CHAPTER

12

Advanced VCL Component Building

IN THIS CHAPTER

The last chapter broke into writing Delphi custom components, and it gave you a solid introduction to the basics. In this chapter, you'll learn how to take component writing to the next level by incorporating advanced design techniques into your Delphi custom components. This chapter provides examples of advanced techniques such as pseudo-visual components, detailed property editors, component editors, and collections.

Pseudo-Visual Components

You've learned about visual components such as `TButton` and `TEdit`, and you've learned about nonvisual components such as `TTable` and `TTimer`. In this section, you'll also learn about a type of component that kind of falls in between visual and nonvisual components—we'll call these components *pseudo-visual components*.

Extending Hints

Specifically, the pseudo-visual component shown in this section is an extension of a Delphi pop-up hint window. We call this a *pseudo-visual* component because it's not a component that's used visually from the Component Palette at design time, but it does represent itself visually at runtime in the body of pop-up hints.

Replacing the default style hint window in a Delphi application requires that you complete the following four steps:

1. Create a descendant of `THintWindow`.
2. Destroy the old hint window class.
3. Assign the new hint window class.
4. Create the new hint window class.

Creating a `THintWindow` Descendant

Before you write the code for a `THintWindow` descendant, you must first decide how you want your new hint window class to behave differently from the default one. In this case, you'll create an elliptical hint window rather than the default square one. This actually demonstrates another cool technique: creating nonrectangular windows! Listing 12.1 shows the `RndHint.pas` unit, which contains the `THintWindow` descendant `TDDGHintWindow`.

LISTING 12.1 `RndHint.pas`—Illustrates an Elliptical Hint

```
unit RndHint;

interface

uses Windows, Classes, Controls, Forms, Messages, Graphics;
```

LISTING 12.1 Continued

```
type
  TDDGHintWindow = class(THintWindow)
  private
    FRegion: THandle;
    procedure FreeCurrentRegion;
  public
    destructor Destroy; override;
    procedure ActivateHint(Rect: TRect; const AHint: string); override;
    procedure Paint; override;
    procedure CreateParams(var Params: TCreateParams); override;
  end;

implementation

destructor TDDGHintWindow.Destroy;
begin
  FreeCurrentRegion;
  inherited Destroy;
end;

procedure TDDGHintWindow.FreeCurrentRegion;
{ Regions, like other API objects, should be freed when you are  }
{ through using them.  Note, however, that you cannot delete a   }
{ region which is currently set in a window, so this method sets }
{ the window region to 0 before deleting the region object.      }
begin
  if FRegion <> 0 then begin           // if Region is alive...
    SetWindowRgn(Handle, 0, True);     // set win region to 0
    DeleteObject(FRegion);             // kill the region
    FRegion := 0;                      // zero out field
  end;
end;

procedure TDDGHintWindow.ActivateHint(Rect: TRect; const AHint: string);
{ Called when the hint is activated by putting the mouse pointer }
{ above a control. }
begin
  with Rect do
    Right := Right + Canvas.TextWidth('WWWW');  // add some slop
  BoundsRect := Rect;
  FreeCurrentRegion;
  with BoundsRect do
    { Create a round rectangular region to display the hint window }
    FRegion := CreateRoundRectRgn(0, 0, Width, Height, Width, Height);
```

LISTING 12.1 Continued

```
  if FRegion <> 0 then
    SetWindowRgn(Handle, FRegion, True);         // set win region
  inherited ActivateHint(Rect, AHint);           // call inherited
end;

procedure TDDGHintWindow.CreateParams(var Params: TCreateParams);
{ We need to remove the border created on the Windows API-level }
{ when the window is created. }
begin
  inherited CreateParams(Params);
  Params.Style := Params.Style and not ws_Border;  // remove border
end;

procedure TDDGHintWindow.Paint;
{ This method gets called by the WM_PAINT handler.  It is }
{ responsible for painting the hint window. }
var
  R: TRect;
begin
  R := ClientRect;                   // get bounding rectangle
  Inc(R.Left, 1);                    // move left side slightly
  Canvas.Font.Color := clInfoText;  // set to proper color
  { paint string in the center of the round rect }
  DrawText(Canvas.Handle, PChar(Caption), Length(Caption), R,
           DT_NOPREFIX or DT_WORDBREAK or DT_CENTER or DT_VCENTER);
end;

initialization
  Application.ShowHint := False;     // destroy old hint window
  HintWindowClass := TDDGHintWindow; // assign new hint window
  Application.ShowHint := True;      // create new hint window
end.
```

The overridden `CreateParams()` and `Paint()` methods are fairly straightforward. `CreateParams()` provides an opportunity to adjust the structure of the window styles before the hint window is created on an API level. In this method, the `WS_BORDER` style is removed from the window class in order to prevent a rectangular border from being drawn around the window. The `Paint()` method is responsible for rendering the window. In this case, the method must paint the hint's `Caption` property into the center of the caption window. The color of the text is set to `clInfoText`, which is the system-defined color of hint text.

An Elliptical Window

The ActivateHint() method contains the magic for creating the nonrectangular hint window. Well, it's not really magic. Actually, two API calls make it happen: CreateRoundRectRgn() and SetWindowRgn().

CreateRoundRectRgn() defines a rounded rectangular region within a particular window. A *region* is a special API object that allows you to perform special painting, hit testing, filling, and clipping in one area. In addition to CreateRoundRectRgn(), a number of other Win32 API functions create different types of regions, including the following:

- CreateEllipticRgn()
- CreateEllipticRgnIndirect()
- CreatePolygonRgn()
- CreatePolyPolygonRgn()
- CreateRectRgn()
- CreateRectRgnIndirect()
- CreateRoundRectRgn()
- ExtCreateRegion()

Additionally, the CombineRgn() function can be used to combine multiple regions into one complex region. All these functions are described in detail in the Win32 API online help.

SetWindowRgn() is then called, passing the recently created region handle as a parameter. This function causes the operating system to take ownership of the region, and all subsequent drawing in the specified window will occur only within the region. Therefore, if the region defined is a rounded rectangle, painting will occur only within that rounded rectangular region.

CAUTION

You need to be aware of two side effects when using SetWindowRgn(). First, because only the portion of the window within the region is painted, your window probably won't have a frame or title bar. You must be prepared to provide the user with an alternative way to move, size, and close the window without the aid of a frame or title bar. Second, because the operating system takes ownership of the region specified in SetWindowRgn(), you must be careful not to manipulate or delete the region while it's in use. The TDDGHintWindow component handles this by calling its FreeCurrentRegion() method before the window is destroyed or a new window is created.

Enabling the `THintWindow` Descendant

The initialization code for the `RndHint` unit does the work of making the `TDDGHintWindow` component the application-wide active hint window. Setting `Application.ShowHint` to `False` causes the old hint window to be destroyed. At that point, you must assign your `THintWindow` descendant class to the `HintWindowClass` global variable. Then, setting `Application.ShowHint` back to `True` causes a new hint window to be created—this time it will be an instance of your descendant class.

Deploying `TDDGHintWindow`

Deploying this pseudo-visual component is different from normal visual and non-visual components. Because all the work for instantiating the component is performed in the `initialization` part of its unit, the unit shouldn't be added to a design package for use on the Component Palette but merely added to the `uses` clause of one of the source files in your project.

Animated Components

Once upon a time while writing a Delphi application, we thought to ourselves, "This is a really cool application, but our About dialog is kind of boring. We need something to spice it up a little." Suddenly, a light bulb came on and an idea for a new component was born We would create a scrolling credits marquee window to incorporate into our About dialogs.

The Marquee Component

Let's take a moment to analyze how the marquee component works. The marquee control: is able to take a bunch of strings and scroll them across the component on command, like a real-life marquee. You'll use `TCustomPanel` as the base class for this `TddgMarquee` component because it already has the basic built-in functionality you need, including a pretty 3D, beveled border.

`TddgMarquee` paints some text strings to a bitmap residing in memory and then copies portions of the memory bitmap to its own canvas to simulate a scrolling effect. It does this using the `BitBlt()` API function to copy a component-sized portion of the memory canvas to the component, starting at the top. Then, it moves down a couple pixels on the memory canvas and copies that image to the control. It moves down again, copies again, and repeats the process over and over so that the entire contents of the memory canvas appear to scroll through the component.

Now is the time to identify any additional classes you might need to integrate into the `TddgMarquee` component in order to bring it to life. There are really only two such classes. First, you need the `TStringList` class to hold all the strings you want to scroll. Second, you must have a memory bitmap on which you can render all the text strings. VCL's own `TBitmap` component will work nicely for this purpose.

Writing the Component

As with the previous components in this chapter, the code for `TddgMarquee` should be approached with a logical plan of attack. In this case, we break up the code work into reasonable parts. The `TddgMarquee` component: can be divided into five major parts:

- The mechanism that renders the text onto the memory canvas
- The mechanism that copies the text from the memory canvas to the marquee window
- The timer that keeps track of when and how to scroll the window to perform the animation
- The class constructor, destructor, and associated methods
- The finishing touches, such as various helper properties and methods

Drawing on an Offscreen Bitmap

When creating an instance of `TBitmap`, you need to know how big it must be to hold the entire list of strings in memory. You do this by first figuring out how high each line of text will be and then multiplying by the number of lines. To find the height and spacing of a line of text in a particular font, use the `GetTextMetrics()` API function by passing it the canvas's handle. A `TTextMetric` record to be filled in by the function:

```
var
  Metrics: TTextMetric;
begin
  GetTextMetrics(Canvas.Handle, Metrics);
```

NOTE

The `GetTextMetrics()` API function modifies a `TTextMetric` record that contains a great deal of quantitative information about a device context's currently selected font. This function gives you information not only on font height and width but also on whether the font is boldfaced, italicized, struck out, or even what the character set name is.

The `TextHeight()` method of `TCanvas` won't work here. That method only determines the height of a specific line of text rather than the spacing for the font in general.

The `tmHeight` field of the Metrics record gives the height of a character cell in the canvas's current font. If you add to that value the `tmInternalLeading` field—to allow for some space between lines—you get the height for each line of text to be drawn on the memory canvas:

```
LineHi := Metrics.tmHeight + Metrics.tmInternalLeading;
```

The height necessary for the memory canvas then can be determined by multiplying LineHi by the number of lines of text and adding that value to two times the height of the TddgMarquee control (to create the blank space at the beginning and end of the marquee). Suppose that the TStringList in which all the strings live is called FItems; now place the memory canvas dimensions in a TRect structure:

```
var
  VRect: TRect;
begin
  { VRect rectangle represents entire memory bitmap }
  VRect := Rect(0, 0, Width, LineHi * FItems.Count + Height * 2);
end;
```

After being instantiated and sized, the memory bitmap is initialized further by setting the font to match the Font property of TddgMarquee, filling the background with a color determined by the Color property of TddgMarquee, and setting the Style property of Brush to bsClear.

TIP

When you render text on TCanvas, the text background is filled with the current color of TCanvas.Brush. To cause the text background to be invisible, set TCanvas.Brush.Style to bsClear.

Most of the preliminary work is now in place, so it's time to render the text on the memory bitmap. The most straightforward way to output the text onto a canvas is to use the TextOut() method of TCanvas; however, you have more control over the formatting of the text when you use the more complex DrawText() API function. Because it requires control over justification, TddgMarquee will use the DrawText() function. An enumerated type is ideal to represent the text justification:

```
type
  TJustification = (tjCenter, tjLeft, tjRight);
```

The following code shows the PaintLine() method for TddgMarquee, which makes use of DrawText() to render text onto the memory bitmap. In this method, FJust represents an instance variable of type TJustification. Here's the code:

```
procedure TddgMarquee.PaintLine(R: TRect; LineNum: Integer);
{ this method is called to paint each line of text onto MemBitmap }
const
  Flags: array[TJustification] of DWORD = (DT_CENTER, DT_LEFT, DT_RIGHT);
var
  S: string;
```

```
begin
  { Copy next line to local variable for clarity }
  S := FItems.Strings[LineNum];
  { Draw line of text onto memory bitmap }
  DrawText(MemBitmap.Canvas.Handle, PChar(S), Length(S), R,
    Flags[FJust] or DT_SINGLELINE or DT_TOP);
end;
```

Painting the Component

Now that you know how to create the memory bitmap and paint text onto it, the next step is learning how to copy that text to the `TddgMarquee` canvas.

The `Paint()` method of a component is invoked in response to a Windows `WM_PAINT` message. The `Paint()` method is what gives your component life; you use the `Paint()` method to paint, draw, and fill to determine the graphical appearance of your components.

The job of `TddgMarquee.Paint()` is to copy the strings from the memory canvas to the canvas of `TddgMarquee`. This feat is accomplished by the `BitBlt()` API function, which copies the bits from one device context to another.

To determine whether `TddgMarquee` is currently running, the component will maintain a Boolean instance variable called `FActive` that reveals whether the marquee's scrolling capability has been activated. Therefore, the `Paint()` method paints differently depending on whether the component is active:

```
procedure TddgMarquee.Paint;
{ this virtual method is called in response to a }
{ Windows paint message }
begin
  if FActive then
    { Copy from memory bitmap to screen }
    BitBlt(Canvas.Handle, 0, 0, InsideRect.Right, InsideRect.Bottom,
      MemBitmap.Canvas.Handle, 0, CurrLine, srcCopy)
  else
    inherited Paint;
end;
```

If the marquee is active, the component uses the `BitBlt()` function to paint a portion of the memory canvas onto the `TddgMarquee` canvas. Notice the `CurrLine` variable, which is passed as the next-to-last parameter to `BitBlt()`. The value of this parameter determines which portion of the memory canvas to transfer onto the screen. By continuously incrementing or decrementing the value of `CurrLine`, you can give `TddgMarquee` the appearance that the text is scrolling up or down.

Animating the Marquee

The visual aspects of the TddgMarquee component are now in place. The rest of the work involved in getting the component working is just hooking up the plumbing, so to speak. At this point, TddgMarquee requires some mechanism to change the value of CurrLine every so often and to repaint the component. This trick can be accomplished fairly easily using Delphi's TTimer component.

Before you can use TTimer, of course, you must create and initialize the class instance. TddgMarquee will have a TTimer instance called FTimer, and you'll initialize it in a procedure called DoTimer:

```
procedure DoTimer;
{ procedure sets up TddgMarquee's timer }
begin
  FTimer := TTimer.Create(Self);
  with FTimer do
  begin
    Enabled := False;
    Interval := TimerInterval;
    OnTimer := DoTimerOnTimer;
  end;
end;
```

In this procedure, FTimer is created, and it's disabled initially. Its Interval property then is assigned to the value of a constant called TimerInterval. Finally, the OnTimer event for FTimer is assigned to a method of TddgMarquee called DoTimerOnTimer. This is the method that will be called when an OnTimer event occurs.

NOTE

When assigning values to events in your code, you need to follow two rules:

- The procedure you assign to the event must be a method of some object instance. It can't be a standalone procedure or function.
- The method you assign to the event must accept the same parameter list as the event type. For example, the OnTimer event for TTimer is of type TNotifyEvent. Because TNotifyEvent accepts one parameter, Sender, of type TObject, any method you assign to OnTimer must also take one parameter of type TObject.

The DoTimerOnTimer() method is defined as follows:

```
procedure TddgMarquee.DoTimerOnTimer(Sender:  TObject);
{ This method is executed in response to a timer event }
```

```
begin
  IncLine;
  { only repaint within borders }
  InvalidateRect(Handle, @InsideRect, False);
end;
```

In this method, a procedure named `IncLine()` is called; this procedure increments or decrements the value of `CurrLine` as necessary. Then the `InvalidateRect()` API function is called to "invalidate" (or *repaint*) the interior portion of the component. We chose to use `InvalidateRect()` rather than the `Invalidate()` method of `TCanvas` because `Invalidate()` causes the entire canvas to be repainted rather than just the portion within a defined rectangle, as is the case with `InvalidateRect()`. This method, because it doesn't continuously repaint the entire component, eliminates much of the flicker that would otherwise occur. Remember: Flicker is bad.

The `IncLine()` method, which updates the value of `CurrLine` and detects whether scrolling has completed, is defined as follows:

```
procedure TddgMarquee.IncLine;
{ this method is called to increment a line }
begin
  if not FScrollDown then       // if Marquee is scrolling upward
  begin
    { Check to see if marquee has scrolled to end yet }
    if FItems.Count * LineHi + ClientRect.Bottom -
      ScrollPixels  >= CurrLine then
      { not at end, so increment current line }
      Inc(CurrLine, ScrollPixels)
    else SetActive(False);
  end
  else begin                   // if Marquee is scrolling downward
    { Check to see if marquee has scrolled to end yet }
    if CurrLine >= ScrollPixels then
      { not at end, so decrement current line }
      Dec(CurrLine, ScrollPixels)
    else SetActive(False);
  end;
end;
```

The constructor for `TddgMarquee` is actually quite simple. It calls the inherited `Create()` method, creates a `TStringList` instance, sets up `FTimer`, and then sets all the default values for the instance variables. Once again, you must remember to call the inherited `Create()` method in your components. Failure to do so means your components will miss out on important and useful functionality, such as handle and canvas creation, streaming, and Windows message response. The following code shows the `TddgMarquee` constructor, `Create()`:

```
constructor TddgMarquee.Create(AOwner: TComponent);
{ constructor for TddgMarquee class }

  procedure DoTimer;
  { procedure sets up TddgMarquee's timer }
  begin
    FTimer := TTimer.Create(Self);
    with FTimer do
    begin
      Enabled := False;
      Interval := TimerInterval;
      OnTimer := DoTimerOnTimer;
    end;
  end;

begin
  inherited Create(AOwner);
  FItems := TStringList.Create;  { instantiate string list }
  DoTimer;                       { set up timer }
  { set instance variable default values }
  Width := 100;
  Height := 75;
  FActive := False;
  FScrollDown := False;
  FJust := tjCenter;
  BevelWidth := 3;
end;
```

The `TddgMarquee` destructor is even simpler: The method deactivates the component by passing `False` to the `SetActive()` method, frees the timer and the string list, and then calls the inherited `Destroy()` method:

```
destructor TddgMarquee.Destroy;
{ destructor for TddgMarquee class }
begin
  SetActive(False);
  FTimer.Free;             // free allocated objects
  FItems.Free;
  inherited Destroy;
end;
```

TIP

As a rule of thumb, when you override constructors, you usually call `inherited` first, and when you override destructors, you usually call `inherited` last. It might help to remember "first in, last out." This ensures that the class has been set up before you modify it and that all dependent resources have been cleaned up before you dispose of the class.

Exceptions to this rule exist; however, you should generally stick to it unless you have good reason not to.

The `SetActive()` method, which is called by both the `IncLine()` method and the destructor (in addition to serving as the writer for the `Active` property), serves as a vehicle that starts and stops the marquee scrolling up the canvas:

```
procedure TddgMarquee.SetActive(Value: Boolean);
{ called to activate/deactivate the marquee }
begin
  if Value and (not FActive) and (FItems.Count > 0) then
  begin
    FActive := True;                // set active flag
    MemBitmap := TBitmap.Create;
    FillBitmap;                     // Paint Image on bitmap
    FTimer.Enabled := True;         // start timer
  end
  else if (not Value) and FActive then
  begin
    FTimer.Enabled := False;   // disable timer,
    if Assigned(FOnDone)       // fire OnDone event,
      then FOnDone(Self);
    FActive := False;          // set FActive to False
    MemBitmap.Free;            // free memory bitmap
    Invalidate;                // clear control window
  end;
end;
```

An important feature of `TddgMarquee` that's lacking thus far is an event that tells the user when scrolling is complete. Never fear—this feature is very straightforward to add by way of an event: `FOnDone`. The first step to adding an event to your component is to declare an instance variable of some event type in the `private` portion of the class definition. You'll use the `TNotifyEvent` type for the `FOnDone` event:

```
FOnDone: TNotifyEvent;
```

The event should then be declared in the `published` part of the class as a property:

```
property OnDone: TNotifyEvent read FOnDone write FOnDone;
```

Recall that the `read` and `write` directives specify from which function or variable a given property should get or set its value.

Taking just these two small steps will cause an entry for `OnDone` to be displayed in the Events page of the Object Inspector at design time. The only other thing that needs to be done is to call the user's handler for `OnDone` (if a method is assigned to `OnDone`), as demonstrated by `TddgMarquee` with this line of code in the `Deactivate()` method:

```
if Assigned(FOnDone) then FOnDone(Self); // fire OnDone event
```

This line basically reads, "If the component user has assigned a method to the `OnDone` event, call that method and pass the `TddgMarquee` class instance (`Self`) as a parameter."

Listing 12.2 shows the completed source code for the `Marquee` unit. Notice that because the component descends from a `TCustom`*`XXX`* class, you need to publish many of the properties provided by `TCustomPanel`.

LISTING 12.2 `Marquee.pas`—Illustrates the `TddgMarquee` Component

```
unit Marquee;

interface

uses
  SysUtils, Windows, Classes, Forms, Controls, Graphics,
  Messages, ExtCtrls, Dialogs;

const
  ScrollPixels = 3;     // num of pixels for each scroll
  TimerInterval = 50;   // time between scrolls in ms

type
  TJustification = (tjCenter, tjLeft, tjRight);

  EMarqueeError = class(Exception);

  TddgMarquee = class(TCustomPanel)
  private
    MemBitmap: TBitmap;
    InsideRect: TRect;
    FItems: TStringList;
    FJust: TJustification;
```

LISTING 12.2 Continued

```
    FScrollDown: Boolean;
    LineHi : Integer;
    CurrLine : Integer;
    VRect: TRect;
    FTimer: TTimer;
    FActive: Boolean;
    FOnDone: TNotifyEvent;
    procedure SetItems(Value: TStringList);
    procedure DoTimerOnTimer(Sender:  TObject);
    procedure PaintLine(R: TRect; LineNum: Integer);
    procedure SetLineHeight;
    procedure SetStartLine;
    procedure IncLine;
    procedure SetActive(Value: Boolean);
  protected
    procedure Paint; override;
    procedure FillBitmap; virtual;
  public
    property Active: Boolean read FActive write SetActive;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property ScrollDown: Boolean read FScrollDown write FScrollDown;
    property Justify: TJustification read FJust write FJust default tjCenter;
    property Items: TStringList read FItems write SetItems;
    property OnDone: TNotifyEvent read FOnDone write FOnDone;
    { Publish inherited properties: }
    property Align;
    property Alignment;
    property BevelInner;
    property BevelOuter;
    property BevelWidth;
    property BorderWidth;
    property BorderStyle;
    property Color;
    property Ctl3D;
    property Font;
    property Locked;
    property ParentColor;
    property ParentCtl3D;
    property ParentFont;
    property Visible;
    property OnClick;
    property OnDblClick;
```

LISTING 12.2 Continued

```
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnResize;
  end;

implementation

constructor TddgMarquee.Create(AOwner: TComponent);
{ constructor for TddgMarquee class }

  procedure DoTimer;
  { procedure sets up TddgMarquee's timer }
  begin
    FTimer := TTimer.Create(Self);
    with FTimer do
    begin
      Enabled := False;
      Interval := TimerInterval;
      OnTimer := DoTimerOnTimer;
    end;
  end;

begin
  inherited Create(AOwner);
  FItems := TStringList.Create;  { instantiate string list }
  DoTimer;                       { set up timer }
  { set instance variable default values }
  Width := 100;
  Height := 75;
  FActive := False;
  FScrollDown := False;
  FJust := tjCenter;
  BevelWidth := 3;
end;

destructor TddgMarquee.Destroy;
{ destructor for TddgMarquee class }
begin
  SetActive(False);
  FTimer.Free;             // free allocated objects
  FItems.Free;
  inherited Destroy;
end;
```

LISTING 12.2 Continued

```
procedure TddgMarquee.DoTimerOnTimer(Sender: TObject);
{ This method is executed in response to a timer event }
begin
  IncLine;
  { only repaint within borders }
  InvalidateRect(Handle, @InsideRect, False);
end;

procedure TddgMarquee.IncLine;
{ this method is called to increment a line }
begin
  if not FScrollDown then       // if Marquee is scrolling upward
  begin
    { Check to see if marquee has scrolled to end yet }
    if FItems.Count * LineHi + ClientRect.Bottom -
      ScrollPixels  >= CurrLine then
      { not at end, so increment current line }
      Inc(CurrLine, ScrollPixels)
    else SetActive(False);
  end
  else begin                   // if Marquee is scrolling downward
    { Check to see if marquee has scrolled to end yet }
    if CurrLine >= ScrollPixels then
      { not at end, so decrement current line }
      Dec(CurrLine, ScrollPixels)
    else SetActive(False);
  end;
end;

procedure TddgMarquee.SetItems(Value: TStringList);
begin
  if FItems <> Value then
    FItems.Assign(Value);
end;

procedure TddgMarquee.SetLineHeight;
{ this virtual method sets the LineHi instance variable }
var
  Metrics : TTextMetric;
begin
  { get metric info for font }
  GetTextMetrics(Canvas.Handle, Metrics);
  { adjust line height }
  LineHi := Metrics.tmHeight + Metrics.tmInternalLeading;
end;
```

LISTING 12.2 Continued

```
procedure TddgMarquee.SetStartLine;
{ this virtual method initializes the CurrLine instance variable }
begin
  // initialize current line to top if scrolling up, or...
  if not FScrollDown then CurrLine := 0
  // bottom if scrolling down
  else CurrLine := VRect.Bottom - Height;
end;

procedure TddgMarquee.PaintLine(R: TRect; LineNum: Integer);
{ this method is called to paint each line of text onto MemBitmap }
const
  Flags: array[TJustification] of DWORD = (DT_CENTER, DT_LEFT, DT_RIGHT);
var
  S: string;
begin
  { Copy next line to local variable for clarity }
  S := FItems.Strings[LineNum];
  { Draw line of text onto memory bitmap }
  DrawText(MemBitmap.Canvas.Handle, PChar(S), Length(S), R,
    Flags[FJust] or DT_SINGLELINE or DT_TOP);
end;

procedure TddgMarquee.FillBitmap;
var
  y, i : Integer;
  R: TRect;
begin
  SetLineHeight;                 // set height of each line
  { VRect rectangle represents entire memory bitmap }
  VRect := Rect(0, 0, Width, LineHi * FItems.Count + Height * 2);
  { InsideRect rectangle represents interior of beveled border }
  InsideRect := Rect(BevelWidth, BevelWidth, Width - (2 * BevelWidth),
    Height - (2 * BevelWidth));
  R := Rect(InsideRect.Left, 0, InsideRect.Right, VRect.Bottom);
  SetStartLine;
  MemBitmap.Width := Width;      // initialize memory bitmap
  with MemBitmap do
  begin
    Height := VRect.Bottom;
    with Canvas do
    begin
      Font := Self.Font;
```

LISTING 12.2 Continued

```
      Brush.Color := Color;
      FillRect(VRect);
      Brush.Style := bsClear;
    end;
  end;
  y := Height;
  i := 0;
  repeat
    R.Top := y;
    PaintLine(R, i);
    { increment y by the height (in pixels) of a line }
    inc(y, LineHi);
    inc(i);
  until i >= FItems.Count;      // repeat for all lines
end;

procedure TddgMarquee.Paint;
{ this virtual method is called in response to a }
{ Windows paint message }
begin
  if FActive then
    { Copy from memory bitmap to screen }
    BitBlt(Canvas.Handle, 0, 0, InsideRect.Right, InsideRect.Bottom,
      MemBitmap.Canvas.Handle, 0, CurrLine, srcCopy)
  else
    inherited Paint;
end;

procedure TddgMarquee.SetActive(Value: Boolean);
{ called to activate/deactivate the marquee }
begin
  if Value and (not FActive) and (FItems.Count > 0) then
  begin
    FActive := True;                // set active flag
    MemBitmap := TBitmap.Create;
    FillBitmap;                     // Paint Image on bitmap
    FTimer.Enabled := True;         // start timer
  end
  else if (not Value) and FActive then
  begin
    FTimer.Enabled := False;   // disable timer,
    if Assigned(FOnDone)       // fire OnDone event,
      then FOnDone(Self);
    FActive := False;          // set FActive to False
```

Listing 12.2 Continued

```
    MemBitmap.Free;            // free memory bitmap
    Invalidate;                // clear control window
  end;
end;

end.
```

Tip

Notice the `default` directive and value used with the `Justify` property of `TddgMarquee`. This use of `default` optimizes streaming of the component, which improves the component's design-time performance. You can give default values to properties of any ordinal type (`Integer`, `Word`, `Longint`, as well as enumerated types, for example), but you can't give them to nonordinal property types such as strings, floating-point numbers, arrays, records, and classes.

You also need to initialize the default values for the properties in your constructor. Failure to do so will cause streaming problems.

Testing `TddgMarquee`

Although it's very exciting to finally have this component written and in the testing stages, don't get carried away by trying to add it to the Component Palette just yet. It has to be debugged first. You should do all preliminary testing with the component by creating a project that creates and uses a dynamic instance of the component. Listing 12.3 depicts the main unit for a project called `TestMarq`, which is used to test the `TddgMarquee` component. This simple project consists of a form that contains two buttons.

Listing 12.3 `TestU.pas`—Tests the `TddgMarquee` Component

```
unit Testu;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Marquee, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
```

LISTING 12.3 Continued

```
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    Marquee1: TddgMarquee;
    procedure MDone(Sender: TObject);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.MDone(Sender: TObject);
begin
  Beep;
end;

procedure TForm1.FormCreate(Sender:  TObject);
begin
  Marquee1 := TddgMarquee.Create(Self);
  with Marquee1 do
  begin
    Parent := Self;
    Top := 10;
    Left := 10;
    Height := 200;
    Width := 150;
    OnDone := MDone;
    Show;
    with Items do
    begin
      Add('Greg');
      Add('Peter');
      Add('Bobby');
      Add('Marsha');
      Add('Jan');
      Add('Cindy');
    end;
  end;
end;
```

LISTING 12.3 Continued

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Marquee1.Active := True;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Marquee1.Active :=  False;
end;

end.
```

TIP

Always create a test project for your new components. *Never* try to do initial testing on a component by adding it to the Component Palette. By trying to debug a component that resides on the palette, not only will you waste time with a lot of gratuitous package rebuilding, but it's possible to crash the IDE as a result of a bug in your component.

After you squash all the bugs you find in this program, it's time to add it to the Component Palette. As you might recall, doing so is easy: Simply choose Component, Install Component. . . from the main menu and then fill in the unit filename and package name in the Install Component dialog. Click OK and Delphi will rebuild the package to which the component was added and update the Component Palette. Of course, your component will need to expose a `Register()` procedure in order to be placed on the Component Palette. The `TddgMarquee` component is registered in the `DDGReg.pas` unit of the `DDGDsgn` package on the CD-ROM accompanying this book.

Writing Property Editors

Chapter 11, "VCL Component Building," shows how properties are edited in the Object Inspector for most of the common property types. The means by which a property is edited is determined by its *property editor*. Several predefined property editors are used for the existing properties. However, there might be a situation in which none of the predefined editors meet your needs, such as when you've created a custom property. Given this situation, you'll need to create your own editor for that property.

You can edit properties in the Object Inspector in two ways. One is to allow the user to edit the value as a text string. The other is to use a dialog that performs the editing of the property. In some cases, you'll want to allow both editing capabilities for a single property.

Here are the steps required for writing a property editor:

1. Create a descendant property editor object.
2. Edit the property as text.
3. Edit the property as a whole with a dialog (optional).
4. Specify the property editor's attributes.
5. Register the property editor.

The following sections cover each of these steps.

Creating a Descendant Property Editor Object

Delphi defines several property editors in the unit `DesignEditors.pas`, all of which descend from the base class `TPropertyEditor`. When you create a property editor, your property editor must descend from `TPropertyEditor` or one of its descendants. Table 12.1 shows the `TPropertyEditor` descendants that are used with the existing properties.

TABLE 12.1 Property Editors Defined in `DesignEditors.pas`

Property Editor	*Description*
`TOrdinalProperty`	The base class for all ordinal property editors, such as `TIntegerProperty`, `TEnumProperty`, `TCharProperty`, and so on.
`TIntegerProperty`	The default property editor for integer properties of all sizes.
`TCharProperty`	The property editor for properties that are a `char` type and a subrange of `char`; that is, `'A'..'Z'`.
`TEnumProperty`	The default property for all user-defined enumerated types.
`TFloatProperty`	The default property editor for floating-point numeric properties.
`TStringProperty`	The default property editor for string type properties.
`TSetElementProperty`	The default property editor for individual `set` elements. Each element in the set is displayed as an individual Boolean option.

TABLE 12.1 Continued

Property Editor	*Description*
`TSetProperty`	The default property editor for `set` properties. The set expands into separate set elements for each element in the set.
`TClassProperty`	The default property editor for properties that are, themselves, objects.
`TMethodProperty`	The default property editor for properties that are method pointers—that is, *events*.
`TComponentProperty`	The default property editor for properties that refer to a component. This isn't the same as the `TClassProperty` editor. Instead, this editor allows the user to specify a component to which the property refers—that is, `ActiveControl`.
`TColorProperty`	The default property editor for properties of the type `TColor`.
`TFontNameProperty`	The default property editor for font names. This editor displays a drop-down list of fonts available on the system.
`TFontProperty`	The default property editor for properties of type `TFont`, which allows the editing of subproperties. `TFontProperty` allows the editing of subproperties because it derives from `TClassProperty`.
`TInt64Property`	The default property editor for all `Int64` and its derivatives.
`TNestedProperty`	This property editor uses its parent's property editor.
`TClassProperty`	The default property editor for objects.
`TMethodProperty`	The default property editor for methods.
`TInterfaceProperty`	The default property editor for interface references.
`TComponentNameProperty`	Property editor for the `Name` property. It restricts the `Name` property from being displayed when more than one component is selected.
`TDateProperty`	The default property editor for the date portion of a `TDateTime` type property.
`TTimePropery`	The property editor for the time portion of a `TDateTime` property.
`TDateTimeProperty`	The property editor for a `TDateTime` property type.
`TVariantProperty`	The property editor for variant types.

The property editor from which your property editor must descend depends on how the property is going to behave when it's edited. In some cases, for example, your property might require the same functionality as `TIntegerProperty`, but it might also require additional logic in the editing process. Therefore, it would be logical that your property editor descend from `TIntegerProperty`.

TIP

Bear in mind that there are cases in which you don't need to create a property editor that depends on your property type. For example, subrange types are checked automatically (for example, `1..10` is checked for by `TIntegerProperty`), enumerated types get drop-down lists automatically, and so on. You should try to use type definitions instead of custom property editors because they're enforced by the language at compile time as well as by the default property editors.

Editing the Property As Text

The property editor has two basic purposes: One is to provide a means for the user to edit the property; this is obvious. The other not-so-obvious purpose is to provide the string representation of the property value to the Object Inspector so that it can be displayed accordingly.

When you create a descendant property editor class, you must override the `GetValue()` and `SetValue()` methods. `GetValue()` returns the string representation of the property value for the Object Inspector to display. `SetValue()` sets the value based on its string representation as it's entered in the Object Inspector.

As an example, examine the definition of the `TIntegerProperty` class type as it's defined in `DSGNINTF.PAS`:

```
TIntegerProperty = class(TOrdinalProperty)
public
  function GetValue: string; override;
  procedure SetValue(const Value: string); override;
end;
```

Here, you see that the `GetValue()` and `SetValue()` methods have been overridden. The `GetValue()` implementation is as follows:

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

Here's the `SetValue()` implementation:

```
procedure TIntegerProperty.SetValue(const Value: String);
var
  L: Longint;
begin
  L := StrToInt(Value);
  with GetTypeData(GetPropType)^ do
    if (L < MinValue) or (L > MaxValue) then
      raise EPropertyError.CreateResFmt(SOutOfRange, [MinValue, MaxValue]);
  SetOrdValue(L);
end;
```

`GetValue()` returns the string representation of an integer property. The Object Inspector uses this value to display the property's value. `GetOrdValue()` is a method defined by `TPropertyEditor` and is used to retrieve the value of the property referenced by the property editor.

`SetValue()` takes the string value entered by the user and assigns it to the property in the correct format. `SetValue()` also performs some error checking to ensure that the value is within a specified range of values. This illustrates how you might perform error checking with your descendant property editors. The `SetOrdValue()` method assigns the value to the property referenced by the property editor.

`TPropertyEditor` defines several methods similar to `GetOrdValue()` for getting the string representation of various types. Additionally, `TPropertyEditor` contains the equivalent "set" methods for setting the values in their respective format. `TPropertyEditor` descendants inherit these methods. These methods are used for getting and setting the values of the properties that the property editor references. Table 12.2 shows these methods.

TABLE 12.2 Read/Write Property Methods for `TPropertyEditor`

Property Type	*"Get" Method*	*"Set" Method*
Floating point	`GetFloatValue()`	`SetFloatValue()`
Event	`GetMethodValue()`	`SetMethodValue()`
Ordinal	`GetOrdValue()`	`SetOrdValue()`
String	`GetStrValue()`	`SetStrValue()`
Variant	`GetVarValue()`	`SetVarValue()`, `SetVarValueAt()`

To illustrate creating a new property editor, we'll have some more fun with the solar system example introduced in the last chapter. This time, we've created a simple component, `TPlanet`, to represent a single planet. `TPlanet` contains the property `PlanetName`. Internal storage for

PlanetName is going to be of type integer and will hold the planet's position in the solar system. However, it will be displayed in the Object Inspector as the name of the planet.

So far this sounds easy, but here's the catch: We want to enable the user to type two values to represent the planet. The user should be able to type the planet name as a string, such as Venus, VENUS, or VeNuS. He should also be able to type the position of the planet in the solar system. Therefore, for the planet Venus, the user would type the numeric value 2.

The component TPlanet is as follows:

```
type
  TPlanetName = type Integer;

  TPlanet = class(TComponent)
  private
    FPlanetName: TPlanetName;
  published
    property PlanetName: TPlanetName read FPlanetName write FPlanetName;
  end;
```

As you can see, there's not much to this component. It has only one property: PlanetName of the type TPlanetName. Here, the special definition of TPlanetName is used so that it's given its own runtime type information, yet it's still treated like an integer type.

This functionality doesn't come from the TPlanet component; rather, it comes from the property editor for the TPlanetName property type. This property editor is shown in Listing 12.4.

LISTING 12.4 PlanetPE.PAS—The Source Code for TPlanetNameProperty

```
unit PlanetPE;

interface

uses
  Windows, SysUtils, DsgnIntF;

type
  TPlanetNameProperty = class(TIntegerProperty)
  public
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;

implementation

const
  { Declare a constant array containing planet names }
```

LISTING 12.4 Continued

```
  PlanetNames: array[1..9] of String[7] =
   ('Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn',
    'Uranus', 'Neptune', 'Pluto');


function TPlanetNameProperty.GetValue: string;
begin
  Result := PlanetNames[GetOrdValue];
end;

procedure TPlanetNameProperty.SetValue(const Value: String);
var
  PName: string[7];
  i, ValErr: Integer;
begin
  PName := UpperCase(Value);
  i := 1;
  { Compare the Value with each of the planet names in the PlanetNames
    array. If a match is found, the variable i will be less than 10 }
  while (PName <> UpperCase(PlanetNames[i])) and (i < 10) do
    inc(i);
  { If i is less than 10, a valid planet name was entered. Set the value
    and exit this procedure. }
  if i < 10 then  // A valid planet name was entered.
  begin
    SetOrdValue(i);
    Exit;
  end
  { If i was greater than 10, the user might have typed in a planet number, or
    an invalid planet name. Use the Val function to test if the user typed in
    a number, if an ValErr is non-zero, an invalid name was entered,
    otherwise, test the range of the number entered for (0 < i < 10). }
  else begin
    Val(Value, i, ValErr);
    if ValErr <> 0 then
      raise Exception.Create(Format('Sorry, Never heard of the planet %s.',
        [Value]));
    if (i <= 0) or (i >= 10) then
       raise Exception.Create('Sorry, that planet is not in OUR solar
system.');
    SetOrdValue(i);
  end;
end;

end.
```

First, we create our property editor, `TPlanetNameProperty`, which descends from `TIntegerProperty`. By the way, it's necessary to include the `DesignEditors and DesignIntf` units in the `uses` clause of this unit.

We've defined an array of string constants to represent the planets in the solar system by their position from the sun. These strings will be used to display the string representation of the planet in the Object Inspector.

As stated earlier, we have to override the `GetValue()` and `SetValue()` methods. In the `GetValue()` method, we just return the string from the `PlanetNames` array, which is indexed by the property value. Of course, this value must be within the range of 1–9. We handle this by not allowing the user to enter a number out of that range in the `SetValue()` method.

`SetValue()` gets a string as it's entered from the Object Inspector. This string can either be a planet name or a number representing a planet's position. If a valid planet name or planet number is entered, as determined by the code logic, the value assigned to the property is specified by the `SetOrdValue()` method. If the user enters an invalid planet name or planet position, the code raises the appropriate exception.

That's all there is to defining a property editor. Well, not quite; it must still be registered before it becomes known to the property to which you want to attach it.

Registering the New Property Editor

You register a property editor by using the appropriately named procedure `RegisterPropertyEditor()`. This method is declared as follows:

```
procedure RegisterPropertyEditor(PropertyType: PTypeInfo;
  ComponentClass: TClass; const PropertyName: string;
  EditorClass: TPropertyEditorClass);
```

The first parameter, `PropertyType`, is a pointer to the Runtime Type Information of the property being edited. This information is obtained by using the `TypeInfo()` function. `ComponentClass` is used to specify to which class this property editor will apply. `PropertyName` specifies the property name on the component, and the `EditorClass` parameter specifies the type of property editor to use. For the `TPlanet.PlanetName` property, the function looks like this:

```
RegisterPropertyEditor(TypeInfo(TPlanetName), TPlanet, 'PlanetName',
  TPlanetNameProperty);
```

TIP

Although, for the purpose of illustration, this particular property editor is registered for use only with the `TPlanet` component and `'PlanetName'` property name, you might choose to be less restrictive in registering your custom property editors. By setting the `ComponentClass` parameter to `nil` and the `PropertyName` parameter to `''`, your property editor will work for any component's property of type `TPlanetName`.

You can register the property editor along with the registration of the component in the component's unit, as shown in Listing 12.5.

LISTING 12.5 `Planet.pas`—The `TPlanet` Component

```
unit Planet;

interface

uses
  Classes, SysUtils;

type
  TPlanetName = type Integer;

  TddgPlanet = class(TComponent)
  private
    FPlanetName: TPlanetName;
  published
    property PlanetName: TPlanetName read FPlanetName write FPlanetName;
  end;

implementation

end.
```

TIP

Placing the property editor registration in the `Register()` procedure of the component's unit will force all the property editor code to be linked in with your component when it's put into a package. For complex components, the design-time tools might take up more code space than the components themselves. Although code size isn't much of an issue for a small component such as this, keep in mind that everything that's listed in the `interface` section of your component's unit (such as the `Register()` procedure) as well as everything it touches (such as the property editor class type) will tag along with your component when it's compiled into a package. For this reason, you might want to perform registration of your property editor in a separate unit. Furthermore, some component writers choose to create both design-time and runtime packages for their components, whereas the property editors and other design-time tools reside only in the design-time package. You'll note that the packages containing this book's code do this using the `DdgRT6` runtime package and the `DDGDT6` design package.

Editing the Property as a Whole with a Dialog

Sometimes it's necessary to provide more editing capability than the in-place editing of the Object Inspector. This is when it becomes necessary to use a dialog as a property editor. An example of this would be the `Font` property for most Delphi components. Certainly, the makers of Delphi could have forced the user to type the font name and other font-related information. However, it would be unreasonable to expect the user to know this information. It's far easier to provide the user with a dialog where he can set these various attributes related to the font and see an example before selecting it.

To illustrate using a dialog to edit a property, we're going to extend the functionality of the `TddgRunButton` component created in Chapter 11. Now the user will be able to click an ellipsis button in the Object Inspector for the `CommandLine` property, which will invoke an Open File dialog from which the user can select a file for `TddgRunButton` to represent.

Sample Dialog Property Editor: Extending `TddgRunButton`

The `TddgRunButton` component is shown in Listing 11.13 in Chapter 11. We won't show it again here, but there are a few things we want to point out. The `TddgRunButton.CommandLine` property is of type `TCommandLine`, which is defined as follows:

```
TCommandLine = type string;
```

Again, this is a special declaration that attaches unique Runtime Type Information to this special type. This allows you to define a property editor specific to the `TCommandLine` type. Additionally, because `TCommandLine` is treated as a string, the property editor for editing string properties still applies to the `TCommandLine` type as well.

Also, as we illustrate the property editor for the `TCommandLine` type, keep in mind that `TddgRunButton` already has included the necessary error checking of property assignments in the properties' access methods. Therefore, it isn't necessary to repeat this error checking in the property editor's logic.

Listing 12.6 shows the definition of the `TCommandLineProperty` property editor.

LISTING 12.6 `RunBtnPE.pas`—The Unit Containing `TCommandLineProperty`

```
unit runbtnpe;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, DsgnIntF, TypInfo;

type

  { Descend from the TStringProperty class so that this editor
    inherits the string property editing capabilities }
  TCommandLineProperty = class(TStringProperty)
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;

implementation

function TCommandLineProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]; // Display a dialog in the Edit method
end;

procedure TCommandLineProperty.Edit;
{ The Edit method displays a TOpenDialog from which the user obtains
  an executable file name that gets assigned to the property }
var
  OpenDialog: TOpenDialog;
begin
  { Create the TOpenDialog }
  OpenDialog := TOpenDialog.Create(Application);
  try
    { Show only executable files }
    OpenDialog.Filter := 'Executable Files|*.EXE';
    { If the user selects a file, then assign it to the property.  }
    if OpenDialog.Execute then
      SetStrValue(OpenDialog.FileName);
  finally
    OpenDialog.Free // Free the TOpenDialog instance.
  end;
end;


end.
```

Examination of `TCommandLineProperty` shows that the property editor itself is very simple. First, notice that it descends from `TStringProperty` so that the string-editing capabilities are maintained. Therefore, in the Object Inspector, it isn't necessary to invoke the dialog. The user can just type the command line directly. Also, we didn't override the `SetValue()` and `GetValue()` methods because `TStringProperty` already handles this correctly. However, it was necessary to override the `GetAttributes()` method in order for the Object Inspector to know that this property is capable of being edited with a dialog. `GetAttributes()` merits further discussion.

Specifying the Property Editor's Attributes

Every property editor must tell the Object Inspector how a property is to be edited and what special attributes (if any) must be used when editing a property. Most of the time, the inherited attributes from a descendant property editor will suffice. In certain circumstances, however, you must override the `GetAttributes()` method of `TPropertyEditor`, which returns a set of property attribute flags (`TPropertyAttribute` flags) that indicate special property-editing attributes. The various `TPropertyAttribute` flags are shown in Table 12.3.

TABLE 12.3 `TPropertyAttribute` Flags

Attribute	*How the Property Editor Works with the Object Inspector*
`paValueList`	Returns an enumerated list of values for the property. The `GetValues()` method populates the list. A drop-down arrow button appears to the right of the property value. This applies to enumerated properties such as `TForm.BorderStyle` and integer `const` groups such as `TColor` and `TCharSet`.
`paSubProperties`	Subproperties are displayed indented below the current property in outline format. `paValueList` must also be set. This applies to set properties and class properties such as `TOpenDialog.Options` and `TForm.Font`.
`paDialog`	An ellipsis button is displayed to the right of the property in the Object Inspector, which, when clicked, causes the property editor's `Edit()` method to invoke a dialog. This applies to properties such as `TForm.Font`.
`paMultiSelect`	Properties are displayed when more than one component is selected on the Form Designer, allowing the user to change the property values for multiple components at once. Some properties aren't appropriate for this capability, such as the `Name` property.
`paAutoUpdate`	`SetValue()` is called on each change made to the property. If this flag isn't set, `SetValue()` is called when the user presses Enter or moves off the property in the Object Inspector. This applies to properties such as `TForm.Caption`.

TABLE 12.3 Continued

Attribute	*How the Property Editor Works with the Object Inspector*
`paFullWidthName`	Tells the Object Inspector that the value doesn't need to be rendered and, as such, the name should be rendered the full width of the inspector.
`paSortList`	The Object Inspector sorts the list returned by `GetValues()`.
`paReadOnly`	The property value can't be changed.
`paRevertable`	The property can be reverted to its original value. Some properties, such as nested properties, shouldn't be reverted. `TFont` is an example of this.

NOTE

You should take a look at `DesignEditors.pas` and examine which `TPropertyAttribute` flags are set for various property editors.

Setting the `paDialog` Attribute for `TCommandLineProperty`

Because `TCommandLineProperty` is to display a dialog, you must tell the Object Inspector to use this capability by setting the `paDialog` attribute in the `TCommandLineProperty.GetAttributes()` method. This will place an ellipsis button to the right of the `CommandLine` property value in the Object Inspector. When the user clicks this button, the `TCommandLineProperty.Edit()` method will be called.

Registering the `TCommandLineProperty`

The final step required for implementing the `TCommandLineProperty` property editor is to register it using the `RegisterProperyEditor()` procedure discussed earlier in this chapter. This procedure was added to the `Register()` procedure in `DDGReg.pas` in the `DDGDsgn` package:

```
RegisterComponents('DDG', [TddgRunButton]);
  RegisterPropertyEditor(TypeInfo(TCommandLine), TddgRunButton,
     '', TCommandLineProperty);
```

Also, note that the units `DsgnIntf` and `RunBtnPE` had to be added to the `uses` clause.

Component Editors

Component editors extend the design-time behavior of your components by allowing you to add items to the local menu associated with a particular component and by allowing you to change the default action when a component is double-clicked in the Form Designer. You might already be familiar with component editors without knowing it if you've ever used the fields editor provided with the `TTable`, `TQuery`, and `TStoredProc` components.

TComponentEditor

You might not be aware of this, but a different component editor is created for each component that's selected in the Form Designer. The type of component editor created depends on the component's type, although all component editors descend from `TComponentEditor`. This class is defined in the `DesignEditors` unit as follows:

```
  TComponentEditor = class(TBaseComponentEditor, IComponentEditor)
  private
    FComponent: TComponent;
    FDesigner: IDesigner;
  public
    constructor Create(AComponent: TComponent; ADesigner: IDesigner); override;
    procedure Edit; virtual;
    procedure ExecuteVerb(Index: Integer); virtual;
    function GetComponent: TComponent;
    function GetDesigner: IDesigner;
    function GetVerb(Index: Integer): string; virtual;
    function GetVerbCount: Integer; virtual;
    function IsInInlined: Boolean;
procedure Copy; virtual;
    procedure PrepareItem(Index: Integer; const AItem: IMenuItem); virtual;
    property Component: TComponent read FComponent;
    property Designer: IDesigner read GetDesigner;
  end;
```

Properties

The `Component` property of `TComponentEditor` is the instance of the component you're in the process of editing. Because this property is of the generic `TComponent` type, you must typecast the property in order to access fields introduced by descendant classes.

The `Designer` property is the instance of `IDesigner` that's currently hosting the application at design time. You'll find the complete definition for this class in the `DesignEditors.pas` unit.

Methods

The `Edit()` method is called when the user double-clicks the component at design time. Often, this method will invoke some sort of design dialog. The default behavior for this method is to call `ExecuteVerb(0)` if `GetVerbCount()` returns a value of `1` or greater. You must call `Designer.Modified()` if you modify the component from this (or any) method.

The use of the term *verb* as it applies to object methods applies to actions an object can take. Delphi has no knowledge of new objects or components initially, and needs to "learn" about them as they are added. With this in mind, it was designed with several methods that can be used to identify an object's actions. The `GetVerbCount`, `GetVerb`, and `ExecuteVerb` methods are generic methods intended for a wide variety of components, and they are the calls you will use to tell Delphi about your component.

The `GetVerbCount()` method is called to retrieve the number of items that are to be added to the local menu.

`GetVerb()` accepts an integer, `Index`, and returns a string containing the text that should appear on the local menu in the position corresponding to `Index`.

When an item is chosen from the local menu, the `ExecuteVerb()` method is called. This method receives the zero-based index of the item selected from the local menu in the `Index` parameter. You should respond by performing whatever action is necessary based on the verb the user selected from the local. menu.

The `Paste()` method is called whenever the component is pasted to the Clipboard. Delphi places the component's filed stream image on the Clipboard, but you can use this method to paste data on the Clipboard in a different type of format.

`TDefaultEditor`

If a custom component editor isn't registered for a particular component, that component will use the default component editor, `TDefaultEditor`. `TDefaultEditor` overrides the behavior of the `Edit()` method so that it searches the properties of the component and generates (or navigates to) the `OnCreate`, `OnChanged`, or `OnClick` event (whichever it finds first). If none of these events exists for this component, the first event defined will be selected.

A Simple Component

Consider the following simple custom component:

```
type
  TComponentEditorSample = class(TComponent)
  protected
    procedure SayHello; virtual;
    procedure SayGoodbye; virtual;
  end;

procedure TComponentEditorSample.SayHello;
begin
  MessageDlg('Hello, there!', mtInformation, [mbOk], 0);
end;

procedure TComponentEditorSample.SayGoodbye;
begin
  MessageDlg('See ya!', mtInformation, [mbOk], 0);
end;
```

As you can see, this little guy doesn't do much: It's a nonvisual component that descends directly from `TComponent`, and it contains two methods, `SayHello()` and `SayGoodbye()`, that simply display message dialogs.

A Simple Component Editor

To make the component a bit more exiting, you'll create a component editor that calls into the component and executes its methods at design time. The minimum `TComponentEditor` methods that must be overridden are `ExecuteVerb()`, `GetVerb()`, and `GetVerbCount()`. The code for this component editor is as follows:

```
type
  TSampleEditor = class(TComponentEditor)
  private
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

procedure TSampleEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TComponentEditorSample(Component).SayHello;    // call function
    1: TComponentEditorSample(Component).SayGoodbye;  // call function
  end;
end;

function TSampleEditor.GetVerb(Index: Integer): string;
begin
  case Index of
    0: Result := 'Hello';       // return hello string
    1: Result := 'Goodbye';     // return goodbye string
  end;
end;

function TSampleEditor.GetVerbCount: Integer;
begin
  Result := 2;      // two possible verbs
end;
```

The `GetVerbCount()` method returns `2`, indicating that there are two different verbs the component editor is prepared to execute. `GetVerb()` returns a string for each of these verbs to appear on the local menu. The `ExecuteVerb()` method calls the appropriate method inside the component, based on the verb index it receives as a parameter.

Registering a Component Editor

Like components and property editors, component editors must also be registered with the IDE within a unit's `Register()` method. To register a component editor, call the aptly named `RegisterComponentEditor()` procedure, which is defined as follows:

```
procedure RegisterComponentEditor(ComponentClass: TComponentClass;
  ComponentEditor: TComponentEditorClass);
```

The first parameter to this function is the component type for which you want to register a component editor, and the second parameter is the component editor itself.

Listing 12.7 shows the `CompEdit.pas` unit, which includes the component, component editor, and registration calls.

LISTING 12.7 `CompEdit.pas`—Illustrates a Component Editor

```
unit CompEdit;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  DsgnIntf;

type
  TComponentEditorSample = class(TComponent)
  protected
    procedure SayHello; virtual;
    procedure SayGoodbye; virtual;
  end;

  TSampleEditor = class(TComponentEditor)
  private
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

implementation

{ TComponentEditorSample }

procedure TComponentEditorSample.SayHello;
begin
  MessageDlg('Hello, there!', mtInformation, [mbOk], 0);
end;
```

LISTING 12.7 Continued

```
procedure TComponentEditorSample.SayGoodbye;
begin
  MessageDlg('See ya!', mtInformation, [mbOk],  0);
end;

{ TSampleEditor }

const
  vHello = 'Hello';
  vGoodbye = 'Goodbye';

procedure TSampleEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TComponentEditorSample(Component).SayHello;    // call function
    1: TComponentEditorSample(Component).SayGoodbye;  // call function
  end;
end;

function TSampleEditor.GetVerb(Index: Integer): string;
begin
  case Index of
    0: Result := vHello;       // return hello string
    1: Result := vGoodbye;      // return goodbye string
  end;
end;

function TSampleEditor.GetVerbCount: Integer;
begin
  Result := 2;      // two possible verbs
end;

end.
```

Streaming Nonpublished Component Data

Chapter 11 indicates that the Delphi IDE automatically knows how to stream the published properties of a component to and from a DFM file. What happens, however, when you have nonpublished data that you want to be persistent by keeping it in the DFM file? Fortunately, Delphi components provide a mechanism for writing and reading programmer-defined data to and from the DFM file.

Defining Properties

The first step in defining persistent nonpublished "properties" is to override a component's `DefineProperties()` method. This method is inherited from `TPersistent`, and it's defined as follows:

```
procedure DefineProperties(Filer: TFiler); virtual;
```

By default, this method handles reading and writing published properties to and from the DFM file. You can override this method, and, after calling `inherited`, you can call the `TFiler` method `DefineProperty()` or `DefineBinaryProperty()` once for each piece of data you want to become part of the DFM file. These methods are defined, respectively, as follows:

```
procedure DefineProperty(const Name: string; ReadData: TReaderProc;
    WriteData: TWriterProc; HasData: Boolean); virtual;

procedure DefineBinaryProperty(const Name: string; ReadData,
    WriteData: TStreamProc; HasData: Boolean); virtual;
```

`DefineProperty()` is used to make standard data types such as strings, integers, Booleans, chars, floats, and enumerated types persistent. `DefineBinaryProperty()` is used to provide access to raw binary data, such as a graphic or sound, written to the DFM file.

For both of these functions, the `Name` parameter identifies the property name that should be written to the DFM file. This doesn't have to be the same as the internal name of the data field you're accessing. The `ReadData` and `WriteData` parameters differ in type between `DefineProperty()` and `DefineBinaryProperty()`, but they serve the same purpose: These methods are called in order to write or read data to or from the DFM file. (We'll discuss these in more detail in just a moment.) The `HasData` parameter indicates whether the "property" has data that it needs to store.

The `ReadData` and `WriteData` parameters of `DefineProperty()` are of type `TReaderProc` and `TWriterProc`, respectively. These types are defined as follows:

```
type
  TReaderProc = procedure(Reader: TReader) of object;
  TWriterProc = procedure(Writer: TWriter) of object;
```

`TReader` and `TWriter` are specialized descendants of `TFiler` that have additional methods for reading and writing native types. Methods of these types provide the conduit between published component data and the DFM file.

The `ReadData` and `WriteData` parameters of `DefineBinaryProperty()` are of type `TStreamProc`, which is defined as follows:

```
type
  TStreamProc = procedure(Stream: TStream) of object;
```

Because TStreamProc type methods receive only TStream as a parameter, this allows you to read and write binary data very easily to and from the stream. Like the other method types described earlier, methods of this type provide the conduit between nonstandard data and the DFM file.

An Example of DefineProperty()

In order to bring all this rather technical information together, Listing 12.8 shows the DefProp.pas unit. This unit illustrates the use of DefineProperty() by providing storage for two private data fields: a string and an integer.

LISTING 12.8 DefProp.pas Illustrated Using the DefineProperty() Function

```
unit DefProp;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TDefinePropTest = class(TComponent)
  private
    FString: String;
    FInteger: Integer;
    procedure ReadStrData(Reader: TReader);
    procedure WriteStrData(Writer: TWriter);
    procedure ReadIntData(Reader: TReader);
    procedure WriteIntData(Writer: TWriter);
  protected
    procedure DefineProperties(Filer: TFiler); override;
  public
    constructor Create(AOwner: TComponent); override;
  end;

implementation

constructor TDefinePropTest.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Put data in private fields }
  FString := 'The following number is the answer...';
  FInteger := 42;
end;
```

LISTING 12.8 Continued

```
procedure TDefinePropTest.DefineProperties(Filer: TFiler);
begin
  inherited DefineProperties(Filer);
  { Define new properties and reader/writer methods }
  Filer.DefineProperty('StringProp', ReadStrData, WriteStrData,
    FString <> '');
  Filer.DefineProperty('IntProp', ReadIntData, WriteIntData, True);
end;

procedure TDefinePropTest.ReadStrData(Reader: TReader);
begin
  FString := Reader.ReadString;
end;

procedure TDefinePropTest.WriteStrData(Writer: TWriter);
begin
  Writer.WriteString(FString);
end;

procedure TDefinePropTest.ReadIntData(Reader: TReader);
begin
  FInteger := Reader.ReadInteger;
end;

procedure TDefinePropTest.WriteIntData(Writer: TWriter);
begin
  Writer.WriteInteger(FInteger);
end;

end.
```

CAUTION

Always use the `ReadString()` and `WriteString()` methods of `TReader` and `TWriter` to read and write string data. Never use the similar-looking `ReadStr()` and `WriteStr()` methods because they'll corrupt your DFM file.

`TddgWaveFile`: An Example of `DefineBinaryProperty()`

We mentioned earlier that a good time to use `DefineBinaryProperty()` is when you need to store graphic or sound information along with a component. In fact, VCL uses this technique for storing images associated with components—the `Glyph` of a `TBitBtn`, for example, or the

Icon of a TForm. In this section, you'll learn how to use this technique when storing the sound associated with the TddgWaveFile component.

NOTE

TddgWaveFile is quite a full-featured component, complete with a custom property, property editor, and component editor to allow you to play sounds at design time. You'll be able to pick through the code for all this a little later in the chapter, but for now we're going to focus the discussion on the mechanism for storing the binary property.

The DefineProperties() method for TddgWaveFile is as follows:

```
procedure TddgWaveFile.DefineProperties(Filer: TFiler);
{ Defines binary property called "Data" for FData field. }
{ This allows FData to be read from and written to DFM file. }

  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then
      Result := not (Filer.Ancestor is TddgWaveFile) or
        not Equal(TddgWaveFile(Filer.Ancestor))
    else
      Result := not Empty;
  end;

begin
  inherited DefineProperties(Filer);
  Filer.DefineBinaryProperty('Data', ReadData, WriteData, DoWrite);
end;
```

This method defines a binary property called Data, which is read and written using the component's ReadData() and WriteData() methods. Additionally, data is written only if the return value of DoWrite() is True. (You'll learn more about DoWrite() in just a moment.)

The ReadData() and WriteData() methods are defined as follows:

```
procedure TddgWaveFile.ReadData(Stream: TStream);
{ Reads WAV data from DFM stream. }
begin
  LoadFromStream(Stream);
end;

procedure TddgWaveFile.WriteData(Stream: TStream);
{ Writes WAV data to DFM stream }
```

```
begin
  SaveToStream(Stream);
end;
```

As you can see, there isn't much to these methods; they simply call the `LoadFromStream()` and `SaveToStream()` methods, which are also defined by the `TddgWaveFile` component. The `LoadFromStream()` method is as follows:

```
procedure TddgWaveFile.LoadFromStream(S: TStream);
{ Loads WAV data from stream S.  This procedure will free }
{ any memory previously allocated for FData. }
begin
  if not Empty then
    FreeMem(FData, FDataSize);
  FDataSize := 0;
  FData := AllocMem(S.Size);
  FDataSize := S.Size;
  S.Read(FData^, FDataSize);
end;
```

This method first checks to see whether memory has been previously allocated by testing the value of the `FDataSize` field. If it's greater than zero, the memory pointed to by the `FData` field is freed. At that point, a new block of memory is allocated for `FData`, and `FDataSize` is set to the size of the incoming data stream. The contents of the stream are then read into the `FData` pointer.

The `SaveToStream()` method is much simpler; it's defined as follows:

```
procedure TddgWaveFile.SaveToStream(S: TStream);
{ Saves WAV data to stream S. }
begin
  if FDataSize > 0 then
    S.Write(FData^, FDataSize);
end;
```

This method writes the data pointed to by pointer `FData` to `TStream S`.

The local `DoWrite()` function inside the `DefineProperties()` method determines whether the `Data` property needs to be streamed. Of course, if `FData` is empty, there's no need to stream data. Additionally, you must take extra measures to ensure that your component works correctly with form inheritance: You must check to see whether the `Ancestor` property for `Filer` is non-`nil`. If it is and it points to an ancestor version of the current component, you must check to see whether the data you're about to write is different from the ancestor. If you don't perform these additional tests, a copy of the data (the wave file, in this case) will be written in each of the descendant forms, and changes to the ancestor's wave file won't be copied to the descendant forms.

Listing 12.9 shows `Wavez.pas`, which includes the complete source code for the component.

LISTING 12.9 `Wavez.pas`—Illustrates a Component Encapsulating a Wave File

```
unit Wavez;

interface

uses
  SysUtils, Classes;

type
  { Special string "descendant" used to make a property editor. }
  TWaveFileString = type string;

  EWaveError = class(Exception);

  TWavePause = (wpAsync, wpsSync);
  TWaveLoop = (wlNoLoop, wlLoop);

  TddgWaveFile = class(TComponent)
  private
    FData: Pointer;
    FDataSize: Integer;
    FWaveName: TWaveFileString;
    FWavePause: TWavePause;
    FWaveLoop: TWaveLoop;
    FOnPlay: TNotifyEvent;
    FOnStop: TNotifyEvent;
    procedure SetWaveName(const Value: TWaveFileString);
    procedure WriteData(Stream: TStream);
    procedure ReadData(Stream: TStream);
  protected
    procedure DefineProperties(Filer: TFiler); override;
  public
    destructor Destroy; override;
    function Empty: Boolean;
    function Equal(Wav: TddgWaveFile): Boolean;
    procedure LoadFromFile(const FileName: String);
    procedure LoadFromStream(S:  TStream);
    procedure Play;
    procedure SaveToFile(const FileName: String);
    procedure SaveToStream(S: TStream);
    procedure Stop;
  published
    property WaveLoop: TWaveLoop read FWaveLoop write FWaveLoop;
```

LISTING 12.9 Continued

```
    property WaveName: TWaveFileString read FWaveName write SetWaveName;
    property WavePause: TWavePause read FWavePause write FWavePause;
    property OnPlay: TNotifyEvent read FOnPlay write FOnPlay;
    property OnStop: TNotifyEvent read FOnStop write FOnStop;
  end;

implementation

uses MMSystem, Windows;

{ TddgWaveFile }

destructor TddgWaveFile.Destroy;
{ Ensures that any allocated memory is freed }
begin
  if not Empty then
    FreeMem(FData, FDataSize);
  inherited Destroy;
end;

function StreamsEqual(S1, S2: TMemoryStream): Boolean;
begin
  Result := (S1.Size = S2.Size) and CompareMem(S1.Memory, S2.Memory, S1.Size);
end;

procedure TddgWaveFile.DefineProperties(Filer: TFiler);
{ Defines binary property called "Data" for FData field. }
{ This allows FData to be read from and written to DFM file. }

  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then
      Result := not (Filer.Ancestor is TddgWaveFile) or
        not Equal(TddgWaveFile(Filer.Ancestor))
    else
      Result := not Empty;
  end;

begin
  inherited DefineProperties(Filer);
  Filer.DefineBinaryProperty('Data', ReadData, WriteData, DoWrite);
end;

function TddgWaveFile.Empty: Boolean;
```

LISTING 12.9 Continued

```
begin
  Result := FDataSize = 0;
end;

function TddgWaveFile.Equal(Wav: TddgWaveFile): Boolean;
var
  MyImage, WavImage: TMemoryStream;
begin
  Result := (Wav <> nil) and (ClassType = Wav.ClassType);
  if Empty or Wav.Empty then
  begin
    Result := Empty and Wav.Empty;
    Exit;
  end;
  if Result then
  begin
    MyImage := TMemoryStream.Create;
    try
      SaveToStream(MyImage);
      WavImage := TMemoryStream.Create;
      try
        Wav.SaveToStream(WavImage);
        Result := StreamsEqual(MyImage, WavImage);
      finally
        WavImage.Free;
      end;
    finally
      MyImage.Free;
    end;
  end;
end;

procedure TddgWaveFile.LoadFromFile(const FileName: String);
{ Loads WAV data from FileName. Note that this procedure does }
{ not set the WaveName property. }
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmOpenRead);
  try
    LoadFromStream(F);
  finally
    F.Free;
  end;
end;
```

LISTING 12.9 Continued

```
procedure TddgWaveFile.LoadFromStream(S: TStream);
{ Loads WAV data from stream S.  This procedure will free }
{ any memory previously allocated for FData. }
begin
  if not Empty then
    FreeMem(FData, FDataSize);
  FDataSize := 0;
  FData :=  AllocMem(S.Size);
  FDataSize := S.Size;
  S.Read(FData^, FDataSize);
end;

procedure TddgWaveFile.Play;
{ Plays the WAV sound in FData using the parameters found in }
{ FWaveLoop and FWavePause. }
const
  LoopArray: array[TWaveLoop] of DWORD = (0, SND_LOOP);
  PauseArray: array[TWavePause] of DWORD = (SND_ASYNC, SND_SYNC);
begin
  { Make sure component contains data }
  if Empty then
    raise EWaveError.Create('No wave data');
  if Assigned(FOnPlay) then FOnPlay(Self);    // fire event
  { attempt to play wave sound }
  if not PlaySound(FData, 0, SND_MEMORY or PauseArray[FWavePause] or
                   LoopArray[FWaveLoop]) then
    raise EWaveError.Create('Error playing sound');
end;

procedure TddgWaveFile.ReadData(Stream: TStream);
{ Reads WAV data from DFM stream. }
begin
  LoadFromStream(Stream);
end;

procedure TddgWaveFile.SaveToFile(const FileName: String);
{ Saves WAV data to file FileName. }
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName,  fmCreate);
  try
    SaveToStream(F);
  finally
```

LISTING 12.9 Continued

```
    F.Free;
  end;
end;

procedure TddgWaveFile.SaveToStream(S: TStream);
{ Saves WAV data to stream S. }
begin
  if not Empty then
    S.Write(FData^, FDataSize);
end;

procedure TddgWaveFile.SetWaveName(const Value: TWaveFileString);
{ Write method for WaveName property. This method is in charge of }
{ setting WaveName property and loading WAV data from file Value. }
begin
  if Value <> '' then begin
    FWaveName := ExtractFileName(Value);
    { don't load from file when loading from DFM stream }
    { because DFM stream will already contain data. }
    if (not (csLoading in ComponentState)) and FileExists(Value) then
      LoadFromFile(Value);
  end
  else begin
    { if Value is an empty string, that is the signal to free }
    { memory allocated for WAV data. }
    FWaveName := '';
    if not Empty then
      FreeMem(FData, FDataSize);
    FDataSize := 0;
  end;
end;

procedure TddgWaveFile.Stop;
{ Stops currently playing WAV sound }
begin
  if Assigned(FOnStop) then FOnStop(Self);  // fire event
  PlaySound(Nil, 0, SND_PURGE);
end;

procedure TddgWaveFile.WriteData(Stream: TStream);
{ Writes WAV data to DFM stream }
begin
  SaveToStream(Stream);
end;

end.
```

Property Categories

As you learned back in Chapter 1, “Programming in Delphi,” a feature new as of Delphi 5 is *property categories*. This feature provides a means for the properties of VCL components to be specified as belonging to particular categories and for the Object Inspector to be sorted by these categories. Properties can be registered as belonging to a particular category using the `RegisterPropertyInCategory()` and `RegisterPropertiesInCategory()` functions declared in the `DesignIntf` unit. The former enables you to register a single property for a category, whereas the latter allows you to register multiple properties with one call.

`RegisterPropertyInCategory()` is overloaded in order to provide four different versions of this function to suit your exact needs. All the versions of this function take a `TPropertyCategoryClass` as the first parameter, describing the category. From there, each of these versions takes a different combination of property name, property type, and component class to enable you to choose the best method for registering your properties. The various versions of `RegisterPropertyInCategory()` are shown here:

```
function RegisterPropertyInCategory(ACategoryClass: TPropertyCategoryClass;
  const APropertyName: string): TPropertyFilter; overload;
function RegisterPropertyInCategory(ACategoryClass: TPropertyCategoryClass;
  AComponentClass: TClass; const APropertyName: string): TPropertyFilter
  overload;
function RegisterPropertyInCategory(ACategoryClass: TPropertyCategoryClass;
  APropertyType: PTypeInfo; const APropertyName: string): TPropertyFilter;
  overload;
function RegisterPropertyInCategory(ACategoryClass: TPropertyCategoryClass;
  APropertyType: PTypeInfo): TPropertyFilter; overload;
```

These functions are also smart enough to understand wildcard symbols, so you can, for example, add all properties that match `'Data*'` to a particular category. Refer to the online help for the `TMask` class for a complete list of supported wildcard characters and their behavior.

`RegisterPropertiesInCategory()` comes in three overloaded variations:

```
function RegisterPropertiesInCategory(ACategoryClass: TPropertyCategoryClass;
  const AFilters: array of const): TPropertyCategory; overload;
function RegisterPropertiesInCategory(ACategoryClass: TPropertyCategoryClass;
  AComponentClass: TClass; const AFilters: array of string): TPropertyCategory;
  overload;
function RegisterPropertiesInCategory(ACategoryClass: TPropertyCategoryClass;
  APropertyType: PTypeInfo; const AFilters: array of string):
TPropertyCategory;
  overload;
```

Category Classes

The TPropertyCategoryClass type is a class reference for a TPropertyCategory. TPropertyCategory is the base class for all standard property categories in VCL. There are 12 standard property categories, and these classes are described in Table 12.4.

TABLE 12.4 Standard Property Category Classes

Class Name	*Description*
TactionCategory	Properties related to runtime actions. The Enabled and Hint properties of TControl 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-and-drop and docking operations. The DragCursor and DragKind properties of TControl are in this category.
THelpCategory	Properties related to using online help and hints. The HelpContext and Hint properties of TWinControl are in this category.
TLayoutCategory	Properties related to the visual display of a control at design time. The Top and Left properties of TControl are in this category.
TLegacyCategory	Properties related to obsolete operations. The Ctl3D and ParentCtl3D properties of TWinControl 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 TControl are in this category.
TLocalizableCategory	Properties related to database operations. The DatabaseName and SQL properties of TQuery are in this category.
TMiscellaneousCategory	Properties that either do not fit a category, do not need to be categorized, or are not explicitly registered to a specific category. The AllowAllUp and Name properties of TSpeedButton are in this category.

TABLE 12.4 Continued

Class Name	*Description*
`TVisualCategory`	Properties related to the visual display of a control at runtime; the `Align` and `Visible` properties of `TControl` are in this category.
`TInputCategory`	Properties related to the input of data (they need not be related to database operations). The `Enabled` and `ReadOnly` properties of `TEdit` are in this category.

As an example, let's say that you've written a component called `TNeato` with a property called `Keen`, and you want to register the `Keen` property as a member of the Action category represented by `TActionCategory`. You could do this by adding a call to `RegisterPropertyInCategory()` to the `Register()` procedure for your control, as shown here:

```
RegisterPropertyInCategory(TActionCategory, TNeato, 'Keen');
```

Custom Categories

As you've already learned, a property category is represented in code as a class that descends from `TPropertyCategory`. How difficult is it, then, to create your own property categories in this way? It's quite easy, actually. In most cases, all you need to do is override the `Name()` and `Description()` virtual class functions of `TPropertyCategory` to return information specific to your category.

As an illustration, we'll create a new Sound category that will be used to categorize some of the properties of the `TddgWaveFile` component, which you learned about earlier in this chapter. This new category class, called `TSoundCategory`, is shown in Listing 12.10. This listing contains `WavezEd.pas`, which is a file that contains the component's category, property editor, and component editor.

LISTING 12.10 `WavezEd.pas`—Illustrates a Property Editor for the Wave File Component

```
unit WavezEd;

interface

uses DsgnIntf;

type
  { Category for some of TddgWaveFile's properties }
  TSoundCategory = class(TPropertyCategory)
```

LISTING 12.10 Continued

```
  public
    class function Name: string; override;
    class function Description: string; override;
  end;

  { Property editor for TddgWaveFile's WaveName property }
  TWaveFileStringProperty = class(TStringProperty)
  public
    procedure Edit; override;
    function GetAttributes: TPropertyAttributes; override;
  end;

  { Component editor for TddgWaveFile.  Allows user to play and stop }
  { WAV sounds from local menu in IDE. }
  TWaveEditor = class(TComponentEditor)
  private
    procedure EditProp(PropertyEditor: TPropertyEditor);
  public
    procedure Edit; override;
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

implementation

uses TypInfo, Wavez, Classes, Controls, Dialogs;

{ TSoundCategory }

class function TSoundCategory.Name: string;
begin
  Result := 'Sound';
end;

class function TSoundCategory.Description: string;
begin
  Result := 'Properties dealing with the playing of sounds'
end;

{ TWaveFileStringProperty }

procedure TWaveFileStringProperty.Edit;
{ Executed when user clicks the ellipses button on the WavName   }
```

LISTING 12.10 Continued

```
{ property in the Object Inspector.  This method allows the user }
{ to pick a file from an OpenDialog and sets the property value. }
begin
  with TOpenDialog.Create(nil) do
    try
      { Set up properties for dialog }
      Filter := 'Wav files|*.wav|All files|*.*';
      DefaultExt := '*.wav';
      { Put current value in the FileName property of dialog }
      FileName := GetStrValue;
      { Execute dialog and set property value if dialog is OK }
      if Execute then
        SetStrValue(FileName);
    finally
      Free;
    end;
end;

function TWaveFileStringProperty.GetAttributes: TPropertyAttributes;
{ Indicates the property editor will invoke a dialog. }
begin
  Result := [paDialog];
end;

{ TWaveEditor }

const
  VerbCount = 2;
  VerbArray: array[0..VerbCount - 1] of string[7] = ('Play', 'Stop');

procedure TWaveEditor.Edit;
{ Called when user double-clicks on the component at design time. }
{ This method calls the GetComponentProperties method in order to }
{ invoke the Edit method of the WaveName property editor. }
var
  Components: TDesignerSelectionList;
begin
  Components := TDesignerSelectionList.Create;
  try
    Components.Add(Component);
    GetComponentProperties(Components, tkAny, Designer, EditProp);
  finally
    Components.Free;
  end;
end;
```

LISTING 12.10 Continued

```
procedure TWaveEditor.EditProp(PropertyEditor: TPropertyEditor);
{ Called once per property in response to GetComponentProperties }
{ call.  This method looks for the WaveName property editor and  }
{ calls its Edit method. }
begin
  if PropertyEditor is TWaveFileStringProperty then begin
    TWaveFileStringProperty(PropertyEditor).Edit;
    Designer.Modified;    // alert Designer to modification
  end;
end;

procedure TWaveEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TddgWaveFile(Component).Play;
    1: TddgWaveFile(Component).Stop;
  end;
end;

function TWaveEditor.GetVerb(Index: Integer): string;
begin
  Result := VerbArray[Index];
end;

function TWaveEditor.GetVerbCount: Integer;
begin
  Result := VerbCount;
end;

end.
```

With the category class defined, all that needs to be done is register the properties for the category using one of the registration functions. This is done in the `Register()` procedure for `TddgWaveFile` using the following line of code:

```
  RegisterPropertiesInCategory(TSoundCategory, TddgWaveFile,
    ['WaveLoop', 'WaveName', 'WavePause']);
```

Lists of Components: `TCollection` and `TCollectionItem`

It's common for components to maintain or own a list of items such as data types, records, objects, or even other components. In some cases, it's suitable to encapsulate this list within its own object and then make this object a property of the owner component. An example of this arrangement is the Lines property of a TMemo component. Lines is a TStrings object type that encapsulates a list of strings. With this arrangement, the TStrings object is responsible for the streaming mechanism used to store its lines to the form file when the user saves the form.

What if you wanted to save a list of items such as components or objects that weren't already encapsulated by an existing class such as TStrings? Well, you could create a class that performs the streaming of the listed items and then make that a property of the owner component. Alternatively, you could override the default streaming mechanism of the owner component so that it knows how to stream its list of items. However, a better solution would be to take advantage of the TCollection and TCollectionItem classes.

The TCollection class is an object used to store a list of TCollectionItem objects. TCollection, itself, isn't a component but rather a descendant of TPersistent. Typically, TCollection is associated with an existing component.

To use TCollection to store a list of items, you would derive a descendant class from TCollection, which you could call TNewCollection. TNewCollection will serve as a property type for a component. Then, you must derive a class from the TCollectionItem class, which you could call TNewCollectionItem. TNewCollection will maintain a list of TNewCollectionItem objects. The beauty of this is that data belonging to TNewCollectionItem that needs to be streamed only needs to be published by TNewCollectionItem. Delphi already knows how to stream published properties.

An example of where TCollection is used is with the TStatusBar component. TStatusBar is a TWinControl descendant. One of its properties is Panels. TStatusBar.Panels is of type TStatusPanels, which is a TCollection descendant and defined as follows:

```
type
  TStatusPanels = class(TCollection)
  private
    FStatusBar: TStatusBar;
    function GetItem(Index: Integer): TStatusPanel;
    procedure SetItem(Index: Integer; Value: TStatusPanel);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(StatusBar: TStatusBar);
    function Add: TStatusPanel;
    property Items[Index: Integer]: TStatusPanel read GetItem write SetItem;
      default;
  end;
```

TStatusPanels stores a list of TCollectionItem descendants, TStatusPanel, as defined here:

```
type
  TStatusPanel = class(TCollectionItem)
  private
    FText: string;
    FWidth: Integer;
    FAlignment: TAlignment;
    FBevel: TStatusPanelBevel;
    FStyle: TStatusPanelStyle;
    procedure SetAlignment(Value: TAlignment);
    procedure SetBevel(Value: TStatusPanelBevel);
    procedure SetStyle(Value: TStatusPanelStyle);
    procedure SetText(const Value: string);
    procedure SetWidth(Value: Integer);
  public
    constructor Create(Collection: TCollection); override;
    procedure Assign(Source: TPersistent); override;
  published
    property Alignment: TAlignment read FAlignment
      write SetAlignment default taLeftJustify;
    property Bevel: TStatusPanelBevel read FBevel
      write SetBevel default pbLowered;
    property Style: TStatusPanelStyle read FStyle write SetStyle
      default psText;
    property Text: string read FText write SetText;
    property Width: Integer read FWidth write SetWidth;
  end;
```

The TStatusPanel properties in the published section of the class declaration will automatically be streamed by Delphi. TStatusPanel takes a TCollection parameter in its Create() constructor, and it associates itself with that TCollection. Likewise, TStatusPanels takes the TStatusBar component in its constructor to which it associates itself. The TCollection engine knows how to deal with the streaming of TCollectionItem components and also defines some methods and properties for manipulating the items maintained in TCollection. You can look these up in the online help.

To illustrate how you might use these two new classes, we've created the TddgLaunchPad component. TddgLaunchPad will enable the user to store a list of TddgRunButton components, which we created in Chapter 11.

TddgLaunchPad is a descendant of the TScrollBox component. One of the properties of TddgLaunchPad is RunButtons, a TCollection descendant. RunButtons maintains a list of TRunBtnItem components. TRunBtnItem is a TCollectionItem descendant whose properties are used to create a TddgRunButton component, which is placed on TddgLaunchPad. In the following sections, we'll discuss how we created this component.

Defining the `TCollectionItem` Class: `TRunBtnItem`

The first step is to define the item to be maintained in a list. For `TddgLaunchPad`, this would be a `TddgRunButton` component. Therefore, each `TRunBtnItem` instance must associate itself with a `TddgRunButton` component. The following code shows a partial definition of the `TRunBtnItem` class:

```
type
  TRunBtnItem = class(TCollectionItem)
  private
    FCommandLine: String;   // Store the command line
    FLeft: Integer;         // Store the positional properties for the
    FTop: Integer;          //    TddgRunButton.
    FRunButton: TddgRunButton; // Reference to a TddgRunButton
     …
  public
    constructor Create(Collection: TCollection); override;
  published
    { The published properties will be streamed }
    property CommandLine: String read FCommandLine write SetCommandLine;
    property Left: Integer read FLeft write SetLeft;
    property Top: Integer read FTop write SetTop;
  end;
```

Notice that `TRunBtnItem` keeps a reference to a `TddgRunButton` component, yet it only streams the properties required to build a `TddgRunButton`. At first you might think that because `TRunBtnItem` associates itself with a `TddgRunButton`, it could just publish the component and let the streaming engine do the rest. Well, this poses some problems with the streaming engine and how it handles the streaming of `TComponent` classes differently from `TPersistent` classes. The fundamental rule here is that the streaming system is responsible for creating new instances for every `TComponent`-derived classname it finds in a stream, whereas it assumes that `TPersistent` instances already exist and doesn't attempt to instantiate new ones. Following this rule, we stream the information required of the `TddgRunButton` and then we create the `TddgRunButton` in the `TRunBtnItem` constructor, which we'll illustrate shortly.

Defining the `TCollection` Class: `TRunButtons`

The next step is to define the object that will maintain this list of `TRunBtnItem` components. We already said that this object must be a `TCollection` descendant. We call this class `TRunButtons`; its definition is as follows:

```
type
  TRunButtons = class(TCollection)
  private
    FLaunchPad: TddgLaunchPad; // Keep a reference to the TddgLaunchPad
    function GetItem(Index: Integer): TRunBtnItem;
    procedure SetItem(Index: Integer; Value: TRunBtnItem);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(LaunchPad: TddgLaunchPad);
    function Add: TRunBtnItem;
    procedure UpdateRunButtons;
    property Items[Index: Integer]: TRunBtnItem read GetItem
      write SetItem; default;
  end;
```

TRunButtons associates itself with a TddgLaunchPad component that we'll show a bit later. It does this in its Create() constructor, which, as you can see, takes a TddgLaunchPad component as its parameter. Notice the various properties and methods that have been added to allow the user to manipulate the individual TRunBtnItem classes. In particular, the Items property is an array to the TRunBtnItem list.

The use of the TRunBtnItem and TRunButtons classes will become clearer as we discuss the implementation of the TddgLaunchPad component.

Implementing the TddgLaunchPad, TRunBtnItem, and TRunButtons Objects

The TddgLaunchPad component has a property of the type TRunButtons. Its implementation, as well as the implementation of TRunBtnItem and TRunButtons, is shown in Listing 12.11.

LISTING 12.11 LnchPad.pas—Illustrates the TddgLaunchPad Implementation

```
unit LnchPad;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, RunBtn, ExtCtrls;

type
  TddgLaunchPad = class;

  TRunBtnItem = class(TCollectionItem)
  private
    FCommandLine: string;   // Store the command line
    FLeft: Integer;         // Store the positional properties for the
    FTop: Integer;          // TddgRunButton.
```

LISTING 12.11 Continued

```
    FRunButton: TddgRunButton; // Reference to a TddgRunButton
    FWidth: Integer;        // Keep track of the width and height
    FHeight: Integer;
    procedure SetCommandLine(const Value: string);
    procedure SetLeft(Value: Integer);
    procedure SetTop(Value: Integer);
  public
    constructor Create(Collection: TCollection); override;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    property Width: Integer read FWidth;
    property Height: Integer read FHeight;
  published
    { The published properties will be streamed }
    property CommandLine: String read FCommandLine
      write SetCommandLine;
    property Left: Integer read FLeft write SetLeft;
    property Top: Integer read FTop write SetTop;
  end;

  TRunButtons = class(TCollection)
  private
    FLaunchPad: TddgLaunchPad; // Keep a reference to the TddgLaunchPad
    function GetItem(Index: Integer): TRunBtnItem;
    procedure SetItem(Index: Integer; Value: TRunBtnItem);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(LaunchPad: TddgLaunchPad);
    function Add: TRunBtnItem;
    procedure UpdateRunButtons;
    property Items[Index: Integer]: TRunBtnItem read
      GetItem write SetItem; default;
  end;

  TddgLaunchPad = class(TScrollBox)
  private
    FRunButtons: TRunButtons;
    TopAlign: Integer;
    LeftAlign: Integer;
    procedure SetRunButtons(Value: TRunButtons);
    procedure UpdateRunButton(Index: Integer);
  public
    constructor Create(AOwner: TComponent); override;
```

Listing 12.11 Continued

```
    destructor Destroy; override;
    procedure GetChildren(Proc: TGetChildProc; Root: TComponent); override;
  published
    property RunButtons: TRunButtons read FRunButtons write SetRunButtons;
  end;

implementation

{ TRunBtnItem }

constructor TRunBtnItem.Create(Collection: TCollection);
{ This constructor gets the TCollection that owns this TRunBtnItem.  }
begin
  inherited Create(Collection);
  { Create an FRunButton instance. Make the launch pad the owner
    and parent. Then initialize its various properties. }
  FRunButton := TddgRunButton.Create(TRunButtons(Collection).FLaunchPad);
  FRunButton.Parent := TRunButtons(Collection).FLaunchPad;
  FWidth := FRunButton.Width;   // Keep track of the width and the
  FHeight := FRunButton.Height; //   height.
end;

destructor TRunBtnItem.Destroy;
begin
  FRunButton.Free;   // Destroy the TddgRunButton instance.
  inherited Destroy; // Call the inherited Destroy destructor.
end;

procedure TRunBtnItem.Assign(Source: TPersistent);
{ It is necessary to override the TCollectionItem.Assign method so that
  it knows how to copy from one TRunBtnItem to another. If this is done,
  then don't call the inherited Assign(). }
begin
  if Source is TRunBtnItem then
  begin
    { Instead of assigning the command line to the FCommandLine storage
      field, make the assignment to the property so that the accessor
      method will be called. The accessor method as some side-effects
      that we want to occur. }
    CommandLine := TRunBtnItem(Source).CommandLine;
    { Copy values to the remaining fields. Then exit the procedure. }
    FLeft := TRunBtnItem(Source).Left;
    FTop := TRunBtnItem(Source).Top;
```

LISTING 12.11 Continued

```
    Exit;
  end;
  inherited Assign(Source);
end;

procedure TRunBtnItem.SetCommandLine(const Value: string);
{ This is the write accessor method for TRunBtnItem.CommandLine. It
  ensures that the private TddgRunButton instance, FRunButton, gets
  assigned the specified string from Value }
begin
  if FRunButton <> nil then
  begin
    FCommandLine := Value;
    FRunButton.CommandLine := FCommandLine;
    { This will cause the TRunButtons.Update method to be called
      for each TRunBtnItem }
    Changed(False);
  end;
end;

procedure TRunBtnItem.SetLeft(Value: Integer);
{ Access method for the TRunBtnItem.Left property. }
begin
  if FRunButton <> nil then
  begin
    FLeft := Value;
    FRunButton.Left := FLeft;
   end;
end;

procedure TRunBtnItem.SetTop(Value: Integer);
{ Access method for the TRunBtnItem.Top property }
begin
  if FRunButton <> nil then
  begin
    FTop := Value;
    FRunButton.Top := FTop;
   end;
end;

{ TRunButtons }

constructor TRunButtons.Create(LaunchPad: TddgLaunchPad);
{ The constructor points FLaunchPad to the TddgLaunchPad parameter.
  LauchPad is the owner of this collection. It is necessary to keep
```

LISTING 12.11 Continued

```
  a reference to LauchPad as it will be accessed internally. }
begin
  inherited Create(TRunBtnItem);
  FLaunchPad := LaunchPad;
end;

function TRunButtons.GetItem(Index: Integer): TRunBtnItem;
{ Access method for TRunButtons.Items which returns the TRunBtnItem
  instance. }
begin
  Result := TRunBtnItem(inherited GetItem(Index));
end;

procedure TRunButtons.SetItem(Index: Integer; Value: TRunBtnItem);
{ Access method for TddgRunButton.Items which makes the assignment to
  the specified indexed item. }
begin
  inherited SetItem(Index, Value)
end;

procedure TRunButtons.Update(Item: TCollectionItem);
{ TCollection.Update is called by TCollectionItems
  whenever a change is made to any of the collection items. This is
  initially an abstract method. It must be overridden to contain
  whatever logic is necessary when a TCollectionItem has changed.
  We use it to redraw the item by calling TddgLaunchPad.UpdateRunButton.}
begin
  if Item <> nil then
    FLaunchPad.UpdateRunButton(Item.Index);
end;

procedure TRunButtons.UpdateRunButtons;
{ UpdateRunButtons is a public procedure that we made available so that
  users of TRunButtons can force all run-buttons to be re-drawn. This
  method calls TddgLaunchPad.UpdateRunButton for each TRunBtnItem
  instance. }
var
  i: integer;
begin
  for i := 0 to Count - 1 do
    FLaunchPad.UpdateRunButton(i);
end;

function TRunButtons.Add: TRunBtnItem;
```

LISTING 12.11 Continued

```
{ This method must be overridden to return the TRunBtnItem instance when
  the inherited Add method is called. This is done by typcasting the
  original result }
begin
  Result := TRunBtnItem(inherited Add);
end;

{ TddgLaunchPad }

constructor TddgLaunchPad.Create(AOwner: TComponent);
{ Initializes the TRunButtons instance and internal variables
  used for positioning of the TRunBtnItem as they are drawn }
begin
  inherited Create(AOwner);
  FRunButtons := TRunButtons.Create(Self);
  TopAlign := 0;
  LeftAlign := 0;
end;

destructor TddgLaunchPad.Destroy;
begin
  FRunButtons.Free;  // Free the TRunButtons instance.
  inherited Destroy; // Call the inherited destroy method.
end;

procedure TddgLaunchPad.GetChildren(Proc: TGetChildProc; Root: TComponent);
{ Override GetChildren to cause TddgLaunchPad to ignore any TRunButtons
  that it owns since they do not need to be streamed in the context
  TddgLaunchPad. The information necessary for creating the TddgRunButton
  instances is already streamed as published properties of the
  TCollectionItem descendant, TRunBtnItem. This method prevents the
  TddgRunButton's from being streamed twice. }
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    { Ignore the run buttons and the scrollbox }
    if not (Controls[i] is TddgRunButton) then
      Proc(TComponent(Controls[I]));
end;

procedure TddgLaunchPad.SetRunButtons(Value: TRunButtons);
{ Access method for the RunButtons property }
```

LISTING 12.11 Continued

```
begin
  FRunButtons.Assign(Value);
end;

procedure TddgLaunchPad.UpdateRunButton(Index: Integer);
{ This method is responsible for drawing the TRunBtnItem instances.
  It ensures that the TRunBtnItem's do not extend beyond the width
  of the TddgLaunchPad. If so, it creates rows. This is only in effect
  as the user is adding/removing TRunBtnItems. The user can still
  resize the TddgLaunchPad so that it is smaller than the width of a
  TRunBtnItem }
begin
  { If the first item being drawn, set both positions to zero. }
  if Index = 0 then
  begin
    TopAlign := 0;
    LeftAlign := 0;
  end;
  { If the width of the current row of TRunBtnItems is more than
    the width of the TddgLaunchPad, then start a new row of TRunBtnItems. }
  if (LeftAlign + FRunButtons[Index].Width) > Width then
  begin
    TopAlign := TopAlign + FRunButtons[Index].Height;
    LeftAlign := 0;
 end;
 FRunButtons[Index].Left := LeftAlign;
 FRunButtons[Index].Top := TopAlign;
 LeftAlign := LeftAlign + FRunButtons[Index].Width;
end;

end.
```

Implementing `TRunBtnItem`

The `TRunBtnItem.Create()` constructor creates an instance of `TddgRunButton`. Each `TRunBtnItem` in the collection will maintain its own `TddgRunButton` instance. The following two lines in `TRunBtnItem.Create()` require further explanation:

```
FRunButton := TddgRunButton.Create(TRunButtons(Collection).FLaunchPad);
FRunButton.Parent := TRunButtons(Collection).FLaunchPad;
```

The first line creates a `TddgRunButton` instance, `FRunButton`. The owner of `FRunButton` is `FLaunchPad`, which is a `TddgLaunchPad` component and a field of the `TCollection` object passed in as a parameter. It's necessary to use the `FLaunchPad` as the owner of `FRunButton`.

Neither a TRunBtnItem instance nor a TRunButtons object can be owners because they descend from TPersistent. Remember, an owner must be a TComponent.

We want to point out a problem that arises by making FLaunchPad the owner of FRunButton. By doing this, we effectively make FLaunchPad the owner of FRunButton at design time. The normal behavior of the streaming engine will cause Delphi to stream FRunButton as a component owned by the FLaunchPad instance when the user saves the form. This isn't a desired behavior because FRunButton is already being created in the constructor of TRunBtnItem, based on the information that's also streamed in the context of TRunBtnItem. This is a vital tidbit of information. Later, you'll see how we prevent TddgRunButton components from being streamed by TddgLaunchPad in order to remedy this undesired behavior.

The second line assigns FLaunchPad as the parent to FRunButton so that FLaunchPad can take care of drawing FRunButton.

The TRunBtnItem.Destroy() destructor frees FRunButton before calling its inherited destructor.

Under certain circumstances, it becomes necessary to override the TRunBtnItem.Assign() method that's called. One such instance is when the application is first run and the form is read from the stream. In the Assign() method, we tell the TRunBtnItem instance to assign the streamed values of its properties to the properties of the component (in this case TddgRunButton) that it encompasses.

The other methods are simply access methods for the various properties of TRunBtnItem; they are explained in the code's comments.

Implementing TRunButtons

TRunButtons.Create() simply points FLaunchPad to the TddgLaunchPad parameter passed to it so that LaunchPad can be referred to later.

TRunButtons.Update() is a method that's invoked whenever a change has been made to any of the TRunBtnItem instances. This method contains logic that should occur due to that change. We use it to call the method of TddgLaunchPad that redraws the TRunBtnItem instances. We've also added a public method, UpdateRunButtons(), to allow the user to force a redraw.

The remaining methods of TRunButtons are property access methods, which are explained in the code's comments in Listing 12.11.

Implementing TddgLaunchPad

The constructor and destructor for TddgLaunchPad are simple. TddgLaunchPad.Create() creates an instance of the TRunButtons object and passes itself as a parameter. TddgLaunchPad.Destroy() frees the TRunButtons instance.

The overriding of the `TddgLaunchPad.GetChildren()` method is important to note here. This is where we prevent the `TddgRunButton` instances stored by the collection from being streamed as owned components of `TddgLaunchPad`. Remember that this is necessary because they shouldn't be created in the context of the `TddgLaunchPad` object but rather in the context of the `TRunBtnItem` instances. Because no `TddgRunButton` components are passed to the `Proc` procedure, they won't be streamed or read from a stream.

The `TddgLaunchPad.UpdateRunButton()` method is where the `TddgRunButton` instances maintained by the collection are drawn. The logic in this code ensures that they never extend beyond the width of `TddgLaunchPad`. Because `TddgLaunchPad` is a descendant of `TScrollBox`, scrolling will occur vertically.

The other methods are simply property-access methods and are commented in the code in Listing 12.11.

Finally, we register the property editor for the `TRunButtons` collection class in this unit's `Register()` procedure. The next section discusses this property editor and illustrates how to edit a list of components from a dialog property editor.

Editing the List of `TCollectionItem` Components with a Dialog Property Editor

Now that we've defined the `TddgLaunchPad` component, the `TRunButtons` collection class, and the `TRunBtnItem` collection class, we must provide a way for the user to add `TddgRunButton` components to the `TRunButtons` collection. The best way to do this is through a property editor that manipulates the list maintained by the `TRunButtons` collection.

This dialog directly manipulates the `TRunBtnItem` components maintained by the `RunButtons` collection of `TddgLaunchPad`. The various `CommandLine` strings for each `TddgRunButton` enclosed in `TRunBtnItem` are displayed in `PathListBox`. A `TddgRunButton` component reflects the currently selected item in the list box to allow the user to test the selection. The dialog also contains buttons to allow the user to add or remove an item, accept the changes, and cancel the operation. As the user makes changes in the dialog, the changes are reflected on the `TddgLaunchPad`.

TIP

A convention for property editors is to include an Apply button to invoke changes on the form. We didn't show this here, but you might consider adding such a button to the `RunButtons` property editor as an exercise. To see how an Apply button works, take a look at the property editor for the `Panels` property of the `TStatusBar` component from the Win32 page of the Component Palette.

Listing 12.12 shows the source code for the TddgLaunchPad-RunButtons property editor and its dialog.

LISTING 12.12 `LPadPE.pas`—The `TRunButtons` Property Editor

```
unit LPadPE;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Buttons, RunBtn, StdCtrls, LnchPad, DesignIntf, DesignEditors,
  ExtCtrls, TypInfo;

type

  { First declare the editor dialog }
  TLaunchPadEditor = class(TForm)
    PathListBox: TListBox;
    AddBtn: TButton;
    RemoveBtn: TButton;
    CancelBtn: TButton;
    OkBtn: TButton;
    Label1: TLabel;
    pnlRBtn: TPanel;
    procedure PathListBoxClick(Sender: TObject);
    procedure AddBtnClick(Sender: TObject);
    procedure RemoveBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
  private
    TestRunBtn: TddgRunButton;
    FLaunchPad: TddgLaunchPad;   // To be used as a backup
    FRunButtons: TRunButtons; // Will refer to the actual TRunButtons
    Modified: Boolean;
    procedure UpdatePathListBox;
  end;

  { Now declare the TPropertyEditor descendant and override the
    required methods }
  TRunButtonsProperty = class(TPropertyEditor)
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure Edit; override;
  end;
```

LISTING 12.12 Continued

```
{ This function will be called by the property editor. }
function EditRunButtons(RunButtons: TRunButtons): Boolean;

implementation

{$R *.DFM}

function EditRunButtons(RunButtons: TRunButtons): Boolean;
{ Instantiates the TLaunchPadEditor dialog which directly modifies
  the TRunButtons collection. }
begin
  with TLaunchPadEditor.Create(Application) do
    try
      FRunButtons := RunButtons; // Point to the actual TRunButtons
      { Copy the TRunBtnItems to the backup FLaunchPad which will be
        used as a backup in case the user cancels the operation }
      FLaunchPad.RunButtons.Assign(RunButtons);
      { Draw the listbox with the list of TRunBtnItems. }
      UpdatePathListBox;
      ShowModal; // Display the form.
      Result := Modified;
    finally
      Free;
    end;
end;

{ TLaunchPadEditor }

procedure TLaunchPadEditor.FormCreate(Sender:  TObject);
begin
  { Created the backup instances of TLaunchPad to be used if the user
    cancels editing the TRunBtnItems }
  FLaunchPad := TddgLaunchPad.Create(Self);

  // Create the TddgRunButton instance and align it to the
  // enclosing panel.
  TestRunBtn := TddgRunButton.Create(Self);
  TestRunBtn.Parent := pnlRBtn;

  TestRunBtn.Width  := pnlRBtn.Width;
  TestRunBtn.Height := pnlRBtn.Height;
end;

procedure TLaunchPadEditor.FormDestroy(Sender: TObject);
```

LISTING 12.12 Continued

```
begin
  TestRunBtn.Free;
  FLaunchPad.Free; // Free the TLaunchPad instance.
end;

procedure TLaunchPadEditor.PathListBoxClick(Sender: TObject);
{ When the user clicks on an item in the list of TRunBtnItems, make
  the test TRunButton reflect the currently selected item }
begin
  if PathListBox.ItemIndex > -1 then
    TestRunBtn.CommandLine := PathListBox.Items[PathListBox.ItemIndex];
end;

procedure TLaunchPadEditor.UpdatePathListBox;
{ Re-initializes the PathListBox so that it reflects the list of
  TRunBtnItems }
var
  i: integer;
begin
  PathListBox.Clear; // First clear the list box.
  for i := 0 to FRunButtons.Count - 1 do
    PathListBox.Items.Add(FRunButtons[i].CommandLine);
end;

procedure TLaunchPadEditor.AddBtnClick(Sender: TObject);
{ When the add button is clicked, launch a TOpenDialog to retrieve
  an executable filename and path. Then add this file to the
  PathListBox. Also, add a new FRunBtnItem. }
var
  OpenDialog: TOpenDialog;
begin
  OpenDialog := TOpenDialog.Create(Application);
  try
    OpenDialog.Filter := 'Executable Files|*.EXE';
    if OpenDialog.Execute then
    begin
      { add to the PathListBox. }
      PathListBox.Items.Add(OpenDialog.FileName);
      FRunButtons.Add; // Create a new TRunBtnItem instance.
      { Set focus to the new item in PathListBox }
      PathListBox.ItemIndex := FRunButtons.Count - 1;
      { Set the command line for the new TRunBtnItem to that of the
        file name gotten as specified by PathListBox.ItemIndex }
      FRunButtons[PathListBox.ItemIndex].CommandLine :=
```

LISTING 12.12 Continued

```
        PathListBox.Items[PathListBox.ItemIndex];
      { Invoke the PathListBoxClick event handler so that the test
        TRunButton will reflect the newly added item }
      PathListBoxClick(nil);
      Modified := True;
    end;
  finally
    OpenDialog.Free
  end;
end;

procedure TLaunchPadEditor.RemoveBtnClick(Sender: TObject);
{ Remove the selected path/filename from PathListBox as well as the
  corresponding TRunBtnItem from FRunButtons }
var
  i: integer;
begin
  i := PathListBox.ItemIndex;
  if i >= 0 then
  begin
    PathListBox.Items.Delete(i);  // Remove the item from the listbox
    FRunButtons[i].Free;          // Remove the item from the collection
    TestRunBtn.CommandLine := ''; // Erase the test run button
    Modified := True;
  end;
end;

procedure TLaunchPadEditor.CancelBtnClick(Sender: TObject);
{ When the user cancels the operation, copy the backup LaunchPad
  TRunBtnItems back to the original TLaunchPad instance. Then,
  close the form by setting ModalResult to mrCancel. }
begin
  FRunButtons.Assign(FLaunchPad.RunButtons);
  Modified := False;
  ModalResult := mrCancel;
end;

{ TRunButtonsProperty }

function TRunButtonsProperty.GetAttributes: TPropertyAttributes;
{ Tell the Object Inspector that the property editor will use a
  dialog. This will cause the Edit method to be invoked when the user
  clicks the ellipsis button in the Object Inspector.  }
```

Listing 12.12 Continued

```
begin
  Result := [paDialog];
end;

procedure TRunButtonsProperty.Edit;
{ Invoke the EditRunButton() method and pass in the reference to the
  TRunButton's instance being edited. This reference can be obtain by
  using the GetOrdValue method. Then redraw the LaunchDialog by calling
  the TRunButtons.UpdateRunButtons method. }
begin
  if EditRunButtons(TRunButtons(GetOrdValue)) then
    Modified;
  TRunButtons(GetOrdValue).UpdateRunButtons;
end;

function TRunButtonsProperty.GetValue: string;
{ Override the GetValue method so that the class type of the property
  being edited is displayed in the Object Inspector. }
begin
  Result := Format('(%s)',  [GetPropType^.Name]);
end;

end.
```

This unit first defines the `TddgLaunchPadEditor` dialog and then the `TRunButtonsProperty` property editor. We're going to discuss the property editor first because it's the property editor that invokes the dialog.

The `TRunButtonsProperty` property editor isn't much different from the dialog property editor we showed earlier. Here, we override the `GetAttributes()`, `Edit()`, and `GetValue()` methods.

`GetAttributes()` simply sets the `TPropertyAttributes` return value to specify that this editor invokes a dialog. Again, this will place an ellipsis button on the Object Inspector.

The `GetValue()` method uses the `GetPropType()` function to return a pointer to the Runtime Type Information for the property being edited. It returns the name field of this information that represents the property's type string. The string is displayed in the Object Inspector within parentheses, which is a convention used by Delphi.

Finally, the `Edit()` method calls a function defined in this unit, `EditRunButtons()`. As a parameter, it passes the reference to the `TRunButtons` property by using the `GetOrdValue` function. When the function returns, the method `UpdateRunButton()` is invoked to cause `RunButtons` to be redrawn to reflect any changes.

The `EditRunButtons()` function creates the `TddgLaunchPadEditor` instance and points its `FRunButtons` field to the `TRunButtons` parameter passed to it. It uses this reference internally to make changes to the `TRunButtons` collection. The function then copies the `TRunButtons` collection of the property to an internal `TddgLaunchPad` component, `FLaunchPad`. It uses this instance as a backup in case the user cancels the edit operation.

Earlier we talked about the possibility of adding an Apply button to this dialog. To do so, you can edit the `FLaunchPad` component's `RunButtons` collection instance instead of directly modifying the actual collection. This way, if the user cancels the operation, nothing happens; if the user clicks Apply or OK, the changes are invoked.

The form's `Create()` constructor creates the internal `TddgLaunchPad` instance. The `Destroy()` destructor ensures that it's freed when the form is destroyed.

`PathListBoxClick()` is the `OnClick` event handler for `PathListBox`. This method makes `TestRunBtn` (the test `TddgRunButton`) reflect the currently selected item in `PathListBox`, which displays a path to the executable file. The user can click this `TddgRunButton` instance to launch the application.

`UpdatePathListBox()` initializes `PathListBox` with the items in the collection.

`AddButtonClick()` is the `OnClick` event handler for the Add button. This event handler invokes a File Open dialog to retrieve an executable filename from the user and adds the path of this filename to `PathListBox`. It also creates a `TRunBtnItem` instance in the collection and assigns the path to its `CommandLine` property, which in turn does the same for the `TddgRunButton` component it encloses.

`RemoveBtnClick()` is the `OnClick` event handler for the Remove button. It removes the selected item from `PathListBox` as well as the `TRunBtnItem` instance from the collection.

`CancelBtnClick()` is the `OnClick` event handler for the Cancel button. It copies the backup collection from `FLaunchPad` to the actual `TRunButtons` collection and closes the form.

The `TCollection` and `TCollectionItems` objects are extremely useful and offer themselves to being used for a variety of purposes. Get to know them well, and next time you need to store a list of components, you'll already have a solution.

Summary

This chapter let you in on some of the more advanced tricks and techniques for Delphi component design. Among other things, you learned about extending hints and animating components as well as component editors, property editors, and component collections. Armed with this information, as well as the more conventional information you learned in the preceding chapter, you should be able to write a component to suit just about any of your programming needs.