Overview of component creation

Topic groups See also

This set of topics provides an overview of component design and the process of writing components for Delphi applications. The material here assumes that you are familiar with Delphi and its standard components.

The main topics discussed are

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

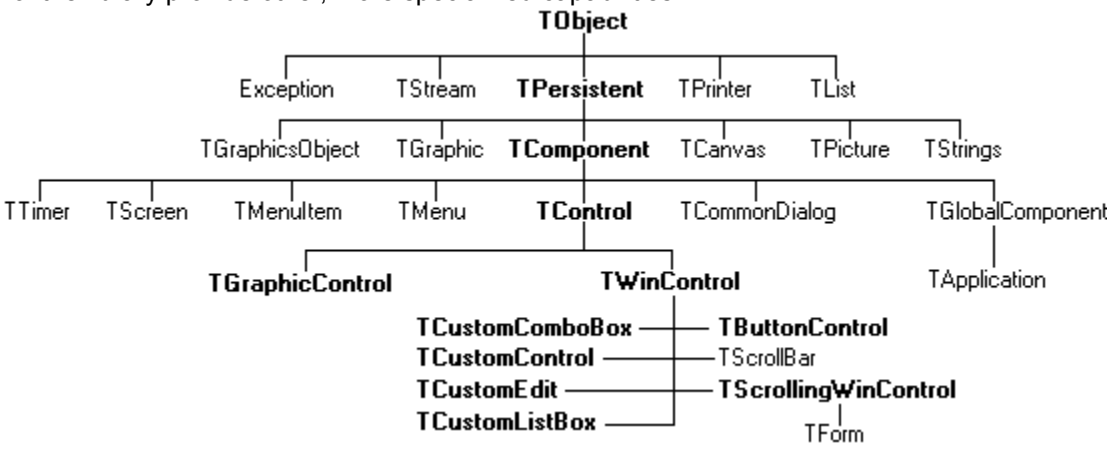
For information on installing new components, see Installing component packages.

The Visual Component Library

Topic groups

Delphi's components are all part of a class hierarchy called the Visual Component Library (VCL). The following figure shows the relationship of selected classes that make up the VCL. For a more detailed discussion of class hierarchies and the inheritance relationships among classes, see Object-oriented programming for component writers

The *TComponent* class 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.



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

Components and classes

Topic groups

Because components are classes, component writers work with objects at a different level from application developers. Creating new components requires that you derive new classes.

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

- You access parts of the class that are inaccessible to application programmers.
- You add new parts (such as properties) to your components.

Because of these differences, you need to be aware of more conventions and think about how application developers will use the components you write.

How do you create components?

Topic groups

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

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

The following table summarizes the different kinds of components and the classes you use as starting points for each.

You can also derive classes that are not components and cannot be manipulated on a form. Delphi includes many such classes, like *TRegIniFile* and *TFont*.

Modifying existing controls

Topic groups

The simplest way to create a component is to customize an existing one. You can derive a new component from any of the components provided with Delphi.

Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, the VCL includes an abstract class (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 list box that does not have some of the properties of the standard *TListBox* class. You cannot remove (hide) a property inherited from an ancestor class, so you need to derive your component from something above *TListBox* in the hierarchy. Rather than force you to start from the abstract *TWinControl* class and reinvent all the list box functions, the VCL provides *TCustomListBox*, which implements the properties of a list box but does not publish all of them. When you derive a component from an abstract class like *TCustomListBox*, you publish only the properties you want to make available in your component and leave the rest protected.

The section Creating properties explains publishing inherited properties. The section Modifying an existing component and the section Customizing a grid show examples of modifying existing controls.

Creating original controls

Topic groups

Windowed controls are objects that appear at runtime and that the user can interact with. Each windowed control has a window handle, accessed through its *Handle* property, that lets Windows identify and operate on the control. The handle allows the control to receive input focus and can be passed to Windows API functions.

All windowed controls descend from the *TWinControl* class. These include most standard Windows controls, such as pushbuttons, list boxes, and edit boxes. While you could derive an original control (one that's not related to any existing control) directly from *TWinControl*, Delphi provides the *TCustomControl* component for this purpose. *TCustomControl* is a specialized windowed control that makes it easier to draw complex visual images.

The section Customizing a grid presents an example of creating a windowed control.

Creating graphic controls

Topic groups

If your control does not need to receive input focus, you can make it a graphic control. Graphic controls are similar to windowed controls, but have no window handles, and therefore consume fewer system resources. Components like *TLabel*, which never receive input focus, are graphic controls. Although these controls cannot receive focus, you can design them to react to mouse messages.

Delphi supports the creation of custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract class derived from *TControl*. Although you can derive controls directly from *TControl*, it is better to start from *TGraphicControl*, which provides a canvas to paint on and handles WM_PAINT messages; all you need to do is override the *Paint* method.

The section Creating a graphic component presents an example of creating a graphic control.

Subclassing Windows controls

Topic groups

In traditional Windows programming, you create custom controls by defining a new *window class* and registering it with Windows. The window class (which is similar to the *objects* or *classes* in object-oriented programming) contains information shared among instances of the same sort of control; you can base a new window class on an existing class, which is called *subclassing*. You then put your control 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 window class. So if you already have a library of custom controls that you want to use in Delphi applications, you can create Delphi components that behave like your controls, and derive new controls from them just as you would with any other component.

For examples of the techniques used in subclassing Windows controls, see the components in the STDCTLS unit that represent standard Windows controls, such as *TEdit*.

Creating nonvisual components

Topic groups

Nonvisual components are used as interfaces for elements like databases (*TDataSet*) and system clocks (*TTimer*), and as placeholders for dialog boxes (*TCommonDialog* and its descendants). Most of the components you write are likely to be visual controls. Nonvisual components can be derived directly from *TComponent*, the abstract base class for all components.

What goes into a component?

Topic groups

To make your components reliable parts of the Delphi environment, you need to follow certain conventions in their design. This section discusses the following topics:

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

Removing dependencies

Topic groups

One quality that makes components usable is the absence of restrictions on what they can do at any point in their code. By their nature, components are incorporated into applications in varying combinations, orders, and contexts. You should design components that function in any situation, without preconditions.

An excellent example of removing dependencies is the *Handle* property of *TWinControl*. If you have 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 do not try to access a window or control until you have created it by calling the *CreateWindow* API function. Delphi windowed controls relieve users from this concern by ensuring that a valid window handle is always available when needed. By using a property to represent the window handle, the control can check whether the window has been created; if the handle is not valid, the control creates a window and returns the handle. Thus, whenever an application's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing background tasks like creating the window, Delphi components allow developers to focus on what they really want to do. Before passing a window handle to an API function, there is no need to verify that the handle exists or to create the window. The application developer can assume that things will work, instead of constantly checking for things that might go wrong.

Although it can take time to create components that are free of dependencies, it is generally time well spent. It not only spares application developers from repetition and drudgery, but it reduces your documentation and support burdens.

Properties, methods, and events

Topic groups

Aside from the visible image manipulated in the Form designer, the most obvious attributes of a component are its properties, events, and methods. Each of these has a section devoted to it in this file, but the discussion that follows explains some of the motivation for their use.

Properties

Properties give the application developer the illusion of setting or reading the value of a variable, while allowing the component writer to hide the underlying data structure or to implement special processing when the value is accessed.

There are several advantages to using properties:

- Properties are available at design time. The application developer can set or change initial values of properties without having to write code.
- Properties can check values or formats as the application developer assigns them. Validating input at design time prevents errors.
- The component can construct appropriate values on demand. Perhaps the most common type of error programmers make is to reference a variable that has not been initialized. By representing data with a property, you can ensure that a value is always available on demand.
- Properties allow you to hide data under a simple, consistent interface. You can alter the way information is structured in a property without making the change visible to application developers.

The section Overview of component creation explains how to add properties to your components.

Events

An event is a special property that invokes code in response to input or other activity at runtime. Events give the application developer a way to attach specific blocks of code to specific runtime occurrences, such as mouse actions and keystrokes. The code that executes when an event occurs is called an *event handler*.

Events allow application developers to specify responses to different kinds of input without defining new components.

The section Creating events explains how to implement standard events and how to define new ones.

Methods

Class methods are procedures and functions that operate on a class rather than on specific instances of the class. For example, every component's constructor method (*Create*) is a class method. Component methods are procedures and functions that operate on the component instances themselves. Application developers use methods to direct a component to perform a specific action or return a value not contained by any property.

Because they require execution of code, methods can be called only at runtime. Methods are useful for several reasons:

- Methods encapsulate the functionality of a component in the same object where the data resides.
- Methods can hide complicated procedures under a simple, consistent interface. An application developer can call a component's *AlignControls* method without knowing how the method works or how it differs from the *AlignControls* method in another component.
- Methods allow updating of several properties with a single call.

The section Creating methods explains how to add methods to your components.

Graphics encapsulation

Topic groups

Delphi simplifies Windows graphics by encapsulating various graphic tools into a canvas. The canvas represents the drawing surface of a window or control and contains other classes, 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 have written a graphical Windows application, you are familiar with the requirements imposed by Windows' graphics device interface (GDI). For example, GDI limits the number of device contexts available and requires that you restore graphic objects to their initial state before destroying them.

With Delphi, you do not have to worry about these things. To draw on a form or other component, you access the component's *Canvas* property. If you want to customize a pen or brush, you set its color or style. When you finish, Delphi disposes of the resources. Delphi also caches resources to avoid recreating them if your application frequently uses the same kinds of resource.

You still have full access to the Windows GDI, but you will often find that your code is simpler and runs faster if you use the canvas built into Delphi components. Graphics features are detailed in the section Using graphics in components.

Registration Overview

Topic groups

Before you can install your components in the Delphi IDE, you have to register them. Registration tells Delphi where you want your component to appear on the Component palette. You can also customize the way Delphi stores your components in the form file. For information on registering a component, see Registering components.

Creating a new component

Topic groups

You can create a new component two ways:

- Using the Component wizard
- Creating a component manually

You can use either of these methods to create a minimally functional component ready to install on the Component palette. After installing, you can add your new component to a form and test it at both design time and runtime. You can then add more features to the component, update the Component palette, and continue testing.

There are several basic steps that you perform whenever you create a new component. These steps are described below; other examples in this document assume that you know how to perform them.

1. Creating a unit for the new component.
2. Deriving your component from an existing component type.
3. Adding properties, methods, and events.
4. Registering your component with Delphi.
5. Creating a Help file for your component and its properties, methods, and events.
6. Creating a package (a special dynamic-link library) so that you can install your component in the Delphi IDE.

When you finish, the complete component includes these files:

- A package (.BPL) or package collection (.DPC) file
- A compiled package (.DCP) file
- A compiled unit (.DCU) file
- A palette bitmap (.DCR) file
- A Help (.HLP) file

Creating a help file to instruct component users on how to use the component is optional. Including a help file is mandatory only if one is created.

Using the Component wizard

Topic groups

The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify only these things:

- The class from which it is derived
- The class name for the new component
- The Component palette page you want it to appear on
- The name of the unit in which the component is created
- The search path where the unit is found
- The name of the package in which you want to place the component

The Component wizard performs the same tasks you would when creating a component manually:

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

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

To start the Component wizard, choose one of these two methods:

- Choose Component|New Component.
- Choose File|New and double-click on Component

Fill in the fields in the Component wizard:

1. In the Ancestor Type field, specify the class from which you are deriving your new component.
2. In the Class Name field, specify the name of your new component class.
3. In the Palette Page field, specify the page on the Component palette on which you want the new component to be installed.
4. In the Unit file name field, specify the name of the unit you want the component class declared in.
5. If the unit is not on the search path, edit the search path in the Search Path field as necessary.

To place the component in a new or existing package, click Component|Install and use the dialog box that appears to specify a package.

Warning: If you derive a component from a VCL class whose name begins with “custom” (such as *TCustomControl*), do not try to place the new component on a form until you have overridden any abstract methods in the original component. Delphi cannot create instance objects of a class that has abstract properties or methods.

To see the source code for your unit, click View Unit. (If the Component wizard is already closed, open the unit file in the Code editor by selecting File|Open.) Delphi creates a new unit containing the class declaration and the *Register* procedure, and adds a **uses** clause that includes all the standard Delphi units. The unit looks like this:

```
unit MyControl;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
type
  TMyControl = class(TCustomControl)
  private
  { Private declarations }
  protected
  { Protected declarations }
  public
  { Public declarations }
  published
  { Published declarations }
end;
procedure Register;
implementation
```

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

Creating a component manually

Topic groups

The easiest way to create a new component is to use the Component wizard. You can, however, perform the same steps manually.

To create a component manually, follow these steps:

1. Creating a unit file
2. Deriving the component
3. Registering the component

Creating a unit

Topic groups

A unit is a separately compiled module of Object Pascal code. Delphi uses units for several purposes. Every form has its own unit, and most components (or groups of related components) have their own units as well.

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, choose File|New to and double-click on Unit. Delphi creates a new unit file and opens it in the Code editor.

To open an existing unit, choose File|Open and select the source code unit that you want to add your component to.

Note: When adding a component to an existing unit, make sure that the unit contains only component code. For example, adding component code to a unit that contains a form causes errors in the Component palette.

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

Deriving the component

Topic groups Example

Every component is a class derived from *TComponent*, from one of its more specialized descendants (such as *TControl* or *TGraphicControl*), or from an existing component class. The section How do you create components? describes which class to derive different kinds of components from.

Deriving classes is explained in more detail in The section Defining new classes.

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

A simple component class is a nonvisual component descended directly from *TComponent*.

Example: Deriving a component

To create a simple component class, add the following class declaration to the **interface** part of your component unit:

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

So far the new component does nothing different from *TComponent*. You have created a framework on which to build your new component.

Registering the component

Topic groups Example

Registration is a simple process that tells Delphi which components to add to its component library, and on which pages of the Component palette they should appear. For a more detailed discussion of the registration process, see Making components available at design time

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 are adding a component to a unit that already contains components, it should already have a *Register* procedure declared, so you do not 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 are 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*.

Example: Registering a component

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

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

This *Register* procedure places *TMyControl* on the Samples page of the Component palette.

Once you register a component, you can compile it into a package (see Registering components) and install it on the Component palette.

Testing uninstalled components

Topic groups Example

You can test the runtime behavior of a component before you install it on the Component palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Component palette.

You test an uninstalled component by emulating the actions performed by Delphi when the component is selected from the palette and placed on a form.

To test an uninstalled component,

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.

   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 *Parent* property.

   Setting the *Parent* property is always the first thing to do after constructing a control. The parent is the component that contains the control visually; usually it is the form on which the control appears, but it 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 control.

Warning: If your component is not a control (that is, if *TControl* is not one of its ancestors), skip this step. If you accidentally set the form's *Parent* property (instead of the component's) to *Self*, you can cause an operating-system problem.

6. Set any other component properties as desired.

Example: Testing uninstalled components

Suppose you want to test a new component of type *TMyControl* in a unit named *MyControl*. 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, MyControl;                                { 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 }
    MyControl1: TMyControl1;                                    { 2. Add an object
field }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  MyControl1 := TMyControl.Create(Self);                    { 4. Construct the
component }
  MyControl1.Parent := Self;            { 5. Set Parent property if component is a
control }
  MyControl1.Left := 12;                                    { 6. Set other
properties... )
  ...                                                                ...continue
as needed }
end;
end.
```

Object-oriented programming for component writers

Topic groups

If you have written applications with Delphi, you know that a class contains both data and code, and that you can manipulate classes at design time and at runtime. In that sense, you've become a component user.

When you create new components, you deal with classes in ways that application developers never need to. You also try to hide the inner workings of the component from the developers who will use it. By choosing appropriate ancestors for your components, designing interfaces that expose only the properties and methods that developers need, and following the other guidelines in this chapter, you can create versatile, reusable components.

Before you start creating components, you should be familiar with these topics, which are related to object-oriented programming (OOP):

- Defining new classes
- Ancestors, descendants, and class hierarchies
- Controlling access
- Dispatching methods
- Abstract class members
- Classes and pointers

Defining new classes

Topic groups

The difference between component writers and application developers is that component writers create new classes while application developers manipulate instances of classes.

A class is essentially a type. As a programmer, you are always working with types and instances, even if you do not use that terminology. For example, you create variables of a type, such as *Integer*. Classes are usually more complex than simple data types, but they work the same way: By assigning different values to instances of the same type, you can perform different tasks.

For example, it is quite common to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of the class *TButton*, but by assigning different values to their *Caption* properties and different handlers to their *OnClick* events, you make the two instances behave differently.

Deriving new classes

Topic groups

There are two reasons to derive a new class:

- To change class defaults to avoid repetition
- To add new capabilities to a class

In either case, the goal is to create reusable objects. If you design components with reuse in mind, you can save work later on. Give your classes usable default values, but allow them to be customized.

Changing class defaults to avoid repetition

Topic groups

Most programmers try to avoid repetition. Thus, if you find yourself rewriting the same lines of code over and over, you place the code in a subroutine or function, or build a library of routines that you can use in many programs. The same reasoning holds for components. If you find yourself changing the same properties or making the same method calls, you can create a new component that does these things by default.

For example, suppose that each time you create an application, you add a dialog box to perform a particular operation. Although it is not difficult to recreate the dialog each time, it is also not necessary. You can design the dialog once, set its properties, and install a wrapper component associated with it onto the Component palette. By making the dialog into a reusable component, you not only eliminate a repetitive task, but you encourage standardization and reduce the likelihood of errors each time the dialog is recreated.

Modifying an existing component shows an example of changing a component's default properties.

Note: If you want to modify only the published properties of an existing component, or to save specific event handlers for a component or group of components, you may be able to accomplish this more easily by creating a *component template*.

Adding new capabilities to a class

Topic groups

A common reason for creating new components is to add capabilities not found in existing components. When you do this, you derive the new component from either an existing component or an abstract base class, such as *TComponent* or *TControl*.

Derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you cannot take them away; so if an existing component class contains properties that you do*not* want to include in yours, you should derive from that component's ancestor.

For example, if you want to add features to a list box, you could derive your component from *TListBox*. However, if you want to add new features but exclude some capabilities of the standard list box, you need to derive your component from *TCustomListBox*, the ancestor of *TListBox*. Then you can recreate (or make visible) only the list-box capabilities you want, and add your new features.

Customizing a grid shows an example of customizing an abstract component class.

Declaring a new component class

Topic groups Example

In addition to standard components, Delphi provides many abstract classes designed as bases for deriving new components. The How do you create components? topic shows the classes you can start from when you create your own components.

To declare a new component class, add a class declaration to the component's unit file.

Example: Declaring a new component class

Here is the declaration of a simple graphical component:

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

A finished component declaration usually includes property, event, and method declarations before the **end**. But a declaration like the one above is also valid, and provides a starting point for the addition of component features.

Ancestors and descendants

Topic groups

Application developers take for granted that every control has properties named *Top* and *Left* that determine its position on the form. To them, it may not matter that all controls inherit these properties from a common ancestor, *TControl*. When you create a component, however, you must know which class to derive it from so that it inherits the appropriate features. And you must know everything that your control inherits, so you can take advantage of inherited features without recreating them.

The class from which you derive a component is called its *immediate ancestor*. Each component inherits from its immediate ancestor, and from the immediate ancestor of its immediate ancestor, and so forth. All of the classes from which a component inherits are called its *ancestors*; the component is a *descendant* of its ancestors.

Together, all the ancestor-descendant relationships in an application constitute a hierarchy of classes. Each generation in the hierarchy contains more than its ancestors, since a class inherits everything from its ancestors, then adds new properties and methods or redefines existing ones.

If you do not specify an immediate ancestor, Delphi derives your component from the default ancestor, *TObject*. *TObject* is the ultimate ancestor of all classes in the object hierarchy.

The general rule for choosing which object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

Controlling access

Topic groups

There are five levels of *access control*—also called *visibility*—on properties, methods, and fields. Visibility determines which code can access which parts of the class. By specifying visibility, you define the *interface* to your components.

The table below shows the levels of visibility, from most restrictive to most accessible:

Visibility	Meaning	Used for
private	Accessible only to code in the unit where the class is defined.	Hiding implementation details.
protected	Accessible to code in the unit(s) where the class and its descendants are defined.	Defining the component writer's interface.
public	Accessible to all code.	Defining the runtime interface.
automated	Accessible to all code. Automation type information is generated.	OLE automation only.
published	Accessible to all code and from the Object Inspector.	Defining the design-time interface.

Declare members as **private** if you want them to be available only within the class where they are defined; declare them as **protected** if you want them to be available only within that class and its descendants. Remember, though, that if a member is available anywhere within a unit file, it is available *everywhere* in that file. Thus, if you define two classes in the same unit, the classes will be able to access each other's private methods. And if you derive a class in a different unit from its ancestor, all the classes in the new unit will be able to access the ancestor's protected methods.

Hiding implementation details

Topic groups Example

Declaring part of a class as **private** makes that part invisible to code outside the class's unit file. Within the unit that contains the declaration, code can access the part as if it were public.

Example: Hiding implementation details

Here is an example that shows how declaring a field as **private** hides it from application developers. The listing shows two form units. Each form has a handler for its *OnCreate* event which assigns a value to a private field. The compiler allows assignment to the field only in the form where it is declared.

```
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

Topic groups

Declaring part of a class as **protected** makes that part visible only to the class itself and its descendants (and to other classes that share their unit files).

You can use **protected** declarations to define a *component writer's interface* to the class. Application units do not have access to the protected parts, but derived classes do. This means that component writers can change the way a class works without making the details visible to application developers.

Defining the runtime interface

Topic groups Example

Declaring part of a class as **public** makes that part visible to any code that has access to the class as a whole.

Public parts are available at runtime to all code, so the public parts of a class define its *runtime interface*. The runtime interface is useful for items that are not meaningful or appropriate at design time, such as properties that depend on runtime input or which are read-only. Methods that you intend for application developers to call must also be public.

Example: Defining the runtime interface

Here is an example that shows two read-only properties declared as part of a component's runtime 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 TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

Defining the design-time interface

Topic groups Example

Declaring part of a class as **published** makes that part public and also generates runtime type information. Among other things, runtime type information allows the Object Inspector to access properties and events.

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

Read-only properties cannot be part of the design-time interface because the application developer cannot assign values to them directly. Read-only properties should therefore be public, rather than published.

Example: Defining the design-time interface

Here is an example of a published property called *Temperature*. 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;
```

Dispatching methods

Topic groups

Dispatch refers to the way a program determines where a method should be invoked when it encounters a method call. The code that calls a method looks like any other procedure or function call. But classes have different ways of dispatching methods.

The three types of method dispatch are

- Static
- Virtual
- Dynamic

Static methods

Topic groups Example

All methods are static unless you specify otherwise when you declare them. Static methods work like regular procedures or functions. 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 runtime, which takes somewhat longer.

A static method does not change when inherited by a descendant class. If you declare a class that includes a static method, then derive a new class from it, the derived class shares exactly the same method at the same address. This means that you cannot override static methods; a static method always does exactly the same thing no matter what class it is called in. If you declare a method in a derived class with the same name as a static method in the ancestor class, the new method simply replaces the inherited one in the derived class.

Example: Static methods

In the following code, the first component declares two static methods. The second declares two static methods with the same names that replace the methods inherited from the first component.

```
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

Topic groups

Virtual methods employ a more complicated, and more flexible, dispatch mechanism than static methods. A virtual method can be redefined in descendant classes, but still be called in the ancestor class. The address of a virtual method isn't determined at compile time; instead, the object where the method is defined looks up the address at runtime.

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

When you derive a new class from an existing one, the new class gets its own VMT, which includes all the entries from the ancestor's VMT plus any additional virtual methods declared in the new class.

Overriding methods

Topic groups Example

Overriding a method means extending or refining it, rather than replacing it. A descendant class can override any of its inherited virtual methods.

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

Overriding a method causes a compilation error if

- The method does not exist in the ancestor class.
- The ancestor's method of that name is static.
- The declarations are not otherwise identical (number and type of arguments parameters differ).

Example: Overriding methods

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

Topic groups

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 that objects consume. However, dispatching dynamic methods is somewhat slower than dispatching regular virtual methods. If a method is called frequently, or if its execution is time-critical, you should probably declare it as virtual rather than dynamic.

Objects must store the addresses of their dynamic methods. But instead of receiving entries in the virtual method table, dynamic methods are listed separately. The dynamic method list contains entries only for methods introduced or overridden by a particular class. (The virtual method table, in contrast, includes all of the object's virtual methods, both inherited and introduced.) Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, working backwards through the inheritance tree.

To make a method dynamic, add the directive **dynamic** after the method declaration.

Abstract class members

Topic groups

When a method is declared as **abstract** in an ancestor class, you must surface it (by redeclaring and implementing it) in any descendant component before you can use the new component in applications. Delphi cannot create instances of a class that contains abstract members. For more information about surfacing inherited parts of classes, see Creating properties and Creating methods.

Classes and pointers

Topic groups

Every class (and therefore every component) is really a pointer. The compiler automatically dereferences class pointers for you, so most of the time you do not need to think about this. The status of classes as pointers becomes important when you pass a class as a parameter. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references; passing a class by reference amounts to passing a reference to a reference.

Creating properties

Topic groups

Properties are the most visible parts of components. The application developer can see and manipulate them at design time and get immediate feedback as the components react in the Form designer. Well-designed properties make your components easier for others to use and easier for you to maintain.

To make the best use of properties in your components, you should understand the following:

- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining properties
- Creating array properties
- Storing and loading properties

Why create properties?

Topic groups

From the application developer's standpoint, properties look like variables. Developers can set or read the values of properties as if they were fields. (About the only thing you can do with a variable that you cannot do with a property is pass it as a **var** parameter.)

Properties provide more power than simple fields because

- Application developers can set properties at design time. Unlike methods, which are available only at runtime, properties let the developer customize components before running an application. Properties can appear in the Object Inspector, which simplifies the programmer's job; instead of handling several parameters to construct an object, you let Delphi read the values from the Object Inspector. The Object Inspector also validates property assignments as soon as they are made.
- Properties can hide implementation details. For example, data stored internally in an encrypted form can appear unencrypted as the value of a property; although the value is a simple number, the component may look up the value in a database or perform complex calculations to arrive at it. Properties let you attach complex effects to outwardly simple assignments; what looks like an assignment to a field can be a call to a method which implements elaborate processing.
- Properties can be virtual. Hence, what looks like a single property to an application developer may be implemented differently in different components.

A simple example is the *Top* property of all controls. Assigning a new value to *Top* does not just change a stored value; it repositions and repaints the control. And the effects of setting a property need not be limited to an individual component; for example, setting the *Down* property of a speed button to *True* sets *Down* property of all other speed buttons in its group to *False*.

Types of properties

Topic groups

A property can be of any type. Different types are displayed differently in the Object Inspector, which validates property assignments as they are made at design time.

Property type	Object Inspector treatment
Simple	Numeric, character, and string properties appear as numbers, characters, and strings. The application developer can edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) appear as editable strings. The developer can also cycle through the possible values by double-clicking the value column, and there is a drop-down list that shows all possible values.
Set	Properties of set types appear as sets. By double-clicking on the property, the developer can expand the set and treat each element as a Boolean value (true if it is included in the set).
Object	Properties that are themselves classes often have their own property editors, specified in the component's registration procedure. If the class held by a property has its own published properties, the Object Inspector lets the developer to expand the list (by double-clicking) to include these 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 them. You can specify a property editor when you register your components.

Publishing inherited properties

Topic groups Example

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

To make a protected or public property available at design time in the Object Inspector, you must redeclare the property as published. Redeclaring means adding a declaration for the inherited property to the declaration of the descendant class.

Example: Publishing an inherited property

If you derive a component from *TWinControl*, for example, it inherits the protected *DockSite* property. By redeclaring *DockSite* in your new component, you can change the level of protection to either public or published.

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

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

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

Redeclarations can make a property less restricted, but not more restricted. Thus you can make a protected property public, but you cannot hide a public property by redeclaring it as protected.

Defining component properties

Topic groups

This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include

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

The property declaration

Topic groups Example

A property is declared in the declaration of its component class. To declare a property, you specify three things:

- The name of the property.
- The type of the property.
- The methods used to read and write the value of the property. If no write method is declared, the property is read-only.

Properties declared in a **published** section of the component's class declaration are editable in the Object Inspector at design time. The value of a published property is saved with the component in the form file. Properties declared in a **public** section are available at runtime and can be read or set in program code.

Example: Property declaration

Here is a typical declaration for a property called *Count*.

```
type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;                        { used for internal storage }
    procedure SetCount (Value: Integer);    { write method }
  public
    property Count: Integer read FCount write SetCount;
  end;
```

Internal data storage (properties)

Topic groups

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

- Property data is stored in class fields.
- The fields used to store property data are private and should be accessed only from within the component itself. Derived components should use the inherited property; they do not need direct access to the property's internal data storage.
- Identifiers for these fields consist of the letter F followed by the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a field called *FWidth*.

The principle that underlies these conventions is that only the implementation methods for a property should access the data behind it. 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 derived components.

Direct access

Topic groups Example

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 you want to make a property available in the Object Inspector but changes to its value trigger no immediate processing.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part. This allows the status of the component to be updated when the property value changes.

Example: A property that uses direct access

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

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

Access methods (properties)

Topic groups Example

You can specify an access method instead of a field in the **read** and **write** parts of a property declaration. Access methods should be protected, and are usually declared as **virtual**; this allows descendant components to override the property's implementation.

Avoid making access methods public. Keeping them protected ensures that application developers do not inadvertently modify a property by calling one of these methods.

Example: Property access methods

Here is a class that declares three properties using the index specifier, which allows all three properties to have the same read and write access methods:

```
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;
  private
    function GetDateElement(Index: Integer): Integer; { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  ...
```

Because each element of the date (day, month, and year) is an int, and because setting each requires encoding the date when set, the code avoids duplication by sharing the read and write methods for all three properties. You need only one method to read a date element, and another to write the date element.

Here is the read method that obtains the date element:

```
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;
```

This is the write method that sets the appropriate date element:

```
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;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);        { encode the modified date }
    Refresh;                                         { update the visible calendar }
  end;
end;
```

The read method

Topic groups

The read method for a property is a function that takes no parameters (except as noted below) 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 called *Count* would be *GetCount*. The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

The only exceptions to the no-parameters rule are for array properties and properties that use index specifiers (see Creating array properties), both of which pass their index values as parameters. (Use index specifiers to create a single read method that is shared by several properties. For more information about index specifiers, see the Object Pascal Language Guide.)

If you do not declare a read method, the property is write-only. Write-only properties are seldom used.

The write method

Topic groups

The write method for a property is a procedure that takes a single parameter (except as noted below) 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 write method's name is *Set* followed by the name of the property. For example, the write method for a property called *Count* would be *SetCount*. The value passed in the parameter becomes the new value of the property; the write method must perform any manipulation needed to put the appropriate data in the property's internal storage.

The only exceptions to the single-parameter rule are for array properties and properties that use index specifiers, both of which pass their index values as a second parameter. (Use index specifiers to create a single write method that is shared by several properties. For more information about index specifiers, see the Object Pascal Language Guide.)

If you do not declare a write method, the property is read-only. For a published property to be usable at design time, it must be defined as read/write.

Write methods commonly test whether a new value differs from the current value before changing the property. For example, here is 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

Topic groups

When you declare a property, you can specify a *default value* for it. Delphi uses the default value to determine whether to store the property in a form file. If you do not specify a default value for a property, Delphi always stores the property.

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

```
property Cool Boolean read GetCool write SetCool default True;
```

Note: Declaring a default value does not set the property to that value. The component's constructor method should initialize property values when appropriate. However, since objects always initialize their fields to 0, it is not strictly necessary for the constructor to set integer properties to 0, string properties to null, or Boolean properties to *False*.

Specifying no default value

Topic groups Example

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. For example,

```
property FavoriteFlavor string nodefault;
```

When you declare a property for the first time, there is no need to include **nodefault**. The absence of a declared default value means that there is no default.

Example: A property with a default value

Here is the declaration of a component that includes a single Boolean property called *IsTrue* with a default value of *True*. Below the declaration (in the **implementation** section of the unit) is the constructor that initializes the property.

```
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);               { call the inherited constructor }
  FIsTrue := True;                        { set the default value }
end;
```

Creating array properties

Topic groups Example

Some properties lend themselves to being indexed like arrays. For example, the *Lines* property of *TMemo* is an indexed list of the strings that make up the text of the memo; you can treat it as an array of strings. *Lines* provides natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties are declared like other properties, except that

- The declaration includes one or more indexes with specified types. The indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, must be methods. They cannot be fields.

The read and write methods for an array property take additional parameters that correspond to the indexes. The parameters must be in the same order and of the same type as the indexes specified in the declaration.

There are a few important differences between array properties and arrays. Unlike the index of an array, the index of 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 reference only individual elements of an array property, not the entire range of the property.

Example: Array property

The following example shows 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;
```

Storing and loading properties

Topic groups

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 developers 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 will not need to do anything to make your components work with form files because the ability to store a representation and load from it are part of the inherited behavior of components. Sometimes, however, you might want to alter the way a component stores itself or the way it initializes when loaded; so you should understand the underlying mechanism.

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

- Using the store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading
- Storing and loading unpublished properties

Using the store-and-load mechanism

Topic groups

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, setting 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

Topic groups Example

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

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 constructor assigns the necessary value. A property whose value is not set by a component's constructor assumes a zero value—that is, whatever value the property assumes when its storage memory is set to 0. Thus numeric values default to 0, Boolean values to *False*, pointers to **nil**, and so on. If there is any doubt, assign a value in the constructor method.

Example: Specifying a default 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

Topic groups Example

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

Example: Stored properties

The following code 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

Topic groups

After a component reads all its property values from its stored description, it calls a virtual method named *Loaded*, which performs any required initializations. The call to *Loaded* occurs before the form and its controls are shown, so you do not 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 to do in any *Loaded* method is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you initialize 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;                                    { call the inherited method
first}
  try
    if FStreamedConnected then Open                          { reestablish
connections }
    else CheckSessionName(False);
  except
    if csDesigning in ComponentState then                          { at design
time... }
      Application.HandleException(Self)                { let Delphi handle the
exception }
    else raise;                                                  { otherwise,
reraise }
  end;
end;
```

Storing and loading unpublished properties

Topic groups

By default, only published properties are loaded and saved with a component. However, it is possible to load and save unpublished properties. This allows you to have persistent properties that do not appear in the Object Inspector. It also allows components to store and load property values that Delphi does not know how to read or write because the value of the property is too complex. For example, the *TStrings* object can't rely on Delphi's automatic behavior to store and load the strings it represents and must use the following mechanism.

You can save unpublished properties by adding code that tells Delphi how to load and save your property's value.

To write your own code to load and save properties, use the following steps:

1. Create methods to store and load the property value.
2. Override the *DefineProperties* method, passing those methods to a filer object.

Creating methods to store and load property values

Topic groups

To store and load unpublished properties, you must first create a method to store your property value and another to load your property value. You have two choices:

- Create a method of type *TWriterProc* to store your property value and a method of type *TReaderProc* to load your property value. This approach lets you take advantage of Delphi's built-in capabilities for saving and loading simple types. If your property value is built out of types that Delphi knows how to save and load, use this approach.
- Create two methods of type *TStreamProc*, one to store and one to load your property's value. *TStreamProc* takes a stream as an argument, and you can use the stream's methods to write and read your property values.

For example, consider a property that represents a component that is created at runtime. Delphi knows how to write this value, but does not do so automatically because the component is not created in the form designer. Because the streaming system can already load and save components, you can use the first approach. The following methods load and store the dynamically created component that is the value of a property named *MyCompProperty*:

```
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
  ifReader.ReadBoolean then
    MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
  Writer.WriteBoolean(MyCompProperty <> nil);
  if MyCompProperty <> nil then
    Writer.WriteComponent(MyCompProperty);
end;
```

Overriding the DefineProperties method

Topic groups

Once you have created methods to store and load your property value, you can override the component's *DefineProperties* method. Delphi calls this method when it loads or stores the component. In the *DefineProperties* method, you must call the *DefineProperty* method or the *DefineBinaryProperty* method of the current filer, passing it the method to use for loading or saving your property value. If your load and store methods are of type *TWriterProc* and type *TReaderProc*, then you call the filer's *DefineProperty* method. If you created methods of type *TStreamProc*, call *DefineBinaryProperty* instead.

No matter which method you use to define the property, you pass it the methods that store and load your property value as well as a boolean value indicating whether the property value needs to be written. If the value can be inherited or has a default value, you do not need to write it.

For example, given the *LoadCompProperty* method of type *TReaderProc* and the *StoreCompProperty* method of type *TWriterProc*, you would override *DefineProperties* as follows:

```
procedure TSampleComponent.DefineProperties(Filer: TFiler);
  functionDoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
    begin
      if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
        Result := MyCompProperty <> nil
      else if MyCompProperty = nil or
         TSampleComponent(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name
then
        Result := True
      else Result := False;
    end
    else { no inherited value -- check for default (nil) value }
      Result := MyCompProperty <> nil;
  end;
begin
  inherited; { allow base classes to define properties }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty,
DoWrite);
end;
```

Creating events

Topic groups

An event is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the application developer. Events let application developers customize the behavior of components without having to change the classes themselves. This is known as *delegation*.

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

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

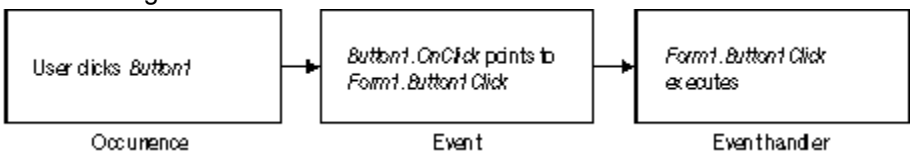
Events are implemented as properties, so you should already be familiar with the material in Creating properties before you attempt to create or change a component's events.

What are events?

Topic groups

An event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer that points to a method in a specific class instance.

From the application developer's perspective, an event is just a name related to a system occurrence, such as *OnClick*, to which specific code can be attached. 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 in the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.To write an event, you need to understand the following:



- Events are method pointers.
- Events are properties.
- Event types are method-pointer types
- Event-handler types are procedures
- Event handlers are optional.

Events are method pointers

Topic groups

Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in an instance object. As a component writer, you can treat the method pointer as a placeholder: When your code detects that an event occurs, 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. When the application developer assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a method in a specific instance object. That object is usually the form that contains the component, but it need not be.

Calling the click-event handler

Topic groups

All controls, for example, inherit a dynamic method called *Click* for handling click events:

```
procedure Click; dynamic;
```

The implementation of *Click* calls the user's click-event handler, if one exists. 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

Topic groups

Components use properties to implement their events. Unlike most other properties, events do not use methods to implement their read and write parts. Instead, event properties use a private class field of the same type as the property.

By convention, the field's name is the name of the property 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 the next section, Event types are method-pointer types.

As with any other property, you can set or change the value of an event at runtime. 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 types are method-pointer types

Topic groups

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 is appropriate, or define one of your own.

Event handler types are procedures

Topic groups

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 should be procedures.

Although an event handler cannot be a function, you can still get information from the application developer'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 OnKeyPress 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 may 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

Topic groups

When creating events, remember that developers using your components may not attach handlers to them. This means that your component should not fail or generate errors simply because there is no handler attached to a particular event. (The mechanics of calling handlers and dealing with events that have no attached handler are explained in Calling the event.)

Events happen almost constantly in a Windows application. Just moving the mouse pointer across a visual component makes Windows send numerous mouse-move messages, which the component translates into *OnMouseMove* events. In most cases, developers do not want to handle the mouse-move events, and this should not cause a problem. So the components you create should not require handlers for their events.

Moreover, application developers can write any code they want in an event handler. The components in the VCL have events written in such a way as to minimize the chance of an event handler generating errors. Obviously, you cannot protect against logic errors in application code, but you can ensure that data structures are initialized before calling events so that application developers do not try to access invalid data.

Implementing the standard events

Topic groups

The controls that come with Delphi inherit events for the most common Windows occurrences. These are called the *standard events*. Although all these events are built into the controls, they are often **protected**, meaning developers cannot 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:

- Identifying standard events
- Making events visible
- Changing the standard event handling

Identifying standard events

Topic groups

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

Standard events for all controls

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

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

The standard events have corresponding protected virtual methods declared in *TControl*, with names that correspond to the event names. For example, *OnClick* events call a method named *Click*, and *OnEndDrag* events call a method named *DoEndDrag*.

Standard events for standard controls

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

OnEnter	*OnKeyDown*	*OnKeyPress*
OnKeyUp	*OnExit*	

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

Making events visible

Topic groups Example

The declarations of the standard events in *TControl* and *TWinControl* are protected, as are the methods that correspond to them. If you are inheriting from one of these abstract classes and want to make their events accessible at runtime or design time, you need to redeclare the events as either public or published.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that is defined in *TControl* but not made visible, and surface it by declaring it as public or published.

Example: Making an event visible

For example, to create a component that surfaces the *OnClick* event at design time, you would add the following to the component's class declaration.

```
type
  TMyControl = class(TCustomControl)
  ...
  published
    property OnClick;
  end;
```

Changing the standard event handling

Topic groups Example

If you want to change the way your component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As an application developer, that is exactly what you would do. But when you are creating a component, you must keep the event available for developers who use the component.

This is 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 application developer's code.

The order in which you call the methods is significant. As a rule, call the inherited method first, allowing the application developer's event-handler to execute before your customizations (and in some cases, to keep the customizations from executing). There may be times when you want to execute your code before calling the inherited method, however. 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 them.

Example: Changing the standard event handler

Suppose you are writing a component and you want to modify the way it responds to mouse clicks. Instead of assigning a handler to the *OnClick* event as a application developer would, you override the protected method *Click*:

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

Defining your own events

Topic groups

Defining entirely new events is relatively unusual. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you will need to define an event for it.

There are the issues you will need to consider when defining an event:

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

Triggering the event

Topic groups

You need to know 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 occurrences. For example, a scroll bar has an *OnChange* event, which is triggered by several kinds of occurrence, including keystrokes, mouse clicks, and changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the proper events.

Two kinds of events

Topic groups

There are two kinds of occurrence you might need to provide events for: user interactions and state changes. User-interaction events are nearly always triggered by a message from Windows, indicating that the user did something your component may need to respond to. State-change events may also be related to messages from Windows (focus changes or enabling, for example), but they can also occur through changes in properties or other code. You have total control over the triggering of the events you define. Define the events with care so that developers are able to understand and use them.

Defining the handler type

Topic groups

Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for events you define yourself are either simple notifications or event-specific types. It is 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. 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 is not enough to know which event happened and what component it happened to. For example, if the event is a key-press event, it is likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters for additional information.

If your event was generated in response to a message, it is 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 by reference the value of the key pressed in a 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 force typed characters to uppercase, for example.

Declaring the event

Topic groups

Once you have determined the type of your event handler, you are 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 most events in Delphi begin with “On.” This is just a convention; the compiler does not 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.

Developers expect to find events in the alphabetical list of names starting with “On.” Using other kinds of names is likely to confuse them.

Calling the event

Topic groups

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

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

There are two other considerations when calling the event:

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

HeadingTitle

Topic groups Example

You should never create a situation in which an empty event handler causes an error, nor should the proper functioning of your component depend on a particular response from the application's event-handling code.

Example: Calling an event handler

An empty handler should produce the same result as no handler at all. So the code for calling an application's event 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

Topic groups Example

For some kinds of events, developers may want to replace the default handling or even suppress all responses. To allow this, you need to pass an argument by reference to the handler and check for a certain value when the handler returns.

This is in keeping with the rule that an empty handler should have the same effect as no handler at all. Because an empty handler will not change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

Example: Overriding default event handling

When handling key-press events, for example, application developers 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 this looks like

```
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 deals 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.

Creating methods

Topic groups

Component methods are procedures and functions built into the structure of a class. 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. These guidelines include

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

In general, components should not contain many methods and you should minimize the number of methods that an application needs to call. The features you might be inclined to implement as methods are often better encapsulated into properties. Properties provide an interface that suits the Delphi environment and are accessible at design time.

Avoiding interdependencies

Topic groups

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

This list gives you an idea of the kinds of dependencies to avoid:

- Methods that the user *must* call to use the component
- Methods that must 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 second method so that if an application calls it when the component is in a bad state, the method corrects the 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 problems. A warning message, for example, is preferable to a system failure if the user does not accommodate your dependencies.

Naming methods

Topic groups

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

If you are accustomed to writing code that only you or a small group of programmers use, 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. Use meaningful verbs.

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

Topic groups

All parts of classes, including fields, methods, and properties, have a level of protection or “visibility,” as explained in Controlling access. Choosing the appropriate visibility for a method is simple.

Most methods you write in your components are **public** or **protected**. You rarely need to make a method **private**, unless it is truly specific to that type of component, to the point that even derived components should not have access to it.

Methods that should be public

Topic groups

Any method that application developers need to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid tying up system resources or putting Windows in a state where it cannot respond to the user.

Note: Constructors and destructors should always be **public**.

Methods that should be protected

Topic groups

Any implementation methods for the component should be **protected** so that applications cannot call them at the wrong time. If you have methods that application code should not call, but that are called in derived classes, declare them 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 is a chance that applications will call it before setting up the data. On the other hand, by making it **protected**, you ensure that applications cannot call it directly. You can then set up other, **public** methods that ensure that data setup occurs before calling the **protected** method.

Property-implementation methods should be declared as virtual **protected** methods. Methods that are so declared allow the application developers to override the property implementation, either augmenting its functionality or replacing it completely. Such properties are fully polymorphic. Keeping access methods **protected** ensures that developers do not accidentally call them, inadvertently modifying a property.

Abstract methods

Topic groups

Sometimes a method is declared as **abstract** in a Delphi component. In the VCL, abstract methods usually occur in classes whose names begin with “custom”, such as *TCustomGrid*. Such classes are themselves abstract, in the sense that they are intended only for deriving descendant classes.

While you can create an instance object of a class that contains an abstract member, it is not recommended. Calling the abstract member leads to an *EAbstractError* exception.

The **abstract** directive is used to indicate parts of classes that should be surfaced and defined in descendant components; it forces Component writers to redeclare the abstract member in descendant classes before actual instances of the class can be created.

Making methods virtual

Topic groups

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 application developers, you can probably make all your methods nonvirtual. On the other hand, if you create abstract components from which other components will be derived, consider making the added methods **virtual**. This way, derived components can override the inherited **virtual** methods.

Declaring methods

Topic groups Example

Declaring a method in a component is the same as declaring any class 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.

Example: Declaring methods

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;
```

Using graphics in components

Topic groups

Windows provides a powerful Graphics Device Interface (GDI) for drawing device-independent graphics. The GDI, however, imposes extra requirements on the programmer, such as managing graphic resources. Delphi takes care of all the GDI drudgery, allowing you to focus on productive work instead of searching for lost handles or unreleased resources.

As with any part of the Windows API, you can call GDI functions directly from your Delphi application. But you will probably find that using Delphi's encapsulation of the graphic functions is faster and easier.

The topics in this section include

- Overview of graphics
- Using the canvas
- Working with pictures
- Off-screen bitmaps
- Responding to changes

Overview of graphics

Topic groups Example

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, brushes, and fonts. After rendering your graphic images, you must restore the device context to its original state before disposing of it.

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

The canvas manages all these resources for you, so you need not concern yourself with creating, selecting, and releasing things like 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 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 uses an existing one.

An example of this is an application that has dozens of forms open, with hundreds of controls. Each of these controls might have one or more *TFont* properties. Though this could result in hundreds or thousands of instances of *TFont* objects, most applications wind up using only two or three font handles thanks to the VCL font cache.

Example: Simplified graphics

Here are two examples of how simple Delphi's graphics code can be. 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

Topic groups

The canvas class encapsulates Windows graphics at several levels, including high-level functions for drawing individual lines, shapes, and text; intermediate properties for manipulating the drawing capabilities of the canvas; and low-level access to the Windows GDI.

The following table summarizes the capabilities of the canvas.

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

Working with pictures

Topic groups

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 stand-alone graphic images, such as bitmaps, metafiles, and icons, including automatic management of palettes.

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

- Using a picture, graphic, or canvas
- Loading and storing graphics
- Handling palettes

Using a picture, graphic, or canvas

Topic groups

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

- A *canvas* represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a stand-alone class.
- A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. Delphi defines classes *TBitmap*, *TIcon*, and *TMetafile*, all descended from a generic *TGraphic*. You can also define your own graphic classes. 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* is a container for a graphic, meaning it could contain any of the graphic classes. 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 class. 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 class 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 class exposed through *TPicture*. If you need access to the specifics of the graphic class itself, you can refer to the picture's *Graphic* property.

Loading and storing graphics

Topic groups Example

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.

Example: Loading a bitmap

To load a bitmap into an image control's picture, 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 recognizes .BMP as the standard extension for bitmap files, so it creates its graphic as a *TBitmap*, then calls that graphic's *LoadFromFile* method. Because the graphic is a bitmap, it loads the image from the file as a bitmap.

Handling palettes

Topic groups

When running on a palette-based device (typically, a 256-color video mode), 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, but controls that contain “rich color” 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 foremost 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. If you have a palette handle, however, Delphi controls can manage it for you.

Specifying a palette for a control

Topic groups

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 these things for your application:

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

Responding to palette changes

Topic groups

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*.

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 further, 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

Topic groups

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

The bitmap class in Delphi, which represents bitmapped images in resources and files, can also work as an off-screen image.

There are two main aspects to working with off-screen bitmaps:

- Creating and managing off-screen bitmaps.
- Copying bitmapped images.

Creating and managing off-screen bitmaps

Topic groups

When creating complex graphic images, you should 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 off-screen 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 off-screen
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;
```

Copying bitmapped images

Topic groups

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.

The following table summarizes the image-copying methods in canvas objects.

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

Responding to changes

Topic groups Example

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 them 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 class's *OnChange* event.

Example: Responding to changes

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();                                        { erase and repaint the
component }
end;
```

Handling messages

Topic groups

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. It is possible, however, that you will need to handle messages that Delphi does not already handle or that you will create your own messages.

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

Topic groups

All Delphi classes have a built-in mechanism for handling messages, called *message-handling methods* or *message handlers*. The basic idea of message handlers is that the class 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 Visual Component Library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular class into method calls. You should never need to alter this message-dispatch mechanism. All you will need to do is create message-handling methods. See the section Declaring a new message-handling method for more on this subject.

What's in a Windows message?

Topic groups

A Windows message is a data record that contains several fields. The most important of these is an integer-size value that identifies the message. Windows defines many messages, and the *Messages* unit declares identifiers for all of them. 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 in the Windows API 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 *nVirtKey* and *lKeyData*, which gives much more specific information than *wParam* and *lParam*.

For each type of message, Delphi defines a record type 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 structure, you do not have to worry about which word is which, because you refer to the parameters by the names *XPos* and *YPos* instead of *lParamLo* and *lParamHi*.

Dispatching messages

Topic groups

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:

- Each component inherits a complete message-dispatching system.
- The dispatch system has default handling. You define handlers only 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 does not 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 structure from Windows to a virtual method called *WndProc* and handling any exceptions by calling the application class’s *HandleException* method.

MainWndProc is a nonvirtual 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 the *WndProc* method of *TWinControl* passes along keyboard events only if the component is not being dragged. Ultimately, *WndProc* calls *Dispatch*, a nonvirtual method inherited from *TObject*, which determines which method to call to handle the message.

Dispatch uses the *Msg* field of the message structure 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 does not define a handler for that message, *Dispatch* calls *DefaultHandler*.

Changing message handling

Topic groups

Before changing the message handling of your components, make sure that is 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 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

Topic groups Example

To change the way a component handles a particular message, you override the message-handling method for that message. If the component does not 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. For clarity, however, it is best to follow the convention of naming message-handling methods after the messages they handle.

Example: Overriding a message handler

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

Topic groups

Once inside a message-handling method, your component has access to all the parameters of the message structure. Because the parameter passed to the message handler is a **var** parameter, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the *Result* field for the message: the value returned by the *SendMessage* call that sends the message.

Because the type of the *Message* parameter in 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

Topic groups

Under some circumstances, you might want your components to ignore messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message, 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. An override of *WndProc* for a control derived from *TWinControl* looks like this:

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

The *TControl* component 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.

The WndProc method

Topic groups

Here is part of the *WndProc* method for *TControl*, for example:

```
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

Topic groups

Because 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

1. Declaring a message identifier.
2. Declaring a message-record type.

Declaring a message identifier

Topic groups Example

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_APP represents the starting number for user-defined messages. When defining message identifiers, you should base them on WM_APP.

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 which messages Windows already defines for that control.

Example: User-defined messages

The following code shows two user-defined messages.

```
const
  WM_MYFIRSTMESSAGE = WM_APP + 400;
  WM_MYSECONDMESSAGE = WM_APP + 401;
```

Declaring a message-structure type

Topic groups Example

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 do not 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* parameter, and the next two bytes as unused.

   Or

   Define the next four bytes to correspond to the *Longint* parameter.
4. Add a final field called *Result*, of type *Longint*.

Example: Message structure

For example, here is the message record for all mouse messages, *TWMMouse*, which uses a variant record to define two sets of names for the same parameters.

```
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;
```

Declaring a new message-handling method

Topic groups Example

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

- Your component needs to handle a Windows message that is not 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. Within the message method implementation, write code for any handling specific to the component.
6. Call the inherited message handler.

Example: Message handler

Here is the declaration of a message handler for a user-defined message called CM_CHANGECOLOR.

```
const
  CM_CHANGECOLOR = WM_APP + 400;
type
  TMyComponent = class(TControl)
  ...
protected
  procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
end;
procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;
  inherited;
end;
```

Making components available at design time

Topic groups

This chapter describes the steps for making the components you create available in the IDE. Making your components available at design time requires several steps:

- Registering components
- Adding palette bitmaps
- Providing Help for your component
- Adding property editors
- Adding component editors
- Compiling components into packages

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 steps that are always necessary are registration and compilation.

Once your components have been registered and compiled into packages, they can be distributed to other developers and installed in the IDE. For information on installing packages in the IDE, see Installing component packages.

Registering components

Topic groups

Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you can register them all at once.

To register a component, add a *Register* procedure to the unit. Within the *Register* procedure, you register the components and determine where to install them on the Component palette.

Note: If you create your component by choosing Component|New Component in the IDE, the code required to register your component is added automatically.

The steps for manually registering a component are:

- Declaring the Register procedure
- Writing the Register procedure

Declaring the register procedure

Topic groups

Registration involves writing a single procedure in the unit, which must have the name *Register*. The *Register* procedure must appear in the interface part of the unit.

The following code shows the outline of a simple unit that creates and registers new components:

```
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.
```

Within the *Register* procedure, call *RegisterComponents* for each component you want to add to the Component palette. If the unit contains several components, you can register them all in one step.

Writing the Register function

Topic groups

Inside the *Register* procedure of 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 at the same time.

To register a component, call the *RegisterComponents* procedure once for each page of the Component palette to which you want to add components. *RegisterComponents* involves three important things:

1. Specifying the components
2. Specifying the palette page
3. Using the RegisterComponents function

Specifying the components

Topic groups

Within the Register procedure, pass the component names in an open array, which you can construct inside the call to RegisterComponents.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

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]);                         { ...one on
another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);    { ...and one on the
Standard page }
end;
```

Specifying the palette page

Topic groups

The palette-page name is a string. If the name you give for the palette page does not already exist, Delphi creates a new page with that name. Delphi stores the names of the standard pages in string-list resources so that international versions of the product can name the pages in their native languages. If you want to install a component on one of the standard pages, you should obtain the string for the page name by calling the *LoadStr* function, passing the constant representing the string resource for that page, such as *srSystem* for the System page.

Using the RegisterComponents function

Topic groups

Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a function that takes three parameters: the name of a Component palette page, the array of component classes, and the index of the last entry in the array.

Set the Page parameter to the name of the page on the component palette where the components should appear. If the named page already exists, the components are added to that page. If the named page does not exist, Delphi creates a new palette page with that name.

Call RegisterComponents from the implementation of the Register procedure in one of the units that defines the custom components. The units that define the components must then be compiled into a package and the package must be installed before the custom components are added to the component palette.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);                    {add to
system page}
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);
{ new page}
end;
```

Adding palette bitmaps

Topic groups

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.

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

For each component 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 files, so Delphi can find the bitmaps when it installs the components on the Component 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 for your component

Topic groups

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

You can provide a small Help file to describe your components, and your help file becomes part of the user's overall Delphi Help system.

See the section Creating the Help file for information on how to compose the help file for use with a component.

Creating the help file

Topic groups

You can use any tool you want to create the source file for a Windows Help file (in .rtf format). Delphi includes the Microsoft Help Workshop, which compiles your Help files and provides an online help authoring guide. You can find complete information about creating Help files in the online guide for Help Workshop.

Composing help files for components consists of the steps:

- Creating the entries
- Making component help context-sensitive
- Adding component help files

Creating the entries

Topic groups

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

1. **Each component should have a help topic.**

   The component topic should show which unit the component is declared in and briefly describe the component. The component topic should link to secondary windows that describe the component's position in the object hierarchy and list all of its properties, events, and methods. Application developers access this topic by selecting the component on a form and pressing F1. For an example of a component topic, place any component on a form and press F1.

   The component topic must have a # footnote with a value unique to the topic. The # footnote uniquely identifies each topic by the Help system.

   The component topic should have a K footnote for keyword searching in the help system Index that includes the name of the component class. For example, the keyword footnote for the *TMemo* component is "TMemo."

   The component topic should also have a $ footnote that provides the title of the topic. The title appears in the Topics Found dialog box, the Bookmark dialog box, and the History window.

2. **Each component should include the following secondary navigational topics:**

   - A hierarchy topic with links to every ancestor of the component in the component hierarchy.
   - A list of all properties available in the component, with links to entries describing those properties.
   - A list of all events available in the component, with links to entries describing those events.
   - A list of methods available in the component, with links to entries describing those methods.

   Links to object classes, properties, methods, or events in the Delphi help system can be made using Alinks. When linking to an object class, the Alink uses the class name of the object, followed by an underscore and the string "object". For example, to link to the *TCustomPanel* object, use the following:

   ```
   !AL(TCustomPanel_object,1)
   ```

   When linking to a property, method, or event, precede the name of the property, method, or event by the name of the object that implements it and an underscore. For example, to link to the *Text* property which is implemented by *TControl*, use the following:

   ```
   !AL(TControl_Text,1)
   ```

   To see an example of the secondary navigation topics, display the help for any component and click on the links labeled hierarchy, properties, methods, or events.

3. **Each property, method, and event that is declared within the component should have a topic.**

   A property, event, or method topic should show the declaration of the item and describe its use. Application developers see these topics either by highlighting the item in the Object Inspector and pressing F1 or by placing the cursor in the Code editor on the name of the item and pressing F1. To see an example of a property topic, select any item in the Object Inspector and press F1.

   The property, event, and method topics should include a K footnote that lists the name of the property, method, or event, and its name in combination with the name of the component. Thus, the *Text* property of *TControl* has the following K footnote:

   ```
   Text,TControl;TControl,Text;Text,
   ```

   The property, method, and event topics should also include a $ footnote that indicates the title of the topic, such as TControl.Text.

All of these topics should have a topic ID that is unique to the topic, entered as a # footnote.

Making component help context-sensitive

Topic groups

Each component, property, method, and event topic must have an A footnote. The A footnote is used to display the topic when the user selects a component and presses F1, or when a property or event is selected in the Object Inspector and the user presses F1. The A footnotes must follow certain naming conventions:

If the Help topic is for a component, the A footnote consists of two entries separated by a semicolon using this syntax:

```
ComponentClass_Object;ComponentClass
```

where *ComponentClass* is the name of the component class.

If the Help topic is for a property or event, the A footnote consists of three entries separated by semicolons using this syntax:

```
ComponentClass_Element;Element_Type;Element
```

where *ComponentClass* is the name of the component class, *Element* is the name of the property, method, or event, and *Type* is the either Property, Method, or Event

For example, for a property named *BackgroundColor* of a component named *TMyGrid*, the A footnote is

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

Adding component help files

Topic groups

To add your Help file to Delphi, use the OpenHelp utility (called oh.exe) located in the bin directory or accessed using Help|Customize in the IDE.

You will find information about using OpenHelp in the OpenHelp.hlp file, including adding your Help file to the Help system.

Adding property editors

Topic groups

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 these five steps:

1. Deriving a property-editor class
2. Editing the property as text
3. Editing the property as a whole
4. Specifying editor attributes
5. Registering the property editor

Deriving a property-editor class

Topic groups Example

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

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 that are the most useful for user-defined properties.

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.
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	Classes. Displays the name of the class and allows expansion of the class'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.

Example: Property editor class

The following example shows the declaration of a simple property editor named *TMyPropertyEditor*:

```
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 classes 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 the following table.

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.

Setting the property value

Topic groups Example

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 throw an exception and not use the improper value.

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

SetValue should convert the string and validate the value before calling one of the Set methods.

Example: Editing properties as text

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

Topic groups Example

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 classes. 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 class's *Edit* method.

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. Because 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.

Within your implementation of the *Edit* method, follow these steps:

1. Construct the editor you are using for the property.
2. Read the current value and assign it to the property using a Get method.
3. When the user selects a new value, assign that value to the property using a Set method.
4. Destroy the editor.

Example: Editing a property as a whole

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

Topic groups Example

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

Flag	Related method	Meaning if included
paValueList	GetValues	The editor can give a list of enumerated values.
paSubProperties	GetProperties	The property has subproperties that can display.
paDialog	Edit	The editor can display a dialog box for editing the entire property.
paMultiSelect	N/A	The property should display when the user selects more than one component.
paAutoUpdate	SetValue	Updates the component after every change instead of waiting for approval of the value.
paSortList	N/A	The Object Inspector should sort the value list.
paReadOnly	N/A	Users cannot modify the property value.
paRevertable	N/A	Enables the Revert to Inherited menu item on the Object Inspector's context menu. The menu item tells the property editor to discard the current property value and return to some previously established default or standard value.

Example: Specifying editor attributes

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

Topic groups Example

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.

Example: Registering a property editor

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 have 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 type *TComponentName*) 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*.

Adding component editors

Topic groups

Component editors determine what happens when the component is double-clicked in the designer and add commands to the context menu that appears when the component is right-clicked. They can also copy your component to the Windows clipboard in custom formats.

If you do not give your components a component editor, Delphi uses the default component editor. The default component editor is implemented by the class *TDefaultEditor*. *TDefaultEditor* does not add any new items to a component's context menu. When the component is double-clicked, *TDefaultEditor* searches the properties of the component and generates (or navigates to) the first event handler it finds.

To add items to the context menu, change the behavior when the component is double-clicked, or add new clipboard formats, derive a new class from *TComponentEditor* and register its use with your component. In your overridden methods, you can use the *Component* property of *TComponentEditor* to access the component that is being edited.

Adding a custom component editor consists of the steps:

- Adding items to the context menu
- Changing the double-click behavior
- Adding clipboard formats
- Registering the component editor

Adding items to the context menu

Topic groups

When the user right-clicks the component, the *GetVerbCount* and *GetVerb* methods of the component editor are called to build context menu. You can override these methods to add commands (verbs) to the context menu.

Adding items to the context menu requires the steps:

- Specifying menu items
- Implementing commands

Specifying menu items

Topic groups

Override the *GetVerbCount* method to return the number of commands you are adding to the context menu. Override the *GetVerb* method to return the strings that should be added for each of these commands. When overriding *GetVerb*, add an ampersand (&) to a string to cause the following character to appear underlined in the context menu and act as a shortcut key for selecting the menu item. Be sure to add an ellipsis (...) to the end of a command if it brings up a dialog. *GetVerb* has a single parameter that indicates the index of the command.

The following code overrides the *GetVerbCount* and *GetVerb* methods to add two commands to the context menu.

```
function TMyEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;
function TMyEditor.GetVerb(Index: Integer): String;
begin
  case Index of
    0: Result := "&DoThis ...";
    1: Result := "Do&That";
  end;
end;
```

Note: Be sure that your *GetVerb* method returns a value for every possible index indicated by *GetVerbCount*.

Implementing commands

Topic groups

When the command provided by *GetVerb* is selected in the designer, the *ExecuteVerb* method is called. For every command you provide in the *GetVerb* method, implement an action in the *ExecuteVerb* method. You can access the component that is being edited using the *Component* property of the editor.

For example, the following *ExecuteVerb* method implements the commands for the *GetVerb* method in the previous example.

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
  MySpecialDialog: TMyDialog;
begin
  case Index of
    0: begin
         MyDialog := TMySpecialDialog.Create(Application);       { instantiate the
editor }
         if MySpecialDialog.Execute then;                 { if the user OKs the
dialog... }
           MyComponent.FThisProperty := MySpecialDialog.ReturnValue;   { ...use the
value }
         MySpecialDialog.Free;                                    { destroy the
editor }
       end;
    1: That;                                                 { call the That
method }
  end;
end;
```

Changing the double-click behavior

Topic groups

When the component is double-clicked, the *Edit* method of the component editor is called. By default, the *Edit* method executes the first command added to the context menu. Thus, in the previous example, double-clicking the component executes the *DoThis* command.

While executing the first command is usually a good idea, you may want to change this default behavior. For example, you can provide an alternate behavior if

- you are not adding any commands to the context menu.
- you want to display a dialog that combines several commands when the component is double-clicked.

Override the *Edit* method to specify a new behavior when the component is double-clicked. For example, the following *Edit* method brings up a font dialog when the user double-clicks the component:

```
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
end;
```

Note: If you want a double-click on the component to display the Code editor for an event handler, use *TDefaultEditor* as a base class for your component editor instead of *TComponentEditor*. Then, instead of overriding the *Edit* method, override the protected *TDefaultEditor.EditProperty* method instead. *EditProperty* scans through the event handlers of the component, and brings up the first one it finds. You can change this to look a particular event instead. For example:

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
```

Adding clipboard formats

Topic groups

By default, when a user chooses Copy while a component is selected in the IDE, the component is copied in Delphi's internal format. It can then be pasted into another form or data module. Your component can copy additional formats to the Clipboard by overriding the *Copy* method.

For example, the following *Copy* method allows a *TImage* component to copy its picture to the Clipboard. This picture is ignored by the Delphi IDE, but can be pasted into other applications.

```
procedure TMyComponent.Copy;
var
  MyFormat : Word;
  AData,APalette : THandle;
begin
  TImage(Component).Picture.Bitmap.SaveToClipBoardFormat(MyFormat, AData, APalette);
  ClipBoard.SetAsHandle(MyFormat, AData);
end;
```

Registering the component editor

Topic groups

Once the component editor is defined, it can be registered to work with a particular component class. A registered component editor is created for each component of that class when it is selected in the form designer.

To create the association between a component editor and a component class, call *RegisterComponentEditor*. *RegisterComponentEditor* takes the name of the component class that uses the editor, and the name of the component editor class that you have defined. For example, the following statement registers a component editor class named *TMyEditor* to work with all components of type *TMyComponent*:

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

Place the call to *RegisterComponentEditor* in the *Register* procedure where you register your component. For example, if a new component named *TMyComponent* and its component editor *TMyEditor* are both implemented in the same unit, the following code registers the component and its association with the component editor.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
```

Property categories

Topic groups

In the Delphi IDE, the Object Inspector affords the programmer the ability to selectively hide and display properties based on property categories. The properties of new custom components can also be fit into this scheme by registering properties in categories. Do this at the same time the component is being registered by calling one of the property registration functions RegisterPropertyInCategory or RegisterPropertiesInCategory. Use the former to register a single property. Use the latter to register multiple properties in a single function call. These functions are defined in the unit DsgnIntf.

Note that it is not mandatory that you register properties or that you register all of the properties of a custom component when some are registered. Any property not explicitly associated with a category is simply deemed to be in the TMiscellaneousCategory category. These properties would be displayed or hidden in the Object Inspector based on that default categorization.

Delphi supplies thirteen stock property categories, in the form of property classes. Register a property of a new custom component in one of these provided categories or create your on property category classes based on these built-in classes.

In addition to these two functions for registering properties, there is an IsPropertyInCategory function. This function is useful for such endeavors as creating localization utilities, in which you must determine whether a property is registered in a given property category.

- Registering one property at a time
- Registering multiple properties at once
- Property category classes
- Using the IsPropertyInCategory function

Registering one property at a time

Topic groups

Register one property at a time and associate it with a property category using the RegisterPropertyInCategory function. RegisterPropertyInCategory comes in four overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with the property category.

The first variation allows you to identify the property by the property's name. The line below registers a property related to visual display of the component, identifying the property by its name, "AutoSize".

```
RegisterPropertyInCategory(TVisualCategory, 'AutoSize');
```

The second variation identifies the property using the characteristics component class type and property name. The example below registers (into the category THelpCategory) a property named "HelpContext" of a component of the custom class TMyButton.

```
RegisterPropertyInCategory(THelpCategory, TMyButton, 'HelpContext');
```

The third variation uses the property's type and the property's name to identify the property. The example below registers a property based on a combination of its type, Integer, and its name, "Width".

```
RegisterPropertyInCategory(TVisualCategory, TypeInfo(Integer), 'Width');
```

The last variation identifies the property using only its property type. The example below registers a property based on its type, Integer.

```
RegisterPropertyInCategory(TVisualCategory, TypeInfo(Integer));
```

See the section Property category classes for a list of the available property categories and a brief description of their uses.

Registering multiple properties at once

Topic groups

Register multiple properties at one time and associate them with a property category using the RegisterPropertiesInCategory function. RegisterPropertiesInCategory comes in three overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with property categories.

The first variation allows you to identify properties for association with a property category based on property name. A list of property names is passed as an array of String. Each property identified by name in the list is registered with the specified property category. In the example below, four properties are registered in the category THelpCategory. These four properties are identified by name using the Strings “HelpContext”, “Hint”, “ParentShowHint”, and “ShowHint”.

```
RegisterPropertiesInCategory(THelpCategory, ['HelpContext', 'Hint',
'ParentShowHint',   'ShowHint']);
```

The second variation identifies the properties by their type. In the example below, all of the properties in the custom component that are of type String are registered in the category TLocalizableCategory.

```
RegisterPropertiesInCategory(TLocalizableCategory, TypeInfo(String));
```

The third variation allows you to pass a list of various criteria, not all of which need be the same type, to use to identify properties to register. The list is passed as an array of constants. In the example below, any property that either has the name “Text” or belongs to a class of type TEdit is registered in the category TLocalizableCategory.

```
RegisterPropertiesInCategory(TLocalizableCategory, ['Text', TEdit]);
```

See the section Property category classes for a list of the available property categories and a brief description of their uses.

Property category classes

Topic groups

Built-in property categories

Delphi provides a built-in set of twelve property categories with which you can associate properties in custom components. Use one of these property category class names for the ACategoryClass parameter of the RegisterPropertyInCategory and RegisterPropertiesInCategory functions.:

Category	Purpose
TActionCategory	Properties related to runtime actions; the Enabled and Hint properties of TEdit are in this category.
TDatabaseCategory	Properties related to database operations; the DatabaseName and SQL properties of TQuery are in this category.
TDragNDropCategory	Properties related to drag-n-drop and docking operations; the DragCursor and DragKind properties of TImage are in this category.
THelpCategory	Properties related to using online help or hints; the HelpContext and Hint properties of TMemo are in this category.
TLayoutCategory	Properties related to the visual display of a control at design-time; the Top and Left properties of TDBEdit are in this category.
TLegacyCategory	Properties related to obsolete operations; the Ctl3D and ParentCtl3D properties of TComboBox are in this category.
TLinkageCategory	Properties related to associating or linking one component to another; the DataSet property of TDataSource is in this category.
TLocaleCategory	Properties related to international locales; the BiDiMode and ParentBiDiMode properties of TMainMenu are in this category.
TLocalizableCategory	Properties that may require modification in localized versions of an application. Many string properties (such as *Caption*) are in this category, as are properties that determine the size and position of controls.
TMiscellaneousCategory	Properties that do not fit a category or do not need to be categorized (and properties not explicitly registered to a specific category); the AllowAllUp and Name properties of TSpeedButton are in this category.
TVisualCategory	Properties related to the visual display of a control at runtime; the Align and Visible properties of TScrollBox are in this category.
TInputCategory	Properties related to the input of data (need not be related to database operations); the Enabled and ReadOnly properties of TEdit are in this category.

Deriving new property categories

You can create new property categories of your own design by deriving a class from either the base class *TPropertyCategory* or one of the built-in descendants. See the section Property category classes for a list of the available property categories and a brief description of their uses.

When deriving a new property category class, override the Name method. The Name method provides the name of the category for display in the Object Inspector. This method must be superseded with a method that returns the name of the custom category. The Name method may simply return a String value or it may retrieve a value from a resource. The latter is useful for easily internationalizing a custom component and its categories.

Given the new Name method below for a custom category class, the text “My Special” would appear in the Object Inspector when property categories are displayed (and at least one of the current object’s properties is registered in this property class).

```
class function TMySpecialCategory.Name: String;
begin
  Result := 'My Special';
end;
```

Using the IsPropertyInCategory function

Topic groups

An application can query the existing registered properties to determine whether a given property is already registered in a specified category. This can be especially useful in situations like a localization utility that checks the categorization of properties preparatory to performing its localization operations. Two overloaded variations of the IsPropertyInCategory function are available, allowing for different criteria in determining whether a property is in a category.

The first variation allows you to base the comparison criteria on a combination of the class type of the owning component and the property's name. In the command line below, for IsPropertyInCategory to return True, the property must belong to the class TEdit, have the name "Text", and be in the property category TLocalizableCategory.

```
IsItThere := IsPropertyInCategory(TLocalizableCategory, TEdit, 'Text');
```

The second variation allows you to base the comparison criteria on a combination of the class name of the owning component and the property's name. In the command line below, for IsPropertyInCategory to return True, the property must belong to the class TEdit, have the name "Text", and be in the property category TLocalizableCategory.

```
IsItThere := IsPropertyInCategory(TLocalizableCategory, 'TEdit', 'Text');
```

Compiling components into packages

Topic groups

Once your components are registered, you must compile them as packages before they can be installed in the IDE. A package can contain one or several components as well as custom property editors. For more information about packages, see Working with packages and components.

To create and compile a package, see Creating and editing packages. Put the source-code units for your custom components in the package's Contains list. If your components depend on other packages, include those packages in the Requires list.

To install your components in the IDE, see Installing component packages.

Modifying an existing component

Topic groups

The easiest way to create a component is to derive it from a component that does nearly everything you want, then make whatever changes you need. What follows is a simple example that modifies the standard memo component to create a memo that does not wrap words by default.

The value of the memo component's *WordWrap* property is initialized to *True*. If you frequently use non-wrapping memos, you can create a new memo component that does not wrap words by default.

Note: To modify published properties or save specific event handlers for an existing component, it is often easier to use a *component template*

Modifying an existing component takes only two steps:

- Creating and registering the component
- Modifying the component class

Creating and registering the component

Topic groups

Creation of every component begins the same way: you create a unit, derive a component class, register it, and install it on the Component palette. Creating a new component.

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 compile and install the new component now, it behaves exactly like its ancestor, *TMemo*. In the next section, you will make a simple change to your component.

Modifying the component object

Topic groups

Once you have created a new component class, you can modify it in almost any way. In this case, you will change only the initial value of one property in the memo component. This involves two small changes to the component class:

- Overriding the constructor.
- Specifying the new default property value.

The constructor actually sets the value of the property. The default tells Delphi what values to store in the form (.DFM) file. Delphi stores only values that differ from the default, so it is important to perform both steps.

Overriding the constructor

Topic groups

When a component is placed on a form at design time, or when an application constructs a component at runtime, the component's constructor sets the property values. When a component is loaded from a form file, the application sets any properties changed at design time.

Note: When you override a constructor, the new constructor must call the inherited constructor before doing anything else. For more information, see Overriding methods.

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 constructor override to the forward 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 is now initialized to *False*.

If you change an initial property value, you should 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

Topic groups

When Delphi stores a description of a form in a form file, it stores the values only of properties that differ from their defaults. Storing only the differing values keeps the form files small and makes loading the form faster. If you create a property or change the default value, it is a good idea to update the property declaration to include the new default. Form files, loading, and default values are explained in more detail in Making components available at design time.

To change the default value of a property, redeclare the property name, followed by the directive **default** and the new default value. 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 the **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 does not affect the workings of the component. You must still initialize the value in the component's constructor. Redeclaring the default ensures that Delphi knows when to write *WordWrap* to the form file.

Creating a graphic component

Topic groups

A graphic control is a simple kind of component. Because a purely graphic control never receives focus, it does not have or need a window handle. Users can still manipulate the control with the mouse, but there is no keyboard interface.

The graphic component presented in this chapter is *TShape*, the shape component is 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 and registering the component
- Publishing inherited properties
- Adding graphic capabilities

Creating and registering the component

Topic groups

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. Creating a new component.

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

Topic groups

Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor class you want to surface in the 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 and Making events visible. Both processes involve redeclaring just the name of the properties in the published part of the class declaration.

For the shape control, you can publish the three mouse events, the three drag-and-drop events, and the 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

Topic groups

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. You have two tasks to perform when creating a graphic control:

1. Determining what to draw.
2. Drawing the component image.

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

Determining what to draw

Topic groups

A graphic control can change its appearance to reflect a dynamic condition, including user input. A graphic control that always looks the same should probably not be a component at all. If you want a static image, you can import the image instead of using a 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 your control a property that determines the shape it draws, add a property called *Shape*. This requires

1. Declaring the property type.
2. Declaring the property.
3. Writing the implementation method.

Creating properties is explained in more detail in Creating properties.

Declaring the property type

Topic groups

When you declare a property of a user-defined type, you must declare the type first, before the class 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 class'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 class.

Declaring the property

Topic groups

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 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 will declare a field that holds the current shape, then declare a property that reads that field and writes to it 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

Topic groups

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

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;                                        { force a repaint with the new
shape }
  end;
end;
```

Overriding the constructor and destructor

Topic groups

To change default property values and initialize owned classes for your component, you must 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 fairly small, so you can change the width and height in the constructor. Changing default property values is explained in more detail in Modifying an existing component.

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

Add the overridden constructor to the declaration of the component class:

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

1 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;
```

2 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

Topic groups

By default, a canvas has a thin black pen and a solid white brush. To let developers change the pen and brush, you must provide classes for them to manipulate at design time, then copy the classes into the canvas during painting. Classes such as an auxiliary pen or brush are called *owned classes* because the component owns them and is responsible for creating and destroying them.

Managing owned classes requires

1. Declaring the class fields.
2. Declaring the access properties.
3. Initializing owned classes.
4. Setting owned classes' properties.

Declaring the class fields

Topic groups

Each class a component owns must have a class field declared for it in the component. The class field ensures that the component always has a pointer to the owned object so that it can destroy the class 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 applications (or other components) need access to the owned objects, you can declare **published** or **public** properties for this purpose.

Add fields for a pen and brush 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

Topic groups

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

To the shape control, add properties that provide access to the pen and brush fields. You will also declare 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;
```

To directly assign the contents of *Value* to *FBrush*...

```
  FBrush := Value;
```

...would overwrite the internal pointer for *FBrush*, lose memory, and create a number of ownership problems.

Initializing owned classes

Topic groups

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

Because you have 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 class:

```
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 classes' properties

Topic groups

As the final step in handling the pen and brush classes, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush classes 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

Topic groups

The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a 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 of the unit.

For the shape control, add the following declaration to the class declaration:

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

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

```
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

Topic groups

The standard shape control does one more thing that your sample shape control does not yet do: it handles squares and circles as well as rectangles and ellipses. To do that, you need to write code that finds the shortest side and centers the image.

Here is 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;
```

Customizing a grid

Topic groups

Delphi provides 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 will see how to create a small one-month calendar from the basic grid component, *TCustomGrid*.

Creating the calendar involves these tasks:

- 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 component is similar to the *TCalendar* component on the Samples page of the Component palette.

Creating and registering the component

Topic groups

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. Creating a new component.

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
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, 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 will find that it appears on the Samples page. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note: While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the *DrawCell* method is described in "Filling in the cells" below.

Publishing inherited properties

Topic groups

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.

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 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;
```

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 install the modified calendar component to the Component palette and use it in an application, you will 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

Topic groups

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 have not published the grid properties *ColCount* and *RowCount*, because it is highly unlikely that users of the calendar will want to display anything other than seven days per week. You still must set the initial values of those properties so that the week always has seven days, however.

To change the initial values of the component's properties, override the constructor to set the desired values. The constructor must be virtual.

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] + [goDrawFocusSelected];  {disable range
selection}
end;
```

The calendar now has seven columns and seven rows, with the top row fixed, or nonscrolling.

Resizing the cells

Topic groups

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 will add a message-handling method to the component.

Creating a message-handling method is described in detail in the section Creating new message handlers.

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 when the calendar is resized, it displays all the cells in the largest size that will fit in the control.

Filling in the cells

Topic groups

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 runtime library contains an array with short day names, so for the calendar, 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
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);    { use RTL
strings }
end;
```

Tracking the date

Topic groups

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 runtime access to that value, but also provide Day, *Month*, and *Year* properties that users of the calendar component can set at design time.

Tracking the date in the calendar consists of the processes:

- Storing the internal date
- Accessing the day, month, and year
- Generating the day numbers
- Selecting the current day

Storing the internal date

Topic groups

To store the date for the calendar, you need a private field to hold the date and a runtime-only property that provides access to that date.

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 runtime 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

Topic groups

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 because 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;
      3: ADay := Value;
      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 runtime 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

Topic groups

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. Use the *IsLeapYear* function to determine whether the year is a leap year. Use the *MonthDays* array in the SysUtils unit to get the number of days in the month.

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 will 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 a class 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 > MonthDays[IsLeapYear(Year), Month]) 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 will see the proper information for the current month.

Selecting the current day

Topic groups

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 are now reusing the *ADay* variable previously set by decoding the date.

Navigating months and years

Topic groups

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 runtime. However, such manipulations are generally only cumbersome when performed repeatedly, and that is fairly rare at design time.

For the calendar, add the following four methods for next and previous month and year. Each of these methods uses the *IncMonth* function in a slightly different manner to increment or decrement *CalendarDate*, by increments of a month or a year. After incrementing or decrementing *CalendarDate*, decode the date value to fill the Year, Month, and Day properties with corresponding new values.

```
procedure TCalendar.NextMonth;
begin
  DecodeDate(IncMonth(CalendarDate, 1), Year, Month, Day);
end;
procedure TCalendar.PrevMonth;
begin
  DecodeDate(IncMonth(CalendarDate, -1), Year, Month, Day);
end;
procedure TCalendar.NextYear;
begin
  DecodeDate(IncMonth(CalendarDate, 12), Year, Month, Day);
end;
procedure TCalendar.PrevYear;
begin
  DecodeDate(CalendarDate, -12), Year, Month, Day);
end;
```

Be sure to add the declarations of the new methods to the class declaration.

Now when you create an application that uses the calendar component, you can easily implement browsing through months or years.

Navigating days

Topic groups

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.

The process of navigating days consists of

- Moving the selection
- Providing an OnChange event
- Excluding blank cells

Moving the selection

Topic groups

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;
var
  TempDay: Integer;
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

Topic groups

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

Topic groups

As the calendar is written, the user can select a blank cell, but the date does not change. 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.

Making a control data-aware

Topic groups

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, lookup controls, and grids. You can also make your own controls data aware. For more information about using data-aware controls, see Using data controls.

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 section first illustrates the simplest case, making a read-only control that links to a single field in a dataset. The specific control used will be the calendar created in Customizing a grid, *TSampleCalendar*. You can also use the standard calendar control on the Samples page of the Component palette, *TCalendar*.

The section then continues with an explanation of how to make the new data-browsing control a data-editing control.

Creating a data-browsing control

Topic groups

Creating a data-aware calendar control, whether it is a read-only control or one in which the user can change the underlying data in the dataset, involves the following steps:

- Creating and registering the component.
- Adding the data link.
- Responding to data changes.

Creating and registering the component

Topic groups Example

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in Creating a new component.

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

- Call the component's unit *DBCal*.
- Derive a new component class called *TDBCalendar*, descended from *TSampleCalendar*. The section Customizing a grid shows you how to create the *TSampleCalendar* component.
- Register *TDBCalendar* on the Samples page of the Component palette.

Example: Creating and registering a component

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

Topic groups

Because this data calendar will be read-only with respect to the data, it makes sense to make the control itself read-only, so users will not make changes within the control and expect them to be reflected in the database.

Making the calendar read-only involves,

- Adding the ReadOnly property.
- Allowing needed updates.

Note that if you started with the *TCalendar* component from Delphi's Samples page instead of *TSampleCalendar*, it already has a *ReadOnly* property, so you can skip these steps.

Adding the read-only property

Topic groups

By adding a *ReadOnly* property, you will 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 41-22.

```
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 will find that the component ignores clicks and keystrokes. It also fails to update the selection position when you change the date.

Allowing needed updates

Topic groups

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 because *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 TDBCalendar.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

Topic groups

The connection between a control and a database is handled by a class called a *data link*. The datalink class 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 datalink class. That is, the control has the responsibility for constructing and destroying the data link. For details on management of owned classes, see Creating a graphic component.

Establishing a data link as an owned class requires these three steps:

1. Declaring the class field
2. Declaring the access properties
3. Initializing the data link

Declaring the class field

Topic groups

A component needs a field for each of its owned classes, as explained in Declaring the class fields. 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;
```

Before you can compile the application, you need to add DB and DBCtrls to the unit's **uses** clause.

Declaring the access properties

Topic groups See also Example

Every data-aware control has a *DataSource* property that specifies which data-source class in the application provides the data to the control. In addition, a control that accesses a single field needs a *DataField* property to specify that field in the data source.

Unlike the access properties for the owned classes in the example in Creating a graphic component, these access properties do not provide access to the owned classes themselves, but rather to corresponding properties in the owned class. That is, you will create properties that enable the control and its data link to share the same data source and 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 datalink class

Example: Declaring access properties

```
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 class when the calendar control is constructed, and destroy the data link before destroying the calendar.

Initializing the data link

Topic groups See also

A data-aware control needs access to its data link throughout its existence, so it must construct the datalink object as part of its own constructor, and destroy the datalink object before it is itself destroyed.

Override the *Create* and *Destroy* methods of the calendar to construct and destroy the datalink 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
  FDataLink := TFieldDataLink.Create;                    { construct the datalink
object }
  inherited Create(AOwner);                 { always call the inherited constructor
first }
  FReadOnly := True;                                                { this is
already here }
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

Topic groups Example

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.

Datalink classes all have events named *OnDataChange*. When the data source indicates a change in its data, the datalink 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 will add a method to the calendar, then designate it as the handler for the data link's *OnDataChange*.

Declare and implement the *DataChange* method, then assign it to the data link's *OnDataChange* event in the constructor. In the destructor, detach the *OnDataChange* handler before destroying the object.

Example: Responding to data changes

```
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 datalink
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.AsDateTime;   { otherwise, set calendar to
the date }
end;
```

You now have a data-browsing control.

Creating a data-editing control

Topic groups

When you create a data-editing control, you create and register the component and add the data link just as you do for a data-browsing control. You also respond to data changes in the underlying field in a similar manner, but you must handle a few more issues.

For example, you probably want your control to respond to both key and mouse events. Your control must respond when the user changes the contents of the control. When the user exits the control, you want the changes made in the control to be reflected in the dataset.

The data-editing control described here is the same calendar control described in the first part of the chapter. The control is modified so that it can edit as well as view the data in its linked field.

Modifying the existing control to make it a data-editing control involves:

- Changing the default value of FReadOnly.
- Handling mouse-down and key-down messages.
- Updating the field datalink class.
- Modifying the Change method.
- Updating the dataset.

Changing the default value of FReadOnly

Topic groups

Because this is a data-editing control, the *ReadOnly* property should be set to *False* by default. To make the *ReadOnly* property *False*, change the value of *FReadOnly* in the constructor:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  ...
  FReadOnly := False;  { set the default value }
  ...
end;
```

Handling mouse-down and key-down messages

Topic groups

When the user of the control begins interacting with it, the control receives either mouse-down messages (WM_LBUTTONDOWN, WM_MBUTTONDOWN, or WM_RBUTTONDOWN) or a key-down message (WM_KEYDOWN) from Windows. To enable a control to respond to these messages, you must write handlers that respond to these messages.

- Responding to mouse-down messages
- Responding to key-down messages

Responding to mouse-down messages

Topic groups

A *MouseDown* method is a protected method for a control's *OnMouseDown* event. The control itself calls *MouseDown* in response to a Windows mouse-down message. When you override the inherited *MouseDown* method, you can include code that provides other responses in addition to calling the *OnMouseDown* event.

To override *MouseDown*, add the *MouseDown* method to the *TDBCalendar* class:

```
type
  TDBCalendar = class(TSampleCalendar);
   ...
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y:
Integer);
      override;
   ...
  end;
procedure TDBCalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
  begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
  end;
end;
```

When *MouseDown* responds to a mouse-down message, the inherited *MouseDown* method is called only if the control's *ReadOnly* property is *False* and the datalink object is in edit mode, which means the field can be edited. If the field cannot be edited, the code the programmer put in the *OnMouseDown* event handler, if one exists, is executed.

Responding to key-down messages

Topic groups

A *KeyDown* method is a protected method for a control's *OnKeyDown* event. The control itself calls *KeyDown* in response to a Windows key-down message. When overriding the inherited *KeyDown* method, you can include code that provides other responses in addition to calling the *OnKeyDown* event.

To override *KeyDown*, follow these steps:

1 Add a *KeyDown* method to the *TDBCalendar* class:

```
type
  TDBCalendar = class(TSampleCalendar);
   ...
  protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
   ...
  end;
```

2 Implement the *KeyDown* method:

```
procedure KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, VK_END,
    VK_HOME, VK_PRIOR, VK_NEXT]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
  begin
    MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
  end;
end;
```

When *KeyDown* responds to a mouse-down message, the inherited *KeyDown* method is called only if the control's *ReadOnly* property is *False*, the key pressed is one of the cursor control keys, and the datalink object is in edit mode, which means the field can be edited. If the field cannot be edited or some other key is pressed, the code the programmer put in the *OnKeyDown* event handler, if one exists, is executed.

Updating the field datalink object

Topic groups

There are two types of data changes:

- A change in a field value that must be reflected in the data-aware control.
- A change in the data-aware control that must be reflected in the field value.

The *TDBCalendar* component already has a *DataChange* method that handles a change in the field's value in the dataset by assigning that value to the *CalendarDate* property. The *DataChange* method is the handler for the *OnDataChange* event. So the calendar component can handle the first type of data change.

Similarly, the field datalink class also has an *OnUpdateData* event that occurs as the user of the control modifies the contents of the data-aware control. The calendar control has a *UpdateData* method that becomes the event handler for the *OnUpdateData* event. *UpdateData* assigns the changed value in the data-aware control to the field data link.

1. To reflect a change made to the value in the calendar in the field value, add an *UpdateData* method to the private section of the calendar component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
   ...
  end;
```

2. Implement the *UpdateData* method:

```
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;        { set field link to calendar
date }
end;
```

3. Within the constructor for *TDBCalendar*, assign the *UpdateData* method to the *OnUpdateData* event:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
```

Modifying the Change method

Topic groups

The *Change* method of the *TDBCalendar* is called whenever a new date value is set. *Change* calls the *OnChange* event handler, if one exists. The component user can write code in the *OnChange* event handler to respond to changes in the date.

When the calendar date changes, the underlying dataset should be notified that a change has occurred. You can do that by overriding the *Change* method and adding one more line of code. These are the steps to follow:

1 Add a new *Change* method to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure Change; override;
   ...
  end;
```

2 Write the *Change* method, calling the *Modified* method that informs the dataset the data has changed, then call the inherited *Change* method:

```
TDBCalendar.Change;
begin
  FDataLink.Modified;                                    { call the Modified
method }
  inherited Change;                                  { call the inherited Change
method }
end;
```

Updating the dataset

Topic groups

So far, a change within the data-aware control has changed values in the field datalink class. The final step in creating a data-editing control is to update the dataset with the new value. This should happen after the person changing the value in the data-aware control exits the control by clicking outside the control or pressing the *Tab* key.

VCL has defined message control IDs for operations on controls. For example, the CM_EXIT message is sent to the control when the user exits the control. You can write message handlers that respond to the message. In this case, when the user exits the control, the *CMExit* method, the message handler for CM_EXIT, responds by updating the record in the dataset with the changed values in the field datalink class. For more information about message handlers, see Handling messages.

To update the dataset within a message handler, follow these steps:

1 Add the message handler to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
   ...
  end;
```

2 Implement the *CMExit* method so it looks something like this:

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                          { tell data link to update
database }
  except
    on Exception do SetFocus;                       { if it failed, don't let focus
leave }
 end;
  inherited;
end;
```

Making a dialog box a component

Topic groups

You will find it convenient to make a frequently used dialog box into a component that you add to the Component palette. Your dialog box components will work just like the components that represent the standard Windows common dialog boxes. The goal is to create a simple component that a user can add to a project and set properties for at design time.

Making a dialog box a component requires these steps:

1. Defining the component interface
2. Creating and registering the component
3. Creating the component interface
4. Testing the component

The Delphi “wrapper” component associated with the dialog box creates and executes the dialog box at runtime, passing along the data the user specified. The dialog-box component is therefore both reusable and customizable.

In this section, you will see how to create a wrapper component around the generic About Box form provided in the Delphi Object Repository.

Note: Copy the files ABOUT.PAS and ABOUT.DFM into your working directory.

There are not 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

Topic groups

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 is 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.

In the case of the About box, you do not need to return any information, so the wrapper's properties only have to contain the information needed to display the About box properly. Because there are four separate fields in the About box that the application might affect, you will provide four string-type properties to provide for them.

Creating and registering the component

Topic groups

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Component palette. This process is outlined in Creating a new component.

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.
```

The new component now has only the capabilities built into *TComponent*. It is the simplest nonvisual component. In the next section, you will create the interface between the component and the dialog box.

Creating the component interface

Topic groups

These are the steps to create the component interface:

1. Including the form unit files
2. Adding interface properties
3. Adding the Execute method

Including the form unit

Topic groups

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 class. In the case of the About box, the form class 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

Topic groups Example

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 class 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 are 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 data at design time so that the wrapper can pass it to the dialog box at runtime.

Declaring an interface property requires two additions to the component's class declaration:

- A private class field, which is a variable the wrapper uses to store the value of the property
- The published 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 class 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.

Example: Adding interface properties

For example, to declare an integer-type interface property called *Year*, you would declare it as follows:

```
type
  TMyWrapper = class(TComponent)
  private
    FYear: Integer;                                { field to hold the Year-property
data }
  published
    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)
  private
    FProductName, FVersion, FCopyright, FComments: string;               { declare
fields }
  published
    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 will 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

Topic groups Example

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 will 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*.

Example: Adding an Execute method

Here is 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.

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. Because the About box does not return any information to the application, there is no need to do anything after calling *ShowModal*. Write the About-box wrapper's *Execute* method so that it looks like this:

Within the public part of the *TAboutDlg* class, add the declaration for the *Execute* method:

```
type
  TAboutDlg = class(TComponent)
public
  function Execute: Boolean;
end;
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

Topic groups

Once you have 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.

Related topic groups

Component Writer's Guide

- Overview of component creation
- Object-oriented programming for component writers
- Creating properties
- Creating events
- Creating methods
- Using graphics in components
- Handling messages
- Property categories
- Making components available at design time
- Modifying an existing component
- Creating a graphic component
- Customizing a grid
- Making a control data aware
- Making a dialog box a component

Overview of component creation

Related topic groups

- Overview of component creation
- The Visual Component Library
- Components and classes
- How do you create components?
- Modifying existing controls
- Creating original controls
- Creating graphic controls
- Subclassing Windows controls
- Creating nonvisual components
- What goes into a component?
- Removing dependencies
- Properties, methods, and events
- Graphics encapsulation
- Registration Overview
- Creating a new component
- Using the Component wizard
- Creating a component manually
- Creating a unit
- Deriving the component
- Registering the component
- Testing uninstalled components

Object-oriented programming for component writers

Related topic groups

- Object-oriented programming for component writers: Overview
- Defining new classes
- Deriving new classes
- Changing class defaults to avoid repetition
- Adding new capabilities to a class
- Declaring a new component class
- Ancestors and descendants
- Controlling access
- Hiding implementation details
- Defining the developer's interface
- Defining the runtime interface
- Defining the design-time interface
- Dispatching methods
- Static methods
- Virtual methods
- Overriding methods
- Dynamic methods
- Abstract class members
- Classes and pointers

Creating properties

Related topic groups

- Creating properties: Overview
- Why create properties?
- Types of properties
- Publishing inherited properties
- Defining component properties
- The property declaration
- Internal data storage (properties)
- Direct access
- Access methods (properties)
- The read method
- The write method
- Default property values
- Specifying no default value
- Creating array properties
- Storing and loading properties
- Using the store-and-load mechanism
- Specifying default values
- Determining what to store
- Initializing after loading
- Storing and loading unpublished properties
- Creating methods to store and load property values
- Overriding the DefineProperties method

Creating events

Related topic groups

- Creating events: Overview
- What are events?
- Events are method pointers
- Calling the click-event handler
- Events are properties
- Event types are method-pointer types
- Event handler types are procedures
- Event handlers are optional
- Implementing the standard events
- Identifying standard events
- Making events visible
- Changing the standard event handling
- Defining your own events
- Triggering the event
- Two kinds of events
- Defining the handler type
- Declaring the event
- Calling the event
- Empty handlers must be valid
- Users can override default handling

Creating methods

Related topic groups

- Creating methods: Overview
- Avoiding interdependencies
- Naming methods
- Protecting methods
- Methods that should be public
- Methods that should be protected
- Abstract methods
- Making methods virtual
- Declaring methods

Using graphics in components

Related topic groups

- Using graphics in components: Overview
- Overview of graphics
- Using the canvas
- Working with pictures
- Using a picture, graphic, or canvas
- Loading and storing graphics
- Handling palettes
- Specifying a palette for a control
- Responding to palette changes
- Offscreen bitmaps
- Creating and managing off-screen bitmaps
- Copying bitmapped images
- Responding to changes

Handling messages

Related topic groups

- Handling messages: Overview
- Understanding the message-handling system
- What's in a Windows message?
- Dispatching messages
- Changing message handling
- Overriding the handler method
- Using message parameters
- Trapping messages
- The WndProc method
- Creating new message handlers
- Declaring a message identifier
- Declaring a message-structure type
- Declaring a new message-handling method

Property categories

Related topic groups

- Property categories
- Registering one property at a time
- Registering multiple properties at once
- Property category classes
- Using the IsPropertyInCategory function

Making components available at design time

Related topic groups

- Making components available at design time: Overview
- Registering components
- Declaring the register procedure
- Writing the Register procedure
- Specifying the components
- Specifying the palette page
- Using the RegisterComponents function
- Adding palette bitmaps
- Providing Help for your component
- Creating the help file
- Creating the entries
- Making component help context-sensitive
- Adding component help files
- Adding property editors
- Deriving a property-editor class
- Setting the property value
- Editing the property as a whole
- Specifying editor attributes
- Registering the property editor
- Adding component editors
- Adding items to the context menu
- Specifying menu items
- Implementing commands
- Changing the double-click behavior
- Adding clipboard formats
- Registering the component editor
- Property categories
- Registering one property at a time
- Registering multiple properties at once
- Property category classes
- Using the IsPropertyInCategory function
- Compiling components into packages

Modifying an existing component

Related topic groups

- Modifying an existing component: Overview
- Creating and registering the component
- Modifying the component object
- Overriding the constructor
- Specifying the new default property value

Creating a graphic component

Related topic groups

- Creating a graphic component
- Creating and registering the component
- Publishing inherited properties
- Adding graphic capabilities
- Determining what to draw
- Declaring the property type
- Declaring the property
- Writing the implementation method
- Overriding the constructor and destructor
- Publishing the pen and brush
- Declaring the class fields
- Declaring the access properties
- Initializing owned classes
- Setting owned classes' properties
- Drawing the component image
- Refining the shape drawing

Customizing a grid

Related topic groups

- Customizing a grid: Overview
- Creating and registering the component
- Publishing inherited properties
- Changing initial values
- Resizing the cells
- Filling in the cells
- Tracking the date
- Storing the internal date
- Accessing the day, month, and year
- Generating the day numbers
- Selecting the current day
- Navigating months and years
- Navigating days
- Moving the selection
- Providing an OnChange event
- Excluding blank cells

Making a control data aware

Related topic groups

- Making a control data aware
- Creating a data-browsing control
- Creating and registering the component
- Making the control read-only
- Adding the read-only property
- Allowing needed updates
- Adding the data link
- Declaring the class field
- Declaring the access properties for a data-aware control
- Initializing the data link
- Responding to data changes
- Creating a data-editing control
- Changing the default value of FReadOnly
- Handling mouse-down and key-down messages
- Responding to mouse-down messages
- Responding to key-down messages
- Updating the field datalink object
- Modifying the Change method
- Updating the dataset

Making a dialog box a component

Related topic groups

- Making a dialog box a component: Overview
- Defining the component interface
- Creating and registering the component
- Creating the component interface
- Including the form unit
- Adding interface properties
- Adding the Execute method
- Testing the component

Link not found

The topic you requested is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

The topic you requested is now loading. If it does not appear within a few seconds, the topic is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.