) and the character set specified by the code page of the user's machine (called the OEM character set).

## Double byte character sets

The ideographic character sets used in Asia cannot use the simple 1:1 mapping between characters in the language and the one byte (8-bit) *char* type. These languages have too many characters to be represented using the 1-byte *char*. Instead, characters are represented by a mix of 1- and 2-byte character codes.

The first byte of every 2-byte character code is taken from a reserved range that depends on the specific character set. The second byte can sometimes be the same as the character code for a separate 1-byte character, or it can fall in the range reserved for the first byte of 2-byte characters. Thus, the only way to tell whether a particular byte in a string represents a single character or part of a 2-byte character is to read the string, starting at the beginning, parsing it into 2-byte characters when a lead byte from the reserved range is encountered.

When writing code for Asian locales, you must be sure to handle all string manipulation using functions that are enabled to parse strings into 1- and 2-byte characters. Delphi provides you with a number of runtime library functions that allow you to do this. These functions are as follows:

| | | |
|---|---|---|
| AdjustLineBreaks | AnsiStrLower | ExtractFileDir |
| AnsiCompareFileName | AnsiStrPos | ExtractFileExt |
| AnsiExtractQuotedStr | AnsiStrRScan | ExtractFileName |
| AnsiLastChar | AnsiStrScan | ExtractFilePath |
| AnsiLowerCase | AnsiStrUpper | ExtractRelativePath |
| AnsiLowerCaseFileName | AnsiUpperCase | FileSearch |
| AnsiPos | AnsiUpperCaseFileName | IsDelimiter |
| AnsiQuotedStr | ByteToCharIndex | IsPathDelimiter |
| AnsiStrComp | ByteToCharLen | LastDelimiter |
| AnsiStrIComp | ByteType | StrByteType |
| AnsiStrLastChar | ChangeFileExt | StringReplace |
| AnsiStrLComp | CharToByteIndex | WrapText |
| AnsiStrLIComp | CharToByteLen | |

Remember that the length of the strings in bytes does not necessarily correspond to the length of the string in characters. Be careful not to truncate strings by cutting a 2-byte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character can't be known up front. Instead, always pass a pointer to a character or a string.

# Wide characters

One approach to working with ideographic character sets is to convert all characters to a wide character encoding scheme such as Unicode. Wide characters are two bytes instead of one, so that the character set can represent many more different characters.

Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems. There is a direct relationship between the number of bytes in the string and the number of characters in the string. You do not need to worry about cutting characters in half or mistaking the second half of a character for the start of a different character.

The biggest disadvantage of working with wide characters is that Windows 95 only supports a few wide character API function calls. Because of this, the VCL components represent all string values as single byte or MBCS strings. Translating between the wide character system and the MBCS system every time you set a string property or read its value would require tremendous amounts of extra code and slow your application down. However, you may want to translate into wide characters for some special string processing algorithms that need to take advantage of the 1:1 mapping between characters and *WideChars*.

## Including bi-directional functionality in applications

Some languages do not follow the left to right reading order commonly found in western languages, but rather read words right to left and numbers left to right. These languages are termed bi-directional (BiDi) because of this separation. The most common bi-directional languages are Arabic and Hebrew, although other Middle East languages are also bi-directional.

*TApplication* has two properties, *BiDiKeyboard* and *NonBiDiKeyboard*, that allow you to specify the keyboard layout. In addition, the VCL supports bi-directional localization through the *BiDiMode* and *ParentBiDiMode* properties. The following table lists VCL objects that have these properties:

| Component palette page | VCL object |
|---|---|
| Standard | TButton |
| | TCheckBox |
| | TComboBox |
| | TEdit |
| | TGroupBox |
| | TLabel |
| | TListBox |
| | TMainMenu |
| | TMemo |
| | TPanel |
| | TPopupMenu |
| | TRadioButton |
| | TRadioGroup |
| | TScrollBar |
| Additional | TBitBtn |
| | TCheckListBox |
| | TDrawGrid |
| | TMaskEdit |
| | TScrollBox |
| | TSpeedButton |
| | TStaticLabel |
| | TStringGrid |
| Win32 | TDateTimePicker |
| | THeaderControl |
| | TListView |
| | TMonthCalendar |
| | TPageControl |
| | TRichEdit |
| | TStatusBar |
| | TTabControl |
| Data Controls | TDBCheckBox |
| | TDBComboBox |
| | TDBEdit |
| | TDBGrid |
| | TDBListBox |
| | TDBLookupComboBox |
| | TDBLookupListBox |
| | TDBMemo |

|             |                                          |
|-------------|------------------------------------------|
|             | TDBRadioGroup                            |
|             | TDBRichEdit                              |
|             | TDBText                                  |
| QReport     | TQRDBText                                |
|             | TQRExpr                                  |
|             | TQRLabel                                 |
|             | TQRMemo                                  |
|             | TQRSysData                               |
| Other classes | TApplication (has no *ParentBiDiMode*) |
|             | TForm                                    |
|             | THintWindow (has no *ParentBiDiMode*)    |
|             | TStatusPanel                             |
|             | THeaderSection                           |

**Notes:**    *THintWindow* picks up the *BiDiMode* of the control that activated the hint.

## Bi-directional properties

*TApplication*'s *BiDiKeyboard* and *NonBiDiKeyboard*, support bi-directional localization.

The property *BiDiMode* is a new enumerated type, *TBiDiMode*, with four states: *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign*, and *bdRightToLeftReadingOnly.*

### bdLeftToRight

*bdLeftToRight* draws text using left to right reading order, and the alignment and scroll bars are not changed. For instance, when entering right to left text, such as Arabic or Hebrew, the cursor goes into push mode and the text is entered right to left. Latin text, such as English or French, is entered left to right. *bdLeftToRight* is the default value.

### bdRightToLeft

*bdRightToLeft* draws text using right to let reading order, the alignment is changed and the scroll bar is moved. Text is entered as normal for right-to-left languages such as Arabic or Hebrew. When the keyboard is changed to a Latin language, the cursor goes into push mode and the text is entered left-to-right.

### bdRightToLeftNoAlign

*bdRightToLeftNoAlign* draws text using right to left reading order, the alignment is not changed, and the scroll bar is moved.

### bdRightToLeftReadingOnly

*bdRightToLeftReadingOnly* draws text using right to left reading order, and the alignment and scroll bars are not changed.

## ParentBiDiMode property

*ParentBiDiMode* is a Boolean property. When *True* (the default) the control looks to its parent to determine what *BiDiMode* to use. If the control is a *TForm* object, the form uses the *BiDiMode* setting from *Application*. If all the *ParentBiDiMode* properties are *True*, when you change *Application's BiDiMode* property, all forms and controls in the project are updated with the new setting.

# FlipChildren method

The *FlipChildren* method allows you to flip the position of a container control's children. Container controls are controls that can accept other controls, such as *TForm*, *TPanel*, and *TGroupbox*. *FlipChildren* has a single boolean parameter, *AllLevels*. When *False*, only the immediate children of the container control are flipped. When *True*, all the levels of children in the container control are flipped.

Delphi flips the controls by changing the Left property and the alignment of the control. If a control's left side is five pixels from the left edge of its parent control, after it is flipped the edit control's right side is five pixels from the right edge of the parent control. If the edit control is left aligned, calling *FlipChildren* will make the control right aligned.

To flip a control at design-time select Edit|Flip Children and select either All or Selected, depending on whether you want to flip all the controls, or just the children of the selected control. You can also flip a control by selecting the control on the form, right-clicking, and selecting Flip Children from the context menu.

**Note:** Selecting an edit control and issuing a Flip Children|Selected command does nothing. This is because edit controls are not containers.

## Additional methods

There are several other methods useful for developing applications for bi-directional users.

| Method | Description |
| --- | --- |
| OkToChangeFieldAlignment | Used with database controls. Checks to see if the alignment of a control can be changed. |
| DBUseRightToLeftAlignment | A wrapper for database controls for checking alignment. |
| ChangeBiDiModeAlignment | Changes the alignment parameter passed to it. No check is done for *BiDiMode* setting, it just converts left alignment into right alignment and vice versa, leaving center-aligned controls alone. |
| IsRightToLeft | Returns *True* if any of the right to left options are selected. If it returns *False* the control is in left to right mode. |
| UseRightToLeftReading | Returns *True* if the control is using right to left reading. |
| UseRightToLeftAlignment | Returns *True* if the control is using right to left alignment. It can be overriden for customization. |
| UseRightToLeftScrollBar | Returns *True* if the control is using a left scroll bar. |
| DrawTextBiDiModeFlags | Returns the correct draw text flags for the BiDiMode of the control. |
| DrawTextBiDiModeFlagsReadingOnly | Returns the correct draw text flags for the *BiDiMode* of the control, limiting the flag to its reading order. |
| AddBiDiModeExStyle | Adds the appropriate *ExStyle* flags to the control that is being created. |

## Locale-specific features

You can add extra features to your application for specific locales. In particular, for Asian language environments, you may want your application to control the input method editor (IME) that is used to convert the keystrokes typed by the user into character strings.

VCL components offer you support in programming the IME. Most windowed controls that work directly with text input have an ImeName property that allows you to specify a particular IME that should be used when the control has input focus. They also provide an ImeMode property that specifies how the IME should convert keyboard input. ImeName introduces several protected methods that you can use to control the IME from classes you define. In addition, the global Screen variable provides information about the IMEs available on the user's system.

The global *Screen* variable also provides information about the keyboard mapping installed on the user's system. You can use this to obtain locale-specific information about the environment in which your application is running.

## Designing the user interface

When creating an application for several foreign markets, it is important to design your user interface so that it can accommodate the changes that occur during translation.

The following topics are discussed in this section:

- Text
- Graphic images
- Formats and sort order
- Keyboard mappings

# Text

All text that appears in the user interface must be translated. English text is almost always shorter than its translations. Design the elements of your user interface that display text so that there is room for the text strings to grow. Create dialogs, menus, status bars, and other user interface elements that display text so that they can easily display longer strings. Avoid abbreviations—they do not exist in languages that use ideographic characters.

Short strings tend to grow in translation more than long phrases. The following table provides a rough estimate of how much expansion you should plan for given the length of your English strings:

| Length of English string (in characters) | Expected increase |
| --- | --- |
| 1-5 | 100% |
| 6-12 | 80% |
| 13-20 | 60% |
| 21-30 | 40% |
| 31-50 | 20% |
| over 50 | 10% |

## Graphic images

Ideally, you will want to use images that do not require translation. Most obviously, this means that graphic images should not include text, which will always require translation. If you must include text in your images, it is a good idea to use a label object with a transparent background over an image rather than including the text as part of the image.

There are other considerations when creating graphic images. Try to avoid images that are specific to a particular culture. For example, mailboxes in different countries look very different from each other. Religious symbols are not appropriate if your application is intended for countries that have different dominant religions. Even color can have different symbolic connotations in different cultures.

## Formats and sort order

The date, time, number, and currency formats used in your application should be localized for the target locale. If you use only the Windows formats, there is no need to translate formats, as these are taken from the user's Windows Registry. However, if you specify any of your own format strings, be sure to declare them as resourced constants so that they can be localized.

The order in which strings are sorted also varies from country to country. Many European languages include diacritical marks that are sorted differently, depending on the locale. In addition, in some countries, 2-character combinations are treated as a single character in the sort order. For example, in Spanish, the combination *ch* is sorted like a single unique letter between *c* and *d*. Sometimes a single character is sorted as if it were two separate characters, such as the German *eszett*.

# Keyboard mappings

Be careful with key-combinations shortcut assignments. Not all the characters available on the US keyboard are easily reproduced on all international keyboards. Where possible, use number keys and function keys for shortcuts, as these are available on virtually all keyboards.

## Isolating resources

The most obvious task in localizing an application is translating the strings that appear in the user interface. To create an application that can be translated without altering code everywhere, the strings in the user interface should be isolated into a single module. Delphi automatically creates a .DFM file that contains the resources for your menus, dialogs, and bitmaps.

In addition to these obvious user interface elements, you will need to isolate any strings, such as error messages, that you present to the user. String resources are not included in the .DFM file. You can isolate them by declaring constants for them using the **resourcestring** keyword. For more information about resource string constants, see the Object Pascal Language Guide. It is best to include all resource strings in a single, separate unit.

For information on using resource DLLs in your applications see "Creating resource DLLs" and "Using resource DLLs."

## Creating resource DLLs

Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource DLL. A resource DLL contains all the resources and only the resources for a program. Resource DLLs allow you to create a program that supports many translations simply by swapping the resource DLL.

Use the Resource DLL wizard to create a resource DLL for your program. The Resource DLL wizard requires an open, saved, compiled project. It will create an RC file that contains the string tables from used RC files and **resourcestring** strings of the project, and generate a project for a resource only DLL that contains the relevant forms and the created RES file. The RES file is compiled from the new RC file.

You should create a resource DLL for each translation you want to support. Each resource DLL should have a file name extension specific to the target locale. The first two characters indicate the target language, and the third character indicates the country of the locale. If you use the Resource DLL wizard, this is handled for you. Otherwise, use the following code obtain the locale code for the target translation:

```
unit locales;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    LocaleList: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
function GetLocaleData(ID: LCID; Flag: DWORD): string;
var
  BufSize: Integer;
begin
  BufSize := GetLocaleInfo(ID, Flag, nil, 0);
  SetLength(Result, BufSize);
  GetLocaleinfo(ID, Flag, PChar(Result), BufSize);
  SetLength(Result, BufSize - 1);
end;
{ Called for each supported locale. }
function LocalesCallback(Name: PChar): Bool; stdcall;
var
  LCID: Integer;
begin
  LCID := StrToInt('$' + Copy(Name, 5, 4));
  Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
  Result := Bool(1);
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  EnumSystemLocales(@LocalesCallback, LCID_SUPPORTED);
end;
end.
```

## Using resource DLLs

The executable, DLLs, and packages that make up your application contain all the necessary resources. However, to replace those resources by localized versions, you need only ship your application with localized resource DLLs that have the same name as your EXE, DLL, or BPL files.

When your application starts up, it checks the locale of the local system. If it finds any resource DLLs with the same name as the EXE, DLL, or BPL files it is using, it checks the extension on those DLLs. If the extension of the resource module matches the language and country of the system locale, your application will use the resources in that resource module instead of the resources in the executable, DLL, or package. If there is not a resource module that matches both the language and the country, your application will try to locate a resource module that matches just the language. If there is no resource module that matches the language, your application will use the resources compiled with the executable, DLL, or package.

If you want your application to use a different resource module than the one that matches the locale of the local system, you can set a locale override entry in the Windows registry. Under the HKEY_CURRENT_USER\Software\Borland\Locales key, add your application's path and file name as a string value and set the data value to the extension of your resource DLLs. At startup, the application will look for resource DLLs with this extension before trying the system locale. Setting this registry entry allows you to test localized versions of your application without changing the locale on your system.

For example, the following procedure can be used in an install or setup program to set the registry key value that indicates the locale to use when loading Delphi applications:

```
procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  try
    if Reg.OpenKey('Software\Borland\Locales', True) then
      Reg.WriteString(LocalOverride, FileName);
  finally
    Reg.Free;
end;
```

Within your application, use the global *FindResourceHInstance* function to obtain the handle of the current resource module. For example:

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery,
SizeOf(szQuery));
```

You can ship a single application that adapts itself automatically to the locale of the system it is running on, simply by providing the appropriate resource DLLs.

## Dynamic switching of resource DLLs

In addition to locating a resource DLL at application startup, it is possible to switch resource DLLs dynamically at runtime. To add this functionality to your own applications, you need to include the ReInit unit in your **uses** statement. (ReInit is located in the Richedit sample in the Demos directory.) To switch languages, you should call *LoadResourceModule*, passing the LCID for the new language, and then call *ReinitializeForms*.

For example, the following code switches the interface language to French:

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
```

The advantage of this technique is that the current instance of the application and all of its forms are used. It is not necessary to update the registry settings and restart the application or reacquire resources required by the application, such as logging in to database servers.

When you switch resource DLLs the properties specified in the new DLL overwrite the properties in the running instances of the forms.

**Note:** Any changes made to the form properties at runtime will be lost. Once the new DLL is loaded, default values are not reset. Avoid code that assumes that the form objects are reinitialized to the their startup state, apart from differences due to localization.

## Localizing applications

Once your application is internationalized, you can create localized versions for the different foreign markets in which you want to distribute it.

Ideally, your resources have been isolated into a resource DLL that contains DFM files and a RES file. You can open your forms in the IDE and translate the relevant properties.

**Note:** In a resource DLL project, you cannot add or delete components. It is possible, however, to change properties in ways that could cause runtime errors, so be careful to modify only those properties that require translation. To avoid mistakes, you can configure the Object Inspector to display only localizable properties; to do so, right-click in the Object Inspector and use the View menu to filter out unwanted property categories.

You can open the RC file and translate relevant strings. Use the StringTable editor by opening the RC file from the Project Manager.

If your version of Delphi includes the Integrated Translation Environment, you can use the ITE to manage localization. For more information, see the online Help for the ITE.

## Deploying applications

Once your Delphi application is up and running, you can deploy it. That is, you can make it available for others to run. A number of steps must be taken to deploy an application to another computer so that the application is completely functional. The steps required by a given application vary, depending on the type of application. The following sections describe those steps for deploying applications:

- [Deploying general applications](#)
- [Deploying database applications](#)
- [Deploying Web applications](#)
- [Programming for varying host environments](#)
- [Software license requirements](#)

## Deploying general applications

Beyond the executable file, an application may require a number of supporting files, such as DLLs, package files, and helper applications. In addition, the Windows registry may need to contain entries for an application, from specifying the location of supporting files to simple program settings. The process of copying an application's files to a computer and making any needed registry settings can be automated by an installation program, such as InstallShield Express. These are the main deployment concerns common to nearly all types of applications:

- Using installation programs
- Identifying application files
- Helper applications
- Dll locations

Delphi applications that access databases and those that run across the Web require additional installation steps beyond those that apply to general applications.For additional information on installing database applications, see Deploying database applications. For more information on installing Web applications, see Deploying Web applications. For more information on installing ActiveX controls, see Deploying an ActiveX control on the Web. For information on deploying CORBA applications, see Deploying CORBA applications.

## Using installation programs

Simple Delphi applications that consist of only an executable file are easy to install on a target computer. Just copy the executable file onto the computer. However, more complex applications that comprise multiple files require more extensive installation procedures. These applications require dedicated installation programs.

Setup toolkits automate the process of creating installation programs, often without needing to write any code. Installation programs created with Setup toolkits perform various tasks inherent to installing Delphi applications, including: copying the executable and supporting files to the host computer, making Windows registry entries, and installing the Borland Database Engine for database applications.

InstallShield Express is a setup toolkit that is bundled with Delphi. InstallShield Express is certified for use with Delphi and the Borland Database Engine. InstallShield Express is not automatically installed when Delphi is installed, and must be manually installed to be used to create installation programs. Run the installation program from the Delphi CD to install InstallShield Express. For more information on using InstallShield Express to create installation programs, see the InstallShield Express online help.

Other setup toolkits are available, however, you should only use those certified to deploy the Borland Database Engine.

## Identifying application files

Besides the executable file, a number of other files may need to be distributed with an application.

- Application files, listed by file name extension
- Package files
- ActiveX controls

## Application files, listed by file name extension

The following types of files may need to be distributed with an application.

| Type | File name extension |
| --- | --- |
| Program files | .EXE and .DLL |
| Package files | .BPL and .DCP |
| Help files | .HLP, .CNT, and .TOC (if used) |
| ActiveX files | .OCX (sometimes supported by a DLL) |
| Local table files | .DBF, .MDX, .DBT, .NDX, .DB, .PX, .Y*, .X*, .MB, .VAL, .QBE |

# Package files

If the application uses runtime packages, those package files need to be distributed with the application. InstallShield Express handles the installation of package files the same as DLLs, copying the files and making necessary entries in the Windows registry. Borland recommends installing the runtime package files supplied by Borland in the Windows\System directory. This serves as a common location so that multiple applications would have access to a single instance of the files. For packages you created, it is recommended that you install them in the same directory as the application. Only the .BPL files need to be distributed.

If you are distributing packages to other developers, supply both the .BPL and the .DCP files.

## ActiveX controls

Certain components bundled with Delphi are ActiveX controls. The component wrapper is linked into the application's executable file (or a runtime package), but the .OCX file for the component also needs to be deployed with the application. These components include

- Chart FX, copyright SoftwareFX Inc.
- VisualSpeller Control, copyright Visual Components, Inc.
- Formula One (spreadsheet), copyright Visual Components, Inc.
- First Impression (VtChart), copyright Visual Components, Inc.
- Graph Custom Control, copyright Bits Per Second Ltd.

ActiveX controls of your own creation need to be registered on the deployment computer before use. Installation programs such as InstallShield Express automate this registration process. To manually register an ActiveX control, use the TRegSvr demo application or the Microsoft utility REGSRV32.EXE (not included with all Windows versions).

DLLs that support an ActiveX control also need to be distributed with an application.

## Helper applications

Helper applications are separate programs without which your Delphi application would be partially or completely unable to function. Helper applications may be those supplied with Windows, by Borland, or they might be third-party products. An example of a helper application is the InterBase utility program Server Manager, which administers InterBase databases, users, and security.

If an application depends on a helper program, be sure to deploy it with your application, where possible. Distribution of helper programs may be governed by redistribution license agreements. Consult the documentation for the helper for specific information.

## DLL locations

You can install .DLL files used only by a single application in the same directory as the application. DLLs that will be used by a number of applications should be installed in a location accessible to all of those applications. A common convention for locating such community DLLs is to place them either in the Windows or the Windows\System directory. A better way is to create a dedicated directory for the common .DLL files, similar to the way the Borland Database Engine is installed.

## Deploying database applications

Applications that access databases involve special installation considerations beyond copying the application's executable file onto the host computer. Database access is most often handled by a separate database engine, the files of which cannot be linked into the application's executable file. The data files, when not created beforehand, must be made available to the application. Multi-tier database applications require even more specialized handling on installation, because the files that make up the application are typically located on multiple computers. Two ways of including database access are

- Providing the database engine
- Multi-tiered Distributed Application Services (MIDAS)

## Providing the database engine

Database access for an application is provided by various database engines. An application can use the Borland Database Engine or a third-party database engine. SQL Links is provided (not available in all versions) to enable native access to SQL database systems. The following sections describe installation of the database access elements of an application:

- Borland Database Engine
- Third-party database engines
- SQL Links

## Borland Database Engine

For standard Delphi data components to have database access, the Borland Database Engine (BDE) must be present and accessible. See BDEDEPLOY.TXT for specific rights and limitations on redistributing the BDE.

Borland recommends use of InstallShield Express (or other certified installation program) for installing the BDE. InstallShield Express will create the necessary registry entries and define any aliases the application may require. Using a certified installation program to deploy the BDE files and subsets is important because:

▪        Improper installation of the BDE or BDE subsets can cause other applications using the BDE to fail. Such applications include not only Borland products, but many third-party programs that use the BDE.

▪        Under Windows 95 and Windows NT, BDE configuration information is stored in the Windows registry instead of .INI files, as was the case under 16-bit Windows. Making the correct entries and deletions for install and uninstall is a complex task.

It is possible to install only as much of the BDE as an application actually needs. For instance, if an application only uses Paradox tables, it is only necessary to install that portion of the BDE required to access Paradox tables. This reduces the disk space needed for an application. Certified installation programs, like InstallShield Express, are capable of performing partial BDE installations. Be sure to leave BDE system files that are not used by the deployed application, but that are needed by other programs.

## Third-party database engines

You can use third-party database engines to provide database access for Delphi applications. Consult the documentation or vendor for the database engine regarding redistribution rights, installation, and configuration.

## SQL Links

SQL Links provides the drivers that connect an application (through the Borland Database Engine) with the client software for an SQL database. See DEPLOY.TXT for specific rights and limitations on redistributing SQL Links. As is the case with the Borland Database Engine (BDE), SQL Links must be deployed using InstallShield Express (or other certified installation program).

**Note:** SQL Links only connects the BDE to the client software, not to the SQL database itself. It is still necessary to install the client software for the SQL database system used. See the documentation for the SQL database system or consult the vendor that supplies it for more information on installing and configuring client software.

The following table shows the names of the driver and configuration files SQL Links uses to connect to the different SQL database systems. These files come with SQL Links and are redistributable in accordance with the Delphi license agreement.

| Vendor | Redistributable files |
|---|---|
| Oracle 7 | SQLORA32.DLL and SQL_ORA.CNF |
| Oracle8 | SQLORA8.DLL and SQL_ORA8.CNF |
| Sybase Db-Lib | SQLSYB32.DLL and SQL_SYB.CNF |
| Sybase Ct-Lib | SQLSSC32.DLL and SQL_SSC.CNF |
| Microsoft SQL Server | SQLMSS32.DLL and SQL_MSS.CNF |
| Informix 7 | SQLINF32.DLL and SQL_INF.CNF |
| Informix 9 | SQLINF9.DLL and SQL_INF9.CNF |
| DB/2 | SQLDB232.DLL and SQL_DB2.CNF |
| InterBase | SQLINT32.DLL and SQL_INT.CNF |

Install SQL Links using InstallShield Express or other certified installation program. For specific information concerning the installation and configuration of SQL Links, see the help file SQLLNK32.HLP, by default installed into the main BDE directory.

# Multi-tiered Distributed Application Services (MIDAS)

Multi-tiered Distributed Application Services (MIDAS) consists of the Business Object Broker, OLEnterprise, the Remote DataBroker, and the ConstraintBroker Manager (SQL Explorer). MIDAS provides multi-tier database capability to Delphi applications.

Handle the installation of the executable and related files for a multi-tier application the same as for general applications. Some of the files that comprise MIDAS may need to be installed on the client computer and others on the server computer.For general application installation information, see Deploying general applications. See the text file LICENSE.TXT on the MIDAS CD and the Delphi file DEPLOY.TXT for specific information regarding licensing and redistribution rights for MIDAS.

MIDAS.DLL must be installed onto the client computer and registered with Windows. On the server computer, the files MIDAS.DLL and STDVCL40.DLL must be installed and registered for the Remote DataBroker and DBEXPLOR.EXE for the ConstraintBroker. Installation programs such as InstallShield Express automate the process of registering these DLLs. To manually register the DLLs, use the TRegSvr demo application or the Microsoft utility REGSRV32.EXE (not included with all Windows versions).

The MIDAS deployment CD provides install programs for the client and server portions of OLEnterprise and the Business ObjectBroker. Use only the Setup Launcher on the MIDAS CD to install OLEnterprise.

Following is a list of the minimum required files to be installed onto the server machine.

| | | | |
|---|---|---|---|
| UNINSTALL.EXE | OBJFACT.ICO | W32PTHD.DLL | NBASE.IDL |
| LICENSE.TXT | ODEBKN40.DLL | RPMARN40.DLL | OBJX.EXE |
| README.TXT | ODECTN40.DLL | RPMAWN40.DLL | OLECFG.EXE |
| OLENTER.HLP | RPMEGN40.DLL | RPMCBN40.DLL | OLEWAN40.CAB |
| OLENTER.CNT | ODEDIN40.DLL | RPMCPN40.DLL | OLENTEXP.EXE |
| FILELIST.TXT | ODEEGN40.DLL | BROKER.EXE | OLENTEXP.HLP |
| SETLOG.TXT | ODELTN40.DLL | RPMFEN40.DLL | OLENTEXP.CNT |
| SETLOG.EXE | LIBAVEMI.DLL | RPMUTN40.DLL | BRKCP.EXE |
| OBJPING.EXE | OLEAAN40.DLL | RPMFE.CAT | BROKER.ICO |
| OBJFACT.EXE | OLERAN40.DLL | EXPERR.CAT | |

Following is a list of the required files to be installed onto the client machine.

| | | | |
|---|---|---|---|
| NBASE.IDL | ODEN40.DLL | RPMFEN40.DLL | OLENTEXP.EXE |
| ODECTN40.DLL | RPMARN40.DLL | RPMUTN40.DLL | SETLOG.EXE |
| ODEDIN40.DLL | RPMAWN40.DLL | OLERAN40.DLL | OLECFG.EXE |
| ODEEGN40.DLL | RPMCBN40.DLL | OLEAAN40.DLL | W32PTHD.DLL |
| ODELTN40.DLL | RPMCPN40.DLL | OLEWAN40.CAB | |
| ODEMSG.DLL | RPMEGN40.DLL | OBJX.EXE | |

## Deploying Web applications

Some Delphi applications are designed to be run over the World Wide Web, such as those in the form of Server-side Extension (ISAPI) DLLs, CGI applications, and ActiveForms.

The steps for installing Web applications are the same as those for general applications, except the application's files are deployed on the Web server. For information on installing general applications, see Deploying general applications.

Here are some special considerations for deploying Web applications:

- For database applications, the Borland Database Engine (or alternate database engine) is installed along with the application files on the Web server.
- Security for the directories must not be so high that access to application files, the BDE, or database files is not possible.
- The directory containing an application must have read and execute attributes.
- The application should not use hard-coded paths for accessing database or other files.
- The location of an ActiveX control is indicated by the CODEBASE parameter of the <OBJECT> HTML tag.

## Programming for varying host environments

Due to the nature of the Windows environment, there are a number of factors that vary with user preference or configuration. The following factors can affect an application deployed to another computer:

- Screen resolutions and color depths
- Fonts
- Windows versions

## Screen resolutions and color depths

The size of the Windows desktop and number of available colors on a computer is configurable and dependent on the hardware installed. These attributes are also likely to differ on the deployment computer compared to those on the development computer.

An application's appearance (window, object, and font sizes) on computers configured for different screen resolutions can be handled in various ways:

- Design the application for the lowest resolution users will have (typically, 640x480). Take no special actions to dynamically resize objects to make them proportional to the host computer's screen display. Visually, objects will appear smaller the higher the resolution is set.
- Design using any screen resolution on the development computer and, at runtime, dynamically resize all forms and objects proportional to the difference between the screen resolutions for the development and deployment computers (a screen resolution difference ratio).
- Design using any screen resolution on the development computer and, at runtime, dynamically resize only the application's forms. Depending on the location of visual controls on the forms, this may require the forms be scrollable for the user to be able to access all controls on the forms.

The following topics are discussed in this section:

- Considerations when not dynamically resizing
- Considerations when dynamically resizing forms and controls
- Accommodating varying color depths

## Considerations when not dynamically resizing

If the forms and visual controls that make up an application are not dynamically resized at runtime, design the application's elements for the lowest resolution. Otherwise, the forms of an application run on a computer configured for a lower screen resolution than the development computer may overlap the boundaries of the screen.

For example, if the development computer is set up for a screen resolution of 1024x768 and a form is designed with a width of 700 pixels, not all of that form will be visible within the Windows desktop on a computer configured for a 640x480 screen resolution.

# Considerations when dynamically resizing forms and controls

If the forms and visual controls for an application are dynamically resized, accommodate all aspects of the resizing process to ensure optimal appearance of the application under all possible screen resolutions. Here are some factors to consider when dynamically resizing the visual elements of an application:

▪        The resizing of forms and visual controls is done at a ratio calculated by comparing the screen resolution of the development computer to that of the computer onto which the application installed. Use a constant to represent one dimension of the screen resolution on the development computer: either the height or the width, in pixels. Retrieve the same dimension for the user's computer at runtime using the *TScreen.Height* Screen object's Height or *TScreen.Width* Width property. Divide the value for the development computer by the value for the user's computer to derive the difference ratio between the two computers' screen resolutions.

▪        Resize the visual elements of the application (forms and controls) by reducing or increasing the size of the elements and their positions on forms. This resizing is proportional to the difference between the screen resolutions on the development and user computers. Resize and reposition visual controls on forms automatically by setting the *CustomForm.Scaled* form's Scaled property to *True* and calling *TWincontrol.ScaleBy* its ScaleBy method. The *ScaleBy* method does not change the form's height and width, though. Do this manually by multiplying the current values for the *Height* and *Width* properties by the screen resolution difference ratio.

▪        The controls on a form can be resized manually, instead of automatically with the *TWincontrol.ScaleBy* method, by referencing each visual control in a loop and setting its dimensions and position. The *Height* and *Width* property values for visual controls are multiplied by the screen resolution difference ratio. Reposition visual controls proportional to screen resolution differences by multiplying the *Top* and *Left* property values by the same ratio.

▪        If an application is designed on a computer configured for a higher screen resolution than that on the user's computer, font sizes will be reduced in the process of proportionally resizing visual controls. If the size of the font at design time is too small, the font as resized at runtime may be unreadable. For example, the default font size for a form is 8. If the development computer has a screen resolution of 1024x768 and the user's computer 640x480, visual control dimensions will be reduced by a factor of 0.625 (640 / 1024 = 0.625). The original font size of 8 is reduced to 5 (8 * 0.625 = 5). Text in the application appears jagged and unreadable as Windows displays it in the reduced font size.

▪        Some visual controls, such as *TLabel* and *TEdit*, dynamically resize when the size of the font for the control changes. This can affect deployed applications when forms and controls are dynamically resized. The resizing of the control due to font size changes are in addition to size changes due to proportional resizing for screen resolutions. This effect is offset by setting the AutoSize property of these controls to *False*.

▪        Avoid making use of explicit pixel coordinates, such as when drawing directly to a canvas. Instead, modify the coordinates by a ratio proportionate to the screen resolution difference ratio between the development and user computers. For example, if the application draws a rectangle to a canvas ten pixels high by twenty wide, instead multiply the ten and twenty by the screen resolution difference ratio. This ensures that the rectangle visually appears the same size under different screen resolutions.

## Accommodating varying color depths

To account for all deployment computers not being configured with the same color availability, the safest way is to use graphics with the least possible number of colors. This is especially true for control glyphs, which should typically use 16-color graphics. For displaying pictures, either provide multiple copies of the images in different resolutions and color depths or explain in the application the minimum resolution and color requirements for the application.

# Fonts

Windows comes with a standard set of TrueType and raster fonts. When designing an application to be deployed on other computers, realize that not all computers will have fonts outside the standard Windows set.

Text components used in the application should all use fonts that are likely to be available on all deployment computers.

When use of a nonstandard font is absolutely necessary in an application, you need to distribute that font with the application. Either the installation program or the application itself must install the font on the deployment computer. Distribution of third-party fonts may be subject to limitations imposed by the font creator.

Windows has a safety measure to account for attempts to use a font that does not exist on the computer. It substitutes another, existing font that it considers the closest match. While this may circumvent errors concerning missing fonts, the end result may be a degradation of the visual appearance of the application. It is better to prepare for this eventuality at design time.

To make a nonstandard font available to an application, use the Windows API functions *AddFontResource* and *DeleteFontResource*. Deploy the .FOT file for the nonstandard font with the application.

# Windows versions

When using Windows API functions or accessing areas of the Windows operating system from an application, there is the possibility that the function, operation, or area may not be available on computers with different versions of Windows. For example, Services are only pertinent to the Windows NT operating system. If an application is to act as a Service or interact with one, this would fail if the application is installed under Windows 95.

To account for this possibility, you have a few options:

▪        Specify in the application's system requirements the versions of Windows on which the application can run. It is the user's responsibility to install and use the application only under compatible Windows versions.

▪        Check the version of Windows as the application is installed. If an incompatible version of Windows is present, either halt the installation process or at least warn the installer of the problem.

▪        Check the Windows version at runtime, just prior to executing an operation not applicable to all versions. If an incompatible version of Windows is present, abort the process and alert the user. Alternately, provide different code to run dependent on different versions of Windows. Some operations are performed differently in Windows 95 than in Windows NT. Use the Windows API function *GetVersionEx* to determine the Windows version.

## Software license requirements

The distribution of some files associated with Delphi applications is subject to limitations or cannot be redistributed at all. The following documents describe the legal stipulations regarding the distribution of these files where limitations exist:

- DEPLOY.TXT

  DEPLOY.TXT covers the some of the legal aspects of distributing of various components and utilities, and other product areas that can be part of or associated with your application. DEPLOY.TXT is a text file installed in the main directory. The topics covered include, but are not limited to

- .EXE, .DLL, and .BPL files
- Components and design-time packages
- Borland Database Engine (BDE)
- ActiveX controls
- Sample Images
- Multi-tiered Distributed Application Services (MIDAS)
- SQL Links
- README.TXT

  README.TXT contains last minute information about Delphi possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. README.TXT is a Windows help file installed into the main Delphi directory.

- No-nonsense license agreement

  The Delphi no-nonsense license agreement, a printed document, covers other legal rights and obligations concerning Delphi.

- Third-party product documentation

  Redistribution rights for third-party components, utilities, helper applications, database engines, and other products are governed by the vendor supplying the product. Consult the documentation for the product or the vendor for information regarding the redistribution of the product with Delphi applications prior to distribution.

**Related topic groups**

**Building applications with Delphi**

- Using Object Pascal with the VCL
- Building applications, components, and libraries
- Common programming tasks
- Developing the application user interface
- Working with controls
- Working with graphics
- Working with multimedia
- Writing multi-threaded applications
- Working with packages and components
- Creating international applications
- Deploying applications

**Using Object Pascal with the VCL**

- Using Object Pascal with the VCL: Overview
- Using the object model
- What is an object?
- Examining a Delphi object
- Changing the name of a component
- Inheriting data and code from an object
- Objects, components, and controls
- Scope and qualifiers
- Private, protected, public, and published declarations
- Using object variables
- Creating, instantiating, and destroying objects
- Components and ownership
- Using components
- Delphi's standard components
- Properties common to visual components
- Position and size properties
- Display properties
- Parent properties
- Navigation properties
- Drag-and-drop properties
- Drag-and-dock properties
- Text controls
- Properties common to all text controls
- Properties shared by memo and rich text controls
- Rich text controls
- Specialized input controls
- Scroll bars
- Track bars
- Up-down controls
- Hot key controls
- Splitter control
- Buttons and similar controls
- Button controls
- Bitmap buttons
- Speed buttons
- Check boxes
- Radio buttons
- Toolbars
- Cool bars
- Handling lists
- List boxes and check-list boxes

- [Combo boxes](#)
- [Tree views](#)
- [List views](#)
- [Date-time pickers and month calendars](#)
- [Grouping components](#)
- [Group boxes and radio groups](#)
- [Panels](#)
- [Header controls](#)
- [Header controls](#)
- [Page controls](#)
- [Header controls](#)
- [Visual feedback](#)
- [Labels and static-text components](#)
- [Status bars](#)
- [Progress bars](#)
- [Help and hint properties](#)
- [Grids](#)
- [Draw grids](#)
- [String grids](#)
- [Graphic display](#)
- [Images](#)
- [Shapes](#)
- [Bevels](#)
- [Paint boxes](#)
- [Animation control](#)
- [Windows common dialog boxes](#)
- [Setting component properties](#)
- [Using the Object Inspector](#)
- [Using property editors](#)
- [Setting properties at runtime](#)
- [Calling methods](#)
- [Working with events and event handlers](#)
- [Generating a new event handler](#)
- [Generating a handler for a component's default event](#)
- [Locating event handlers](#)
- [Associating an event with an existing event handler](#)
- [Using the Sender parameter](#)
- [Displaying and coding shared events](#)
- [Associating menu events with event handlers](#)
- [Deleting event handlers](#)
- [Using helper objects](#)
- [Working with lists](#)
- [Working with string lists](#)

- [Loading and saving string lists](#)
- [Creating a new string list](#)
- [Manipulating strings in a list](#)
- [Counting the strings in a list](#)
- [Accessing a particular string](#)
- [Finding the position of a string in the list](#)
- [Iterating through strings in a list](#)
- [Adding a string to a list](#)
- [Deleting a string from a list](#)
- [Copying a complete string list](#)
- [Associating objects with a string list](#)
- [Windows registry and INI files](#)
- [Using streams](#)
- [Using data modules and remote data modules](#)
- [Creating and editing data modules](#)
- [Creating business rules in a data module](#)
- [Accessing a data module from a form](#)
- [Adding a remote data module to an application server project](#)
- [Using the Object Repository](#)
- [Sharing items within a project](#)
- [Adding items to the Object Repository](#)
- [Sharing objects in a team environment](#)
- [Using an Object Repository item in a project](#)
- [Copying an item](#)
- [Inheriting an item](#)
- [Using an item](#)
- [Using project templates](#)
- [Modifying shared items](#)
- [Specifying a default project, new form, and main form](#)
- [Adding custom components to the IDE](#)

## Building applications, components, and libraries

- Creating applications
- Windows applications
- User interface models
- SDI Applications
- MDI applications
- Setting IDE, project, and compilation options
- Programming templates
- Console applications
- Service applications
- Service threads
- Service name properties
- Debugging services
- Creating packages and DLLs
- When to use packages and DLLs
- Writing database applications
- Building distributed applications
- Distributing applications using TCP/IP
- Using sockets in applications
- Creating Web server applications
- Distributing applications using COM and DCOM
- Distributing applications using CORBA
- Distributing database applications

**Common programming tasks**

## Developing the application user interface

- [Registering actions](#)
- [Writing action list editors](#)
- [Demo programs](#)

## Working with controls

## Working with graphics

**Working with multimedia**

- Adding silent video clips to an application

- Example of adding silent video clips

- Adding audio and/or video clips to an application

- Example of adding audio and/or video clips

**Writing multi-threaded applications**

- Using threads: Overview
- Defining thread objects
- Initializing the thread
- Writing the thread function
- Using the main VCL thread
- Using thread-local variables
- Checking for termination by other threads
- Writing clean-up code
- CoordinatingThreads
- Avoiding simultaneous access
- Locking objects
- Using critical sections
- Using the multi-read exclusive-write synchronizer
- Waiting for other threads
- Waiting for a thread to finish executing
- Waiting for a task to be completed
- Executing thread objects
- Overriding the default priority
- Starting and stopping threads
- Using threads in distributed applications
- Using threads in message-based servers
- Using threads with distributed objects
- Debugging multi-threaded applications

**Working with packages and components**

# Creating international applications

## Deploying applications

Related topic groups

- Deploying applications: Overview
- Deploying general applications
- Using installation programs
- Identifying application files
- Application files, listed by file name extension
- Package files
- ActiveX controls
- Helper applications
- DLL locations
- Deploying database applications
- Providing the database engine
- Borland Database Engine
- Third-party database engines
- SQL Links
- Multi-tiered Distributed Application Services (MIDAS)
- Deploying Web applications
- Programming for varying host environments
- Screen resolutions and color depths
- Considerations when not dynamically resizing
- Considerations when dynamically resizing forms and controls
- Accommodating varying color depths
- Fonts
- Windows versions
- Software license requirements

**Link not found**

The topic you requested is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

The topic you requested is now loading. If it does not appear within a few seconds, the topic is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.