Delphi Component Writer's Guide

Delphi for Windows

Introduction

Copyright Agreement

Delphi is not just a development environment for visually creating applications with components. It also includes everything you need to create the components for building applications in the same environment, using the same Object Pascal language.

The Delphi Component Writer's Guide and its accompanying Help file (CWG.HLP) describe everything you need to know to write components for Delphi applications. The printed manual and the Help file contain the same information in different forms.

This material has two purposes:

1. To teach you how to create working components
2. To ensure that the components you write are well-behaved parts of the Delphi environment

Whether you're writing components for your own applications or for commercial distribution, this book will guide you to writing components that fit in well with any Delphi application.

What is a component?

Components are the building blocks of Delphi applications. Although most components represent visible parts of a user interface, components can also represent nonvisual elements in a program, such as timers and databases.

There are three different levels at which to think about components: a functional definition, a technical definition, and a practical definition.

The functional definition of "component"

From the end user's perspective, a component is something to choose from the palette and use in an application by manipulating it in the Forms Designer or in code. From the component writer's perspective, however, a component is an object in code. Although there are few real restrictions on what you can do when writing a component, it's good to keep in mind what the end user expects when using the components you write.

Before you attempt to write components, we strongly recommend that you become familiar with the existing components in Delphi so you can make your components

familiar to users. Your goal should be to make your components "feel" as much like other components as possible.

The technical definition of "component"

At the simplest level, a component is any object descended from the type *TComponent*. *TComponent* defines the most basic behavior that all components must have, such as the ability to appear on the Component palette and operate in the Forms Designer.

But beyond that simple definition are several larger issues. For example, although *TComponent* defines the basic behavior needed to operate the Delphi environment, it can't know how to handle all the specific additions you make to your components. You'll have to specify those yourself.

Although it's not difficult to create well-behaved components, it does require that you pay close attention to the standards and conventions spelled out in this book.

The component writer's definition of "component"

At a very practical level, a component is any element that can "plug into" the Delphi development environment. It can represent almost any level of complexity, from a simple addition to one of the standard components to a vast, complex interface to another hardware or software system. In short, a component can do or be anything you can create in code, as long as it fits into the component framework.

The definition of a component, then, is essentially an interface specification. This manual spells out the framework onto which you build your specialized code to make it work in Delphi.

Defining the limits of "component" is therefore like defining the limits of programming. We can't tell you every kind of component you can create, any more than we can tell you all the programs you can write in a given language. What we can do is tell you how to write your code so that it fits well in the Delphi environment.

What's different about writing components?

There are three important differences between the task of creating a component for use in Delphi and the more common task of creating an application that uses components:

- Component writing is nonvisual
- Component writing requires deeper knowledge of objects
- Component writing follows more conventions

Component writing is nonvisual

The most obvious difference between writing components and building applications with Delphi is that component writing is done strictly in code. Because the visual design of Delphi applications requires completed components, creating those components requires writing Object Pascal code.

Although you can't use the same visual tools for creating components, you can use all the programming features of the Delphi development environment, including the Code Editor, integrated debugger, and ObjectBrowser.

Component writing requires deeper knowledge of objects

Other than the non-visual programming, the biggest difference between creating components and using them is that when you create a new component you need to derive a new object type from an existing one, adding new properties and methods. Component users, on the other hand, use existing components and customize their behavior at design time by changing properties and specifying responses to events.

When deriving new objects, you have access to parts of the ancestor objects unavailable to end users of those same objects. These parts intended only for component writers are collectively called the *protected interface* to the objects. Descendant objects also need to call on their ancestor objects for a lot of their implementation, so component writers need to be familiar with that aspect of object-oriented programming.

Component writing follows more conventions

Writing a component is a more traditional programming task than visual application creation, and there are more conventions you need to follow than when you use existing components. Probably the most important thing to do before you start writing components of your own is to really use the components that come with Delphi, to get a feeling for the obvious things like naming conventions, but also for the kinds of abilities component users will expect when they use your components.

The most important thing that component users expect of components is that they should be able to do almost anything to those components at any time. Writing components that fulfill that expectation is not difficult, but it requires some forethought and adherance to conventions.

Creating a component (summary)

In brief, the process of creating your own component consists of these steps:

1. Create a unit for the new component.
2. Derive a component type from an existing component type.
3. Add properties, methods, and events as needed.
4. Register your component with Delphi.
5. Create a Help file for your component and its properties, methods, and events.

All these steps are covered in detail in this manual. When you finish, the complete component includes four files:

1. A compiled unit (.DCU file)
2. A palette bitmap (.DCR file)
3. A Help file (.HLP file)
4. A Help-keyword file (.KWF file)

Although only the first file is required, the others make your components much more useful and usable.

What's in this book?

The Delphi Component Writer's Guide is divided into two parts. The first part explains all the aspects of building components. The second part provides several complete examples of writing different kinds of components.

Part I, "Creating components"

The chapters in this part of the book describe the various parts of components and how you create them.

- **Chapter 1, "Overview of component creation,"** explains the basic steps involved in the creation of any component. You should read this chapter before starting to create components.
- **Chapter 2, "OOP for component writers,"** presents several topics component writers need to know about programming with objects.
- **Chapter 3, "Creating properties,"** presents the procedures for adding properties to components.
- **Chapter 4, "Creating events,"** describes the process of adding events to components.
- **Chapter 5, "Creating methods,"** explains the process of adding methods to components and describes the conventions component writers should follow in naming and protecting methods.
- **Chapter 6, "Using graphics in components,"** describes the aspects of the Delphi graphics encapsulation that are particularly useful to component writers.
- **Chapter 7, "Handling messages,"** describes the Windows messaging system and the mechanisms built into Delphi components to handle messages.
- **Chapter 8, "Registering components,"** presents the requirements for customizing the way your components interact with the Delphi development environment, including providing Help to component users.

Part II, "Sample components"

The chapters in this part of the book give concrete examples of making components.

- **Chapter 9, "Modifying an existing component,"** demonstrates the simplest way to create a new component, by making modifications to an already-working component.
- **Chapter 10, "Creating a graphic component,"** shows an example of how to create a simple graphical component from scratch.

- **Chapter 11, "Customizing a grid,"** shows how to create a component based on one of the abstract component types in the component library.
- **Chapter 12, "Making a control data-aware,"** demonstrates how to take an existing control and make it into a data-aware browsing control.
- **Chapter 13, "Making a dialog box a component,"** explains how to take a complete, working form and turn it into a reusable dialog box component.
- **Chapter 14, "Building a dialog box into a DLL,"** shows how to take a form and its controls and build it into a dynamic-link library (DLL) that any Windows application can use.

What's not in this book?

Although this book touches on all the aspects that define a Delphi component, it can't possibly cover every aspect of every kind of component you might want to write. If you want to create a component that operates on any system at a low level, you need to understand that system's low-level operations.

For example, if you want to create components that take advantage of the intricacies of the communications functions built into Windows, you need to know enough about communications and the Windows API functions that implement them to make the appropriate calls to those functions from within your component. Similarly, if you want to access data in databases not directly supported by the Borland Database Engine, you need to know how to program the interface for that database so your component can provide access.

On the other hand, if all you want to do is create some slightly customized versions of the standard components provided with Delphi, all you really need is a good working knowledge of the Delphi development environment and its standard components, and some fundamental programming skills.

Manual conventions

The printed manuals for Delphi use the special typefaces and symbols described in Table Intro.1 to indicate special text.

Table Intro.1 Typefaces and symbols in these manuals

Typeface or symbol	Meaning
`Monospace type`	Monospaced text represents text as it appears onscreen or in Object Pascal code. It also represents anything you must type.
[ ]	Square brackets in text or syntax listings enclose optional items. Text of this sort should not be typed verbatim.
Boldface	Boldfaced words in text or code listings represent Object Pascal reserved words or compiler options.
Italics	Italicized words in text represent Object Pascal identifiers, such as variable or type names. Italics are also used to emphasize certain words, such as new terms.

Table Intro.1 Typefaces and symbols in these manuals (continued)

Typeface or symbol	Meaning
Keycaps	This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."
■	This symbol indicates the beginning of a procedure description. The text that follows describes a set of general steps for performing a specified kind of task.
➤	This symbol indicates a specific action you should take, such as a step in an example.

Part

I

Creating components

One of the key features of Delphi is that you can extend the library of components available for your applications from within Delphi itself. The chapters in this part describe all the aspects of component creation.

These are the topics you need to master to create your own components:

- Overview of component creation
- OOP for component writers
- Creating properties
- Creating events
- Creating methods
- Using graphics in components
- Handling messages
- Registering components

Chapter

1

Overview of component creation

This chapter provides a broad overview of component architecture, the philosophy of component design, and the process of writing components for Delphi applications.

The main topics discussed are

- The Visual Component Library
- Components and objects
- How do you create components?
- What goes in a component?
- Creating a new component
- Testing uninstalled components

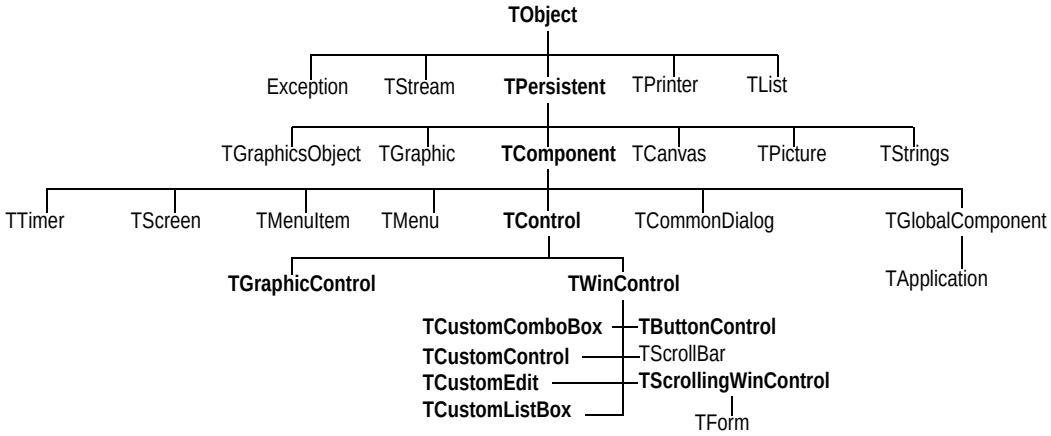
All this material assumes you have some familiarity with using Delphi and its standard components.

The Visual Component Library

Delphi's components are all part of an object hierarchy called the Visual Component Library (VCL). Figure 1.1 shows the relationship of the objects that make up VCL. Chapter 2 discusses object hierarchies and the inheritance relationships between objects.

Note that the type *TComponent* is the shared ancestor of every component in the VCL. *TComponent* provides the minimal properties and events necessary for a component to work in Delphi. The various branches of the library provide other, more specialized capabilities.

Figure 1.1 The Visual Component Library object hierarchy



When you create a component, you add to the VCL by deriving a new object from one of the existing object types in the hierarchy.

Components and objects

Because components are objects, component writers work with objects at a different level than component users do. Creating new components requires that you derive new types of objects. Chapter 2 describes in detail the kinds of object-oriented tasks component writers need to use.

Briefly, there are two main differences between creating components and using components. When creating components,

- You have access to parts of the object that are inaccessible to end users
- You add new parts (such as properties) to your components

Because of these differences, you need to be aware of more conventions, and you need to think in terms of how end users will use the components you write.

How do you create components?

A component can be almost any program element you want to manipulate at design time. Creating a new component means deriving a new component object type from an existing type. You can derive a new component from any existing component, but the following are the most common ways to create new components:

- Modifying existing controls
- Creating original controls
- Creating graphic controls
- Subclassing Windows controls
- Creating nonvisual components

Table 1.1 summarizes the different kinds of components and the object types you use as starting points for each.

Table 1.1 Component creation starting points

To do this	Start with this type
Modify an existing component	Any existing component, such as *TButton* or *TListBox*, or an abstract component type, such as *TCustomListBox*.
Create an original control	*TCustomControl*
Create a graphic control	*TGraphicControl*
Use an existing Windows control	*TWinControl*
Create a non-visual component	*TComponent*

You can also derive other objects that are not components, but you cannot manipulate them in a form. Delphi includes a number of that kind of object, such as *TINIFile* or *TFont*.

Modifying existing controls

The simplest way to create a component is to start from an existing, working component and customize it. You can derive a new component from any of the components provided with Delphi. For instance, you might want to change the default property values of one of the standard controls.

There are certain controls, such as list boxes and grids, that have a number of variations on a basic theme. In those cases, Delphi provides an abstract control type (with the word "custom" in its name, such as *TCustomGrid*) from which to derive customized versions.

For example, you might want to create a special kind of list box that does not have some of the properties of the standard *TListBox* type. You can't remove a property from an ancestor type, so you need to derive your component from something higher in the hierarchy than *TListBox*. Rather than forcing you to go clear back to an abstract control type and reinvent all the list box functions, the Visual Component Library (VCL) provides *TCustomListBox*, which implements all the properties needed for a list box, but does not publish all of them.

When deriving a component from one of the abstract types such as *TCustomListBox*, you publish those properties you want to make available in your component and leave the rest protected. Chapter 3 explains publishing inherited properties.

Chapter 9 and Chapter 11 show examples of modifying existing controls.

Creating original controls

A standard control is an item that's visible at run time, usually one the user can interact with. These standard controls all descend from the object type *TWinControl*. When you create an original control (one that's not related to any existing control), you use *TWinControl* as the starting point.

The key aspect of a standard control is that it has a window handle, embodied in a property called *Handle*. The window handle means that Windows "knows about" the control, so that, among other things,

- The control can receive the input focus
- You can pass the handle to Windows API functions (Windows needs a handle to identify which window to operate on.)

If your control doesn't need to receive input focus, you can make it a graphic control, which saves system resources. The next section describes graphic controls.

All the components that represent standard windows controls, such as push buttons, list boxes, and edit boxes, descend from *TWinControl* except *TLabel*, since label controls never receive the input focus.

Creating graphic controls

Graphic controls are very similar to custom controls, but they don't carry the overhead of being Windows controls. That is, Windows doesn't know about graphic controls. They have no window handles, and therefore consume no system resources. The main restriction on graphic controls is that they cannot receive the input focus.

Delphi supports the creation of custom controls through the type *TGraphicControl*. *TGraphicControl* is an abstract type derived from *TControl*. Although you can derive controls from *TControl*, you're better off deriving from *TGraphicControl*, which provides a canvas to paint on, and handles *WM_PAINT* messages, so all you need to do is override the *Paint* method.

Chapter 10 shows an example of creating a graphic control.

Subclassing Windows controls

Windows has a concept called a *window class* that is somewhat similar to the object-oriented programming concept of object or class. A window class is a set of information shared between different instances of the same sort of window or control in Windows.

When you create a new kind of control (usually called a *custom control*) in traditional Windows programming, you define a new window class and register it with Windows. You can also base a new window class on an existing class, which is called *subclassing*.

In traditional Windows programming, if you wanted to create a custom control, you had to write it in a dynamic-link library (DLL), much like the standard Windows controls, and provide an interface to it.

Using Delphi, you can create a component "wrapper" around any existing Windows class. So if you already have a library of custom controls that you want to use in your Delphi applications, you can create Delphi components that let you use your existing controls and derive new controls from them just as you would any other component.

Although this manual does not include an example of subclassing a Windows control, you can see the techniques used in the components in the *StdCtls* unit that represent the standard Windows controls, such as *TButton*.

Creating nonvisual components

The abstract object type *TComponent* is the base type for all components. The only components you'll create directly from *TComponent* are nonvisual components. Most of the components you'll write will probably be various kinds of visual controls.

TComponent defines all the properties and methods essential for a component to participate in the Form Designer. Thus, any component you derive from *TComponent* will already have design capability built into it.

Nonvisual components are fairly rare. You'll mostly use them as an interface for nonvisual program elements (much as Delphi uses them for database elements) and as placeholders for dialog boxes (such as the file dialog boxes).

Chapter 13 shows an example of creating a nonvisual component.

What goes in a component?

There are few restrictions on what you can put in the components you write. However, there are certain conventions you should follow if you want to make your components easy and reliable for the people who will use them.

This section discusses the philosophies underlying the design of components, including the following topics:

- Removing dependencies
- Properties, events, and methods
- Graphics encapsulation
- Registration

Removing dependencies

Perhaps the most important philosophy behind the creation of Delphi's components is the necessity of removing dependencies. One of the things that makes components so easy for end users to incorporate into their applications is the fact that there are generally no restrictions on what they can do at any given point in their code.

The very nature of components suggests that different users will incorporate them into applications in varying combinations, orders, and environments. You should design your components so that they function in any context, without requiring any preconditions.

An example of removing dependencies

An excellent example of removing dependencies in components is the *Handle* property of windowed controls. If you've written Windows applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you don't access a window or control until you've created it by calling the *CreateWindow* API function. Calling API functions with invalid handles causes a multitude of problems.

Delphi components protect users from worrying about window handles and whether they are valid by ensuring that a valid handle is always available when needed. That is, by using a property for the window handle, the component can check whether the window has been created, and therefore whether there is a valid window handle. If the handle isn't already valid, the property creates the window and returns the handle. Thus, any time a user's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing the background tasks such as creating the window, components allow developers to focus on what they really want to do. If a developer needs to pass a window handle to an API function, it shouldn't be necessary to first check to make sure there's a valid handle and, if necessary, create the window. With component-based programming, the programmer can write assuming that things will work, instead of constantly checking for things that might go wrong.

Although it might take a little more time to create components that don't have dependencies, it's generally time well spent. Not only does it keep users of your components from having to repeatedly perform the same tasks, but it also reduces your documentation and support burdens, since you don't have to provide and explain numerous warnings or resolve the problems users might have with your components.

Properties, events, and methods

Outside of the visible image the component user manipulates in the form at design time, the most obvious attributes of a component are its properties, events, and methods. Each of these is sufficiently important that it has its own chapter in this book, but this section will explain a little of the philosophy of implementing them.

Properties

Properties give the component user the illusion of setting or reading the value of a variable in the component while allowing the component writer to hide the underlying data structure or to implement side effects of accessing the value.

There are several advantages to the component writer in using properties:

- Properties are available at design time.

  This allows the component user to set and change initial values of properties without having to write code.

- Properties can check values or formats as the user assigns them.

  Validating user input prevents errors caused by invalid values.

- The component can construct appropriate values on demand.

  Perhaps the most common type of error programmers make is to reference a variable that hasn't had an initial value assigned. By making the value a property, you can ensure that the value read from the property is always valid.

Chapter 3 explains how to add properties to your components.

Events

Events are connections between occurrences determined by the component writer (such as mouse actions and keystrokes) and code written by component users ("event handlers"). In essence, an event is the component writer's way of providing a hook for the component user to specify what code to execute when a particular occurrence happens.

It is events, therefore, that allow component users to *be* component users instead of component writers. The most common reason for subclassing in traditional Windows applications is that users want to specify a different response to, for example, a Windows message. But in Delphi, component users can specify handlers for predefined events without subclassing, so they don't need to derive their own components.

Chapter 4 explains how to add events for standard Windows occurrences or events you define yourself.

Methods

Methods are procedures or functions built into a component. Component users use methods to direct a component to perform a specific action or return a certain value not covered by a property. Methods are also useful for updating several related properties with a single call.

Because they require execution of code, methods are only available at run time.

Chapter 5 explains how to add methods to your components.

Graphics encapsulation

Delphi takes most of the drudgery out of Windows graphics by encapsulating the various graphic tools into a *canvas*. The canvas represents the drawing surface of a window or control, and contains other objects, such as a pen, a brush, and a font. A canvas is much like a Windows device context, but it takes care of all the bookkeeping for you.

If you've ever written a graphic Windows application, you're familiar with the kinds of requirements Windows' graphics device interface (GDI) imposes on you, such as limits on the number of device contexts available, and restoring graphic objects to their initial state before destroying them.

When working with graphics in Delphi, you need not concern yourself with any of those things. To draw on a form or component, you access the *Canvas* property. If you want to customize a pen or brush, you set the color or style. When you finish, Delphi takes care of disposing of the resources. In fact, it caches resources, so if your application frequently uses the same kinds of resources, it will probably save a lot of creating and recreating.

Of course, you still have full access to the Windows GDI, but you'll often find that your code is much simpler and runs faster if you use the canvas built into Delphi components. Delphi's graphics features are detailed in Chapter 6.

Registration

Before your components can operate in Delphi at design time, you have to register them with Delphi. Registration tells Delphi where you want your component to appear on the Component palette. There are also some customizations you can make to the way Delphi stores your components in the form file. Registration is explained in Chapter 8.

Creating a new component

There are several steps you perform whenever you create a new component. All the examples given that create new components assume you know how to perform these steps.

- You can create a new component two ways:
  - Creating a component manually
  - Using the Component Expert

Once you do either of those, you have at least a minimally functional component ready to install on the Component palette. Installing components is explained in Chapter 2 of the *Delphi User's Guide*. After installing, you can add your new component to a form and test it in both design time and run time. You can then add more features to the component, update the palette, and continue testing.

Creating a component manually

The easiest way to create a new component is to use the Component Expert. However, you can also perform the same steps manually.

- Creating a component manually requires three steps:
  1. Creating a new unit
  2. Deriving the component object
  3. Registering the component

Creating a new unit

A unit is a separately compiled module of Object Pascal code. Delphi uses units for a number of purposes. Every form has its own unit, and most components (or logical groups of components) have their own units as well. Units are discussed in more detail in online Help.

When you create a component, you either create a new unit for the component, or add the new component to an existing unit.

- To create a unit for a component, choose File | New Unit.

  Delphi creates a new file and opens it in the Code Editor.

- To add a component to an existing unit, choose File | Open to choose the source code for an existing unit.

Note When adding a component to an existing unit, make sure that unit already contains only component code. Adding component code to a unit that contains, for example, a form, will cause errors in the Component palette.

Once you have either a new or existing unit for your component, you can derive the component object.

Deriving the component object

Every component is an object descended from the type *TComponent*, from one of its more specialized descendants, such as *TControl* or *TGraphicControl*, or from an existing component type. "How do you create components?" on page 10 describes which types to descend from for different kinds of components.

Deriving new objects is explained in more detail in "Creating new objects" in Chapter 2.

- To derive a component object, add an object type declaration to the **interface** part of the unit that will contain the component.

➤ To create, for example, the simplest component type, a non-visual component descended directly from *TComponent*, add the following type definition to the **interface** part of your component unit:

```
type
  TNewComponent = class(TComponent)
  end;
```

You can now register *TNewComponent*. Note that the new component does nothing different from *TComponent* yet. However, you've created a framework on which you'll build your new component.

Registering the component

Registering a component is a simple process that tells Delphi what components to add to its component library, and which pages of the Component palette the components should appear on. Chapter 8 describes the registration process and its nuances in much more detail.

- To register a component,

1. Add a procedure named *Register* to the **interface** part of the component's unit.

   Register takes no parameters, so the declaration is very simple:

   ```
   procedure Register;
   ```

   If you're adding a component to a unit that already contains components, it should already have a *Register* procedure declared, so you don't need to change the declaration.

2. Write the *Register* procedure in the **implementation** part of the unit, calling *RegisterComponents* for each component you want to register.

   RegisterComponents is a procedure that takes two parameters: the name of a Component palette page and a set of component types. If you're adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

➤ To register a component named *TNewComponent* and place it on the Samples page of the Component palette, add the following *Register* procedure to the unit that contains *TNewComponent*'s declaration:

```
procedure Register;
begin
  RegisterComponents('Samples', [TNewComponent]);
end;
```

Once you register a component, you can install the component onto the Component palette.

Using the Component Expert

You can use the Component Expert to create a new component. Using the Component Expert simplifies the initial stages of creating a new component, as you need only specify three things:

- The name of the new component
- The ancestor type
- The Component palette page you want it to appear on

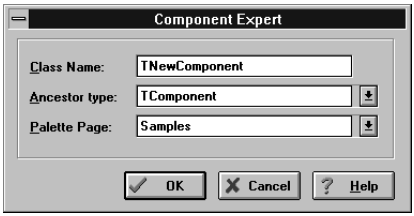
The Component Expert performs the same tasks you would do when creating a component manually, namely

- Creating a new unit
- Deriving the component object
- Registering the component

The Component Expert cannot add components to an existing unit. You must add components to existing units manually.

■ To open the Component Expert, choose File | New Component.

Figure 1.2 The Delphi Component Expert



After you fill in the fields in the Component Expert, choose OK. Delphi creates a new unit containing the type declaration and the *Register* procedure, and adds a **uses** clause that includes all the standard Delphi units.

You should save the unit right away, giving the unit a meaningful name.

Testing uninstalled components

You can test the run-time behavior of a component before you install it on the Component palette. This is particularly useful for debugging newly-created components, but you can use the same technique for testing any component, regardless of whether the component appears on the Component palette.

In essence, you can test an uninstalled component by emulating the actions performed by Delphi when a user places a component from the Component palette on a form.

■ To test an uninstalled component, do the following:

1. Add the name of component's unit to the form unit's **uses** clause.
2. Add an object field to the form to represent the component.

   This is one of the main differences between the way you add components and the way Delphi does it. You add the object field to the public part at the bottom of the form's type declaration. Delphi would add it above, in the part of the type declaration that it manages.

   You should never add fields to the Delphi-managed part of the form's type declaration. The items in that part of the type declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.
3. Attach a handler to the form's *OnCreate* event.
4. Construct the component in the form's *OnCreate* handler.

   When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for destroying the component when the time comes). You will nearly always pass *Self* as the owner. In a method, *Self* is a reference to the object that contains the method. In this case, in the form's *OnCreate* handler, *Self* refers to the form.
5. Assign the component's *Parent* property.

   Setting the *Parent* property is always the first thing to do after constructing the component. The parent is the component that visually contains the component, which is most often the form, but might be a group box or panel. Normally, you'll set *Parent* to *Self*, that is, the form. Always set *Parent* before setting other properties of the component.
6. Set any other component properties as desired.

➤ Suppose you want to test a new component of type *TNewComponent* in a unit named *NewTest*. Create a new project, then follow the steps to end up with a form unit that looks like this:

```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, NewTest;                          { 1. Add NewTest to uses clause }
```

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);            { 3. Attach a handler to OnCreate }
  private
    { Private declarations }
  public
    { Public Declarations }
    NewComponent1: TNewComponent;                          { 2. Add an object field }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  NewComponent1 := TNewComponent.Create(Self);          { 4. Construct the component }
  NewComponent1.Parent := Self;                            { 5. Set Parent property }
  NewComponent.Left := 12;                              { 6. Set other properties... }
  ⋮                                                         { ...continue as needed }
end;
end.
```

Summary

This chapter introduces the concepts explained more thoroughly in the rest of this manual. The remaining chapters in this part expand on the details of the process of creating components and their properties, methods, and events. For concrete examples of component creation, see Part II.

Chapter

2

OOP for component writers

Working with Delphi, you've encountered the idea that an object contains both data and code, and that you can manipulate objects both at design time and run time. In that sense, you've become a component user.

When you create new kinds of components, you deal with objects in ways that end users never need to. Before you start creating components, you need to be familiar with these topics related to object-oriented programming (OOP) in general, presented in this chapter:

- Creating new objects
- Ancestors and descendants
- Controlling access
- Dispatching methods
- Objects and pointers

This material assumes familiarity with the topics discussed in Chapter 6 of the *Delphi User's Guide*. If you have used objects in Borland's previous Pascal products, you should also read through Appendix A for a discussion of changes in the Pascal object model.

Creating new objects

The primary difference between component users and component writers is that users manipulate *instances* of objects, and writers create new *types* of objects. This concept is fundamental to object-oriented programming, and an understanding of the distinction is extremely important if you plan to create your own components.

The concept of types and instances is not unique to objects. Programmers continually work with types and instances, but they don't generally use that terminology. "Type" is a very common idea in Object Pascal programs, and as a programmer you generally create variables of a type. Those variables are instances of the type.

Object types are generally more complex than simple types such as *Integer*, but by assigning different values to instances of the same type, a user can perform quite different tasks.

For example, it's quite common for a user to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of type *TButton*, but by assigning different values to the *Text*, *Default*, and *Cancel* properties and assigning different handlers to the *OnClick* events, the user makes the two instances do very different things.

Deriving new types

The purpose of defining object types is to provide a basis for useful instances. That is, the goal is to create an object that you or other users can use in different applications in different circumstances, or at least in different parts of the same application.

There are two reasons to derive new types:

- Changing type defaults to avoid repetition
- Adding new capabilities to a type

In either case, the goal is to create reusable objects. If you plan ahead and design your objects with future reuse in mind, you can save a lot of later work. Give your object types usable default values, but make them customizable.

The following sections provide a closer look at the two reasons for deriving new types.

Changing type defaults to avoid repetition

In all programming tasks, needless repetition is something to avoid. If you find yourself rewriting the same lines of code over and over, you should either place the code in a subroutine or function, or build a library of routines you'll use in many programs.

The same reasoning holds for components. If you frequently find yourself changing the same properties or making the same method calls, you should probably create a new component type that does those things by default.

For example, it's possible that each time you create an application, you find yourself adding a dialog box form to perform a particular function. Although it's not difficult to recreate the dialog box each time, it's also not necessary. You can design the dialog box once, set its properties, and then install the result onto the Component palette as a reusable component. Not only can this reduce the repetitive nature of the task, it also encourages standardization and reduces the chance of error in recreating the dialog box.

Chapter 9 shows an example of changing the default properties of a component.

Adding new capabilities to a type

The other reason for creating a new kind of component is that you want to add capabilities not already found in the existing components. When you do that, you can either derive from an existing component type (for example, creating a specialized kind of list box) or from an abstract, base type, such as *TComponent* or *TControl*.

As a general rule, derive your new component from the type that contains the closest subset of the features you want. You can add capabilities to an object, but you can't take them away, so if an existing component type contains properties that you *don't* want to include in yours, you should derive from that component's ancestor.

For example, if you want to add some capability to a list box, you would derive your new component from *TListBox*. However, if you want to add some new capability but exclude some existing capabilities of the standard list box, you need to derive your new list box from *TCustomListBox*, the ancestor of *TListBox*. Next, recreate or make visible the list box capabilities you want to include. Finally, add your new features.

Chapter 11 shows an example of customizing an abstract component type.

Declaring a new component type

When you decide that you need to derive a new type of component, you then need to decide what type to derive your new type *from*. As with adding new capabilities to an existing type, the essential rule to follow is this: Derive from the type that contains as much as possible that you want in your component, but which contains nothing that you don't want in your component.

Delphi provides a number of abstract component types specifically designed for component writers to use as bases for deriving new component types. Table 1.1 on page 11 shows the different types you can start from when you create your own components.

- To declare a new component type, add a type declaration to the component's unit.

  Here, for example, is the declaration of a simple graphical component:

```
type
  TSampleShape = class(TGraphicControl)
  end;
```

  A finished component declaration will include property, field, and method declarations before the **end**, but an empty declaration is also valid, and provides a starting point for the addition of component features. Chapter 10 shows an example of creating a complete graphic control.

Ancestors and descendants

From a component user's standpoint, an object is a self-contained entity consisting of properties, methods and events. Component users don't need to know or care about such issues as what object a given component descends from. But these issues are extremely important to you as a component writer.

Component users can take for granted that every component has properties named *Top* and *Left* that determine where the component appears on the form that owns it. To them, it does not matter that all components inherit those properties from a common ancestor, *TComponent*. However, when you create a component, you need to know which object to descend from so as to inherit the appropriate parts. You also need to

know everything your component inherits, so you can take advantage of inherited features without recreating them.

From the definition of object types, you know that when you define an object type, you derive it from an existing object type. The type you derive from is called the *immediate ancestor* of your new object type. The immediate ancestor of the immediate ancestor is called an *ancestor* of the new type, as are all of *its* ancestors. The new type is called a *descendant* of its ancestors.

If you do not specify an ancestor object type, Delphi derives your object from the default ancestor object type, *TObject*. Ultimately, the standard type *TObject* is an ancestor of all objects in Object Pascal.

Object hierarchies

All the ancestor-descendant relationships in an application result in a hierarchy of objects. You can see the object hierarchy in outline form by opening the Object Browser in Delphi.

The most important thing to remember about object hierarchies is that each "generation" of descendant objects contains more than its ancestors. That is, an object inherits everything that its ancestor contains, then adds new data and methods or redefines existing methods.

However, an object cannot remove anything it inherits. For example, if an object has a particular property, all descendants of that object, direct or indirect, will also have that property.

Controlling access

Object Pascal provides four levels of *access control* on the parts of objects. Access control lets you specify what code can access what parts of the object. By specifying levels of access, you define the *interface* to your components. If you plan the interface carefully, you will improve both the usability and reusability of your components.

Unless you specify otherwise, the fields, methods, and properties you add to your objects are *published*, meaning that any code that has access to the object as a whole also has access to those parts of the object, and the compiler generates run-time type information for those items.

The following table shows the levels of access, in order from most restrictive to most accessible:

Table 2.1 Levels of object-part protection

Protection	Used for
private	Hiding implementation details
protected	Defining the developer's interface
public	Defining the run-time interface
published	Defining the design-time interface

All protections operate at the level of units. That is, if a part of an object is accessible (or inaccessible) in one part of a unit, it is also accessible (or inaccessible) everywhere else in the unit. If you want to give special protection to an object or part of an object, you need to put it in its own unit.

Hiding implementation details

Declaring part of an object as **private** makes that part invisible to code outside the unit in which you declare the object type. Within the unit that contains the declaration, code can access the part as if it were **public**.

Private parts of object types are mostly useful for hiding details of implementation from users of the object. Since users of the object can't access the private parts, you can change the internal implementation of the object without affecting user code.

Note This notion of privacy differs from some other object-oriented languages, such as C++, where only "friends" of a class can access the private parts of the class. In that sense, you can consider that all objects and other code in a unit are automatically friends of every object declared in that unit.

Here is an example that shows how declaring a field as **private** prevents users from accessing information. The listing shows two form units, with each form having a handler for its *OnCreate* event. Each of those handlers assigns a value to a private field in one of the forms, but the compiler only allows that assignment in the form that declares the private field.

```
unit HideInfo;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;
type
  TSecretForm = class(TForm)                                    { declare new form }
    procedure FormCreate(Sender: TObject);
  private                                                       { declare private part }
    FSecretCode: Integer;                                     { declare a private field }
  end;
var
  SecretForm: TSecretForm;
implementation
procedure TSecretForm.FormCreate(Sender: TObject);
begin
  FSecretCode := 42;                                          { this compiles correctly }
end;
end.                                                                    { end of unit }

unit TestHide;                                              { this is the main form file }
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  HideInfo;                                             { use the unit with TSecretForm }
type
  TTestForm = class(TForm)
    procedure FormCreate(Sender: TObject);
  end;
```

```
var
  TestForm: TTestForm;
implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
  SecretForm.FSecretCode := 13;        { compiler stops with "Field identifier expected" }
end;
end.                                                                    { end of unit }
```

Although a program using the *HideInfo* unit can use objects of type *TSecretForm*, it cannot access the *FSecretCode* field in any of those objects.

Defining the developer's interface

Declaring part of an object as **protected** makes that part invisible to code outside the unit in which you declare the object type, much like parts declared **private**. The difference with protected parts, however, is that units that contain object types derived from the object type can access the protected parts.

You can use **protected** declarations to define a *developer's interface* to the object. That is, users of the object don't have access to the protected parts, but developers (such as component writers) do. In general, that means you can make interfaces available that allow component writers to change the way an object works without making those details visible to end users.

Defining the run-time interface

Declaring part of an object as **public** makes that part visible to any code that has access to the object as a whole. That is, the public part has no special restrictions on it. If you don't specify any access control (**private**, **protected**, or **public**) on a field, method, or property, that part will be **published**.

Public parts of objects are available at run time to all code, so the public parts of an object define that object's *run-time interface*. The run-time interface is useful for items that aren't meaningful or appropriate at design time, such as properties that depend on actual run-time information or which are read-only. Methods that you intend for users of your components to call should also be declared as part of the run-time interface.

Note that read-only properties cannot operate at design time, so they should appear in the **public** declaration section.

Here is an example that shows two read-only properties declared as part of a component's run-time interface:

```
type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer;                          { implementation details are private }
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer read FTempCelsius;            { properties are public }
```

```
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
⋮
function GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

Since the user cannot change the value of the properties, they cannot appear in the Object Inspector, so they should not be part of the design-time interface.

Defining the design-time interface

Declaring part of an object as **published** makes that part public and also generates run-time type information for the part. Among other things, run-time type information ensures that the Object Inspector can access properties and events.

Because only published parts show up in the Object Inspector, the published parts of an object define that object's *design-time interface*. The design-time interface should include any aspects of the object that a user might want to customize at design time, but must exclude any properties that depend on specific information about the run-time environment.

Note Read-only properties cannot be part of the design-time interface because the user cannot alter them. Read-only properties should be **public**.

Here is an example of a published property. Because it is published, it appears in the Object Inspector at design time.

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer;                          { implementation details are private }
  published
    property Temperature: Integer read FTemperature write FTemperature;      { writable! }
  end;
```

Temperature, the property in this example, is available at design time, so users of the component can adjust the value.

Dispatching methods

Dispatching is the term used to describe how your application determines what code to execute when making a method call. When you write code that calls an object method, it looks like any other procedure or function call. However, objects have three different ways of *dispatching* methods.

The three types of method dispatch are

- Static
- Virtual
- Dynamic

Virtual and dynamic methods work the same way, but the underlying implementation is different. Both of them are quite different from static methods, however. All of the different kinds of dispatching are important to understand when you create components.

Static methods

All object methods are static unless you specify otherwise when you declare them. Static methods work just like regular procedure or function calls. The compiler determines the exact address of the method and links the method at compile time.

The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at run time, which takes somewhat longer.

The other distinction of a static method is that it does not change at all when inherited by another type. That is, if you declare an object type that includes a static method, then derive a new object type from it, the descendant object shares exactly the same method at the same address. Static methods, therefore, always do exactly the same thing, no matter what the actual type of the object.

You cannot override static methods. Declaring a method in a descendant type with the same name as a static method in the object's ancestor *replaces* the ancestor's method.

In the following code, for example, the first component declares two static methods. The second declares two static methods with the same names that replace the methods in the first.

```
type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;      { different from the inherited method, despite same declaration }
    function Flash(HowOften: Integer): Integer;                { this is also different }
  end;
```

Virtual methods

Calling a virtual method is just like calling any other method, but the mechanism for dispatch is a little more complex because it is also more flexible. Virtual methods enable you to redefine methods in descendant objects, but still call the methods the same way. That is, the address of the method isn't determined at compile time. Instead, the object looks up the address of the method at run time.

- To declare a new virtual method, add the directive **virtual** after the method declaration.

The **virtual** directive in a method declaration creates an entry in the object's *virtual method table*, or VMT. The VMT holds the addresses of all the virtual methods in an object type.

When you derive a new object from an existing object type, the new type gets its own VMT, which includes all the entries from its ancestor's VMT, plus any additional virtual methods declared in the new object. In addition, the descendant object can *override* any of its inherited virtual methods.

Overriding methods

Overriding a method means extending or refining it, rather than replacing it. That is, a descendant object type can redeclare and reimplement any of the methods declared in its ancestors. You can't override static methods, because declaring a static method with the same name as an inherited static method replaces the inherited method completely.

- To override a method in a descendant object type, add the directive **override** to the end of the method declaration.

Using **override** will cause a compile-time error if

- The method does not exist in the ancestor object
- The ancestor's method of that name is static
- The declarations are not otherwise identical (names and types of parameters, procedure vs. function, and so on)

The following code shows the declaration of two simple components. The first declares three methods, each with a different kind of dispatching. The other, derived from the first, replaces the static method and overrides the virtual methods.

```
type
  TFirstComponent = class(TCustomControl)
    procedure Move;                                          { static method }
    procedure Flash; virtual;                               { virtual method }
    procedure Beep; dynamic;                        { dynamic virtual method }
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move;                                    { declares new method }
    procedure Flash; override;                   { overrides inherited method }
    procedure Beep; override;                    { overrides inherited method }
  end;
```

Dynamic methods

Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory the object consumes. Dispatching dynamic methods is somewhat slower than dispatching regular virtual methods, however, so if a method call is time-critical or repeated often, you should probably make the method virtual, rather than dynamic.

- To declare a dynamic virtual method, add the directive **dynamic** after the method declaration.

Instead of creating an entry in the object's virtual method table, **dynamic** assigns a number to the method, and stores the address of the associated code. Unlike the virtual method table, which contains the addresses of all of an object's virtual methods, inherited and introduced, the dynamic method list contains entries only for methods introduced or overridden by a particular object type. Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, in reverse order of inheritance.

There is a variation on dynamic methods used for handling messages, including Windows messages, in an application. Chapter 7 discusses these message-handling methods. The dispatch mechanism for message handlers is identical to that for other dynamic methods, but you declare them differently.

Objects and pointers

One thing to be aware of when writing components that you don't need to consider when using existing components is that every Object Pascal object (and therefore every component) is really a pointer. The compiler automatically dereferences the object pointers for you, so in most cases, you never need to think about objects being pointers.

This becomes important, however, when you pass objects as parameters.

In general, you should pass objects by value rather than by reference. That is, when declaring an object as a parameter to a routine, you should not make it a **var** parameter. The reason is that objects are already pointers, which are references. Passing an object as a **var** parameter, then, would be passing a reference to the reference.

Summary

When writing components, you deal with objects in ways that component users never do, but those aspects should never be obvious to the component user.

By choosing appropriate ancestors for your components, carefully designing the interfaces so that they expose all the properties and methods that users need without burdening them with inappropriate items, and following the guidelines for designing methods and passing parameters, you can create useful, reusable components.

The chapters that follow, and the sample components described elsewhere in the book, often refer back to the concepts and practices described in this chapter.

Chapter

3

Creating properties

Properties are the most distinctive parts of components, largely because component users can see and manipulate them at design time and get immediate feedback as the components react in real time. Properties are also important because, if you design them well, they make your components easier for others to use and easier for you to maintain.

In order to best make use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining component properties
- Creating array properties
- Writing property editors

Why create properties?

Properties provide significant advantages, both for you as a component writer and for the users of your components. The most obvious advantage is that properties can appear in the Object Inspector at design time. That simplifies your programming job, because instead of handling several parameters to construct an object, you just read the values assigned by the user.

From the component user's standpoint, properties look like variables. Users can set or read the values of properties much as if those properties were object fields. About the only thing they cannot do with a property that they would with a variable is pass it as a **var** parameter.

From the component writer's standpoint, however, properties provide much more power than simple object fields because

- Users can set properties at design time.

This is very important, because unlike methods, which are only available at run time, properties let users customize components before running an application. In general, your components should not contain a lot of methods; most of them can probably be encapsulated into properties.

- Unlike an object field, a property can hide implementation details from users.

  For example, the data might be stored internally in an encrypted form, but when setting or reading the value of the property, the data would appear unencrypted. Although the value of a property might be a simple number, the component might look up the value from a database or perform complex calculations to arrive at that value.

- Properties allow side effects to outwardly simple assignments.

  What appears to be a simple assignment involving an object field is really a call to a method, and that method could do almost anything.

  A simple but clear example is the *Top* property of all components. Assigning a new value to *Top* doesn't just change some stored value; it causes the component to relocate and repaint itself. The effects of property setting need not be limited to an individual component. Setting the *Down* property of a speed-button component to *True* causes the speed button to set the *Down* properties of all other speed buttons in its group to *False*.

- The implementation methods for a property can be virtual, meaning that what looks like a single property to a component user might do different things in different components.

Types of properties

A property can be of any type that a function can return (since the implementation of the property can use a function). All the standard rules of Pascal type compatibility apply to properties, just as they would to variables. Type compatibility is explained in Chapter 5 of the *Delphi User's Guide*.

The most important aspect of choosing types for your properties is that different types appear differently in the Object Inspector. The Object Inspector uses the type of the property to determine what choices appear to the user. You can specify a different property editor when you register your components, as explained in "Writing property editors" in this chapter.

Table 3.1 How properties appear in the Object Inspector

Property type	Object Inspector treatment
Simple	Numeric, character, and string properties appear in the Object Inspector as numbers, characters, and strings, respectively. The user can type and edit the value of the property directly.
Enumerated	Properties of enumerated types (including *Boolean*) display the value as defined in the source code. The user can cycle through the possible values by double-clicking the value column. There is also a drop-down list that shows all possible values of the enumerated type.

Table 3.1 How properties appear in the Object Inspector (continued)

Property type	Object Inspector treatment
Set	Properties of set types appear in the Object Inspector looking like a set. By expanding the set, the user can treat each element of the set as a Boolean value: *True* if the element is included in the set or *False* if it's not included.
Object	Properties that are themselves objects often have their own property editors. However, if the object that is a property also has published properties, the Object Inspector allows the user to expand the list of object properties and edit them individually. Object properties must descend from *TPersistent*.
Array	Array properties must have their own property editors. The Object Inspector has no built-in support for editing array properties.

Publishing inherited properties

All components inherit properties from their ancestor types. When you derive a new component from an existing component type, your new component inherits all the properties in the ancestor type. If you derive instead from one of the abstract types, many of the inherited properties are either **protected** or **public**, but not **published**.

If you need more information about levels of protection such as **protected**, **private**, and **published**, see "Controlling access" on page 24.

- To make a **protected** or **public** property available to users of your components, you must *redeclare* the property as **published**.

Redeclaring means adding the declaration of an inherited property to the declaration of a descendant object type.

If you derive a component from *TWinControl*, for example, it inherits a *Ctl3D* property, but that property is **protected**, so users of the component cannot access *Ctl3D* at design time or run time. By redeclaring *Ctl3D* in your new component, you can change the level of protection to either **public** or **published**.

The following code shows a redeclaration of *Ctl3D* as **published**, making it available at design time:

```
type
  TSampleComponent = class(TWinControl)
  published
    property Ctl3D;
  end;
```

Note that redeclarations can only make a property less restricted, not more restricted. Thus, you can make a **protected** property **public**, but you cannot "hide" a **public** property by redeclaring it as **protected**.

When you redeclare a property, you specify only the property name, not the type and other information described in "Defining component properties." You can also declare new default values when redeclaring a property, or specify whether to store the property.

Defining component properties

The syntax for property declarations is explained in detail in online Help in the topic for the reserved word **property**. This section focuses on the particulars of declaring properties in Delphi components and the conventions used by the standard components.

Specific topics include

- The property declaration
- Internal data storage
- Direct access
- Access methods
- Default property values

The property declaration

Declaring a property and its implementation is straightforward. You add the property declaration to the declaration of your component object type.

■ To declare a property, you specify three things:

- The name of the property
- The type of the property
- Methods to read and/or set the value of the property

At a minimum, a component's properties should be declared in a **public** part of the component's object-type declaration, making it easy to set and read the properties from outside the component at run time.

To make the property editable at design time, declare the property in a **published** part of the component's object type declaration. Published properties automatically appear in the Object Inspector. Public properties that aren't published are available only at run time.

Here is a typical property declaration:

```
type
  TYourComponent = class(TComponent)
  ⋮
  private
    FCount: Integer;                                    { field for internal storage }
    function GetCount: Integer;                                    { read method }
    procedure SetCount(ACount: Integer);                          { write method }
  public
    property Count: Integer  read GetCount write SetCount;       { property declaration }
  end;
```

The remainder of this section discusses the details for each part of the declaration and the conventions for naming and protecting them.

Internal data storage

There are no restrictions on how you store the data for a property. In general, however, Delphi's components follow these conventions:

- Property data is stored in object fields.
- Identifiers for properties' object fields start with the letter *F*, and incorporate the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in an object field called *FWidth*.
- Object fields for property data should be declared as **private**. This ensures that the component that declares the property has access to them, but component users and descendant components don't.

  Descendant components should use the inherited property itself, not direct access to the internal data storage, to manipulate a property.

The underlying principle behind these conventions is that only the implementation methods for a property should access the data behind that property. If a method or another property needs to change that data, it should do so through the property, *not* by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating descendant components.

Direct access

The simplest way to make property data available is *direct access*. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal storage field without calling an access method. Direct access is useful when the property has no side effects, but you want to make it available in the Object Inspector.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part, usually to update the status of the component based on the new property value.

The following component-type declaration shows a property that uses direct access for both the **read** and **write** parts:

```
type
  TSampleComponent = class(TComponent)
  private                                                { internal storage is private }
    FReadOnly: Boolean;                                { declare field to hold property value }
  published                                            { make property available at design time }
    property ReadOnly: Boolean read FReadOnly write FReadOnly;
  end;
```

Access methods

The syntax for property declarations allows the **read** and **write** parts of a property declaration to specify access methods instead of an object field. Regardless of how a particular property implements its **read** and **write** parts, however, that implementation

should be **private**, and descendant components should use the inherited property for access. This ensures that use of a property will not be affected by changes in the underlying implementation.

Making access methods private also ensures that component users don't accidentally call those methods, inadvertently modifying a property.

The read method

The **read** method for a property is a function that takes no parameters, and returns a value of the same type as the property. By convention, the function's name is "Get" followed by the name of the property. For example, the **read** method for a property named *Count* would be named *GetCount*.

The only exception to the "no parameters" rule is for array properties, which pass their indexes as parameters.

The **read** method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

If you don't declare a **read** method, the property is write-only. Write-only properties are very rare, and generally not very useful.

The write method

The **write** method for a property is always a procedure that takes a single parameter, of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the procedure's name is "Set" followed by the name of the property. For example, the **write** method for a property named *Count* would be named *SetCount*.

The value passed in the parameter is used to set the new value of the property, so the **write** method needs to perform any manipulation needed to put the appropriate values in the internal storage.

If you don't declare a **write** method, the property is read-only.

It's common to test whether the new value actually differs from the current value before setting the value. For example, here's a simple **write** method for an integer property called *Count* that stores its current value in a field called *FCount*:

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

Default property values

When you declare a property, you can optionally declare a *default value* for the property. The default value for a component's property is the value set for that property in the

component's constructor. For example, when you place a component from the Component palette on a form, Delphi creates the component by calling the component's constructor, which determines the initial values of the component's properties.

Delphi uses the declared default value to determine whether to store a property in a form file. For more information on storing properties and the importance of default values, see "Storing and loading properties" on page 80. If you do not specify a default value for a property, Delphi *always* stores the property.

- To declare a default value for a property, append the **default** directive to the property's declaration (or redeclaration), followed by the default value.

Note Declaring a default value in the property declaration does not actually set the property to that value. It is your responsibility as the component writer to ensure that the component's constructor actually sets the property to that value.

Specifying no default value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

- To designate a property as having no default value, append the **nodefault** directive to the property's declaration.

When you declare a property for the first time, there is no need to specify **nodefault**, because the absence of a declared default value means the same thing.

Here is the declaration of a component that includes a single Boolean property named *IsTrue* with a default value of *True*, including the constructor that sets the default value.

```
type
  TSampleComponent = class(TComponent)
  private
    FIsTrue: Boolean;
  public
    constructor Create(AOwner: TComponent); override;
  published
    property IsTrue: Boolean read FIsTrue write FIsTrue default True;
  end;
⋮
constructor TSampleComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor }
  FIsTrue := True;                                          { set the default value }
end;
```

Note that if the default value for *IsTrue* had been *False*, you would not need to set it explicitly in the constructor, since all objects (and therefore, components) always initialize all their fields to zero, and a "zeroed" Boolean value is *False*.

Creating array properties

Some properties lend themselves to being indexed, much like arrays. That is, they have multiple values that correspond to some kind of index value. An example in the

standard components is the *Lines* property of the memo component. *Lines* is an indexed list of the strings that make up the text of the memo, which you can treat as an array of strings. In this case, the array property gives the user natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties work just like other properties, and you declare them in largely the same way. The only differences in declaring array properties are as follows:

- The declaration for the property includes one or more indexes with specified types. Indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, *must* be methods. They cannot be object fields.
- The access methods for reading and writing the property values take additional parameters that correspond to the index or indexes. The parameters must be in the same order and of the same type as the indexes specified in the property declaration.

Although they seem quite similar, there are a few important distinctions between array properties and arrays. Unlike the index of an array, the index type for an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can only reference individual elements of an array property, not the entire range of the property.

The clearest way to see an array property is to look at an example. Here's the declaration of a property that returns a string based on an integer index:

```
type
  TDemoComponent = class(TComponent)
  private
    function GetNumberName(Index: Integer): string;
  public
    property NumberName[Index: Integer]: string read GetNumberName;
  end;
⋮
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
  Result := 'Unknown';
  case Index of
    -MaxInt..-1: Result := 'Negative';
    0: Result := 'Zero';
    1..100: Result := 'Small';
    101..MaxInt: Result := 'Large';
  end;
end;
```

Writing property editors

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing. Depending on the property being edited, you might find it useful to provide either or both kinds.

Writing a property editor requires five steps:

- Deriving a property-editor object
- Editing the property as text
- Editing the property as a whole
- Specifying editor attributes
- Registering the property editor

Deriving a property-editor object

The *DsgnIntf* unit defines several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor object can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor types described in Table 3.2.

- To create a property-editor object, derive a new object type from one of the existing property editor types.

The *DsgnIntf* unit also defines some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones more useful for user-defined properties.

Table 3.2 Predefined property-editor types

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from *TOrdinalProperty*.
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	*Char*-type and subranges of *Char*, such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings, including strings of specified length, such as **string**[20]
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Objects. Displays the name of the object type and allows expansion of the object's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.
TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop-down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop-down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

One of the simplest property editors is *TFloatPropertyEditor*, the editor for properties that are floating-point numbers. Here is its declaration:

```
type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;
```

Editing the property as text

All properties need to provide a string representation of their values for the Object Inspector to display. Most properties also allow the user to type in a new value for the property. Property-editor objects provide virtual methods you can override to convert between the text representation and the actual value.

The methods you override are called *GetValue* and *SetValue*. Your property editor also inherits a set of methods used for assigning and reading different sorts of values, as shown in Table 3.3.

Table 3.3 Methods for reading and writing property values

Property type	"Get" method	"Set" method
Floating point	*GetFloatValue*	*SetFloatValue*
Method pointer (event)	*GetMethodValue*	*SetMethodValue*
Ordinal type	*GetOrdValue*	*SetOrdValue*
String	*GetStrValue*	*SetStrValue*

- When you override a *GetValue* method, you will call one of the "Get" methods, and when you override *SetValue*, you will call one of the "Set" methods.

Displaying the property value

The property editor's *GetValue* method returns a string that represents the current value of the property. The Object Inspector uses this string in the value column for the property. By default, *GetValue* returns `'unknown'`.

- To provide a string representation of your property, override the property editor's *GetValue* method.

  If the property isn't a string value, your *GetValue* must convert the value into a string representation.

Setting the property value

The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should raise an exception and not use the improper value.

- To read string values into properties, override the property editor's *SetValue* method.

Here are the *GetValue* and *SetValue* methods for *TIntegerProperty*. *Integer* is an ordinal type, so *GetValue* calls *GetOrdValue* and converts the result to a string. *SetValue* converts the string to an integer, performs some range checking, and calls *SetOrdValue*.

```
function TIntegerProperty.GetValue: string;
begin
  Result := IntToStr(GetOrdValue);
end;

procedure TIntegerProperty.SetValue(const Value: string);
var
  L: Longint;
begin
  L := StrToInt(Value);                                    { convert string to number }
  with GetTypeData(GetPropType)^ do          { this uses compiler data for type Integer }
    if (L < MinValue) or (L > MaxValue) then               { make sure it's in range... }
      raise EPropertyError.Create(                      { ...otherwise, raise exception }
        FmtLoadStr(SOutOfRange, [MinValue, MaxValue]));
  SetOrdValue(L);                                  { if in range, go ahead and set value }
end;
```

The specifics of the particular examples here are less important than the principle: *GetValue* converts the value to a string; *SetValue* converts the string and validates the value before calling one of the "Set" methods.

Editing the property as a whole

You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves objects. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once.

- To provide a whole-property editor dialog box, override the property-editor object's *Edit* method.

Note that *Edit* methods use the same "Get" and "Set" methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a "Get" method and a "Set" method. Since the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's *Edit* method.

The *Color* properties found in most components use the standard Windows color dialog box as a property editor. Here is the *Edit* method from *TColorProperty*, which invokes the dialog box and uses the result:

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
begin
  ColorDialog := TColorDialog.Create(Application);              { construct the editor }
  try
    ColorDialog.Color := GetOrdValue;                         { use the existing value }
```

```
    if ColorDialog.Execute then                    { if the user OKs the dialog... }
      SetOrdValue(ColorDialog.Color);              { ...use the result to set value }
  finally
    ColorDialog.Free;                                          { destroy the editor }
  end;
end;
```

Specifying editor attributes

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

- To specify editor attributes, override the property editor's *GetAttributes* method.

GetAttributes is a function that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values:

Table 3.4 Property-editor attribute flags

Flag	Meaning if included	Related method
paValueList	The editor can give a list of enumerated values.	*GetValues*
paSubProperties	The property has subproperties that can display.	*GetProperties*
paDialog	The editor can display a dialog box for editing the entire property.	*Edit*
paMultiSelect	The property should display when the user selects more than one component.	N/A

Color properties are more versatile than most, in that they allow several ways for users to choose them in the Object Inspector: typing, selection from a list, and customized editor. *TColorProperty*'s *GetAttributes* method, therefore, includes several attributes in its return value:

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paMultiSelect, paDialog, paValueList];
end;
```

Registering the property editor

Once you create a property editor, you need to register it with Delphi. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

- To register a property editor, call the *RegisterPropertyEditor* procedure.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit.

  This is always a call to the built-in function *TypeInfo*, such as `TypeInfo(TMyComponent)`.

- The type of the component to which this editor applies. If this parameter is **nil**, the editor applies to all properties of the given type.
- The name of the property. This parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

Here is an excerpt from the procedure that registers the editors for the standard components on the Component palette:

```
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(TComponent), nil, '', TComponentProperty);
  RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
    TComponentNameProperty);
  RegisterPropertyEditor(TypeInfo(TMenuItem), TMenu, '', TMenuItemProperty);
end;
```

The three statements in this procedure cover the different uses of *RegisterPropertyEditor*:

- The first statement is the most typical. It registers the property editor *TComponentProperty* for all properties of type *TComponent* (or descendants of *TComponent* that do not have their own editors registered). In general, when you register a property editor, you've created an editor for a particular type, and you want to use it for all properties of that type, so the second and third parameters are **nil** and an empty string, respectively.
- The second statement is the most specific kind of registration. It registers an editor for a specific property in a specific type of component. In this case, the editor is for the *Name* property of all components.
- The third statement is more specific than the first, but not as limited as the second. It registers an editor for all properties of type *TMenuItem* in components of type *TMenu*.

Summary

Properties are the most immediately visible parts of your components, and the ones most useful to developers who use your components. If you follow the guidelines set out in this chapter, your components should fit seamlessly into the Delphi environment.

Chapter

4

Creating events

Events are very important parts of components, although the component writer doesn't usually need to do much with them. An event is a link between an occurrence in the system (such as a user action or a change in focus) that a component might need to respond to and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the component user.

By using events, application developers can customize the behavior of components without having to change the objects themselves. As a component writer, you use events to enable application developers to customize the behavior of your components.

Events for the most common user actions (such as mouse actions) are built into all the standard Delphi components, but you can also define new events and give them their own events. In order to create events in a component, you need to understand the following:

- What are events?
- Implementing the standard events
- Defining your own events

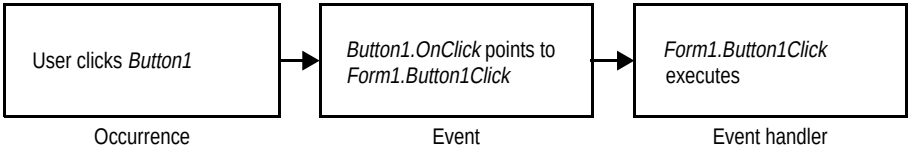
Note Delphi implements events as properties, so you should already be familiar with the material in Chapter 3, "Creating properties" before you attempt to create or change a component's events.

What are events?

Loosely defined, an event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer, a pointer to a specific method in a specific object instance.

From the component user's perspective, an event is just a name related to a system event, such as *OnClick*, that the user can assign a specific method to call. For example, a push button called *Button1* has an *OnClick* method. By default, Delphi generates an event handler called *Button1Click* in the form that contains the button and assigns it to

OnClick. When a click event occurs on the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.



Thus, the component user sees the event as a way of specifying what user-written code the application should call when a specific event occurs.

From the component writer's perspective, there's a lot more to an event. The most important thing to remember, however, is that you're providing a link, a place where the component's user can attach code in response to certain kinds of occurrences. Your components provide outlets where the user can "plug in" specific code.

In order to write an event, you need to understand the following:

- Events are method pointers
- Events are properties
- Event-handler types
- Event handlers are optional

Events are method pointers

Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in a specific object instance. As a component writer, you can treat the method pointer as a placeholder: your code detects that an event occurs, so you call the method (if any) specified by the user for that event.

Method pointers work just like any other procedural type, but they maintain a hidden pointer to an object instance. When the user assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a specific method of a specific object instance. That instance is usually the form that contains the component, but it need not be.

All controls, for example, inherit a dynamic method called *Click* for handling click events. The implementation of *Click* calls the user's click-event handler, if any:

```
procedure TControl.Click;
begin
  if Assigned(OnClick) then OnClick(Self);
end;
```

If the user has assigned a handler to a control's *OnClick* event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

Events are properties

Components use properties to implement their events. Unlike most other properties, events don't use methods to implement their **read** and **write** parts. Instead, event properties use a private object field of the same type as the property.

By convention, the field's name is the same as the name of the property, but preceded by the letter *F*. For example, the *OnClick* method's pointer is stored in a field called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;                { declare a field to hold the method pointer }
    ⋮
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
```

To learn about *TNotifyEvent* and other event types, see "Event-handler types" in the next section.

As with any other property, you can set or change the value of an event at run time. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

Event-handler types

Because an event is a pointer to an event handler, the type of the event property must be a method pointer type. Similarly, any code to be used as an event handler must be an appropriately typed method of an object.

All event-handler methods are procedures. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

Delphi defines method types for all its standard events. When you create your own events, you can use an existing type if that's appropriate, or define one of your own.

Event-handler types are procedures

Although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers *must* be procedures.

Although an event handler cannot be a function, you can still get information back from the user's code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don't require the user's code to change the value.

An example of passing **var** parameters to an event handler is the key-pressed event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two parameters, one to indicate which object generated the event, and one to indicate which key was pressed:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component might want to change the character. One example might be to force all characters to uppercase in an editor. In that case, the user could define the following handler for keystrokes:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

You can also use **var** parameters to let the user override the default handling.

Event handlers are optional

The most important thing to remember when creating events for your components is that users of your components might not attach handlers to the events. That means that your component shouldn't fail or generate errors simply because a user of the component failed to attach a handler to a particular event.

The mechanics of calling handlers and dealing with events that have no attached handler are explained in "Calling the event" on page 53, but the principle has important implications for the design of your components and their events.

The optional nature of event handlers has two aspects:

- Component users do not have to handle events.

  Events happen almost constantly in a Windows application. Just by moving the mouse pointer across a component you cause Windows to send numerous mouse-move messages to the component, which the component translates into *OnMouseMove* events. In most cases, however, users of components don't care to handle the mouse move events, and this does not cause a problem. The component does not depend on the mouse events being handled.

  Similarly, the components you create should not be dependent on users handling the events they generate.

- Component users can write any code they want in an event handler.

  In general, there are no restrictions on the code a user can write in an event handler. The components in the Delphi component library all have events written in such a way that they minimize the chances of user-written code generating unexpected errors. Obviously, you can't protect against logic errors in user code, but you can, for example, ensure that all data structures are initialized before calling events so that users don't try to access invalid information.

Implementing the standard events

All the controls that come with Delphi inherit events for all of the most common Windows events. Collectively, these are called the *standard events*. Although all these events are built into the standard controls, by default they are **protected**, meaning end users can't attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- What are the standard events?
- Making events visible
- Changing the standard event handling

What are the standard events?

There are two categories of standard events: those defined for all controls and those defined only for the standard windows controls.

Standard events for all controls

The most basic events are defined in the object type *TControl*. All controls, whether windowed or graphical or custom, inherit these events. The following table lists all the events available in all controls:

OnClick	*OnDragDrop*	*OnEndDrag*	*OnMouseMove*
OnDblClick	*OnDragOver*	*OnMouseDown*	*OnMouseUp*

All the standard events have corresponding protected dynamic methods declared in *TControl*, with names that correspond to the event names, but without the preceding "On." For example, *OnClick* events call a method named *Click*.

Standard events for standard controls

In addition to the events common to all controls, standard controls (those descended from *TWinControl*) have the following events:

OnEnter	*OnKeyDown*	*OnKeyPress*
OnKeyUp	*OnExit*	

As with the standard events in *TControl*, the windowed-control events have corresponding methods.

Making events visible

The declarations of the standard events are **protected**, as are the methods that correspond to them. If you want to make those events accessible to users either at run time or design time, you need to redeclare the event property as either **public** or **published**.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. Thus, you can take an event that's defined in the standard *TControl* but not made visible to users and promote it to a level that the user can see and use it.

If you create a component, for example, that needs to surface the *OnClick* event at design time, you add the following to the component's type declaration:

```
type
  TMyControl = class(TCustomControl)
    ⋮
  published
    property OnClick;                    { makes OnClick available in Object Inspector }
  end;
```

Changing the standard event handling

If you want to change the way your custom component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As a component user, that's exactly what you would do. However, when you're creating components you can't do that, because you need to keep the event available for the users of the component.

This is precisely the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling, and by calling the inherited method you can maintain the standard handling, including the event for the user's code.

The order in which you call the inherited method is significant. As a general rule, you call the inherited method first, allowing the user's event-handler code to execute before your customizations (and in some cases, to keep from executing the customizations). However, there might be times when you want to execute your code before calling the inherited method. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to the changed status.

Suppose, for example, you're writing a component and you want to modify the way your new component responds to clicks. Instead of assigning a handler to the *OnClick* event as a component user would do, you override the protected method *Click*:

```
procedure TMyControl.Click;
begin
  inherited Click;                { perform standard handling, including calling handler }
  { your customizations go here }
end;
```

Defining your own events

Defining entirely new events is a relatively rare thing. More often you will refine the handling of existing events. However, there are times when a component introduces behavior that is entirely different from any other, so you'll need to define an event for it.

There are three steps involved in defining an event:

- Triggering the event
- Defining the handler type
- Declaring the event
- Calling the event

Triggering the event

The first issue you encounter when defining your own events that you don't need to consider when using the standard events is what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse and Windows sends a *WM_LBUTTONDOWN* message to the application. Upon receiving that message, a component calls its *MouseDown* method, which in turn calls any code the user has attached to the *OnMouseDown* event.

But some events are less clearly tied to specific external events. A scroll bar, for example, has an *OnChange* event, triggered by numerous kinds of occurrences, including keystrokes, mouse clicks, or changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the proper events.

Here are the methods *TControl* uses to handle the *WM_LBUTTONDOWN* message from Windows. *DoMouseDown* is a private implementation method that provides generic handling for left, right, and middle button clicks, translating the parameters of the Windows message into values for the *MouseDown* method.

```
type
  TControl = class(TComponent)
  private
    FOnMouseDown: TMouseEvent;
    procedure DoMouseDown(var Message: TWMMouse; Button: TMouseButton;
      Shift: TShiftState);
    procedure WMLButtonDown(var Message: TWMLButtonDown); message WM_LBUTTONDOWN;
  protected
    procedure MouseDown(Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer); dynamic;
  end;
⋮
procedure TControl.MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Assigned(FOnMouseDown) then
    FOnMouseDown(Self, Button, Shift, X, Y);                    { call handler, if any }
end;

procedure TControl.DoMouseDown(var Message: TWMMouse; Button: TMouseButton;
  Shift: TShiftState);
begin
  with Message do
    MouseDown(Button, KeysToShiftState(Keys) + Shift, XPos, YPos);  { call dynamic method }
end;

procedure TControl.WMLButtonDown(var Message: TWMLButtonDown);
```

```
begin
  inherited;                                              { perform default handling }
  if csCaptureMouse in ControlStyle then MouseCapture := True;
  if csClickEvents in ControlStyle then Include(FControlState, csClicked);
  DoMouseDown(Message, mbLeft, []);                { call the generic mouse-down method }
end;
```

Two kinds of events

There are two kinds of occurrences you might need to provide events for: state changes and user interactions. The mechanics of handling them are the same, but the semantics are slightly different.

User-interaction events will nearly always be triggered by a message from Windows, indicating that the user did something your component might need to respond to. State-change events might also be related to messages from Windows (focus changes or being enabled, for example), but they can also occur through changes in properties or other code. You have total control over the triggering of events you define. You should be consistent and complete so that users of your components know how to use the events.

Defining the handler type

Once you determine that your event occurred, you have to define how you want the event handled. That means determining the type of the event handler. In most cases, handlers for the events you define yourself will be either simple notifications or event-specific types. It's also possible to get information back from the handler.

Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type *TNotifyEvent*, which carries only one parameter, the sender of the event. Thus, all a handler for a notification "knows" about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

Event-specific handlers

In some cases, it's not enough to know just what event happened and what component it happened to. For example, if the event is a key-press event, it's likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters with any necessary information about the event.

If your event was generated in response to a message, it's likely that the parameters you pass to the event handler come directly from the message parameters.

Returning information from the handler

Because all event handlers are procedures, the only way to pass information back from a handler is through a **var** parameter. Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (*OnKeyDown*, *OnKeyUp*, and *OnKeyPress*) pass the value of the key pressed in a **var** parameter named *Key*. The event handler can change *Key* so that the application sees a different key as being involved in the event. This is a way to, for instance, force typed characters to uppercase.

Declaring the event

Once you've determined the type of your event handler, you're ready to declare the method pointer and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

Event names start with "On"

The names of all the standard events in Delphi begin with "On." This is only a convention; the compiler doesn't enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all method-pointer properties are assumed to be events and appear on the Events page.

Component users expect to find events in the alphabetical list of names starting with "On." Using other kinds of names will likely confuse them.

Calling the event

In general, it's best to centralize calls to an event. That is, create a virtual method in your component that calls the user's event handler (if the user assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from your component can customize event handling by overriding that one method, rather than searching through your code looking for places you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid
- Users can override default handling

Empty handlers must be valid

You should never create a situation in which an empty event handler causes an error. That is, the proper functioning of your component should not depend on a particular response from the user's event-handler code. In fact, an empty handler should produce the same result as no handler at all.

Components should never require the user to use them in a particular way. An important aspect of that principle is that component users expect no restrictions on what they can do in an event handler.

Since an empty handler should behave the same as no handler, the code for calling the user's handler should look like this:

```
if Assigned(OnClick) then OnClick(Self);
{ perform default handling }
```

You should *never* have something like this:

```
if Assigned(OnClick) then OnClick(Self)
else { perform default handling };
```

Users can override default handling

For some kinds of events, the user might want to replace the default handling or even suppress all responses. To enable users to do that, you need to pass a **var** parameter to the handler and check for a certain value when the handler returns.

Note that this is in keeping with the notion that empty handlers should have the same effect as no handler at all: since an empty handler won't change the values of any **var** parameters, the default handling always takes place after calling the empty handler.

When handling key-press events, for example, the user can suppress the component's default handling of the keystroke by setting the **var** parameter *Key* to a null character (#0). The logic for supporting that looks like this:

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);
if Key <> #0 then                                    { perform default handling };
```

The actual code is a little different from this because it's dealing with Windows messages, but the logic is the same. By default, the component calls any user-assigned handler, then performs its standard handling. If the user's handler sets *Key* to a null character, the component skips the default handling.

Summary

Events are important parts of your components, as they enable users to customize the behavior of the components without having to derive new components of their own. For most common occurrences in Windows, Delphi's standard components inherit events, so you will at most need to publish an existing event. For components that require special kinds of interactions, however, you can also define and implement your own kinds of events.

Chapter

5

Creating methods

Component methods are no different from any other object's methods. That is, they are just procedures and functions built into the structure of a component object. Although there are essentially no restrictions on what you can do with the methods of a component, Delphi does use some standards you should follow.

The guidelines to follow when writing methods for your components include

- Avoiding interdependencies
- Naming methods
- Protecting methods
- Making methods virtual
- Declaring methods

As a general rule, minimize the number of methods users need to call to use your components. A lot of the features you might be inclined to implement as methods are probably better encapsulated into properties. Doing so provides an interface that suits the Delphi environment, and also lets users access them at design time.

Avoiding interdependencies

At all times when writing components, minimize the preconditions imposed on the component user. To the greatest extent possible, component users should able to do anything they want to a component, whenever they want to do it. There will be times when you can't accommodate that, but your goal should be to come as close as possible.

Although it's impossible to list all the possible kinds of interdependencies you want to avoid, this list gives you an idea of the kinds of things to avoid:

- Methods that the user *must* call in order to use the component
- Methods that need to execute in a particular order
- Methods that put the component into a state or mode where certain events or methods could be invalid

The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that other method in such a way that if the user calls it when the component is in a bad state, the method corrects that state before executing its main code. At a minimum, you should raise an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on *you* to be sure that using the code in incorrect ways does not cause the user problems. A warning message, for example, is preferable to crashing if the user doesn't accommodate your interdependencies.

Naming methods

Delphi imposes no restrictions on what you name methods or their parameters. However, there are a few conventions that make methods easier for users of your components. Keep in mind that the nature of a component architecture dictates that many different kinds of people might use your components.

If you're accustomed to writing code that only you or a small group of programmers uses, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive.

  A name like *PasteFromClipboard* is much more informative than simply *Paste* or *PFC*.

- Procedure names should be active.

  Use active verbs in your procedure names. For example, *ReadFileNames* is much more helpful than *DoFiles*.

- Function names should reflect the nature of what they return.

  Although it might be obvious to you as a programmer that a function named *X* returns the horizontal position of something, a name like *GetHorizontalPosition* is more universally understandable.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

Protecting methods

All parts of objects, including fields, methods, and properties, can have various levels of protection, as explained in "Controlling access" on page 24. Choosing the appropriate level of protection for methods is quite straightforward.

As a general rule, methods you write in your components will be either **public** or **protected**. The exception to that rule is methods that implement properties, which should always be **private**. Otherwise, you rarely need to make a method **private**, unless it is truly specific to that particular type of component, to the point that even components derived from it should not have access to it.

Note There is generally no reason for declaring a method (other than an event handler) as **published**. Doing so looks to the end user exactly as if the method were **public**.

Methods that should be public

Any methods that users of your components need to be able to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid unduly tying up system resources or putting Windows in a state where it can't respond to the user.

Note Constructors and destructors should *always* be public.

Methods that should be protected

Any methods that are implementation methods for the component should be **protected**. That keeps users from calling them at the wrong time. If you have methods that a user's code should not call, but which descendant objects will need to call, you should declare the methods as **protected**.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method public, there's a chance a user will call it before setting up the data. On the other hand, by making it protected, you ensure that the user can't call it directly. You can then set up other, public methods that ensure that data setup occurs before calling the protected method.

Methods that should be private

The one category of methods that should always be private is property-implementation methods. You definitely don't want users calling methods that manipulate the data for a property. They should only access that information by accessing the property itself. You can ensure that by making the property public and its implementation methods private.

If descendant objects need to override the implementation of a method, they can do so, but instead of overriding the implementation methods, they must access the inherited property value through the property itself.

Making methods virtual

Virtual methods in Delphi components are no different from virtual methods in other objects. You make methods virtual when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by end users, you can probably make all your methods static. On the other hand, if you create components of a more abstract nature, which other component writers will use as the starting point for their own components, consider making the added methods virtual. That way, components derived from your components can override the inherited virtual methods.

Declaring methods

Declaring a method in a component is no different from declaring any object method.

- To declare a new method in a component, you do two things:
  - Add the declaration to the component's object-type declaration
  - Implement the method in the **implementation** part of the component's unit

The following code shows a component that defines two new methods: one protected static method and one public virtual method.

```
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger;                              { declare protected static method }
  public
    function CalculateArea: Integer; virtual;          { declare public virtual method }
  end;
⋮
implementation
⋮
procedure TSampleComponent.MakeBigger;                          { implement first method }
begin
  Height := Height + 5;
  Width := Width + 5;
end;

function TSampleComponent.CalculateArea: Integer;              { implement second method }
begin
  Result := Width * Height;
end;
```

Summary

Methods are important parts of components, although not as visible as properties and events. There are few restrictions on what you can do with methods, but you should be careful about how you protect methods you write. Implementation methods that users should not call should always be protected.

Chapter

6

Using graphics in components

Windows provides a powerful Graphics Device Interface (GDI) for drawing device-independent graphics. Unfortunately, GDI imposes a lot of extra requirements on the programmer, such as managing graphic resources. As a result, you spend a lot of time doing things other than what you really want to do: creating graphics.

Delphi takes care of all the GDI drudgery for you, allowing you to spend your time doing productive work instead of searching for lost handles or unreleased resources. Delphi tackles the tedious tasks so you can focus on the productive and creative ones.

Note that, as with any part of the Windows API, you can call GDI functions directly from your Delphi application if you want to. However, you will probably find that using Delphi's encapsulation of the graphic functions is a much more productive way to create graphics.

There are several important topics dealing with graphics in Delphi:

- Overview of Delphi graphics
- Using the canvas
- Working with pictures
- Offscreen bitmaps
- Responding to changes

Overview of Delphi graphics

Delphi encapsulates the Windows GDI at several levels. The most important to you as a component writer is the way components display their images on the screen. When calling GDI functions directly, you need to have a handle to a device context, into which you have selected various drawing tools such as pens and brushes and fonts. After rendering your graphic images, you must then restore the device context to its original state before disposing of it.

Instead of forcing you to deal with graphics at that detailed level, Delphi provides a simple yet complete interface: your component's *Canvas* property. The canvas takes care of making sure it has a valid device context, and releases the context when you're not using it. Similarly, the canvas has its own properties representing the current pen, brush, and font.

The canvas manages all those resources for you, so you need not concern yourself with creating, selecting, and releasing things such as pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting Delphi manage graphic resources is that it can cache resources for later use, which can greatly speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because Delphi caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, Delphi reuses an existing one.

As an example of how much simpler Delphi's graphics code can be, here are two samples of code. The first uses standard GDI functions to draw a yellow ellipse outlined in blue on a window in an application written with ObjectWindows. The second uses a canvas to draw the same ellipse in an application written with Delphi.

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
  PenHandle, OldPenHandle: HPEN;
  BrushHandle, OldBrushHandle: HBRUSH;
begin
  PenHandle := CreatePen(PS_SOLID, 1, RGB(0, 0, 255));                { create blue pen }
  OldPenHandle := SelectObject(PaintDC, PenHandle);            { tell DC to use blue pen }
  BrushHandle := CreateSolidBrush(RGB(255, 255, 0));             { create a yellow brush }
  OldBrushHandle := SelectObject(PaintDC, BrushHandle);     { tell DC to use yellow brush }
  Ellipse(HDC, 10, 10, 50, 50);                                      { draw the ellipse }
  SelectObject(OldBrushHandle);                                 { restore original brush }
  DeleteObject(BrushHandle);                                       { delete yellow brush }
  SelectObject(OldPenHandle);                                    { restore original pen }
  DeleteObject(PenHandle);                                          { destroy blue pen }
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  with Canvas do
  begin
    Pen.Color := clBlue;                                          { make the pen blue }
    Brush.Color := clYellow;                                   { make the brush yellow }
    Ellipse(10, 10, 50, 50);                                        { draw the ellipse }
  end;
end;
```

Using the canvas

The canvas object encapsulates Windows graphics at several levels, ranging from high-level functions for drawing individual lines, shapes, and text to intermediate-level

properties for manipulating the drawing capabilities of the canvas to low-level access to the Windows GDI.

Table 6.1 summarizes the capabilities of the canvas.

Table 6.1 Canvas capability summary

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as *MoveTo*, *LineTo*, *Rectangle*, and *Ellipse*
	Displaying and measuring text	*TextOut*, *TextHeight*, *TextWidth*, and *TextRect* methods
	Filling areas	*FillRect* and *FloodFill* methods
Intermediate	Customizing text and graphics	*Pen*, *Brush*, and *Font* properties
	Manipulating pixels	*Pixels* property
	Copying and merging images	*Draw*, *StretchDraw*, *BrushCopy*, and *CopyRect* methods; *CopyMode* property
Low	Calling Windows GDI functions	*Handle* property

For detailed information on canvas objects and their methods and properties, see online Help.

Working with pictures

Most of the graphics work you do in Delphi is limited to drawing directly on the canvases of components and forms. Delphi also provides for handling standalone graphic images, however, such as bitmaps, metafiles, and icons, including automatic management of palettes.

There are three important aspects to working with pictures in Delphi:

- Pictures, graphics, and canvases
- Graphics in files
- Handling palettes

Pictures, graphics, and canvases

There are three kinds of objects in Delphi that deal with graphics:

- A *canvas*, which represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a standalone object.
- A *graphic*, which represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. Delphi defines object types *TBitmap*, *TIcon*, and *TMetafile*, all descended from a generic *TGraphic*. You can also define your own graphic objects. By defining a minimal standard interface for all graphics, *TGraphic* provides a simple mechanism for applications to use different kinds of graphics easily.
- A *picture*, which is a container for a graphic, meaning it could contain any of the graphic object types. That is, an item of type *TPicture* can contain a bitmap, an icon, a

metafile, or a user-defined graphic type, and an application can access them all in the same way through the picture object. For example, the image control has a property called *Picture*, of type *TPicture*, enabling the control to display images from many kinds of graphics.

Keep in mind that a picture object always has a graphic, and a graphic might have a canvas (the only standard graphic that has a canvas is *TBitmap*). Normally, when dealing with a picture, you work only with the parts of the graphic object exposed through *TPicture*. If you need access to the specifics of the graphic object itself, you can refer to the picture's *Graphic* property.

Graphics in files

All pictures and graphics in Delphi can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

- To load an image into a picture from a file, call the picture's *LoadFromFile* method.
- To save an image from a picture into a file, call the picture's *SaveToFile* method.

LoadFromFile and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves whatever type of file is appropriate for the type of graphic object being saved.

To load a bitmap into an image control's picture, for example, pass the name of a bitmap file to the picture's *LoadFromFile* method:

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
  Image1.Picture.LoadFromFile('RANDOM.BMP');
end;
```

The picture recognized .BMP as the standard extension for bitmap files, so it creates its graphic as a *TBitmap*, then calls that graphic's *LoadFromFile* method. Since the graphic is a bitmap, it loads the image from the file as a bitmap.

Handling palettes

When running on a palette-based device, Delphi controls automatically support palette realization. That is, if you have a control that has a palette, you can use two methods inherited from *TControl* to control how Windows accommodates that palette.

Palette support for controls has these two aspects:

- Specifying a palette for a control
- Responding to palette changes

Most controls have no need for a palette. However, controls that contain graphic images (such as the image control) might need to interact with Windows and the screen device driver to ensure the proper appearance of the control. Windows refers to this process as *realizing* palettes.

Realizing palettes is the process of ensuring that the frontmost window uses its full palette, and that windows in the background use as much of their palettes as possible, then map any other colors to the closest available colors in the "real" palette. As windows move in front of one another, Windows continually realizes the palettes.

Note Delphi itself provides no specific support for creating or maintaining palettes, other than in bitmaps. However, if you have a palette handle, Delphi controls can manage it for you.

Specifying a palette for a control

If you have a palette you want to apply to a control, you can tell your application to use that palette.

- To specify a palette for a control, override the control's *GetPalette* method to return the handle of the palette.

Specifying the palette for a control does two things for your application:

1. It tells the application that your control's palette needs to be realized.
2. It designates the palette to use for realization.

The clearest example of a control using a palette is the image control, *TImage*. The image control gets its palette (if any) from the picture it contains. *TImage* overrides *GetPalette* to return the palette of its picture:

```
type
  TImage = class(TGraphicControl)
  protected
    function GetPalette: HPALETTE; override;                    { override the method }
  ⋮
  end;
⋮
function TImage.GetPalette: HPALETTE;
begin
  Result := 0;                                          { default result is no palette }
  if FPicture.Graphic is TBitmap then                     { only bitmaps have palettes }
    Result := TBitmap(FPicture.Graphic).Palette;                { use it if available }
end;
```

Responding to palette changes

If your control specifies a palette by overriding *GetPalette*, Delphi automatically takes care of responding to palette messages from Windows. The method that handles the palette messages is *PaletteChanged*. For normal operation, you should never need to alter the default behavior of *PaletteChanged*.

The primary role of *PaletteChanged* is to determine whether to realize the control's palette in the foreground or the background. Windows handles this realization of palettes by making the topmost window have a foreground palette, with other windows resolved in background palettes. Delphi goes one step farther, in that it also realizes palettes for controls within a window in tab order. The only time you might need to override this default behavior is if you want a control that is not first in tab order to have the foreground palette.

Offscreen bitmaps

When drawing complex graphic images, a common technique in Windows programming is to create an offscreen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an offscreen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap object in Delphi to represent bitmapped images in resources and files can also work as an offscreen image.

There are two main aspects to working with offscreen bitmaps:

- Creating and managing offscreen bitmaps
- Copying bitmapped images

Creating and managing offscreen bitmaps

When creating complex graphic images, you should generally avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas.

The most common use of an offscreen bitmap is in the *Paint* method of a graphic control. As with any temporary object, the bitmap should be protected with a **try**..**finally** block:

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;                              { override the Paint method }
  end;

procedure TFancyControl.Paint;
var
  Bitmap: TBitmap;                          { temporary variable for the offscreen bitmap }
begin
  Bitmap := TBitmap.Create;                                 { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                                            { destroy the bitmap object }
  end;
end;
```

For an example of painting a complex image on an offscreen bitmap, see the source code for the Gauge control from the Samples page of the Component palette. The gauge draws its different shapes and text on an offscreen bitmap before copying them to the screen. Source code for the gauge is in the the file GAUGES.PAS in the SOURCE\SAMPLES subdirectory.

Copying bitmapped images

Delphi provides four different ways to copy images from one canvas to another. Depending on the effect you want to create, you call different methods.

Table 6.2 summarizes the image-copying methods in canvas objects.

Table 6.2 Image-copying methods

To create this effect	Call this method
Copy an entire graphic	*Draw*
Copy and resize a graphic	*StretchDraw*
Copy part of a canvas	*CopyRect*
Copy a bitmap with raster operations	*BrushCopy*

You can see examples of using all these methods in online Help.

Responding to changes

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish those objects as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

- To respond to changes in a graphic object, assign a method to the object's *OnChange* event.

The shape component publishes properties representing the pen and brush it uses to draw its shape. The component's constructor assigns a method to the *OnChange* event of each, causing the component to refresh its image if either the pen or brush changes:

```
type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                          { construct the pen }
  FPen.OnChange := StyleChanged;                      { assign method to OnChange event }
  FBrush := TBrush.Create;                                    { construct the brush }
```

```
  FBrush.OnChange := StyleChanged;                    { assign method to OnChange event }
end;

procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate(True);                                   { erase and repaint the component }
end;
```

Chapter 10 shows the complete process of creating the shape component, including responding to changes to its pen and brush.

Summary

Delphi's graphics encapsulation makes it easy and fast to handle graphics in components or applications. By handling all the record-keeping tasks, Delphi reduces the number of lines of code you need to write and prevents many common errors. By caching graphic resources, Delphi can make your graphics routines run faster.

Whether you draw lines or shapes or manipulate complex bitmapped images, the Delphi graphics encapsulation makes your job easier and less error-prone.

Chapter

7

Handling messages

One of the keys to traditional Windows programming is handling the *messages* sent by Windows to applications. Delphi handles most of the common ones for you, but in the course of creating components it is possible either that you will need to handle messages that Delphi doesn't already handle or that you will create your own messages and need to handle them.

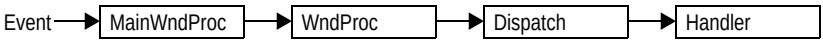
There are three aspects to working with messages:

- Understanding the message-handling system
- Changing message handling
- Creating new message handlers

Understanding the message-handling system

All Delphi objects have a built-in mechanism for handling messages, called *message-handling methods* or *message handlers*. The basic idea of message handlers is that the object receives messages of some sort and dispatches them, calling one of a set of specified methods depending on the message received. If no specific method exists for a particular message, there is a default handler.

The following diagram shows the message-dispatch system:



The Delphi component library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular object into method calls. You should never need to alter this message-dispatch mechanism. All you'll need to do is create message-handling methods.

What's in a Windows message?

A Windows message is a data record that contains several useful fields. The most important of these is an integer-size value that identifies the message. Windows defines a lot of messages, and the *Messages* unit declares identifiers for all of them. The other useful information in a message comes in two parameter fields and a result field.

One parameter contains 16 bits, the other 32 bits. You often see Windows code that refers to those values as *wParam* and *lParam*, for "word parameter" and "long parameter." Often, each parameter will contain more than one piece of information, and you see references to names such as *lParamHi*, which refers to the high-order word in the long parameter.

Originally, Windows programmers had to remember (or look up) what each parameter contained. More recently, Microsoft has named the parameters. This so-called "message cracking" makes it much simpler to understand what information accompanies each message. For example, the parameters to the *WM_KEYDOWN* message are now called *VKey* and *KeyData*, which gives much more specific information than *wParam* and *lParam*.

Delphi defines a specific record type for each different type of message that gives a mnemonic name to each parameter. For example, mouse messages pass the x- and y-coordinates of the mouse event in the long parameter, one in the high-order word, and the other in the low-order word. Using the mouse-message record, however, you need not concern yourself with which word is which, because you refer to the parameters by the names *XPos* and *YPos* instead of *lParamLo* and *lParamHi*.

Dispatching methods

When an application creates a window, it registers a *window procedure* with the Windows kernel. The window procedure is the routine that handles messages for the window. Traditionally, the window procedure contains a huge **case** statement with entries for each message the window has to handle. Keep in mind that "window" in this sense means just about anything on the screen: each window, each control, and so on. Every time you create a new type of window, you have to create a complete window procedure.

Delphi simplifies message dispatching in several ways:

- Every component inherits a complete message-dispatching system.
- The dispatch system has default handling. You only define handlers for messages you need to respond to specially.
- You can modify small parts of the message-handling and rely on inherited methods for most processing.

The greatest benefit of this message dispatch system is that you can safely send any message to any component at any time. If the component doesn't have a handler defined for the message, the default handling takes care of it, usually by ignoring the message.

Tracing the flow of messages

Delphi registers a method called *MainWndProc* as the window procedure for each type of component in an application. *MainWndProc* contains an exception-handling block, passing the message record from Windows to a virtual method called *WndProc* and handling any exceptions by calling the application object's *HandleException* method.

MainWndProc is a static method that contains no special handling for any particular messages. Customizations take place in *WndProc*, since each component type can override the method to suit its particular needs.

WndProc methods check for any special conditions that affect their processing so they can "trap" unwanted messages. For example, while being dragged, components ignore keyboard events, so *TWinControl.WndProc* only passes along keyboard events if the component is not being dragged. Ultimately, *WndProc* calls *Dispatch*, a static method inherited from *TObject*, which determines what method to call to handle the message.

Dispatch uses the *Msg* field of the message record to determine how to dispatch a particular message. If the component defines a handler for that particular message, *Dispatch* calls the method. If the component doesn't define a handler for that message, *Dispatch* calls *DefaultHandler*.

Changing message handling

Before changing the message-handling of your components, make sure that's what you really want to do. Delphi translates most Windows messages into events that both the component writer and the component user can handle. Rather than changing the message-handling behavior, you should probably change the event-handling behavior.

To change the message handling, you override the message-handling method. You can also prevent a component from handling a message under certain circumstances by trapping the message.

Overriding the handler method

To change the way a component handles a particular message, you override the message-handling method for that message. If the component doesn't already handle the particular message, you need to declare a new message-handling method.

- To override a message-handling method, you declare a new method in your component with the same message index as the method it overrides. Do *not* use the **override** directive; you must use the **message** directive and a matching message index.

  Note that the name of the method and the type of the single **var** parameter do not have to match the overridden method. Only the message index is significant. However, for clarity, it is best to follow the convention of naming message-handling methods after the messages they handle.

  For example, to override a component's handling of the *WM_PAINT* message, you redeclare the *WMPaint* method:

```
type
  TMyComponent = class(...)
    ⋮
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
  end;
```

Using message parameters

Once inside a message-handling method, your component has access to all the parameters of the message record. Since the message is always a **var** parameter, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the *Result* field. *Result* is the value the Windows documentation refers to as the "return value" for the message: the value returned by the *SendMessage* call that sends the message.

Because the type of the *Message* parameter to the message-handling method varies with the message being handled, you should refer to the documentation on Windows messages for the names and meanings of individual parameters.

If for some reason you need to refer to the message parameters by their old-style names (*wParam*, *lParam*, and so on), you can typecast *Message* to the generic type *TMessage*, which uses those parameter names.

Trapping messages

Under certain circumstances, you might want your components to ignore certain messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message that way, you override the virtual method *WndProc*.

The *WndProc* method screens messages before passing them to the *Dispatch* method, which in turn determines which method gets to handle the message. By overriding *WndProc*, your component gets a chance to filter out messages before dispatching them.

In general, an override of *WndProc* looks like this:

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
  { tests to determine whether to continue processing }
  inherited WndProc(Message);                                    { dispatch normally }
end;
```

Here is part of the *WndProc* method for *TControl*, for example. *TControl* defines entire ranges of mouse messages that it filters when a user is dragging and dropping controls. Overriding *WndProc* helps this in two ways:

- It can filter ranges of messages instead of having to specify handlers for each one.
- It can preclude dispatching the message at all, so the handlers are never called.

```
procedure TControl.WndProc(var Message: TMessage);
begin
  ⋮
```

```
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                                    { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
      ⋮                                                   { handle others normally }
    end;
  ⋮                                                    {otherwise process normally }
end;
```

Creating new message handlers

Since Delphi provides handlers for most common Windows messages, the time you will most likely need to create new message handlers is when you define your own messages.

Working with user-defined messages has two aspects:

- Defining your own messages
- Declaring a new message-handling method

Defining your own messages

A number of the standard components define messages for internal use. The most common reasons for defining messages are broadcasting information not covered by standard Windows messages and notification of state changes.

■ Defining a message is a two-step process. The steps are

- Declaring a message identifier
- Declaring a message-record type

Declaring a message identifier

A message identifier is an integer-sized constant. Windows reserves the messages below 1,024 for its own use, so when you declare your own messages you should start above that level.

The constant *WM_USER* represents the starting number for user-defined messages. When defining message identifiers, you should base them on *WM_USER*.

Be aware that some standard Windows controls use messages in the user-defined range. These include list boxes, combo boxes, edit boxes, and command buttons. If you derive a component from one of these and want to define a new message for it, be sure to check the *Messages* unit to see what messages Windows already defines for that control.

The following code shows two user-defined messages:

```
const
  WM_MYFIRSTMESSAGE = WM_USER + 0;
  WM_MYSECONDMESSAGE = WM_USER + 1;
```

Declaring a message-record type

If you want to give useful names to the parameters of your message, you need to declare a message-record type for that message. The message record is the type of the parameter passed to the message-handling method. If you don't use the message's parameters, or if you want to use the old-style parameter notation (*wParam*, *lParam*, and so on), you can use the default message record, *TMessage*.

■ To declare a message-record type, follow these conventions:

1. Name the record type after the message, preceded by a *T*.
2. Call the first field in the record *Msg*, of type *TMsgParam*.
3. Define the next two bytes to correspond to the word-size parameter.
4. Define the next four bytes to correspond to the long parameter.
5. Add a final field called *Result*, of type *LongInt*.

For example, here is the message record for all mouse messages, *TWMMouse*:

```
type
  TWMMouse = record
    Msg: TMsgParam;                                  { first is the message ID }
    Keys: Word;                                          { this is the wParam }
    case Integer of                                { two ways to look at the lParam }
      0: (
        XPos: Integer;                           { either as x- and y-coordinates... }
        YPos: Integer);
      1: (
        Pos: TPoint;                                     { ...or as a single point }
        Result: Longint);                          { and finally, the result field }
  end;
```

Note that *TWMMouse* uses a variant record to define two different sets of names for the same parameters.

Declaring a new message-handling method

There are two sets of circumstances that require you to declare new message-handling methods:

- Your component needs to handle a Windows message that isn't already handled by the standard components.
- You have defined your own message for use by your components.

■ To declare a message-handling method, do the following:

1. Declare the method in a **protected** part of the component's class declaration.
2. Make the method a procedure.
3. Name the method after the message it handles, but without any underline characters.
4. Pass a single **var** parameter called *Message*, of the type of the message record.
5. Write code for any handling specific to the component.
6. Call the inherited message handler.

Here's the declaration, for example, of a message handler for a user-defined message called *CM_CHANGECOLOR*:

```
type
  TMyComponent = class(TControl)
    ⋮
    procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
  end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;                          { set color from long parameter }
  inherited;                                              { call inherited handler }
end;
```

Summary

Responding to Windows messages is an important way for some components to interact with Windows or with each other. Delphi's standard components already respond to the most common messages, but your components can override that handling or respond to other messages, including user-defined messages.

The most important things to remember when handling Windows messages are to use the message record types defined in the *Messages* unit and to call inherited message handlers in your message-handling methods.

Chapter

8

Registering components

Writing a component and its properties, methods, and events is only part of the process of component creation. Although a component with only those features can be useful, the real power of components comes from the ability to manipulate them at design time.

In addition to the tasks that make the components do their own work, making your components available at design time requires several steps:

- Registering components with Delphi
- Adding palette bitmaps
- Providing Help on properties and events
- Storing and loading properties

Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only step that is always necessary is registration.

Registering components with Delphi

In order to have Delphi recognize your components and place them on the Component palette, you must *register* each component. Registration works on a unit basis, so if you create several components in a single unit, you register them all at once.

■ To register a component, add a *Register* procedure to the component unit, which has two aspects:

- Declaring the Register procedure
- Implementing the Register procedure

Once you've set up the registration, you can install the components into the palette as described in the *User's Guide*.

Declaring the Register procedure

Registration involves writing a single procedure in the component unit, which must have the name *Register*. *Register* must appear in the **interface** part of the unit so Delphi can locate it. Within the *Register* procedure, you call the procedure *RegisterComponents* for each component you want to register.

The following code, for example, shows the outline of a simple unit that creates and registers new component types:

```
unit MyBtns;
interface
type
  { declare your component types here }

procedure Register;                          { this must appear in the interface section }

implementation
  { component implementation goes here }
procedure Register;
begin
  { register the components }
end;
end.
```

Implementing the Register procedure

Inside the *Register* procedure for a unit containing components, you must register each component you want to add to the Component palette. If the unit contains several components, you can register them all in one step.

- To register a component, call the *RegisterComponents* procedure once for each page of the Component palette you want to add components to. *RegisterComponents* tells Delphi two important things about the components it registers:

  - The name of the palette page you want to install on
  - The names of the components to install

  The palette-page name is a string. If the name you give for the palette page doesn't already exist, Delphi creates a new page with that name.

  You pass the component names in an open array, which you can construct inside the call to *RegisterComponents*.

  The following *Register* procedure, for example, registers a component named *TMyComponent* and places it on a Component palette page called "Miscellaneous":

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent]);
end;
```

  You could also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);          { two on this page }
  RegisterComponents('Assorted', [TThird]);                      { and one on another }
end;
```

Adding palette bitmaps

Every component needs a bitmap to represent the component on the Component palette. If you don't specify your own bitmap, Delphi uses a default bitmap.

Since the palette bitmaps are only needed at design time, you don't compile them into the component unit. Instead, you supply them in a Windows resource file with the same name as the unit, but with the extension .DCR (for "dynamic component resource"). You can create this resource file using the bitmap editor in Delphi. Each bitmap should be 24 pixels square.

For each unit you want to install, supply a palette bitmap file, and within each palette bitmap file, supply a bitmap for each component you register. The bitmap image has the same name as the component. Keep the palette bitmap file in the same directory with the compiled unit, so Delphi can find the bitmaps when it installs the components on the palette.

For example, if you create a component named *TMyControl* in a unit named *ToolBox*, you need to create a resource file called TOOLBOX.DCR that contains a bitmap called TMYCONTROL. The resource names are not case-sensitive, but by convention, they are usually in uppercase letters.

Providing Help on properties and events

When you select a component on a form, or a property or event in the Object Inspector, you can press *F1* to get Help on that item. Users of your components can get the same kind of documentation for your components if you create the appropriate Help files.

Because Delphi uses a special Help engine that handles searches across multiple Help files, you can provide a small Help file with just the information on your components, and users will be able to find your documentation without having to take any special steps. Your Help becomes part of the user's overall Delphi Help system.

■ To provide Help to users of your components, you need to understand two things:

- How Delphi handles Help requests
- Merging your Help into Delphi

How Delphi handles Help requests

Delphi looks up Help requests based on keywords. That is, when a user presses *F1* with a component selected in the Form Designer, Delphi turns the name of the component into a keyword (in this case, "class_" plus the name of the component type), then calls the Windows Help engine (WinHelp) to search for a Help topic that has that keyword.

Keywords are a standard part of the Windows Help system. In fact, WinHelp uses the keywords in a Help file to generate the list in the Search dialog box. Since the keywords used in context-sensitive searches, such as those for a component, are not designed for human readability, you'll enter them as alternate keywords. Alternate keywords do not appear in the Search dialog box.

For example, a user searching for details on a component called *TSomething* might open the WinHelp Search dialog box and type `TSomething`, but would never use the alternate form, class_TSomething, used by the Form Designer's context search. The special keyword class_TSomething is therefore invisible to the user, to avoid cluttering the Search list.

Merging your Help into Delphi

Delphi includes all the tools you need to create and merge Windows Help files, including the Windows Help Compiler, HC.EXE. The mechanics of creating Help files for your Delphi components are no different than those for creating any Help file, but there are some conventions you need to follow to be compatible with the Help for the rest of the library.

- To learn how to create Help files in general, see "Creating Windows Help" in the Help file CWH.HLP.
- To make your Help files compatible with other Delphi component Help, you need to do all four of the following tasks:

  1. Creating the Help file
  2. Adding special footnotes
  3. Creating the keyword file
  4. Merging the Help indexes

  When you finish creating the Help for your components, you have several files:

  - A compiled Help (.HLP) file
  - A Help keyword (.KWF) file
  - One or more Help source (.RTF) files
  - A Help project (.HPJ) file

  The compiled Help and keyword files should go in the directory with the unit that contains your components. If you distribute your components to other users, you should distribute those files along with the compiled unit (.DCU) file. The Help source and Help project files are essentially source code for the Help and keyword files, so be sure to store and archive them accordingly.

Creating the Help file

You can use any tools you want to create a Windows Help file. Delphi's multiple-file search engine can include material from any number of Help files. In addition to the compiled Help file, you need to have the Rich Text Format (RTF) source file available so you can generate the keyword file, although you probably won't distribute the RTF file.

- To make your component's Help work with the Help for the rest of the components in the library, observe the following conventions:

1 Each component has a screen.

   The component screen should give a brief description of the component's purpose, then list separately all the properties, events, and methods available to end users. Application developers access this screen by selecting the component on a form and pressing *F1*. For an example of a component screen, place any component on a form and press *F1*.

   The component screen should have a "K" footnote for keyword searching that includes the name of the component. For example, the keyword footnote for the *TMemo* component reads "TMemo component."

2 Every property, event, and method that the component adds or changes significantly has a screen.

   A property, event, or method screen should indicate what component the item applies to, show the declared syntax of the item, and describe its use. Application developers see these screens either by highlighting the item in the Object Inspector and pressing *F1* or by placing the test cursor in the Code Editor on the name of the item and pressing *F1*. To see an example of a property screen, select any item in the Object Inspector and press *F1*.

   The property, event, or method screen should have a "K" footnote for keyword searching includes the name of the item and what kind of item it is. For example, the keyword footnote for the *Top* property reads "Top property."

Each screen in the Help file will also need special footnotes that Delphi uses for its multiple-file index searches. The next section describes these special footnotes.

Adding special footnotes

Delphi needs special search keys to be able to distinguish between the Help screens for components and other similarly-named items. You should provide the standard keyword-search items for each item ("K" footnotes), but you also need special footnotes for Delphi.

- To add keywords for *F1* searches from the Object Inspector or the Code Editor, add "B" footnotes to your Help file screens.

  "B" footnotes are just like the "K" footnotes used for the standard WinHelp keyword search, but they are used only by the Delphi search engine. The following table shows how to create a "B" footnote for each kind of component Help screen.

Table 8.1 Component-screen Help search footnotes

Screen type	"B" footnote content	Example
Main component screen	'class_' + component type name	class_TMemo
Generic property or event	'prop_' + property name 'event_' + event name	prop_WordWrap event_OnChange
Component-specific property or event	'prop_' + component type name + property name 'event_' + component type name + event name	prop_TMemoWordWrap event_TMemoOnChange

It is important to distinguish between generic screens and component-specific screens. A generic screen is one which applies to the specific property or event in all components. For example, the *Left* property is identical in all components, so its search string is prop_Left. The *BorderStyle* property, on the other hand, is different depending on what component it belongs to, so the *BorderStyle* properties of specific components have their own screens. Thus, the edit box component, *TEdit*, has a screen for its *BorderStyle* property, with a search string of prop_TEditBorderStyle.

Creating the keyword file

When you have created and compiled a Help file for your components and added the special footnotes for keyword searches, you also need to generate a separate keyword file that Delphi can merge into its master search index for topic searches.

- To create a keyword (.KWF) file from your Help source (RTF) file,

  1. At the DOS prompt, go to the directory that contains the Help source (RTF) file.
  2. Run the the keyword-file generation application, KWGEN, followed by the name of the Help project (.HPJ) file for your Help file.

     For example, if your Help project file is named SPECIAL.HPJ, you would type the following at the DOS prompt:

     ```
     KWGEN SPECIAL.HPJ
     ```

     When KWGEN finishes, you will have a keyword file with the same name as the Help file and project file, but with the extension .KWF.
  3. Place the keyword file in the same directory with the compiled unit and Help file.

When you install your components into the Component palette, you'll also want to merge the keywords into the master search index for the Delphi Help system.

Merging the Help indexes

After you create the keyword file from the Help file for your components, you need to merge the keywords into the master Help index for Delphi.

- To merge your keyword file into the Delphi master Help index,

  1. Make sure you have placed your keyword (.KWF) file along with the compiled Help (.HLP) file in the directory with the compiled unit that contains your components.
  2. Run the HELPINST application. HELPINST is a Windows application installed with Delphi.

When HELPINST finishes, the Delphi master help index (.HDX) file includes the keywords for your component's Help screens.

Storing and loading properties

Delphi stores forms and their components in form (.DFM) files. A form file is a binary representation of the properties of a form and its components. When Delphi users add the components you write to their forms, your components must have the ability to

write their properties to the form file when saved. Similarly, when loaded into Delphi or executed as part of an application, the components must restore themselves from the form file.

Most of the time you won't need to do anything to make your components work with form files: the ability to store a representation and load from it are part of the inherited behavior of components. In some cases, however, you might want to alter the way a component stores itself or the way it initializes when loaded, so you should understand something about the underlying mechanism.

These are the aspects of property storage you need to understand:

- The store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading

The store-and-load mechanism

When an application developer designs forms, Delphi saves descriptions of the forms in a form (.DFM) file, which it later attaches to the compiled application. When a user runs the application, it reads in those descriptions.

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its public and published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, which sets all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

Specifying default values

Delphi components only save their property values if those values differ from the default values. If you don't specify otherwise, Delphi assumes a property has no default value, meaning the component always stores the property, whatever its value.

A property whose value is not set by a components constructor assumes a zero value. A zero value means whatever value the property assumes when its storage memory is set to zero. That is, numeric values default to zero, Boolean values to *False*, pointers to **nil**, and so on. If there is any doubt, specify the default value explicitly.

- To specify a default value for a property, add the **default** directive and the new default value to the end of the property declaration.

You can also specify a default value when redeclaring a property. In fact, one reason to redeclare a property is to designate a different default value.

Note Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's *Create* constructor assigns the necessary value.

The following code shows a component declaration that specifies a default value for the *Align* property and the implementation of the component's constructor that sets the default value. In this case, the new component is a special case of the standard panel component that will be used for status bars in a window, so its default alignment should be to the bottom of its owner.

```
type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;      { override to set new default }
  published
    property Align default alBottom;                  { redeclare with new default value }
  end;
⋮
constructor TStatusBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                            { perform inherited initialization }
  Align := alBottom;                                 { assign new default value for Align }
end;
```

Determining what to store

You can control whether Delphi stores each of your components' properties. By default, all properties in the **published** part of the object declaration are stored. You can choose to not store a given property at all, or designate a function that determines at run time whether to store the property.

- To control whether Delphi stores a property, add the **stored** directive to the property declaration, followed by *True*, *False*, or the name of a Boolean method.

You can add a **stored** clause to the declaration or redeclaration of any property.

The following code, for example, shows a component that declares three new properties. One is always stored, one is never stored, and the third is stored depending on the value of a Boolean method:

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public                                                          { normally not stored }
    property Important: Integer stored True;                           { always stored }
  published                                                      { normally stored always }
    property Unimportant: Integer stored False;                         { never stored }
    property Sometimes: Integer stored StoreIt;       { storage depends on function value }
  end;
```

Initializing after loading

After a component reads all its property values from its stored description, it calls a virtual method called *Loaded*, which provides a chance to perform any initializations that might be required. The call to *Loaded* occurs before the form and its controls are shown, so you don't need to worry about initialization causing flicker on the screen.

- To initialize a component after it loads its property values, override the *Loaded* method.

Note The first thing you do in any *Loaded* method you write is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you perform initializations on your own component.

The following code comes from the *TDatabase* component. After loading, the database tries to reestablish any connections that were open at the time it was stored, and specifies how to handle any exceptions that occur while connecting.

```
procedure TDatabase.Loaded;
begin
  inherited Loaded;                          { always call the inherited method first }
  Modified;                                                { this sets internal flags }
  try
    if FStreamedConnected then Open;                        { reestablish connections }
  except
    if csDesigning in ComponentState then                              { at design time... }
      Application.HandleException(Self)          { ...let the Delphi handle the exception }
    else Raise;                                                 { otherwise, reraise }
  end;
end;
```

Summary

Registration enables a component to operate as part of the component library and appear on the Component palette. You can test your components without registering them, but to make them available to other application developers, you need to register them and provide Help on elements you add to the standard components.

You can also control what parts of your component data Delphi stores and loads, making the loading process more efficient and your components more useful.

Part

II

Sample components

The chapters in this part give you concrete examples of creating various kinds of components. Before trying any of these examples, you should at least be familiar with the general description of component creation in Chapter 1, including deriving component objects and registering components.

Each chapter presents a complete example of a different kind of component you can create:

- **Chapter 9, "Modifying an existing component,"** shows the simplest kind of component creation, deriving a new component from an existing, complete component, including changing default property values.
- **Chapter 10, "Creating a graphic component,"** demonstrates the creation of an original component, including adding properties, managing owned objects, and using graphics.
- **Chapter 11, "Customizing a grid,"** shows how to modify one of the abstract component objects to create a custom control, including responding to Windows messages, defining custom property types, and adding methods to the component.
- **Chapter 12, "Making a control data-aware,"** shows how to make a data browser out of an existing control.
- **Chapter 13, "Making a dialog box a component,"** illustrates how to turn a user-designed form into a reusable component.
- **Chapter 14, "Building a dialog box into a DLL,"** describes how to turn a user-designed form into a dynamic-link library (DLL) that any Windows application can use.

Chapter

9

Modifying an existing component

The simplest way to create a component is to start with a component that does nearly everything you want and make whatever small change you need. The most common reason to do this is to change one or more of the default property values in the standard components.

The mechanics of modifying an existing component are quite simple. The actual process of creating the new component is exactly the process described in "Creating a new component" on page 16. Once you've created, registered, and installed the new component, however, you'll make just small changes to the new component object.

The example in this chapter will modify the standard memo component to create a memo that does not wrap words by default. This is a very simple example, but it illustrates all you need to know to modify existing components.

By default, the value of the memo component's *WordWrap* property is *True*. If you use several memos that don't need to wrap words, you can easily create a new memo component that doesn't wrap words by default.

■ Modifying an existing component takes only two steps:

- Creating and registering the component
- Modifying the component object

The other chapters in this part describe creating more complex components. The basic process will always be the same, but with the more complex components, you'll have more steps involved in customizing the new object.

Creating and registering the component

Creation of every component begins the same way: you create a unit, derive a component object, register it, and install it on the Component palette. This process is explained in "Creating a new component" on page 16.

➤ For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *Memos*.
- Derive a new component type called *TWrapMemo*, descended from *TMemo*.
- Register *TWrapMemo* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit Memos;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, StdCtrls;
type
  TWrapMemo = class(TMemo)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TWrapMemo]);
end;
end.
```

If you install the new component now, it will behave exactly like its ancestor, *TMemo*. In the next section, you'll make a simple change to your component.

Modifying the component object

Once you've created a new component object, you can modify it in almost any way. In this case, you will change only the default value of one property in the memo component.

■ Changing the default value of a property involves two small changes to the component object:

- Overriding the constructor
- Specifying the new default property value

Overriding the constructor actually sets the value of the property. Specifying the value as the default tells Delphi what value the constructor sets. Delphi only stores values that differ from their default values, so it is important that you perform both of these steps.

Overriding the constructor

All components set their initial property values when constructed. When you place a component on a form at design time or when a running application constructs a component and reads its property values from a form file, the first thing that happens is that the component's constructor is called, setting the default property values.

In the case of a component loaded from a form file, after constructing the object with its default property values, the application then sets any properties changed at design time, so that when the component appears, it looks and acts as designed.

The constructor, however, always determines the default property values.

- To change the default value of a property, override the component's constructor to set the desired value.

  When you override a constructor, the new constructor must *always* call the inherited constructor before doing anything else. For more information on constructors and overriding methods, see Chapter 2.

➤ For this example, your new component needs to override the constructor inherited from *TMemo* to set the *WordWrap* property to *False*. To achieve this, add the declaration of the constructor override to the object declaration, then write the new constructor in the **implementation** part of the unit:

```
type
  TWrapMemo = class(TMemo)
  public                                              { constructors are always public }
    constructor Create(AOwner: TComponent); override;  { this syntax is always the same }
  end;
⋮
constructor TWrapMemo.Create(AOwner: TComponent);      { this goes after implementation }
begin
  inherited Create(AOwner);                                  { ALWAYS do this first! }
  WordWrap := False;                                      { set the new desired value }
end;
```

Now you can install the new component on the Component palette and add it to a form. Note that the *WordWrap* property now defaults to *False*.

If you change (or create) a new default property value, you must also designate that value as the default. If you fail to match the value set by the constructor to the specified default value, Delphi cannot store and restore the proper value.

Specifying the new default property value

When Delphi stores a description of a form in a form file, it only stores the values of properties that differ from their default values. This has two advantages: keeping the form files small and making form loading faster. If you create a property or change the default value, it's a good idea to update the property declaration to include the new default. Form files, loading, and default values are all explained in more detail in Chapter 8.

- To change the default value of a property, redeclare the property name, followed by the directive **default** and the new default value.

  Note that you don't need to redeclare the entire property, just the name and the default value.

➤ For the word-wrapping memo component, you redeclare the *WordWrap* property in a published part of the object declaration, with a default value of *False*:

```
type
  TWrapMemo = class(TMemo)
  ⋮
  published
    property WordWrap default False;
  end;
```

Specifying the default property value doesn't affect the workings of the component at all. You must still explicitly set the default value in the component's constructor. The difference is in the inner workings of the application: Delphi no longer writes *WordWrap* to the form file if it is *False*, since you've told it that the constructor will set that value automatically.

Summary

Modifying existing components is very simple, and is mostly done to change a few default property values. The rest of the examples in this part (like most components you'll create yourself) are more complex, but the basic process is the same: you derive an object from an existing object type that provides as much of what you want as possible, then add to that object to create the desired result.

The main difference you'll see in future examples is that your starting point isn't already a working component, so you'll have to do a little more work to get beyond the very basic capabilities built into the abstract objects you'll use as starting points.

Chapter

10

Creating a graphic component

Another simple kind of component is a graphic control. Because a purely graphic control never receives keyboard focus, it doesn't have or need a window handle. Users of applications containing graphic controls can still manipulate the control with the mouse, but since the control never receives focus, there's no keyboard interface.

The graphic component presented in this chapter is *TShape*, the shape component on the Additional page of the Component palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. This chapter calls its shape component *TSampleShape* and shows you all the steps involved in creating the shape component.

Creating a graphic component requires three steps:

- Creating and registering the component
- Publishing inherited properties
- Adding graphic capabilities

Creating and registering the component

Creation of every component begins the same way: You create a unit, derive a component object, register it, and install it on the Component palette. This process is explained in "Creating a new component" on page 16.

➤ For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *Shapes*.
- Derive a new component type called *TSampleShape*, descended from *TGraphicControl*.
- Register *TSampleShape* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit Shapes;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TSampleShape = class(TGraphicControl)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.
```

Publishing inherited properties

Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor type you want to make available to the users of your new component. *TGraphicControl* already publishes all the properties that enable the component to function as a control, so all you need to publish is the ability to respond to mouse events and handle drag-and-drop.

Publishing inherited properties and events is explained in "Publishing inherited properties" on page 33 and "Making events visible" on page 49. Both processes involve redeclaring just the name of the properties in the **published** part of the object-type declaration.

➤ For the shape control, you can publish the three mouse events, the three drag-and-drop events, and two drag-and-drop properties:

```
type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;                                { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;                                    { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;                                           { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;
```

The sample shape control now makes mouse and drag-and-drop interactions available to its users.

Adding graphic capabilities

Once you have declared your graphic component and published any inherited properties you want to make available, you can add the graphic capabilities that distinguish your component. There are two steps you'll always perform when creating a graphic control:

- Determining what to draw
- Drawing the component image

In addition, for the shape control example, you'll add some properties that enable application developers using the shape control to customize the appearance of the shape at design time.

Determining what to draw

A graphic control generally has the ability to change its appearance to reflect either a dynamic condition or a user-specified condition or both. A graphic component that always looks the same is probably not well suited to being a component at all. If you want a static image, you should probably import a bitmapped image instead of using a graphic control.

In general, the appearance of a graphic control depends on some combination of its properties. The gauge control, for example, has properties that determine its shape and orientation and whether it shows its progress numerically as well as graphically. Similarly, the shape control has a property that determines what kind of shape it should draw.

➤ To give the shape control a property to determine the shape it draws, add a property called *Shape*, which requires three steps:

- Declaring the property type
- Declaring the property
- Writing the implementation method

Creating properties is explained in more detail in Chapter 3.

Declaring the property type

When you declare a property of a user-defined type, you must be sure to declare the property type first, before the object type that includes the property. The most common sort of user-defined type for properties is enumerated.

For the shape control, you need an enumerated type with an element for each kind of shape the control can draw.

➤ Add the following type definition above the shape control object's declaration:

```
type
  TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
  TSampleShape = class(TGraphicControl)                    { this is already there }
```

You can now use this type to declare a new property in the object.

Declaring the property

When you declare a property, you usually need to declare a private field to store the data for the property, then specify methods for reading and/or writing the property value. Often, you don't need to use a method to read the value, but can just point to the stored data instead.

For the shape control, you'll declare a field that holds the current shape, then declare a property that reads that field and writes through a method call.

➤ Add the following declarations to *TSampleShape*:

```
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType;                          { field to hold property value }
    procedure SetShape(Value: TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

Now all that remains is to add the implementation of *SetShape*.

Writing the implementation method

When either the read or write part of a property definition uses a method call instead of directly accessing the stored property data, you need to implement those methods.

➤ Add the implementation of the *SetShape* method to the **implementation** part of the unit:

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then                          { ignore if this isn't a change }
  begin
    FShape := Value;                                          { store the new value }
    Invalidate(True);                          { force a repaint with the new shape }
  end;
end;
```

Overriding the constructor and destructor

In order to change default property values and initialize owned objects for your component, you need to override the inherited constructor and destructor. In both cases, remember always to call the inherited method in your new constructor or destructor.

Changing default property values

The default size of a graphic control is rather small, so you can change the width and height in the constructor. Changing default property values is explained in more detail in Chapter 9.

➤ In this example, the shape control sets its size to a square 65 pixels on each side.

1 Add the overridden constructor to the declaration of the component object:

```
type
  TSampleShape = class(TGraphicControl)
  public                                              { constructors are always public}
    constructor Create(AOwner: TComponent); override;    { remember override directive }
  end;
```

2 Redeclare the *Height* and *Width* properties with their new default values:

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  published
    property Height default 65;
    property Width default 65;
  end;
```

3 Write the new constructor in the implementation part of the unit:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  Width := 65;
  Height := 65;
end;
```

Publishing the pen and brush

By default, a canvas has a thin, black pen and a solid, white brush. To enable developers using the shape control to change those aspects of the canvas, you must provide objects for them to manipulate at design time, then copy those objects into the canvas when painting. Objects such as an auxiliary pen or brush are called *owned objects* because the component owns them and is responsible for creating and destroying them.

Managing owned objects requires three steps:

- Declaring the object fields
- Declaring the access properties
- Initializing owned objects

Declaring the object fields

Each object a component owns must have an object field declared for it in the component. The field ensures that the component always has a pointer to the owned object so it can destroy the object before destroying itself. In general, a component initializes owned objects in its constructor, and destroys them in its destructor.

Fields for owned objects are nearly always declared as private. If users or components need access to the owned objects, you can declare properties that provide the access.

➤ Add fields for pen and brush objects to the shape control:

```
type
  TSampleShape = class(TGraphicControl)
  private                                          { fields are nearly always private }
    FPen: TPen;                                         { a field for the pen object }
    FBrush: TBrush;                                   { a field for the brush object }
    ⋮
  end;
```

Declaring the access properties

You can provide access to a the owned objects of a component by declaring properties of the type of the objects. That gives developers using the component a way to access the objects either at design time or at run time. In general, the read part of the property just references the object field, but the write part calls a method that enables the component to react to changes in the owned object.

➤ Add properties to the shape control that provide access to the pen and brush fields. You'll also declare the methods for reacting to changes to the pen or brush.

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  private                                           { these methods should be private }
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published                                      { make these available at design time }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
```

Then, write the *SetBrush* and *SetPen* methods in the implementation part of the unit:

```
procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);                        { replace existing brush with parameter }
end;

procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);                            { replace existing pen with parameter }
end;
```

Initializing owned objects

If you add objects to your component, the component's constructor must initialize those objects so that the user can interact with the objects at run time. Similarly, the component's destructor must also destroy the owned objects before destroying the component itself.

➤ Because you added a pen and a brush to the shape control, you need to initialize them in the shape control's constructor and destroy them in the control's destructor:

1 Construct the pen and brush in the shape control constructor:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                          { construct the pen }
  FBrush := TBrush.Create;                                     { construct the brush }
end;
```

2 Add the overridden destructor to the declaration of the component object:

```
type
  TSampleShape = class(TGraphicControl)
  public                                                { destructors are always public}
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;                        { remember override directive }
  end;
```

3 Write the new destructor in the implementation part of the unit:

```
destructor TSampleShape.Destroy;
begin
  FPen.Free;                                                  { destroy the pen object }
  FBrush.Free;                                               { destroy the brush object }
  inherited Destroy;                          { always call the inherited destructor, too }
end;
```

Setting owned objects' properties

As one last step in handling the pen and brush objects, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush objects have *OnChange* events, so you can create a method in the shape control and point both *OnChange* events to it.

➤ Add the following method to the shape control, and update the component's constructor to set the pen and brush events to the new method:

```
type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender: TObject);
  end;
⋮
implementation
⋮
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                          { construct the pen }
  FPen.OnChange := StyleChanged;                  { assign method to OnChange event }
  FBrush := TBrush.Create;                                     { construct the brush }
  FBrush.OnChange := StyleChanged;                { assign method to OnChange event }
end;
```

```
procedure TSampleShape.StyleChanged(Sender: TObject);
begin
  Invalidate(True);                                    { erase and repaint the component }
end;
```

With these changes, the component redraws to reflect changes to either the pen or the brush.

Drawing the component image

The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a virtual method called *Paint* that you override to paint the image you want on your control.

The *Paint* method for the shape control needs to do several things:

- Use the pen and brush selected by the user
- Use the selected shape
- Adjust coordinates so that squares and circles use the same width and height.

■ Overriding the *Paint* method requires two steps:

1. Add *Paint* to the component's declaration.
2. Write the *Paint* method in the **implementation** part.

➤ For the shape control, add the following declaration to the object declaration:

```
type
  TSampleShape = class(TGraphicControl)
  ⋮
  protected
    procedure Paint; override;
  ⋮
  end;
```

Then write the method in the **implementation** part :

```
procedure TSampleShape.Paint;
begin
  with Canvas do
  begin
    Pen := FPen;                                          { copy the component's pen }
    Brush := FBrush;                                      { copy the component's brush }
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height);                   { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4);  { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height);                     { draw round shapes }
    end;
  end;
end;
```

Paint is called whenever the control needs to update its image. Windows tells controls to paint when they first appear or when a window in front of them goes away. In addition, you can force repainting by calling *Invalidate*, as the *StyleChanged* method does.

Refining the shape drawing

The standard shape control does one more thing that your sample shape control doesn't yet do: It handles squares and circles as well as rectangles and ellipses. To do that, you need to adjust the height or width of the shape drawn to match the shorter side of the control.

There's nothing "tricky" or special about handling the square and circle shapes. It's just a matter of writing a little more code to find the shortest side and center the image.

➤ Here's a refined *Paint* method that adjusts for squares and ellipses:

```
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin
  with Canvas do
  begin
    Pen := FPen;                                          { copy the component's pen }
    Brush := FBrush;                                    { copy the component's brush }
    W := Width;                                           { use the component width }
    H := Height;                                        { use the component height }
    if W < H then S := W else S := H;                { save smallest for circles/squares }

    case FShape of                                     { adjust height, width and position }
      sstRectangle, sstRoundRect, sstEllipse:
        begin
          X := 0;                                   { origin is top-left for these shapes }
          Y := 0;
        end;
      sstSquare, sstRoundSquare, sstCircle:
        begin
          X := (W - S) div 2;                                { center these horizontally }
          Y := (H - S) div 2;                                          { and vertically }
          W := S;                                  { use shortest dimension for width }
          H := S;                                                      { and for height }
        end;
    end;

    case FShape of
      sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);                     { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);           { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);                                 { draw round shapes }
    end;
  end;
end;
```

Summary

Creating a graphic control in Delphi is as easy as creating any other control. Use *TGraphicControl* as a starting point, publish any properties you want available in your control, and write a *Paint* method to render the graphic image.

For further suggestions on generating graphic images with the graphics features of Delphi, see Chapter 6.

Chapter

11

Customizing a grid

Delphi provides several kinds of abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. In this chapter, you'll see how to create a small one-month calendar from the basic grid component, *TCustomGrid*.

Creating the calendar takes seven steps:

- Creating and registering the component
- Publishing inherited properties
- Changing initial values
- Resizing the cells
- Filling in the cells
- Navigating months and years
- Navigating days

The resulting calendar component is almost the same as the *TCalendar* component on the Samples page of the Component palette. The main difference is that *TCalendar* provides a bit more flexibility than the *TSampleCalendar* described in this chapter.

Creating and registering the component

Creation of every component begins the same way: You create a unit, derive a component object, register it, and install it on the Component palette. This process is explained in "Creating a new component" on page 16.

➤ For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *CalSamp*.
- Derive a new component type called *TSampleCalendar*, descended from *TCustomGrid*.
- Register *TSampleCalendar* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit CalSamp;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Grids;
type
  TSampleCalendar = class(TCustomGrid)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;
end.
```

If you install the calendar component now, you'll find that it appears on the Samples page, and you can place it on a form and run the application. The only properties available, however, are the most basic control properties. The next step will be to make some of the more specialized properties available to users of the calendar.

Publishing inherited properties

The abstract grid component, *TCustomGrid*, provides a large number of protected properties. You can choose which of those properties you want to make available to users of the calendar control. A number of the properties are very useful for the calendar.

- To make inherited protected properties available to users of your components, redeclare the properties in the **published** part of your component's declaration.

➤ For the calendar control, publish the following properties and events, as shown here:

```
type
  TSampleCalendar = class(TCustomGrid)
  published
    property Align;                                    { publish properties }
    property BorderStyle;
    property Color;
    property Ctl3D;
    property Font;
    property GridLineWidth;
    property ParentColor;
    property ParentFont;
    property OnClick;                                      { publish events }
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
  end;
```

Note that there are a number of other properties you could also publish, but which do not apply to a calendar, such as the *Options* property that would enable the user to choose which grid lines to draw.

If you rebuild the component library and try out the modified calendar component now, you'll find many more properties and events available in the calendar, all fully functional. You can now start adding new capabilities of your own design.

Changing initial values

A calendar is essentially a grid with a fixed number of rows and columns, although not all the rows always contain dates. For this reason, you haven't published the grid properties *ColCount* and *RowCount*, since it is highly unlikely that users of the calendar will want to display anything other than seven days per week. However, you still need to set the initial values of those properties so that the week always has seven days.

- To change the initial values of the component's properties, override the constructor to set the desired values.

  Remember that you need to add the constructor to the public part of the component's object declaration, then write the new constructor in the **implementation** part of the component's unit. The first statement in the new constructor should always be a call to the inherited constructor.

```
type
  TSampleCalendar = class(TCustomGrid
  public
    constructor Create(AOwner: TComponent); override;
  ⋮
  end;
⋮
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                              { call inherited constructor }
  ColCount := 7;                                            { always seven days/week }
  RowCount := 7;                                  { always six weeks plus the headings }
  FixedCols := 0;                                                { no row labels }
  FixedRows := 1;                                          { one row for day names }
  ScrollBars := ssNone;                                       { no need to scroll }
  Options := Options - [goRangeSelect};                    { disable range selection }
end;
```

Now when you add a calendar to a form it has seven rows and seven columns, with the top row fixed. You probably need to resize the control to make all the cells visible, however. In the next section, you'll see how to respond to the resizing message from Windows to adjust the size of the cells to fit into whatever size the user sets.

Resizing the cells

When a user or application changes the size of a window or control, Windows sends a message called *WM_SIZE* to the affected window or control so it can adjust any settings

needed to later paint its image in the new size. Your component can respond to that message by altering the size of the cells so they all fit inside the boundaries of the control. To respond to the *WM_SIZE* message, you'll add a message-handling method to the component.

Creating a message-handling method is described in detail in "Creating new message handlers" on page 71.

➤ In this case, the calendar control needs a response to *WM_SIZE*, so add a protected method called *WMSize* to the control indexed to the *WM_SIZE* message, then write the method so that it calculates the proper cell size to allow all cells to be visible in the new size:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
  ⋮
  end;
⋮
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines: Integer;                                    { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                 { calculate combined size of all lines }
  DefaultColWidth := (Message.Width - GridLines) div 7;     { set new default cell width }
  DefaultRowHeight := (Message.Height - GridLines) div 7;             { and cell height }
end;
```

Now if you add a calendar to a form and resize the calendar, it always displays all the cells for the calendar in the largest size that will fit in the control.

Of course, a calendar with no dates in it is not very useful. In the next section, you'll start adding some content to the calendar.

Filling in the cells

A grid control fills in its contents cell-by-cell. In the case of the calendar, that means calculating which date, if any, belongs in each cell. The default drawing for grid cells takes place in a virtual method called *DrawCell*.

■ To fill in the contents of grid cells, override the *DrawCell* method.

The easiest part to fill in is the heading cells in the fixed row. The run-time library contains an array with short day names, so use the appropriate one for each column:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
      override;
  end;
⋮
```

```
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    TextOut(0, 0, ShortDayNames[ACol + 1]);              { use RTL strings for headings }
end;
```

Tracking the date

For the calendar control to be useful, users and applications must have a mechanism for setting the day, month, and year. Delphi stores dates and times in variables of type *TDateTime*. *TDateTime* is an encoded numeric representation of the date and time, which is useful for programmatic manipulation, but not convenient for human use.

You can therefore store the date in encoded form, providing run-time access to that value, but also provide *Day*, *Month*, and *Year* properties that users of the calendar component can set at design time.

Storing the internal date

To store the date for the calendar, you need a private field to hold the date and a run-time-only property that provides access to that date. In the next section, you'll also add properties to access the day, month, and year individually.

➤ Adding the internal date to the calendar requires three steps:

1 Declare a private field to hold the date:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  ⋮
```

2 Initialize the date field in the constructor:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                    { this is already here }
  ⋮                                                          { other initializations here }
  FDate := Date;                                              { get current date from RTL }
end;
```

3 Declare a run-time property to allow access to the encoded date.

You'll need a method for setting the date, because setting the date requires updating the onscreen image of the control:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  ⋮
```

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                                          { set new date value }
  Refresh;                                               { update the onscreen image }
end;
```

Accessing the day, month, and year

An encoded numeric date is fine for applications, but humans prefer to work with days, months, and years. You can provide alternate access to those elements of the stored, encoded date by creating properties.

Because each element of the date (day, month, and year) is an integer, and since setting each requires encoding the date when set, you can avoid duplicating the code each time by sharing the implementation methods for all three properties. That is, you can write two methods, one to read an element and one to write one, and use those methods to get and set all three properties.

➤ To provide design-time access to the day, month, and year, you do the following:

1. Declare the three properties, assigning each a unique **index** number:

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  ⋮
```

2. Declare and write the implementation methods, setting different elements for each index value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer;         { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ⋮
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);             { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
```

```
begin
  if Value > 0 then                                    { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);              { get current date elements }
    case Index of                                    { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value else Exit;
      3: ADay := Value else Exit;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);             { encode the modified date }
    Refresh;                                          { update the visible calendar }
  end;
end;
```

Now you can set the calendar's day, month, and year at design time using the Object Inspector or at run time using code. Of course, you have not yet added the code to paint the dates into the cells, but now you have the needed data.

Generating the day numbers

Putting numbers into the calendar involves several considerations. The number of days in the month depends on which month it is, and whether the given year is a leap year. In addition, months start on different days of the week, dependent on the month and year.

The previous section described how to keep track of the current month and year. Now, based on those, you can determine whether the specified year is a leap year and the number of days in the month.

➤ Add two more methods to the calendar object: a Boolean function that indicates whether the current year is a leap year, and an integer function that returns the number of days in the current month. Be sure to add the method declarations to the *TSampleCalendar* type declaration.

```
function TSampleCalendar.IsLeapYear: Boolean;
begin
  Result := (Year mod 4 = 0)                           { years divisible by 4 are... }
    and ((Year mod 100 <> 0                             { ...except century years... }
    or (Year mod 400 = 0));                        { ...unless it's divisible by 400 }
end;

function DaysThisMonth: Integer;
const
  DaysPerMonth: array[1..12] of Integer =
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);          { usual numbers of days }
begin
  if FDate = 0 then Result := 0                  { 0 indicates invalid date, so no days }
  else
  begin
    Result := DaysPerMonth[Month];                       { normally, just return number }
    if (Month = February) and IsLeapYear then Inc(Result);     { plus 1 in leap February }
  end;
end;
```

Note that the *DaysThisMonth* function returns zero if the stored date is empty. This allows applications to indicate an invalid date. By indicating that the month has no days, you will produce a blank calendar to represent the invalid date.

Once you have the information on leap years and days per month, you can calculate where in the grid the individual dates go. The calculation is based on the day of the week the month starts on.

Because you'll need the month-offset number for each cell you fill in, the best practice is to calculate it once when you change the month or year, then refer to it each time. You can store the value in an object field, then update that field each time the date changes.

➤ To fill in the days in the proper cells, you do the following:

1 Add a month-offset field to the object and a method that updates the field value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;                                  { storage for the offset }
  ⋮
  protected
    procedure UpdateCalendar; virtual;                    { property for offset access }
  end;
⋮
procedure TSampleCalendar.UpdateCalendar;
var
  AYear, AMonth, ADay: Word;
  FirstDate: TDateTime;                              { date of the first day of the month }
begin
  if FDate <> 0 then                               { only calculate offset if date is valid }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                      { get elements of date }
    FirstDate := EncodeDate(AYear, AMonth, 1);                     { date of the first }
    FMonthOffset := 2 - DayOfWeek(FirstDate);        { generate the offset into the grid }
  end;
  Refresh;                                                  { always repaint the control }
end;
```

2 Add statements to the constructor and the *SetCalendarDate* and *SetDateElement* methods that call the new update method whenever the date changes:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                       { this is already here }
  ⋮                                                          { other initializations here }
  UpdateCalendar;                                                 { set proper offset }
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                                               { this was already here }
  UpdateCalendar;                                  { this previously called Refresh }
end;
```

```
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  ⋮
    FDate := EncodeDate(AYear, AMonth, ADay);                { encode the modified date }
    UpdateCalendar;                                       { this previously called Refresh }
  end;
end;
```

3 Add a method to the calendar that returns the day number when passed the row and column coordinates of a cell:

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;          { calculate day for this cell }
  if (Result < 1) or (Result > DaysThisMonth) then Result := -1;  { return -1 if invalid }
end;
```

Remember to add the declaration of *DayNum* to the component's type declaration.

4 Now that you can calculate where the dates go, you can update *DrawCell* to fill in the dates:

```
procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
  TheText: string;
  TempDay: Integer;
begin
  if ARow = 0 then                                        { if this is the header row ...}
    TheText := ShortDayNames[ACol + 1]                          { just use the day name }
  else
  begin
    TheText := '';                                          { blank cell is the default }
    TempDay := DayNum(ACol, ARow);                           { get number for this cell }
    if TempDay <> -1 then TheText := IntToStr(TempDay);       { use the number if valid }
  end;
  with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;
```

Now if you reinstall the calendar component and place one on a form, you'll see the proper information for the current month. Although you can change the month and year at will using the appropriate properties, there are common changes (such as moving to adjacent months or years) that can easily be made into methods, as shown later in the chapter.

Selecting the current day

Now that you have numbers in the calendar cells, it makes sense to move the selection highlighting to the cell containing the current day. By default, the selection starts on the top left cell, so you need to set the *Row* and *Column* properties both when constructing the calendar initially and when the date changes.

➤ To set the selection on the current day, change the *UpdateCalendar* method to set *Row* and *Column* before calling *Refresh*:

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    ⋮                                          { existing statements to set FMonthOffset }
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh;                                                  { this is already here }
end;
```

Note that you're now reusing the *ADay* variable previously set by decoding the date.

Navigating months and years

Properties are useful for manipulating components, especially at design time. But sometimes there are types of manipulations that are so common or natural, often involving more than one property, that it makes sense to provide methods to handle them. One example of such a natural manipulation is a "next month" feature for a calendar. Handling the wrapping around of months and incrementing of years is simple, but very convenient for the developer using the component.

The only drawback to encapsulating common manipulations into methods is that methods are only available at run time. However, such manipulations are generally only cumbersome when performed repeatedly, and that's fairly rare at design time.

➤ For the calendar, add the following four methods for next and previous month and year:

```
procedure TCalendar.NextMonth;
begin
  if Month < 12 then                                    { if it's not December... }
    Month := succ(Month)                               { ...just increment the month }
  else
  begin
    Year := Year + 1;                                 { otherwise, it's next year... }
    Month := 1;                                                { ...and January }
  end;
end;

procedure TCalendar.PrevMonth;
begin
  if Month > 1 then                                          { if not January... }
    Month := pred(Month)                                  { ...just decrement month }
  else
  begin
    Year := Year - 1;                               { otherwise, it's the previous year... }
    Month := 12;                                             { ...and December }
  end;
end;
```

```
procedure TCalendar.NextYear;
begin
  Year := Year + 1;                                          { increment year }
end;

procedure TCalendar.PrevYear;
begin
  Year := Year - 1;                                          { decrement year }
end;
```

Be sure to add the declarations of the new methods to the object declaration.

Now when you create an application that uses the calendar component, you can easily implement browsing through months or years. The CBROWSE application in the DEMOS\CBROWSE directory shows just such an example.

Navigating days

Within a given month, there are two obvious ways to navigate among the days. The first is to use the arrow keys, and the other is to respond to clicks of the mouse. The standard grid component handles both as if they were clicks. That is, an arrow movement is treated like a click on an adjacent cell.

Moving the selection

The inherited behavior of a grid handles moving the selection in response to either arrow keys or clicks, but if you want to change the selected day, you need to modify that default behavior.

➤ To handle movements within the calendar, override the *Click* method of the grid.

When you override a method such as *Click* that is tied in with user interactions, you will nearly always include a call to the inherited method, so as not to lose the standard behavior.

The following is an overridden *Click* method for the calendar grid. Be sure to add the declaration of *Click* to *TSampleCalendar*, including the **override** directive afterward.

```
procedure TSampleCalendar.Click;
begin
  inherited Click;                          { remember to call the inherited method! }
  TempDay := DayNum(Col, Row);              { get the day number for the clicked cell }
  if TempDay <> -1 then Day := TempDay;                        { change day if valid }
end;
```

Providing an OnChange event

Now that users of the calendar can change the date within the calendar, it makes sense to allow applications to respond to those changes.

➤ Add an *OnChange* event to *TSampleCalendar*.

1 Declare the event, a field to store the event, and a dynamic method to call the event:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  ⋮
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  ⋮
```

2 Write the *Change* method:

```
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
```

3 Add statements calling *Change* to the end of the *SetCalendarDate* and *SetDateElement* methods:

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                          { this is the only new statement }
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    ⋮                                          { many statements setting element values }
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change;                                                        { this is new }
  end;
end;
```

Applications using the calendar component can now respond to changes in the date of the component by attaching handlers to the *OnChange* event.

Excluding blank cells

When navigating the days in the calendar, it is not clear what it means to select a blank cell. As currently written, the calendar moves the selection to the blank cell, but does not change the date of the calendar. It makes sense, then, to disallow selection of the blank cells.

- To control whether a given cell is selectable, override the *SelectCell* method of the grid.

  SelectCell is a function that takes a column and row as parameters, and returns a Boolean value indicating whether the specified cell is selectable.

➤ You can override *SelectCell* to return *False* if the cell does not contain a valid date:

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False            { -1 indicates invalid date }
  else Result := inherited SelectCell(ACol, ARow);      { otherwise, use inherited value }
end;
```

Now if the user clicks a blank cell or tries to move to one with an arrow key, the calendar leaves the current cell selected.

Summary

TCustomGrid provides most of the basic behavior you need in a grid control. By choosing appropriate inherited properties and events to publish, you can produce a very usable custom component very quickly by adding just those parts unique to your component.

Delphi provides abstract components for most of the kinds of controls you will likely want to customize, including buttons, check boxes, grids, list boxes, and combo boxes.

The calendar component also demonstrates several useful component-writing techniques, including responding to Windows messages and sharing implementation methods among similar properties.

Chapter

12

Making a control data-aware

When working with database connections, it is often convenient to have controls that are *data-aware*. That is, the application can establish a link between the control and some part of a database. Delphi includes data-aware labels, edit boxes, list boxes, combo boxes, and grids. You can also make your own controls data-aware.

There are several degrees of data awareness. The simplest is read-only data awareness, or *data browsing*, the ability to reflect the current state of a database. More complicated is editable data awareness, or *data editing*, where the user can edit the values in the database by manipulating the control. Note also that the degree of involvement with the database can vary, from the simplest case, a link with a single field, to more complex cases, such as multiple-record controls.

This chapter will illustrate the simplest case, making a read-only control that links to a single field in a database. The specific control used will be the calendar created in Chapter 11, *TSampleCalendar*. You can also use the standard calendar control on the Samples page of the Component palette, *TCalendar*.

■ Creating a data-aware calendar control involves the following steps:

- Creating and registering the component
- Making the control read-only
- Adding the data link
- Responding to data changes

Creating and registering the component

Creation of every component begins the same way: You create a unit, derive a component object, register it, and install it on the Component palette. This process is explained in "Creating a new component" on page 16.

➤ For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *DBCal*.
- Derive a new component type called *TDBCalendar*, descended from *TSampleCalendar*.
- Register *TDBCalendar* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit DBCal;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Grids, Calendar;
type
  TDBCalendar = class(TSampleCalendar)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TDBCalendar]);
end;
end.
```

You can now proceed with making the new calendar a data browser.

Making the control read-only

Since this data calendar will be read-only with respect to the data, it make sense to make the control itself read-only, so users won't make changes within the control and expect them to be reflected in the database.

➤ Making the calendar read only involves two steps:

- Adding the ReadOnly property
- Allowing needed updates

Adding the ReadOnly property

Adding a read-only option to the calendar control is a straightforward process. By adding a property, you'll provide a way to make the control read-only at design time. When that property is set to *True*, you can make all cells in the control unselectable.

1 Add the property declaration and a **private** field to hold the value:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FReadOnly: Boolean;                                  { field for internal storage }
  public
    constructor Create(AOwner: TComponent); override;    { must override to set default }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
```

```
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  FReadOnly := True;                                          { set the default value }
end;
```

2 Override the *SelectCell* method to disallow selection if the control is read-only. Use of *SelectCell* is explained in "Excluding blank cells" on page 112.

```
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if FReadOnly then Result := False                          { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;
```

Remember to add the declaration of *SelectCell* to the type declaration of *TDBCalendar*, and append the **override** directive.

If you now add the calendar to a form, you'll find that the component ignores clicks and keystrokes. Unfortunately, it also fails to update the selection position when you change the date. In the next section, you'll enable the control to make internal updates while still excluding user changes.

Allowing needed updates

The read-only calendar uses the *SelectCell* method for all kinds of changes, including setting the *Row* and *Col* properties. The *UpdateCalendar* method sets *Row* and *Col* every time the date changes, but since *SelectCell* disallows changes, the selection remains in place, even though the date changes.

To get around this absolute prohibition on changes, you can add an internal Boolean flag to the calendar, and permit changes when that flag is set to *True*:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean;                                 { private flag for internal use }
  protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
  public
    procedure UpdateCalendar; override;               { remember the override directive }
  end;
⋮

function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if (not FUpdating) and FReadOnly then Result := False       { allow select if updating }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;

procedure UpdateCalendar;
begin
  FUpdating := True;                                        { set flag to allow updates }
  try
    inherited UpdateCalendar;                                        { update as usual }
```

```
    finally
      FUpdating := False;                                    { always clear the flag }
    end;
  end;
```

The calendar still disallows user changes, but now correctly reflects changes made in the date by changing the date properties. Now that you have a true read-only calendar control, you are ready to add the data-browsing ability.

Adding the data link

The connection between a control and a database is handled by an object called a *data link*. There are several kinds of data links provided with Delphi. The data-link object that connects a control with a single field in a database is *TFieldDataLink*. There are also data links for entire tables.

A data-aware control *owns* its data-link object. That is, the control has the responsibility for constructing and destroying the data link. Chapter 10 describes the management of owned objects in more detail.

To establish a data link as an owned object, you perform three steps:

1. Declaring the object field
2. Declaring the access properties
3. Initializing the data link

Declaring the object field

A component needs an object field for each of its owned objects, as explained in "Declaring the object fields" on page 95. In this case, the calendar needs a field of type *TFieldDataLink* for its data link.

➤ Declare a field for the data link in the calendar:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FDataLink: TFieldDataLink;
    ⋮
  end;
```

Declaring the access properties

Every data-aware control has a *DataSource* property that specifies which data-source object in the application provides the data to the control. In addition, a control that access a single field needs a *DataField* property to specify that field in the data source.

Unlike the access properties for the owned objects in the example in Chapter 10, however, these access properties do not provide access to the owned objects themselves, but rather to corresponding properties in the owned object. That is, you'll create

properties that enable the control and its data link to share the same data source and data field.

➤ Declare the *DataSource* and *DataField* properties and their implementation methods, then write the methods as "pass-through" methods to the corresponding properties of the data-link object.

```
type
  TDBCalendar = class(TSampleCalendar)
  private                                          { implementation methods are private }
    function GetDataField: string;                 { returns the name of the data field }
    function GetDataSource: TDataSource;          { returns reference to the data source }
    procedure SetDataField(const Value: string);           { assigns name of data field }
    procedure SetDataSource(Value: TDataSource);              { assigns new data source }
  published                                    { make properties available at design time }
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
⋮
function TDBCalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;
```

Now that you have established the links between the calendar and its data link, there is one more important step. You must construct the data link object when the calendar control is constructed, and destroy the data link before destroying the calendar.

Initializing the data link

A data-aware control needs access to its data link throughout its existence, so it must construct the data-link object as part of its own constructor, and destroy the data-link object before it is itself destroyed.

➤ Override the *Create* and *Destroy* methods of the calendar to construct and destroy the data-link object, respectively.

```
type
  TDBCalendar = class(TSampleCalendar)
```

```
  public                                { constructors and destructors are always public }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                { always call the inherited constructor first }
  FReadOnly := True;                                          { this is already here }
  FDataLink := TFieldDataLink.Create;                  { construct the data-link object }
end;

destructor TDBCalendar.Destroy;
begin
  FDataLink.Free;                                  { always destroy owned objects first... }
  inherited Destroy;                                  { ...then call inherited destructor }
end;
```

Now you have a complete data link, but you have not yet told the control what data it should read from the linked field. The next section explains how to do that.

Responding to data changes

Once a control has a data link and properties to specify the data source and data field, it needs to respond to changes in the data in that field, either because of a move to a different record or because of a change made to that field.

Data-link objects all have events named *OnDataChange*. When the data source indicates a change in its data, the data-link object calls any event handler attached to its *OnDataChange* event.

- To update a control in response to data changes, attach a handler to the data link's *OnDataChange* event.

  In this case, you'll add a method to the calendar, then designate it as the handler for the data link's *OnDataChange*.

➤ Declare a *DataChange* method, then assign it to the data link's *OnDataChange* event:

```
type
  TDBCalendar = class(TSampleCalendar)
  private                               { this is an internal detail, so make it private }
    procedure DataChange(Sender: TObject);       { must have proper parameters for event }
  end;
⋮
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                { always call the inherited constructor first }
  FReadOnly := True;                                          { this is already here }
  FDataLink := TFieldDataLink.Create;                  { construct the data-link object }
  FDataLink.OnDataChange := DataChange;                       { attach handler to event }
end;
```

```
destructor TDBCalendar.Destroy;
begin
  FDataLink.OnDataChange := nil;              { detach handler before destroying object }
  FDataLink.Free;                               { always destroy owned objects first... }
  inherited Destroy;                               { ...then call inherited destructor }
end;

procedure TDBCalendar.DataChange(Sender: TObject);
begin
  if FDataLink.Field = nil then                      { if there is no field assigned... }
    CalendarDate := 0                                          { ...set to invalid date }
  else CalendarDate := FDataLink.Field.AsDate;     { otherwise, set calendar to the date }
end;
```

Summary

Making a data-browsing control is relatively simple. You provide a data-link object and properties that allow the control and the data link to share a data source, then specify the name of the field in that data source that the control will represent. By updating the control whenever the underlying data changes, you can browse the database.

Creating data-editing controls is somewhat more complicated. To get an idea of the process, you should first look at the source code for the controls in the *DBCtrls* unit, then look at the DATAEDIT.PAS sample.

Chapter

13

Making a dialog box a component

You will probably find it most convenient to make a frequently used dialog box a component that you can add to the Component palette. Your dialog-box components will work just like the components that represent the standard Windows common dialog boxes.

Making a dialog box a component requires four steps:

1. Defining the component interface
2. Creating and registering the component
3. Creating the component interface
4. Testing the component

The goal is to create a simple component that a user can add to a project and set properties for at design time. The Delphi "wrapper"component associated with the dialog box creates and executes it at run time, passing along the data the user specified. The dialog-box component is therefore both reusable and customizable.

In this chapter, you'll see how to create a wrapper component around the generic About Box form provided in the Delphi Gallery. There aren't many special considerations for designing a dialog box that will be wrapped into a component. Nearly any form can operate as a dialog box in this context.

Defining the component interface

Before you can create the component for your dialog box, you need to decide how you want developers to use it. You create an interface between your dialog box and applications that use it.

For example, look at the properties for the common dialog box components. They enable the developer to set the initial state of the dialog box, such as the caption and initial control settings, then read back any needed information after the dialog box closes. There's no direct interaction with the individual controls in the dialog box, just with the properties in the wrapper component.

The interface must therefore contain enough information that the dialog-box form can appear in the way the developer specifies and return any information the application needs. You can think of the properties in the wrapper component as being persistent data for a transient dialog box. If you've previously written applications using the ObjectWindows library, you'll recognize this as the role played by a transfer record.

In the case of the About box, you don't need to return any information, so the wrapper's properties only have to contain the information needed to display the About box properly. Since there are four separate fields in the About box that the application might affect, you'll provide four string-type properties to provide for them.

Creating and registering the component

Creation of every component begins the same way: You create a unit, derive a component object, register it, and install it on the Component palette. This process is explained in "Creating a new component" on page 16.

➤ For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *AboutDlg*.
- Derive a new component type called *TAboutBoxDlg*, descended from *TComponent*.
- Register *TAboutBoxDlg* on the Samples page of the Component palette.

The resulting unit should look like this:

```
unit AboutDlg;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TAboutBoxDlg = class(TComponent)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TAboutBoxDlg]);
end;
end.
```

If you install the new component now, it will have only the capabilities built into *TComponent*. It is the simplest nonvisual component. You can place it on a form and change its name. In the next section, you'll create the interface between the component and the dialog box.

Creating the component interface

Now that you have created a component and defined the interface between the component and the dialog box it will wrap, you can implement that interface.

■ There are three steps to creating the component interface:

1 Including the form unit
2 Adding interface properties
3 Adding the Execute method

Including the form unit

For your wrapper component to initialize and display the wrapped dialog box, you must add the form's unit to the **uses** clause of the wrapper component's unit.

➤ Append *About* to the **uses** clause of the *AboutDlg* unit.

The **uses** clause now looks like this:

```
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms
  About;
```

The form unit always declares an instance of the form type. In the case of the About box, the form type is *TAboutBox*, and the *About* unit includes the following declaration:

```
var
  AboutBox: TAboutBox;
```

So by adding *About* to the **uses** clause, you make *AboutBox* available to the wrapper component.

Adding interface properties

Before proceeding, decide on the properties your wrapper needs to enable developers to use your dialog box as a component in their applications. Then, you can add declarations for those properties to the component's type declaration.

Properties in wrapper components are somewhat simpler than the properties you would create if you were writing a regular component. Remember that in this case, you're just creating some persistent data that the wrapper can pass back and forth to the dialog box. By putting that data in the form of properties, you enable developers to set the data at design time for the wrapper to pass to the dialog box at run time.

■ Declaring an interface property requires two parts, both of which go in the **published** part of the component's type declaration:

- An object field, which is a variable the wrapper uses to store the value of the property.
- The property declaration itself, which specifies the name of the property and tells it which field to use for storage.

Interface properties of this sort do not need access methods. They use direct access to their stored data. By convention, the object field that stores the property's value has the same name as the property, but with the letter *F* in front. The field and the property *must* be of the same type.

For example, to declare an integer-type interface property called *Year*, you'd declare it as follows:

```
type
  TMyWrapper = class(TComponent)
  published
    FYear: Integer;                                { field to hold the Year-property data }
    property Year: Integer read FYear write FYear;       { property matched with storage }
  end;
```

➤ For this About box, you need four **string**-type properties—one each for the product name, the version information, the copyright information, and any comments:

```
type
  TAboutBoxDlg = class(TComponent)
  published
    FProductName, FVersion, FCopyright, FComments: string;              { declare fields }
    property ProductName: string read FProductName write FProductName;
    property Version: string read FVersion write FVersion;
    property Copyright: string read FCopyright write FCopyright;
    property Comments: string read FComments write FComments;
  end;
```

When you install the component onto the Component palette and place the component on a form, you'll be able to set the properties, and those values will automatically stay with the form. The wrapper can then use those values when executing the wrapped dialog box.

Adding the Execute method

The final part of the component interface is a way to open the dialog box and return a result when it closes. As with the common-dialog-box components, you'll use a Boolean function called *Execute* that returns *True* if the user clicks OK or *False* if the user cancels the dialog box.

The declaration for the *Execute* method always looks like this:

```
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
```

The minimum implementation for *Execute* needs to construct the dialog box form, show it as a modal dialog box, and return either *True* or *False*, depending on the return value from *ShowModal*.

Here's the minimal *Execute* method, for a dialog-box form of type *TMyDialogBox*:

```
function TMyWrapper.Execute: Boolean;
begin
  DialogBox := TMyDialogBox.Create(Application);                    { construct the form }
  try
    Result := (DialogBox.ShowModal = IDOK);     { execute; set result based on how closed }
```

```
    finally
      DialogBox.Free;                                    { dispose of the form }
    end;
  end;
```

Note the use of a **try..finally** block to ensure that the application disposes of the dialog-box object even if an exception occurs. In general, whenever you construct an object this way, you should use a **try..finally** block to protect the block of code and make certain the application frees any resources it allocates. For more information on these protections, see Chapter 7 in the *Delphi User's Guide*.

In practice, there will be more code inside the **try..finally** block. Specifically, before calling *ShowModal*, the wrapper will set some of the dialog box's properties based on the wrapper component's interface properties. After *ShowModal* returns, the wrapper will probably set some of its interface properties based on the outcome of the dialog box execution.

In the case of the About box, you need to use the wrapper component's four interface properties to set the contents of the labels in the About box form. Since the About box doesn't return any information to the application, there's no need to do anything after calling *ShowModal*. The About-box wrapper's *Execute* method looks like this:

```
function TAboutBoxDlg.Execute: Boolean;
begin
  AboutBox := TAboutBox.Create(Application);                { construct About box }
  try
    if ProductName = '' then                          { if product name's left blank... }
      ProductName := Application.Title;               { ...use application title instead }
    AboutBox.ProductName.Caption := ProductName;                { copy product name }
    AboutBox.Version.Caption := Version;                        { copy version info }
    AboutBox.Copyright.Caption := Copyright;                  { copy copyright info }
    AboutBox.Comments.Caption := Comments;                          { copy comments }
    AboutBox.Caption := 'About ' + ProductName;             { set About-box caption }
    with AboutBox do
    begin
      ProgramIcon.Picture.Graphic := Application.Icon;                  { copy icon }
      Result := (ShowModal = IDOK);                       { execute and set result }
    end;
  finally
    AboutBox.Free;                                          { dispose of About box }
  end;
end;
```

Testing the component

Once you've installed the dialog-box component, you can use it as you would any of the common dialog boxes, by placing one on a form and executing it. A quick way to test the About box is to add a command button to a form and execute the dialog box when the user clicks the button.

For example, if you created an About dialog box, made it a component, and added it to the Component palette, you can test it with the following steps:

1. Create a new project.
2. Place an About-box component on the main form.
3. Place a command button on the form.
4. Double-click the command button to create an empty click-event handler.
5. In the click-event handler, type the following line of code:
```
AboutBoxDlg1.Execute;
```
6. Run the application.

When the main form appears, click the command button. The About box appears with the default project icon and the name Project1. Choose OK to close the dialog box.

You can further test the component by setting the various properties of the About-box component and again running the application.

Summary

When you have a dialog box form that appears in a number of applications, you can wrap the dialog-box form into a component and install the component into the Component palette. Using the dialog box as a component ensures that on the one hand, no one accidentally changes the shared dialog box, and that on the other hand, changes in the shared dialog box show up in all the projects that use it. In addition, the dialog box is only created when used, so it does not consume memory or system resources when not in use.

Chapter

14

Building a dialog box into a DLL

When you want to use the same dialog box in multiple applications, especially when the applications aren't all Delphi applications, you can build the dialog box into a dynamic-link library (DLL). Because a DLL is a separate executable file, applications written with tools other than Delphi can call the same DLL. For example, you can call it from applications created with C++, Paradox, or dBASE.

Because the DLL is a standalone file, each DLL contains the overhead of the component library (about 100K). You can minimize this overhead by putting several dialog boxes into a single DLL. For example, suppose you have a suite of applications that all use the same dialog boxes for checking passwords, displaying shared data, or updating status information. You can put all of these dialog boxes into a single DLL, allowing them to share the component-library overhead.

Building a dialog box into a DLL takes three steps:

1. Adding the interface routine
2. Modifying the project file
3. Opening the dialog box from an application

For this example, you'll design a simple password-entry dialog box and save it in a DLL. The following figure shows the completed dialog box:

Figure 14.1 The password-entry dialog box form

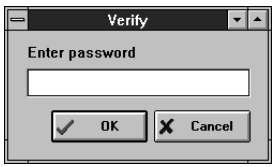


Table 14.1 describes the components that make up the password-entry form and the properties to set for them:

Table 14.1 Password-entry form component properties

Component	Property	Value
form	*Name*	PasswordForm
	BorderStyle	*bsDialog*
	Position	*poScreenCenter*
label	*Name*	Prompt
	Caption	Enter password
edit box	*Name*	PasswordField
	PasswordChar	*
bit-button	*Name*	OKButton
	Kind	*bkOK*
bit-button	*Name*	CancelButton
	Kind	*bkCancel*

When you save the project, name the password-entry form's unit file PASSFORM.PAS. Name the project file PASSWORD.PRJ.

Adding the interface routine

When wrapping a dialog box into a DLL, you need to provide an interface routine for the dialog box. This is similar to creating a wrapper component for a dialog box, but since the application calling for the dialog box might not be written with Delphi, you need to export plain procedure or function interfaces instead of components with methods and properties.

The simplest and most useful way to create the interface routine is to create a single function that takes parameters representing any properties the developer might want to set in the dialog box, returning a value describing the outcome of the dialog-box session. If the dialog box isn't going to return any information to the application, you could use a procedure instead of a function.

You add the declaration of the interface routine to the interface section of the dialog box's form unit, and implement the routine in the implementation section.

Declaring the interface routine

Adding an interface-routine declaration involves declaring a single procedure or function in the interface section of the form unit, following the declaration with the **export** directive to assure the routine is accessible to outside code.

When writing interface routines that will be called from languages other than Object Pascal, be sure to declare parameters and return values using types that are available in the calling language. For example, you should pass strings as null-terminated arrays of characters (the Object Pascal type *PChar*) rather than Object Pascal's native **string** type.

In a form unit containing a dialog-box form of type *TPasswordForm,* you would add the following declaration to the interface section:

```
unit PassForm;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Forms, Controls,
  Forms, Dialogs, StdCtrls, Buttons;
type
  TPasswordForm = class(TForm)
  ⋮                                                    { various declarations go here }
  end;

var
  PasswordForm: TPasswordForm;

function GetPassword(APassword: PChar): WordBool; export;
```

Note that the routine *must* have the **export** directive following it. That ensures that the DLL can export the routine to outside programs.

Implementing the interface routine

Once you've declared the interface routine, all that remains is to implement it. The basic outline of an interface routine looks like this:

```
function DoDialog(...): ReturnType;                      { could be a procedure instead }
begin
  Result := DefaultValue;                               { set a default function result }
  DialogBox := TDialogBoxType.Create(Application);            { construct the instance }
  try
    if DialogBox.ShowModal = IDOK then                       { execute and test result }
      { user accepted dialog box, set Result appropriately }
    else
      { user canceled or closed dialog box, react accordingly };
  finally
    DialogBox.Free;                                  { dispose of the dialog box instance }
  end;
end;
```

Here's an example that executes a password-entry dialog box and returns *True* if the password the user types matches the one passed as a parameter:

```
function GetPassword(APassword: PChar): WordBool;
begin
  Result := False;                                   { start by assuming no correct entry }
  PasswordForm := TPasswordForm.Create(Application);
  try
    if PasswordForm.ShowModal = IDOK then                 { test password if OK clicked }
      if CompareText(PasswordForm.PasswordField.Text, StrPas(APassword)) = 0 then
        Result := True
      else MessageDlg('Invalid password', mtWarning, [mbOK], 0);
  finally
    PasswordForm.Free;
  end;
end;
```

Note that before comparing the strings, you need to convert the null-terminated string into an Object Pascal string by calling the *StrPas* function.

Modifying the project file

Once you've created an interface routine for a dialog box, you're ready to modify the project file to create a DLL instead of an application. There are four simple edits you need to make in the project file:

1. Change the reserved word **program** in the first line of the file to **library**.
2. Remove the *Forms* unit from the project's **uses** clause.
3. Remove all lines of code between the **begin** and **end** at the bottom of the file.
4. Below the **uses** clause, and before the **begin..end** block, add the reserved word **exports**, followed by the name of the dialog box interface routine and a semicolon.

After these modifications, when you compile the project, it produces a DLL instead of an application. Applications can now call the DLL to open the wrapped dialog box.

Here is an example of a typical project file before modification:

```
program Password;

uses
  Forms,
  PassForm in 'PASSFORM.PAS' {PasswordForm};

{$R *.RES}
begin
  Application.CreateForm(TPasswordForm, PasswordForm);
  Application.Run;
end.
```

Here is the same file, modified to create a DLL with an interface function called *GetPassword*:

```
library Password;                                    { 1. reserved word changed }

uses                                                       { 2. removed Forms, }
  PassForm in 'PASSFORM.PAS' {PasswordForm};

exports
  GetPassword;                                       { 4. add exports clause }
{$R *.RES}
begin                                          { 3. remove code from main block }
end.
```

Opening the dialog box from an application

Once you finish wrapping your dialog box into a DLL and compile the DLL, you're ready to use the dialog box from an application.

To use a dialog box from a DLL, you need to do two things:

1 Import the interface routine from the DLL.
2 Call the interface routine.

These steps are the same when calling the DLL from any application, no matter what tools the application was created with. The following sections demonstrate briefly how to call the same function in the same DLL from different environments.

Opening the dialog box from a Delphi application

Importing an interface routine is simple: inside the unit where you need to call the routine, you repeat its declaration, appending the reserved word **external** and the name of the DLL.

For example, to import an interface routine called *GetPassword* from a DLL called PASSWORD.DLL, you add the following to the importing unit:

```
function GetPassword(APassword: PChar): WordBool; far;
external 'PASSWORD';
```

Code in the unit that imports the routine can then call the routine normally.

For example, to test the import of *GetPassword*, you could create a simple project with a single command button on the main form. After importing the interface routine for the password-entry dialog box, you could add the following click-event handler that uses the password-entry dialog box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if GetPassword('PASSWORD') then
    Color := clGreen
  else
    Color := clRed;
end;
```

If the user enters the correct password, the application turns the form green. If the user enters an incorrect password or cancels the dialog box, the form turns red.

Opening the dialog box from a Paradox application

Importing an interface routine into a Paradox for Windows application is similar to importing the routine into a Delphi application. Since Paradox does not have units, you modify the Uses window of the object that will call the routine.

For example, if a button will call the *GetPassword* function in a DLL called PASSWORD.DLL, you modify the button's Uses window to include the following:

```
Uses Password
  GetPassword(APassword CPTR) CWORD
endUses
```

You can then test the import much as you would in Delphi. Create a new form that contains a button object and import *GetPassword* from PASSWORD.DLL. You can then modify the **pushButton** method of the button to call *GetPassword*:

```
method pushButton(var eventInfo Event)
   if GetPassword("PASSWORD") <> 0 then
      #Page2.Color = Green
   else
      #Page2.Color = Red
   endIf
endmethod
```

Opening the dialog box from a dBASE application

Importing an interface routine into a dBASE for Windows application is similar to importing the routine into a Delphi application.

Here is a small dBASE program that imports the *GetPassword* function from PASSWORD.DLL:

```
LOAD DLL PASSWORD.DLL
EXTERN CLOGICAL GetPassword(CSTRING) PASSWORD.DLL
DO PWTEST.WFM
```

The file PWTEST.WFM in that program is a form that contains a single pushbutton with the following procedure:

```
Procedure PUSHBUTTON1_OnClick
IF GetPassword("PASSWORD")
    Form.ColorNormal = "G/G"
ELSE
    Form.ColorNormal = "R/R"
ENDIF
```

Opening the dialog box from a C/C++ application

Importing an interface routine into a C or C++ application takes two steps:

1. Declare the function name in the program's main source code. For example,

```
extern BOOL GetPassword(LPSTR);
```

2. Run the IMPLIB utility on PASSWORD.DLL and add the resulting .LIB file to your project. Compiling the project then automatically includes the imported function.

You can then call the imported function from your application. For example, create a C++ application with the ObjectWindows library that consists of a form that contains a button. In response to the user pressing a button, call the following method that calls *GetPassword*:

```
void TTestWindow::ButtonPressed()
{
  if (GetPassword("PASSWORD"))
    SetBkgndColor(TColor::LtGreen);
  else
    SetBkgndColor(TColor::LtRed);
  Invalidate();
}
```

Summary

By building a dialog-box form into a DLL, you make that form available to multiple applications, including applications written in languages other than Object Pascal.

You can also create DLLs with Delphi that do not include forms. For example, if you have a library of commonly-used routines, you can compile them into a DLL and make them available to various applications.

Appendix

A

Changes in Pascal objects

This appendix summarizes the changes made in the object model between previous Borland Pascal products and Delphi.

The new object model

Delphi introduces a number of changes in the definition of object types. A few of these changes are backward-compatible, and can be incorporated into existing objects. Most, however, are exclusive to new-style objects.

You can declare old-style objects in one unit, new-style objects in another, and use both of them in the same program. Keep in mind, however, that there are certain differences between the objects, and you are responsible for handling those disparities.

Note that new-style object declarations use the reserved word **class**, while old-style objects still use **object**.

The following changes are compatible with objects using the version 7.0 and earlier object model:

- Protected parts (in addition to public and private)
- Published parts

These changes apply only to new (version > 7.0) objects:

- Changes in object declarations
- Changes in object use
- Properties
- Changes in method dispatching

Changes in object declarations

There are several important additions and changes to the way you declare objects and parts of objects. The most obvious change is that you use the reserved word **class**, rather than **object**, in the declaration. Using **object** declares an old-style object, while **class** declares a new-style object.

New-style object declarations differ in several other ways, however, including the following:

- Default ancestor
- Protected parts
- Published parts
- Class methods
- Forward object declaration

Default ancestor

The *System* unit defines an abstract object type called *TObject* that is the default ancestor of all newly declared objects. Because all objects now have a common ancestor, at a very basic level, you can treat them polymorphically.

Note You need not explicitly declare *TObject* as an ancestor.

The following type declaration

```
type
  TMyObject = class
  ⋮
  end;
```

is exactly equivalent to this one:

```
type
  TMyObject = class(TObject)
  ⋮
  end;
```

TObject contains rudimentary constructors and destructors, called *Create* and *Destroy*, respectively.

- *TObject.Create* is a constructor that allocates a dynamic instance of the object on the heap and initializes all its fields to zeros.
- *TObject.Destroy* disposes of the memory allocated by *Create* and destroys the object instance.

TObject also provides several other methods, which are described in online Help.

Protected parts

In addition to public and private parts of classes, you can now declare protected parts. There is a new directive, **protected**, that operates like its counterparts, **private** and **public**, in that it is reserved only in the type declaration of a class.

The protected parts of a class and its ancestor classes can only be accessed through a class declared in the current unit.

Protection combines the advantages of public and private components. As with private components, you can hide implementation details from end users. However, unlike private components, protected components are still available to programmers who want to derive new objects from your objects without the requirement that the derived objects be declared in the same unit.

Protected methods are particularly useful in hiding the implementation of properties.

Published parts

In addition to private, protected, and public parts of objects, you can now declare published parts. There is a new directive, **published**, that operates like its counterparts, **private**, **protected**, and **public**, in that it is reserved only in the type declaration of a class.

Declaring a part of an object as published generates run-time type information for that part, including it in the application's published interface. Inside your application, a published part acts just like a public part. The only difference is that other applications can get information about those parts through the published interface.

The Delphi Object Inspector, for example, uses the published interface of objects in the component palette to determine the properties and events it displays.

Class methods

Delphi allows you to declare individual methods as *class methods*. Class methods are methods of the type, rather than of a particular instance, so you can call the method without having to construct an instance. The implementation of the class method must not depend on the run-time values of any object fields.

An example of a class method is the *NewInstance* method declared in *TObject*. *NewInstance* allocates memory for each instance of a type.

- To declare a class method, you put the reserved word **class** in front of the method declaration.

  For example,

```
type
  TMyObject = class(TObject)
    class function GetClassName: string;
  end;
```

  You can call a class method just like a normal method for an instance of that type, but you can also call the method of the type itself:

```
var
  MyObject: TMyObject;
  AString: string;
begin
  AString := TMyObject.GetClassName;
```

```
  MyObject := TMyObject.Create;
  AString := MyObject.GetClassName;
end;
```

Forward object declaration

You can now pre-declare an object type to make it available to other object-type declarations without fully defining it. This is much like a forward declaration of a procedure or function, in that you declare the symbol, but fully define it later.

- To forward-declare an object type, you do the following:

```
type
  TMyClass = class;
```

The class must then be fully defined within that same type declaration block. A typical use looks something like this:

```
type
  TMyClass = class;
  TYourClass = class(TSomething)
    MyClassField: TMyClass;
    ⋮
  end;
  TMyClass = class(TObject)
    MyField: TMyType;
    ⋮
  end;
```

Changes in object use

There are several changes in the way you use new-model objects.

- Reference model
- Method pointers
- Object-type references
- Run-time type information

Reference model

Because all new-style objects are dynamic instances (that is, allocated on the heap and accessed through a pointer), Delphi dispenses with the need to declare separate object and pointer types and explicitly dereference object pointers.

Thus, in the old object model, you might have declared types as follows:

```
type
  PMyObject = ^TMyObject;
  TMyObject = object(TObject)
    MyField: PMyObject;
    constructor Init;
  end;
```

You would then construct an instance and perhaps access its field as follows:

```
var
  MyObject: PMyObject;
begin
  MyObject := New(PMyObject, Init);
  MyObject^.MyField := ...
end;
```

Delphi greatly simplifies this process by using a reference model that automatically assumes you want to dereference the pointer:

```
type
  TMyObject = class(TObject)
    MyField: TMyObject;
    constructor Create;
  end;
var
  MyObject: TMyObject;
begin
  MyObject := TMyObject.Create;
  MyObject.MyField := ...
end;
```

Method pointers

Delphi now allows you to declare procedural types that are object methods, enabling you to call particular methods of particular object instances at run time. The main advantage to method pointers is that they let you extend an object by delegation (that is, delegating some behavior to another object), rather than by deriving a new object and overriding methods.

Delphi uses method pointers to connect events with specific code in specific places, such as calling a method of a particular form when the user clicks a button. Thus, instead of deriving a new class from *TButton* and overriding its click behavior, the programmer connects the existing object to a method of a specific object instance (generally the form containing the button) that has the desired behavior.

The only difference between a method pointer declaration and a normal procedural type declaration is the use of the reserved words **of object** following the function or procedure prototype. For example,

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
```

An object field of type *TNotifyEvent* is then assignment-compatible with any object method declared as `procedure(Sender: TObject)`. For example,

```
type
  TAnObject = class(TObject)
    FOnClick: TNotifyEvent;
  end;
  TAnotherObject = class(TObject)
    procedure AMethod(Sender: TObject);
  end;
```

```
var
  AnObject: TAnObject;
  AnotherObject: TAnotherObject;
begin
  AnObject := TAnObject.Create;
  AnotherObject := TAnotherObject.Create;
  AnObject.FOnClick := AnotherObject.AMethod;
end;
```

Object-type references

Delphi now allows you to create pointers to object types, known as object-type references. Using object-type references, you can either create instances of the type assigned to the reference or get information about the object type by calling class methods.

To declare an object-type reference, you use the reserved word **class**. For example, to create a reference to the type *TObject*,

```
type
  TObjectRef = class of TObject;
```

Once you declare a reference, you can assign to it any object type assignment-compatible with the declared type. The following example shows legal uses of an object-type reference:

```
type
  TObjectRef = class of TObject;
  TDescendant = class(TObject)
  end;
var
  ObjectRef: TObjectRef;
  AnObject: TObject;
  ADescendant: TDescendant;
begin
  AnObject := TObject.Create;
  ADescendant := TDescendant.Create;
  ObjectRef := TObject;
  {insert code to demonstrate polymorphism }
  ObjectRef := TDescendant;
  {insert code to demonstrate polymorphism }
end;
```

Run-time type information

Delphi provides access to object-type information at run time. Specifically, you can use a new operator, **is**, to determine whether a given object is of a given type or one of its descendants.

The expression

```
AnObject is TObjectType
```

evaluates as *True* if *AnObject* is assignment-compatible with objects of type *TObjectType*, meaning that AnObject is either of type *TObjectType* or of an object type descended from *TObjectType*.

You can also use run-time type information to assure safe typecasting of objects. The expression

```
AnObject as TObjectType
```

is equivalent to the typecast

```
TObjectType(AnObject)
```

except that using **as** raises an *EInvalidCast* exception if *AnObject* is not of a compatible type. The as expression is really equivalent to

```
if AnObject is TObjectType then TObjectType(AnObject)...
```

The **as** typecast is most useful as a shorthand when creating a **with..do** block that includes a typecast. For example, it is much simpler to write

```
with AnObject as TObjectType do ...
```

than the equivalent two-step process:

```
if AnObject is TObjectType then with TObjectType(AnObject) do ...
```

Properties

In addition to fields and methods, objects can also have properties. Properties look to the end user like object fields, but internally they can encapsulate methods that read or write the value of the "field."

Properties let you control access to protected fields or create side effects to changing what otherwise look like fields.

There are four main topics regarding properties:

- Property syntax
- Fields for reading and writing
- Array properties
- Read-only and write-only properties

Property syntax

For example,

```
type
  TAnObject = class(TObject)
    function GetAProperty: TypeX;
    procedure SetAProperty(ANewValue: TypeX);
    property AProperty: TypeX  read GetAProperty write SetAProperty;
  end;
```

When using objects of type *TAnObject*, you treat *AProperty* just as if it were a field of type *TypeX*. References to the property are automatically translated by the compiler into method calls.

To set the value of a property, you use an assignment statement:

```
AnObject.AProperty := AValue;
```

Internally, the compiler translates that to

```
AnObject.SetAProperty(AValue);
```

Similarly, to retrieve the value of a property, you reference the property:

```
AVariable := AnObject.AProperty;
```

Again, the compiler does an automatic translation into a method call:

```
AVariable := AnObject.GetAProperty;
```

Often you'll store the value of the property in a private or protected field, then use the read and write methods to get and set the values. If the read and write methods are virtual, you can easily override them in descendant object types to change the reading and writing behavior of the property without affecting code that uses the property.

Fields for reading and writing

You can access object fields directly to implement the read and/or write parts of a property.

For example, the read part might return the value of a field, while the write part might set the field and produce other side effects:

```
type
  TPropObject = class(TObject)
    FValue: Integer;
    procedure SetValue(NewValue: Integer);
    property AValue: Integer  read FValue write SetValue;
  end;
procedure TPropObject.SetValue(NewValue: Integer);
begin
  FValue := NewValue;
  UpdateScreen;
end;
```

Array properties

You can declare properties that look and act much like arrays, in that they have multiple values of the same type referred to by an index. Unlike an array, however, you can't refer to the property as a whole, only to individual elements in the array. These properties are called *array properties*.

There are two important aspects to array properties:

- Declaring an array property
- Accessing an array property

Declaring an array property

The declaration of an array property is identical to the declaration of any other property, but you also declare an index and a type, which also becomes a parameter to both the **read** and **write** methods.

- To declare an array property, you specify the name of the property, the name and type of the index , and the type of the elements.

  For example, to declare a property that looks like an array of strings, you might do this:

  ```
  property MyStrings[Index: Integer]: string read GetMyString write SetMyString;
  ```

  The **read** method for the property must be a function that takes a single parameter with the same name and type as the property index and returns the same type as the elements:

  ```
  function GetMyString(Index: Integer): string;
  ```

  GetMyString would return the appropriate string for any given index value. How you implement that is entirely hidden from the user. To the user, *MyStrings* really looks like an array of strings.

  The **write** method for an indexed property is a procedure that takes two parameters: the first with the same signature as the property's index, and the second of the same type as the property elements:

  ```
  procedure SetMyString(Index: Integer; const NewElement: string);
  ```

  Note that the value for the element can be passed either by value or by reference.

Accessing an array property

You use an array property in your code just as if it were an array, with the exception that you cannot access the "array" as a whole.

For example, given the following property declaration,

```
property MyStrings[Index: Integer]: string read GetMyString write SetMyString;
```

you can set or retrieve strings as follows:

```
var
  YourString: string;
begin
  YourString := MyStrings[1];
  MyStrings[2] := 'This is a string.';
end;
```

Default array properties

You can declare an array property as the default property, which means you can reference the array without using its name, treating the object itself as if it were indexed. An object can have only one default property.

- To declare a default array property, add the directive **default** after an array property:

```
type
  TMyObject = class(TObject)
    property X[Index: Integer]: Integer read GetElement; default;
  end;
```

To access the default array property, place the index next to the object identifier without specifying the property name. For example, given the above declaration, the following code accesses the default property *X* directly and again as the default property:

```
var
  MyObject: TMyObject;
begin
  MyObject.X[1] := 42;
  MyObject[2] := 1993;
end;
```

Multiple-index properties

An array property, like an array, can have more than one index. The corresponding parameters to the read and write methods must still have the same signatures as the indexes, and must appear in the same order as the indexes.

Read-only and write-only properties

You can make a property read-only or write-only by omitting the **read** or **write** portion of the property declaration. For example, to declare a read-only property, you supply only a **read** method:

```
type
  TAnObject = class(TObject)
    property AProperty: TypeX  read GetAnObject;
  end;
```

By not providing a method for setting the property, you ensure that to the component user, the property is read-only. Any attempt to write to a read-only property or read a write-only property causes a compiler error.

Changes in method dispatching

Delphi makes several changes in the way objects dispatch method calls. Previous versions of the compiler allowed static, virtual, and dynamic virtual methods. In the new object model, you can use static, virtual, dynamic, and message-handling messages.

In addition to changes in the kinds of methods you can declare, there are also changes in the implementation of abstract methods and changes in the way you override virtual and dynamic methods.

- Dynamic methods
- Message-handling methods

- Abstract methods
- Override directive
- Virtual constructors

Dynamic methods

Delphi reintroduces the idea of dynamic method dispatching. Unlike the dynamic methods used in ObjectWindows, which used a variant of the **virtual** directive followed by an integer expression, dynamic method declarations use the directive **dynamic**. The compiler assigns its own index. (It happens to be a negative number, but it's not a number you'll ever need to use.)

Dynamic methods work just like virtual methods, but instead of an entry in the class' virtual method table, the dynamic method is dispatched by using its index number. The only noticeable difference between dynamic methods and virtual methods is that dynamic-method dispatch takes somewhat longer.

Message-handling methods

In addition to dynamic methods, Delphi also has a new, specialized form of dynamic method called a message-handling method.

There are three important aspects to message-handling methods:

- Declaring message handlers
- Dispatching messages
- Calling inherited message handlers

Declaring message handlers

Message-handling methods have four distinguishing characteristics:

1. They are always procedures.
2. They are declared with the **message** directive.
3. They take an integer constant as a dynamic index, following message.
4. They take a single parameter, which must be a **var** parameter.

A typical message-handling method declaration looks like this:

```
type
  TMyControl = class(TWinControl)
    procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
  end;
```

The name of the method and the name and type of the parameter are not important. For example, a descendant type could declare an entirely different handler for the same message:

```
type
  TMyOtherControl = class(TMyControl)
    procedure PaintIt(var Info); message WM_PAINT;
  end;
```

Dispatching messages

The actual message dispatching is done by a method inherited from *TObject* called *Dispatch*.

Calling inherited message handlers

Within a message-handler method, you can call the message-handler inherited from the object's ancestor type. Because the name and parameter of the method might vary, you can call the inherited method just by using **inherited**; you don't have to include the ancestor method's name.

For example, the *WMPaint* method described above might be implemented as follows:

```
procedure TMyControl.WMPaint(var Message: TWMPaint);
begin
  with Message do
  begin
    ⋮
    inherited;
    ⋮
  end;
end;
```

The inherited method called will be the one with the same message index (in this case, *WM_PAINT*). For example, if the method declared above, *TMyOtherControl.PaintIt*, included an inherited statement, the method called would be *TMyControl.WMPaint*.

Default message handler

It is always safe to call inherited within a message-handler method. If the ancestor type does not declare a specific message handler for a particular message index, inherited calls the *TObject* method *DefaultHandler*.

Abstract methods

You can declare virtual or dynamic methods as truly abstract. In previous versions, you could declare a method and call a procedure called *Abstract*, which generated a run-time error. Delphi supports a new directive, **abstract**, which declares a method as truly abstract. That means that you don't implement the method, just declare its heading.

An abstract method declaration looks like this:

```
type
  TMyObject = class
    procedure Something; virtual; abstract;
  end;
```

You can create instances of classes that have abstract methods, but calling those methods generates a run-time error 210.

The **abstract** directive is only valid at the first introduction of a method.

Override directive

Because virtual methods now have two kinds of dispatching, VMT-based and dynamic, methods that override virtual and dynamic methods use the **override** directive instead of repeating **virtual** or **dynamic**.

For example,

```
type
  TAnObject = class
    procedure P; virtual;
  end;
  TAnotherObject = class(TAnObject)
    procedure P; override;
  end;
```

Type *TAnotherObject* could declare its *P* method with either virtual or dynamic, but those mean something different: either of those would introduce a different method *P*, replacing, rather than overriding, the inherited *P*.

Virtual constructors

New-style objects can have virtual constructors. *TObject.Create*, for instance, is not virtual, but many VCL objects have virtual constructors.

Index

N

O

P

R

S

T

U

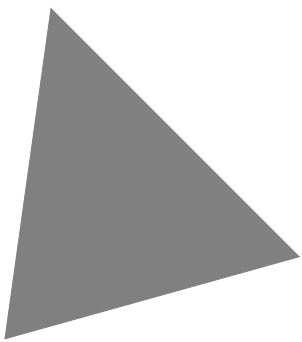
V

W

X

Y

Component Writer's Guide



Delphi™

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Printed in the U.S.A.

1E0R1294
9495969798-987654321
W1

Contents

Tables

Figures