CHAPTER 19

Building a Better Window with System.Windows.Forms

If you have read through the previous 18 chapters, you should have a solid handle on the C# programming language as well as the foundation of the .NET architecture. While you could take your newfound knowledge and begin building the next generation of console applications (boring!), you are more likely to be interested in building an attractive graphical user interface (GUI) to allow users to interact with your system.

This chapter is the first of three aimed at introducing you to the process of building traditional form-based desktop applications. Here, you'll learn how to build a highly stylized main window using the `Form` and `Application` classes. This chapter also illustrates how to capture and respond to user input (i.e., handle mouse and keyboard events) within the context of a GUI desktop environment. Finally, you will learn to construct menu systems, toolbars, status bars, and multiple-document interface (MDI) applications, both by hand and using the designers incorporated into Visual Studio 2005.

Overview of the System.Windows.Forms Namespace

Like any namespace, `System.Windows.Forms` is composed of various classes, structures, delegates, interfaces, and enumerations. Although the difference in appearance between a console UI (CUI) and graphical UI (GUI) seems at first glance like night and day, in reality the process of building a Windows Forms application involves nothing more than learning how to manipulate a new set of types using the C# syntax you already know. From a high level, the hundreds of types within the `System.Windows.Forms` namespace can be grouped into the following broad categories:

- *Core infrastructure*: These are types that represent the core operations of a .NET Forms program (`Form`, `Application`, etc.) and various types to facilitate interoperability with legacy ActiveX controls.
- *Controls*: These are types used to create rich UIs (`Button`, `MenuStrip`, `ProgressBar`, `DataGridView`, etc.), all of which derive from the `Control` base class. Controls are configurable at design time and are visible (by default) at runtime.
- *Components*: These are types that do not derive from the `Control` base class but still provide visual features to a .NET Forms program (`ToolTip`, `ErrorProvider`, etc.). Many components (such as the `Timer`) are not visible at runtime, but can be configured visually at design time.
- *Common dialog boxes*: Windows Forms provides a number of canned dialog boxes for common operations (`OpenFileDialog`, `PrintDialog`, etc.). As you would hope, you can certainly build your own custom dialog boxes if the standard dialog boxes do not suit your needs.

Given that the total number of types within `System.Windows.Forms` is well over 100 strong, it would be redundant (not to mention a terrible waste of paper) to list every member of the Windows Forms family. To set the stage for the next several chapters, however, Table 19-1 lists some of the core .NET 2.0 `System.Windows.Forms` types (consult the .NET Framework 2.0 SDK documentation for full details).

Table 19-1. *Core Types of the* `System.Windows.Forms` *Namespace*

Classes	Meaning in Life
`Application`	This class encapsulates the runtime operation of a Windows Forms application.
`Button`, `CheckBox`, `ComboBox`, `DateTimePicker`, `ListBox`, `LinkLabel`, `MaskedTextBox`, `MonthCalendar`, `PictureBox`, `TreeView`	These classes (in addition to many others) correspond to various GUI widgets. You'll examine many of these items in detail in Chapter 21.
`FlowLayoutPanel`, `TableLayoutPanel`	.NET 2.0 now supplies various "layout managers" that automatically arrange a Form's controls during resizing.
`Form`	This type represents a main window, dialog box, or MDI child window of a Windows Forms application.
`ColorDialog`, `OpenFileDialog`, `SaveFileDialog`, `FontDialog`, `PrintPreviewDialog`, `FolderBrowserDialog`	These are various standard dialog boxes for common GUI operations.
`Menu`, `MainMenu`, `MenuItem`, `ContextMenu`, `MenuStrip`, `ContextMenuStrip`,	These types are used to build topmost and context-sensitive menu systems. The new (.NET 2.0) `MenuStrip` and `ContextMenuStrip` controls allow you to build menus that may contain traditional drop-down menu items as well as other controls (text boxes, combo boxes, and so forth).
`StatusBar`, `Splitter`, `ToolBar`, `ScrollBar`, `StatusStrip`, `ToolStrip`	These types are used to adorn a Form with common child controls.

■ **Note** In addition to `System.Windows.Forms`, the `System.Windows.Forms.dll` assembly defines additional GUI-centric namespaces. For the most part, these additional types are used internally by the Forms engine and/or the designer tools of Visual Studio 2005. Given this fact, we will keep focused on the core `System.Windows.Forms` namespace.

Working with the Windows Forms Types

When you build a Windows Forms application, you may choose to write all the relevant code by hand (using Notepad or TextPad, perhaps) and feed the resulting `*.cs` files into the C# compiler using the `/target:winexe` flag. Taking time to build some Windows Forms applications by hand not only is a great learning experience, but also helps you understand the code generated by the various graphics designers found within various .NET IDEs.

To make sure you truly understand the basic process of building a Windows Forms application, the initial examples in this chapter will avoid the use of graphics designers. Once you feel comfortable with the process of building a Windows Forms application "wizard-free," you will then leverage the various designer tools provided by Visual Studio 2005.

Building a Main Window by Hand

To begin learning about Windows Forms programming, you'll build a minimal main window from scratch. Create a new folder on your hard drive (e.g., C:\MyFirstWindow) and create a new file within this directory named `MainWindow.cs` using your editor of choice.

In the world of Windows Forms, the `Form` class is used to represent any window in your application. This includes a topmost main window in a single-document interface (SDI) application, modeless and modal dialog boxes, and the parent and child windows of a multiple-document interface (MDI) application. When you are interested in creating and displaying the main window in your program, you have two mandatory steps:

1. Derive a new class from `System.Windows.Forms.Form`.
2. Configure your application's `Main()` method to invoke `Application.Run()`, passing an instance of your `Form`-derived type as an argument.

Given this, update your `MainWindow.cs` file with the following class definition:

```
using System;
using System.Windows.Forms;

namespace MyWindowsApp
{
     public class MainWindow : Form
     {
          // Run this application and identify the main window.
          static void Main(string[] args)
          {
               Application.Run(new MainWindow());
          }
     }
}
```

In addition to the always present `mscorlib.dll`, a Windows Forms application needs to reference the `System.dll` and `System.Windows.Forms.dll` assemblies. As you may recall from Chapter 2, the default C# response file (`csc.rsp`) instructs `csc.exe` to automatically include these assemblies during the compilation process, so you are good to go. Also recall that the `/target:winexe` option of `csc.exe` instructs the compiler to generate a Windows executable.

■ **Note** Technically speaking, you can build a Windows application at the command line using the `/target:exe` option; however, if you do, you will find that a command window will be looming in the background (and it will stay there until you shut down the main window). When you specify `/target:winexe`, your executable runs as a native Windows Forms application (without the looming command window).

To compile your C# code file, open a Visual Studio 2005 command prompt and issue the following command:

```
csc /target:winexe *.cs
```

Figure 19-1 shows a test run.

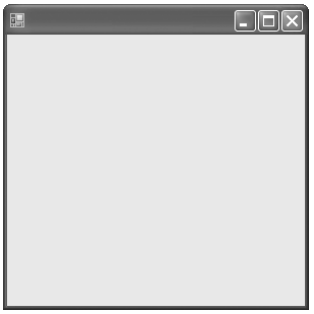

Figure 19-1. *A simple main window à la Windows Forms*

Granted, the Form is not altogether that interesting at this point. But simply by deriving from `Form`, you have a minimizable, maximizable, resizable, and closable main window (with a default system-supplied icon to boot!). Unlike other Microsoft GUI frameworks you may have used in the past (Microsoft Foundation Classes, in particular), there is no need to bolt in hundreds of lines of coding infrastructure (frames, documents, views, applications, or message maps). Unlike a C-based Win32 API Windows application, there is no need to manually implement `WinProc()` or `WinMain()` procedures. Under the .NET platform, those dirty details have been encapsulated within the `Form` and `Application` types.

Honoring the Separation of Concerns

Currently, the `MainWindow` class defines the `Main()` method directly within its scope. If you prefer, you may create a second static class (I named mine `Program`) that is responsible for the task of launching the main window, leaving the `Form`-derived class responsible for representing the window itself:

```
namespace MyWindowsApp
{
    // The main window.
    public class MainWindow : Form { }

    // The application object.
    public static class Program
    {
        static void Main(string[] args)
        {
             // Don't forget to 'use' System.Windows.Forms!
            Application.Run(new MainWindow());
        }
    }
}
```

By doing so, you are abiding by an OO principal termed *the separation of concerns*. Simply put, this rule of OO design states that a class should be in charge of doing the least amount of work possible. Given that you have refactored the initial class into two unique classes, you have decoupled the Form from the class that creates it. The end result is a more portable window, as it can be dropped into any project without carrying the extra baggage of a project-specific `Main()` method.

■ **Source Code** The MyFirstWindow project can be found under the Chapter 19 subdirectory.

The Role of the Application Class

The `Application` class defines numerous static members that allow you to control various low-level behaviors of a Windows Forms application. For example, the `Application` class defines a set of events that allow you to respond to events such as application shutdown and idle-time processing. In addition to the `Run()` method, here are some other methods to be aware of:

- `DoEvents()`: Provides the ability for an application to process messages currently in the message queue during a lengthy operation.
- `Exit()`: Terminates the Windows application and unloads the hosting AppDomain.
- `EnableVisualStyles()`: Configures your application to support Windows XP visual styles. Do note that if you enable XP styles, this method must be called before loading your main window via `Application.Run()`.

The `Application` class also defines a number of properties, many of which are read-only in nature. As you examine Table 19-2, note that most of these properties represent an "application-level" trait such as company name, version number, and so forth. In fact, given what you already know about assembly-level attributes (see Chapter 12), many of these properties should look vaguely familiar.

Table 19-2. *Core Properties of the* `Application` *Type*

Property	Meaning in Life
`CompanyName`	Retrieves the value of the assembly-level `[AssemblyCompany]` attribute
`ExecutablePath`	Gets the path for the executable file
`ProductName`	Retrieves the value of the assembly-level `[AssemblyProduct]` attribute
`ProductVersion`	Retrieves the value of the assembly-level `[AssemblyVersion]` attribute
`StartupPath`	Retrieves the path for the executable file that started the application

Finally, the `Application` class defines various static events, some of which are as follows:

- `ApplicationExit`: Occurs when the application is just about to shut down
- `Idle`: Occurs when the application's message loop has finished processing the current batch of messages and is about to enter an idle state (as there are no messages to process at the current time)
- `ThreadExit`: Occurs when a thread in the application is about to terminate

Fun with the Application Class

To illustrate some of the functionality of the `Application` class, let's enhance your current `MainWindow` to perform the following:

- Reflect over select assembly-level attributes.
- Handle the static `ApplicationExit` event.

The first task is to make use of select properties in the `Application` class to reflect over some assembly-level attributes. To begin, add the following attributes to your `MainWindow.cs` file (note you are now `using` the `System.Reflection` namespace):

```
using System;
using System.Windows.Forms;
using System.Reflection;
```

```
// Some attributes regarding this assembly.
[assembly:AssemblyCompany("Intertech Training")]
[assembly:AssemblyProduct("A Better Window")]
[assembly:AssemblyVersion("1.1.0.0")]

namespace MyWindowsApp
{
   ...
}
```

Rather than manually reflecting over the `[AssemblyCompany]` and `[AssemblyProduct]` attributes using the techniques illustrated in Chapter 12, the `Application` class will do so automatically using various static properties. To illustrate, implement the default constructor of `MainForm` as so:

```
public class MainWindow : Form
{
    public MainWindow()
    {
        MessageBox.Show(Application.ProductName,
            string.Format("This app brought to you by {0}",
            Application.CompanyName));
    }
}
```

When you run this application, you'll see a message box that displays various bits of information (see Figure 19-2).

Figure 19-2. *Reading attributes via the* `Application` *type*

Now, let's equip this Form to respond to the `ApplicationExit` event. When you wish to respond to events from within a Windows Forms application, you will be happy to find that the same event syntax detailed in Chapter 8 is used to handle GUI-based events. Therefore, if you wish to intercept the static `ApplicationExit` event, simply register an event handler using the `+=` operator:

```
public class MainForm : Form
{
     public MainForm()
     {
          ...
          // Intercept the ApplicationExit event.
          Application.ApplicationExit += new EventHandler(MainWindow_OnExit);
     }
     private void MainWindow_OnExit(object sender, EventArgs evArgs)
     {
            MessageBox.Show(string.Format("Form version {0} has terminated.",
                Application.ProductVersion));
     }
}
```

The System.EventHandler Delegate

Notice that the `ApplicationExit` event works in conjunction with the `System.EventHandler` delegate. This delegate must point to methods that conform to the following signature:

```
delegate void EventHandler(object sender, EventArgs e);
```

`System.EventHandler` is the most primitive delegate used to handle events within Windows Forms, but many variations do exist for other events. As far as `EventHandler` is concerned, the first parameter of the assigned method is of type `System.Object`, which represents the object sending the event. The second `EventArgs` parameter (or a descendent thereof) contains any relevant information regarding the current event.

■ **Note** `EventArgs` is the base class to numerous derived types that contain information for a family of related events. For example, mouse events work with the `MouseEventArgs` parameter, which contains details such as the (*x*, *y*) position of the cursor. Many keyboard events work with the `KeyEventArgs` type, which contains details regarding the current keypress, and so forth.

In any case, if you now recompile and run the application, you will find your message box appear upon the termination of the application.

■ **Source Code** The AppClassExample project can be found under the Chapter 19 subdirectory.

The Anatomy of a Form

Now that you understand the role of the `Application` type, the next task is to examine the functionality of the `Form` class itself. Not surprisingly, the `Form` class inherits a great deal of functionality from its parent classes. Figure 19-3 shows the inheritance chain (including the set of implemented interfaces) of a `Form`-derived type using the Visual Studio 2005 Object Browser.

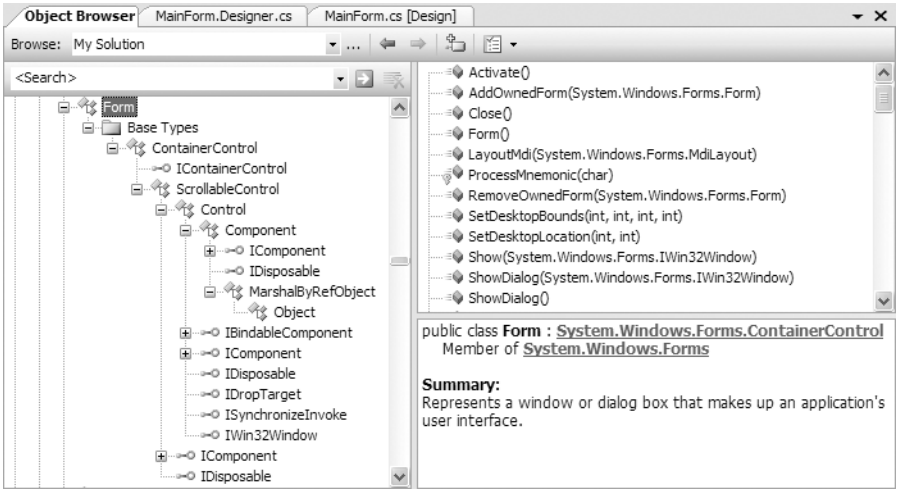

Figure 19-3. *The derivation of the* `Form` *type*

Although the complete derivation of a `Form` type involves numerous base classes and interfaces, do understand that you are *not* required to learn the role of each and every member from each and every parent class or implemented interface to be a proficient Windows Forms developer. In fact, the majority of the members (properties and events in particular) you will use on a daily basis are easily set using the Visual Studio 2005 IDE Properties window. Before we move on to examine some specific members inherited from these parent classes, Table 19-3 outlines the basic role of each base class.

Table 19-3. *Base Classes in the* `Form` *Inheritance Chain*

Parent Class	Meaning in Life
`System.Object`	Like any class in .NET, a Form "is-a" object.
`System.MarshalByRefObject`	Recall during our examination of .NET remoting (see Chapter 18) that types deriving from this class are accessed remotely via a *reference* (not a copy) of the remote type.
`System.ComponentModel.Component`	This class provides a default implementation of the `IComponent` interface. In the .NET universe, a component is a type that supports design-time editing, but is not necessarily visible at runtime.
`System.Windows.Forms.Control`	This class defines common UI members for all Windows Forms UI controls, including the `Form` type itself.
`System.Windows.Forms.ScrollableControl`	This class defines support for auto-scrolling behaviors.
`System.Windows.Forms.ContainerControl`	This class provides focus-management functionality for controls that can function as a container for other controls.
`System.Windows.Forms.Form`	This class represents any custom Form, MDI child, or dialog box.

As you might guess, detailing each and every member of each class in the `Form`'s inheritance chain would require a large book in itself. However, it is important to understand the behavior supplied by the `Control` and `Form` types. I'll assume that you will spend time examining the full details behind each class at your leisure using the .NET Framework 2.0 SDK documentation.

The Functionality of the Control Class

The `System.Windows.Forms.Control` class establishes the common behaviors required by any GUI type. The core members of `Control` allow you to configure the size and position of a control, capture keyboard and mouse input, get or set the focus/visibility of a member, and so forth. Table 19-4 defines some (but not all) properties of interest, grouped by related functionality.

Table 19-4. *Core Properties of the* `Control` *Type*

Properties	Meaning in Life
`BackColor`, `ForeColor`, `BackgroundImage`, `Font`, `Cursor`	These properties define the core UI of the control (colors, font for text, mouse cursor to display when the mouse is over the widget, etc.).
`Anchor`, `Dock`, `AutoSize`	These properties control how the control should be positioned within the container.
`Top`, `Left`, `Bottom`, `Right`, `Bounds`, `ClientRectangle`, `Height`, `Width`	These properties specify the current dimensions of the control.
`Enabled`, `Focused`, `Visible`	These properties each return a Boolean that specifies the state of the current control.
`ModifierKeys`	This static property checks the current state of the modifier keys (Shift, Ctrl, and Alt) and returns the state in a `Keys` type.
`MouseButtons`	This static property checks the current state of the mouse buttons (left, right, and middle mouse buttons) and returns this state in a `MouseButtons` type.
`TabIndex`, `TabStop`	These properties are used to configure the tab order of the control.
`Opacity`	This property determines the opacity of the control, in fractions (0.0 is completely transparent; 1.0 is completely opaque).
`Text`	This property indicates the string data associated with this control.
`Controls`	This property allows you to access a strongly typed collection (`ControlsCollection`) that contains any child controls within the current control.

As you would guess, the `Control` class also defines a number of events that allow you to intercept mouse, keyboard, painting, and drag-and-drop activities (among other things). Table 19-5 lists some (but not all) events of interest, grouped by related functionality.

Table 19-5. *Events of the* `Control` *Type*

Events	Meaning in Life
`Click`, `DoubleClick`, `MouseEnter`, `MouseLeave`, `MouseDown`, `MouseUp`, `MouseMove`, `MouseHover`, `MouseWheel`	Various events that allow you to interact with the mouse
`KeyPress`, `KeyUp`, `KeyDown`	Various events that allow you to interact with the keyboard
`DragDrop`, `DragEnter`, `DragLeave`, `DragOver`	Various events used to monitor drag-and-drop activity
`Paint`	This event allows you to interact with GDI+ (see Chapter 20)

Finally, the `Control` base class also defines a number of methods that allow you to interact with any `Control`-derived type. As you examine the methods of the `Control` type, you will notice that a good number of them have an `On` prefix followed by the name of a specific event (`OnMouseMove`, `OnKeyUp`, `OnPaint`, etc.). Each of these `On`-prefixed virtual methods is the default event handler for its respective event. If you override any of these virtual members, you gain the ability to perform any necessary pre- or postprocessing of the event before (or after) invoking your parent's default implementation:

```
public class MainWindow : Form
{
    protected override void OnMouseDown(MouseEventArgs e)
    {
        // Add code for MouseDown event.

        // Call parent implementation when finished.
        base.OnMouseDown(e);
    }
}
```

While this can be helpful in some circumstances (especially if you are building a custom control that derives from a standard control; see Chapter 21), you will often handle events using the standard C# event syntax (in fact, this is the default behavior of the Visual Studio 2005 designers). When you do so, the framework will call your custom event handler once the parent's implementation has completed:

```
public class MainWindow : Form
{
    public MainWindow()
    {
        MouseDown += new MouseEventHandler(MainWindow_MouseDown);
    }

    void MainWindow_MouseDown(object sender, MouseEventArgs e)
    {
        // Add code for MouseDown event.
    }
}
```

Beyond these `OnXXX()` methods, here are a few other methods to be aware of:

- `Hide()`: Hides the control and sets the `Visible` property to false
- `Show()`: Shows the control and sets the `Visible` property to true
- `Invalidate()`: Forces the control to redraw itself by sending a `Paint` event

To be sure, the `Control` class does define additional properties, methods, and events beyond the subset you've just examined. You should, however, now have a solid understanding regarding the overall functionality of this base class. Let's see it in action.

Fun with the Control Class

To illustrate the usefulness of some members from the `Control` class, let's build a new Form that is capable of handling the following events:

- Respond to the `MouseMove` and `MouseDown` events.
- Capture and process keyboard input via the `KeyUp` event.

To begin, create a new class derived from `Form`. In the default constructor, you'll make use of various inherited properties to establish the initial look and feel. Note you're now `using` the `System.Drawing` namespace to gain access to the `Color` structure (you'll examine this namespace in detail in the next chapter):

```
using System;
using System.Windows.Forms;
using System.Drawing;
```

```
namespace MyWindowsApp
{
    public class MainWindow : Form
    {
        public MainWindow()
        {
            // Use inherited properties to set basic UI.
            Text = "My Fantastic Form";
            Height = 300;
            Width = 500;
            BackColor = Color.LemonChiffon;
            Cursor = Cursors.Hand;
        }
    }

    public static class Program
    {
        static void Main(string[] args)
        {
            Application.Run(new MainWindow());
        }
    }
}
```

Compile your application at this point, just to make sure you have not injected any typing errors:

```
csc /target:winexe *.cs
```

Responding to the MouseMove Event

Next, you need to handle the `MouseMove` event. The goal is to display the current (*x*, *y*) location within the Form's caption area. All mouse-centric events (`MouseMove`, `MouseUp`, etc.) work in conjunction with the `MouseEventHandler` delegate, which can call any method matching the following signature:

```
void MyMouseHandler(object sender, MouseEventArgs e);
```

The incoming `MouseEventArgs` structure extends the general `EventArgs` base class by adding a number of members particular to the processing of mouse activity (see Table 19-6).

Table 19-6. *Properties of the* `MouseEventArgs` *Type*

Property	Meaning in Life
`Button`	Gets which mouse button was pressed, as defined by the `MouseButtons` enumeration
`Clicks`	Gets the number of times the mouse button was pressed and released
`Delta`	Gets a signed count of the number of detents the mouse wheel has rotated
`X`	Gets the *x*-coordinate of a mouse click
`Y`	Gets the *y*-coordinate of a mouse click

Here, then, is the updated `MainForm` class that handles the `MouseMove` event as intended:

```
public class MainForm : Form
{
     public MainForm()
     {
```

```
        ...
        // Handle the MouseMove event
        MouseMove += new MouseEventHandler(MainForm_MouseMove);
    }
    // MouseMove event handler.
    public void MainForm_MouseMove(object sender, MouseEventArgs e)
    {
       Text = string.Format("Current Pos: ({0}, {1})", e.X, e.Y);
    }
}
```

If you now run your program and move the mouse over your Form, you will find the current (*x*, *y*) value display on the caption area (see Figure 19-4).

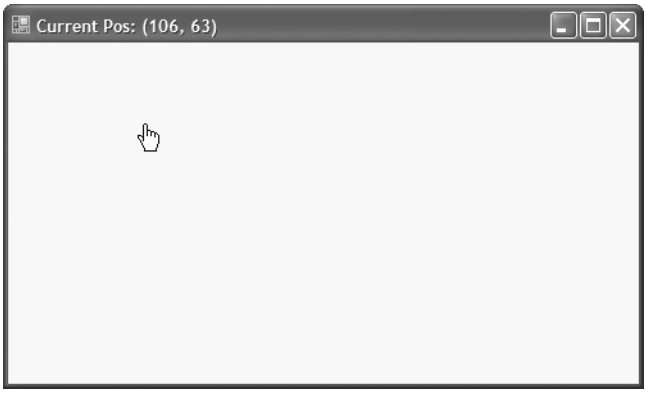

Figure 19-4. *Monitoring mouse movement*

Determining Which Mouse Button Was Clicked

One thing to be aware of is that the `MouseUp` (or `MouseDown`) event is sent whenever *any* mouse button is clicked. If you wish to determine exactly *which* button was clicked (such as left, right, or middle), you need to examine the `Button` property of the `MouseEventArgs` class. The value of the `Button` property is constrained by the related `MouseButtons` enumeration. Assume you have updated your default constructor to handle the `MouseUp` event as so:

```
public MainWindow()
{
...
    // Handle the MouseUp event.
    MouseUp += new MouseEventHandler(MainForm_MouseUp);
}
```

The following `MouseUp` event handler displays which mouse button was clicked inside a message box:

```
public void MainForm_MouseUp (object sender, MouseEventArgs e)
{
    // Which mouse button was clicked?
    if(e.Button == MouseButtons.Left)
        MessageBox.Show("Left click!");
    if(e.Button == MouseButtons.Right)
```

```
            MessageBox.Show("Right click!");
        if (e.Button == MouseButtons.Middle)
            MessageBox.Show("Middle click!");
}
```

Responding to Keyboard Events

Processing keyboard input is almost identical to responding to mouse activity. The `KeyUp` and `KeyDown` events work in conjunction with the `KeyEventHandler` delegate, which can point to any method taking an object as the first parameter and `KeyEventArgs` as the second:

```
void MyKeyboardHandler(object sender, KeyEventArgs e);
```

`KeyEventArgs` has the members of interest shown in Table 19-7.

Table 19-7. *Properties of the* `KeyEventArgs` *Type*

Property	Meaning in Life
`Alt`	Gets a value indicating whether the Alt key was pressed
`Control`	Gets a value indicating whether the Ctrl key was pressed
`Handled`	Gets or sets a value indicating whether the event was fully handled in your handler
`KeyCode`	Gets the keyboard code for a `KeyDown` or `KeyUp` event
`Modifiers`	Indicates which modifier keys (Ctrl, Shift, and/or Alt) were pressed
`Shift`	Gets a value indicating whether the Shift key was pressed

Update your `MainForm` to handle the `KeyUp` event. Once you do, display the name of the key that was pressed inside a message box using the `KeyCode` property.

```
public class MainForm : Form
{
    public MainForm()
    {
        ...
        // Listen for the KeyUp Event.
        KeyUp += new KeyEventHandler(MainForm_KeyUp);
    }
    private void MainForm_KeyUp (object sender, KeyEventArgs e)
    {
        MessageBox.Show(e.KeyCode.ToString(), "Key Pressed!");
    }
}
```

Now compile and run your program. You should be able to determine not only which mouse button was clicked, but also which keyboard key was pressed.

That wraps up our look at the core functionality of the `Control` base class. Next up, let's check out the role of `Form`.

■ **Source Code** The ControlBehaviors project is included under the Chapter 19 subdirectory.

The Functionality of the Form Class

The `Form` class is typically (but not necessarily) the direct base class for your custom `Form` types. In addition to the large set of members inherited from the `Control`, `ScrollableControl`, and `ContainerControl` classes, the `Form` type adds additional functionality in particular to main windows, MDI child windows, and dialog boxes. Let's start with the core properties in Table 19-8.

Table 19-8. *Properties of the* `Form` *Type*

Properties	Meaning in Life
`AcceptButton`	Gets or sets the button on the Form that is clicked when the user presses the Enter key.

`ActiveMDIChild`
`IsMDIChild`
`IsMDIContainer`
	Used within the context of an MDI application.

`CancelButton`	Gets or sets the button control that will be clicked when the user presses the Esc key.
`ControlBox`	Gets or sets a value indicating whether the Form has a control box.
`FormBorderStyle`	Gets or sets the border style of the Form. Used in conjunction with the `FormBorderStyle` enumeration.
`Menu`	Gets or sets the menu to dock on the Form.
`MaximizeBox` `MinimizeBox`	Used to determine if this Form will enable the maximize and minimize boxes.
`ShowInTaskbar`	Determines if this Form will be seen on the Windows taskbar.
`StartPosition`	Gets or sets the starting position of the Form at runtime, as specified by the `FormStartPosition` enumeration.
`WindowState`	Configures how the Form is to be displayed on startup. Used in conjunction with the `FormWindowState` enumeration.

In addition to the expected `On`-prefixed default event handlers, Table 19-9 gives a list of some core methods defined by the `Form` type.

Table 19-9. *Key Methods of the* `Form` *Type*

Method	Meaning in Life
`Activate()`	Activates a given Form and gives it focus.
`Close()`	Closes a Form.
`CenterToScreen()`	Places the Form in the dead-center of the screen.
`LayoutMDI()`	Arranges each child Form (as specified by the `LayoutMDI` enumeration) within the parent Form.
`ShowDialog()`	Displays a Form as a modal dialog box. More on dialog box programming in Chapter 21.

Finally, the `Form` class defines a number of events, many of which fire during the form's lifetime. Table 19-10 hits the highlights.

Table 19-10. *Select Events of the* `Form` *Type*

Events	Meaning in Life
`Activated`	Occurs whenever the Form is *activated*, meaning the Form has been given the current focus on the desktop
`Closed`, `Closing`	Used to determine when the Form is about to close or has closed
`Deactivate`	Occurs whenever the Form is *deactivated*, meaning the Form has lost current focus on the desktop
`Load`	Occurs after the Form has been allocated into memory, but is not yet visible on the screen
`MDIChildActive`	Sent when a child window is activated

The Life Cycle of a Form Type

If you have programmed user interfaces using GUI toolkits such as Java Swing, Mac OS X Cocoa, or the raw Win32 API, you are aware that "window types" have a number of events that fire during their lifetime. The same holds true for Windows Forms. As you have seen, the life of a Form begins when the type constructor is called prior to being passed into the `Application.Run()` method.

Once the object has been allocated on the managed heap, the framework fires the `Load` event. Within a `Load` event handler, you are free to configure the look and feel of the Form, prepare any contained child controls (such as `ListBoxes`, `TreeViews`, and whatnot), or simply allocate resources used during the Form's operation (database connections, proxies to remote objects, and whatnot).

Once the `Load` event has fired, the next event to fire is `Activated`. This event fires when the Form receives focus as the active window on the desktop. The logical counterpart to the `Activated` event is (of course) `Deactivate`, which fires when the Form loses focus as the active window. As you can guess, the `Activated` and `Deactivate` events can fire numerous times over the life of a given `Form` type as the user navigates between active applications.

When the user has chosen to close the Form in question, two close-centric events fire: `Closing` and `Closed`. The `Closing` event is fired first and is an ideal place to prompt the end user with the much hated (but useful) "Are you *sure* you wish to close this application?" message. This confirmational step is quite helpful to ensure the user has a chance to save any application-centric data before terminating the program.

The `Closing` event works in conjunction with the `CancelEventHandler` delegate defined in the `System.ComponentModel` namespace. If you set the `CancelEventArgs.Cancel` property to true, you prevent the Form from being destroyed and instruct it to return to normal operation. If you set `CancelEventArgs.Cancel` to false, the `Close` event fires and the Windows Forms application terminates, which unloads the AppDomain and terminates the process.

To solidify the sequence of events that take place during a Form's lifetime, assume you have a new `MainWindow.cs` file that handles the `Load`, `Activated`, `Deactivate`, `Closing`, and `Close` events within the class constructor (be sure to add a `using` directive for the `System.ComponentModel` namespace to obtain the definition of `CancelEventArgs`):

```
public MainForm()
{
     // Handle various lifetime events.
     Closing += new CancelEventHandler(MainForm_Closing);
     Load += new EventHandler(MainForm_Load);
     Closed += new EventHandler(MainForm_Closed);
     Activated += new EventHandler(MainForm_Activated);
     Deactivate += new EventHandler(MainForm_Deactivate);
}
```

In the `Load`, `Closed`, `Activated`, and `Deactivate` event handlers, you are going to update the value of a new Form-level `System.String` member variable (named `lifeTimeInfo`) with a simple message that displays the name of the event that has just been intercepted. As well, notice that within the `Closed` event handler, you will display the value of this string within a message box:

```
private void MainForm_Load(object sender, System.EventArgs e)
{ lifeTimeInfo += "Load event\n"; }
private void MainForm_Activated(object sender, System.EventArgs e)
{ lifeTimeInfo += "Activate event\n"; }
private void MainForm_Deactivate(object sender, System.EventArgs e)
{ lifeTimeInfo += "Deactivate event\n"; }
private void MainForm_Closed(object sender, System.EventArgs e)
{
    lifeTimeInfo += "Closed event\n";
    MessageBox.Show(lifeTimeInfo);
}
```

Within the `Closing` event handler, you will prompt the user to ensure she wishes to terminate the application using the incoming `CancelEventArgs`:

```
private void MainForm_Closing(object sender, CancelEventArgs e)
{
    DialogResult dr = MessageBox.Show("Do you REALLY want to close this app?",
        "Closing event!", MessageBoxButtons.YesNo);
    if (dr == DialogResult.No)
        e.Cancel = true;
    else
        e.Cancel = false;
}
```

Notice that the `MessageBox.Show()` method returns a `DialogResult` type, which has been set to a value representing the button clicked by the end user (`Yes` or `No`). Now, compile your code at the command line:

```
csc /target:winexe *.cs
```

Now run your application and shift the Form into and out of focus a few times (to trigger the `Activated` and `Deactivate` events). Once you shut down the Form, you will see a message box that looks something like Figure 19-5.

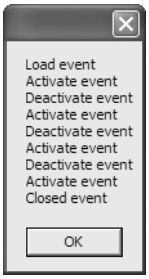

Figure 19-5. *The life and times of a* `Form`*-derived type*

Now, most of the really interesting aspects of the `Form` type have to do with its ability to create and host menu systems, toolbars, and status bars. While the code to do so is not complex, you will be happy to know that Visual Studio 2005 defines a number of graphical designers that take care of

most of the mundane code on your behalf. Given this, let's say good-bye to the command-line compiler for the time being and turn our attention to the process of building Windows Forms applications using Visual Studio 2005.

■ **Source Code** The FormLifeTime project can be found under the Chapter 19 subdirectory.

Building Windows Applications with Visual Studio 2005

Visual Studio 2005 has a specific project type dedicated to the creation of Windows Forms applications. When you select the Windows Application project type, you not only receive an application object with a proper `Main()` method, but also are provided with an initial `Form`-derived type. Better yet, the IDE provides a number of graphical designers that make the process of building a UI child's play. Just to learn the lay of the land, create a new Windows Application project workspace (see Figure 19-6). You are not going to build a working example just yet, so name this project whatever you desire.

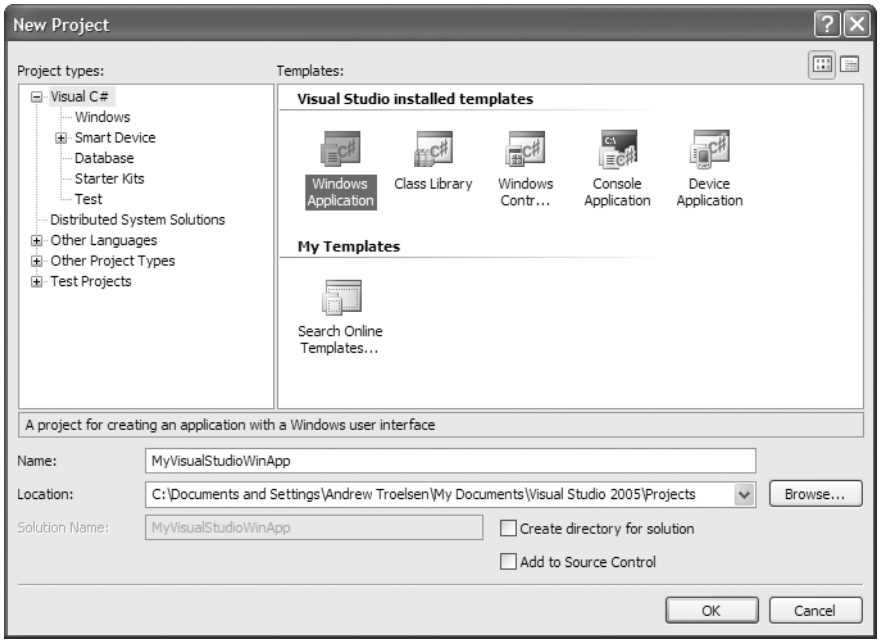

Figure 19-6. *The Visual Studio 2005 Windows Application project*

Once the project has loaded, you will no doubt notice the Forms designer, which allows you to build a UI by dragging controls/components from the Toolbox (see Figure 19-7) and configuring their properties and events using the Properties window (see Figure 19-8).

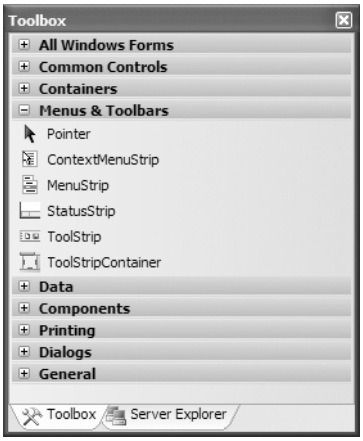

Figure 19-7. *The Visual Studio 2005 Toolbox*

Figure 19-8. *The Visual Studio 2005 Properties window*

As you can see, the Toolbox groups UI controls by various categories. While most are self-explanatory (e.g., Printing contains printing controls, Menus & Toolbars contains recommended menu/toolbar controls, etc.), a few categories deserve special mention:

- *Common Controls*: Members in this category are considered the "recommended set" of common UI controls.
- *All Windows Forms*: Here you will find the full set of Windows Forms controls, including various .NET 1.*x* controls that are considered depreciated.

The second bullet point is worth reiterating. If you have worked with Windows Forms using .NET 1.*x*, be aware that many of your old friends (such as the `DataGrid` control) have been placed under the All Windows Forms category. Furthermore, the common UI controls you may have used under .NET 1.*x* (such as `MainMenu`, `ToolBar`, and `StatusBar`) are *not* shown in the Toolbox by default.

Enabling the Deprecated Controls

The first bit of good news is that these (deprecated) UI elements are still completely usable under .NET 2.0. The second bit of good news is that if you still wish to program with them, you can add them back to the Toolbox by right-clicking anywhere in the Toolbox and selecting Choose Items. From the resulting dialog box, check off the items of interest (see Figure 19-9).

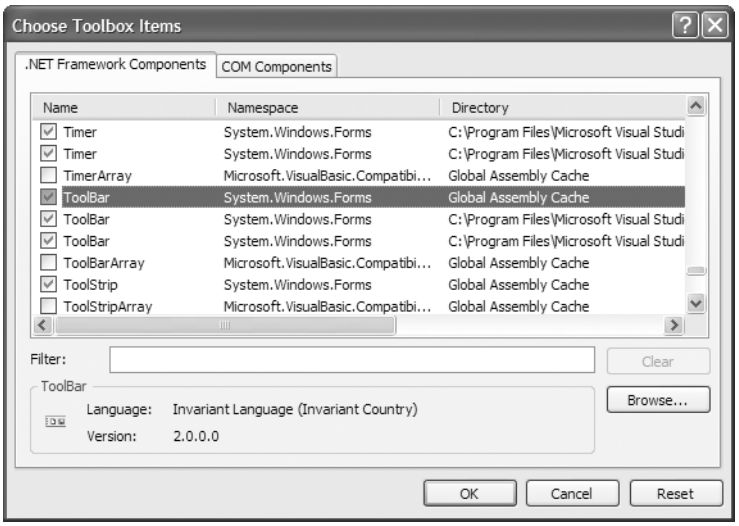

Figure 19-9. *Adding additional controls to the Toolbox*

■ **Note** At first glance, it might appear that there are redundant listings for a given control (such as the `ToolBar`). In reality, each listing is unique, as a control may be versioned (1.0 versus 2.0) and/or may be a member of the .NET Compact Framework. Be sure to examine the directory path to select the correct item.

At this point, I am sure you are wondering why many of these old standbys have been hidden from view. The reason is that .NET 2.0 provides a set of new menu, toolbar, and status bar–centric controls that are now favored. For example, rather than using the legacy `MainMenu` control to build a menu, you can use the `MenuStrip` control, which provides a number of new bells and whistles in addition to the functionality found within `MainMenu`.

■ **Note** In this chapter, I will favor the use of this new recommend set of UI elements. If you wish to work with the legacy `MainMenu`, `StatusBar`, or `ToolBar` types, consult the .NET Framework 2.0 SDK documentation.

Dissecting a Visual Studio 2005 Windows Forms Project

Each `Form` in a Visual Studio 2005 Windows Application project is composed of two related C# files, which can be verified using Solution Explorer (see Figure 19-10).

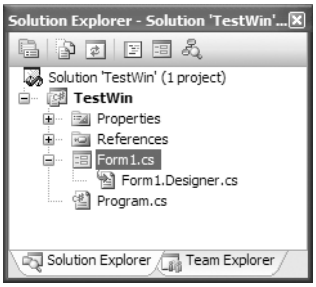

Figure 19-10. *Each Form is composed of two* `*.cs` *files.*

Right-click the `Form1.cs` icon and select View Code. Here you will see a partial class that contains all of the Form's event handlers, constructors, overrides, and any member you author yourself (note that I renamed this initial class from `Form1` to `MainWindow` using the `Rename` refactoring):

```
namespace MyVisualStudioWinApp
{
    public partial class MainWindow : Form
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

The default constructor of your Form makes a call to a method named `InitializeComponent()`, which is defined within the related `*.Designer.cs` file. This method is maintained on your behalf by Visual Studio 2005, and it contains all of the code representing your design-time modifications.

To illustrate, switch back to the Forms designer and locate the `Text` property in the Properties window. Change this value to something like **My Test Window**. Now open your `Form1.Designer.cs` file and notice that `InitializeComponent()` has been updated accordingly:

```
private void InitializeComponent()
{
...
    this.Text = "My Test Window";
}
```

In addition to maintaining `InitializeComponent()`, the `*.Designer.cs` file will define the member variables that represent each control placed on the designer. Again, to illustrate, drag a `Button` control onto the Forms designer. Now, using the Properties window, rename your member variable from `button1` to `btnTestButton` via the `Name` property.

■ **Note** It is always a good idea to rename the controls you place on the designer before handling events. If you fail to do so, you will most likely end up with a number of nondescript event handlers, such as `button27_Click`, given that the default names simply suffix a numerical value to the variable name.

Handling Events at Design Time

Notice that the Properties window has a button depicting a lightning bolt. Although you are always free to handle Form-level events by authoring the necessary logic by hand (as done in the previous examples), this event button allows you to visually handle an event for a given control. Simply select the control you wish to interact with from the drop-down list box (mounted at the top of the Properties window), locate the event you are interested in handling, and type in the name to be used as an event handler (or simply double-click the event to generate a default name of the form `ControlName_EventName`).

Assuming you have handled the `Click` event for the `Button` control, you will find that the `Form1.cs` file contains the following event handler:

```
public partial class MainWindow : Form
{
    public MainWindow()
    {
        InitializeComponent();
    }
    private void btnButtonTest_Click(object sender, EventArgs e)
    {
    }
}
```

As well, the `Form1.Designer.cs` file contains the necessary infrastructure and member variable declaration:

```
partial class MainWindow
{
...
    private void InitializeComponent()
    {
...
        this.btnButtonTest.Click +=
            new System.EventHandler(this.btnButtonTest_Click);
    }
    private System.Windows.Forms.Button btnButtonTest;
}
```

■ **Note** Every control has a *default event*, which refers to the event that will be handled if you double-click the item on the control using the Forms designer. For example, a Form's default event is `Load`, and if you double-click anywhere on a Form type, the IDE will automatically write code to handle this event.

The Program Class

Beyond the Form-centric files, a Visual Studio 2005 Windows application defines a second class that represents the application object (e.g., the type defining the `Main()` method). Notice that the `Main()` method has been configured to call `Application.EnableVisualStyles()` as well as `Application.Run()`:

```
static class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
```

```
        Application.Run(new MainWindow());
    }
}
```

■ **Note** The `[STAThread]` attribute instructs the CLR to host any legacy COM objects (including ActiveX controls) in a *single-threaded apartment* (*STA*). If you have a background in COM, you may recall that the STA was used to ensure access to a COM type occurred in a synchronous (hence, thread-safe) manner.

Autoreferenced Assemblies

Finally, if you examine Solution Explorer, you will notice that a Windows Forms project automatically references a number of assemblies, including `System.Windows.Forms.dll` and `System.Drawing.dll`. Again, the details of `System.Drawing.dll` will be examined in the next chapter.

Working with MenuStrips and ContextMenuStrips

As of .NET 2.0, the recommended control for building a menu system is `MenuStrip`. This control allows you to create "normal" menu items such as File ➤ Exit, and you may also configure it to contain any number of relevant controls within the menu area. Here are some common UI elements that may be contained within a `MenuStrip`:

- `ToolStripMenuItem`: A traditional menu item
- `ToolStripComboBox`: An embedded `ComboBox`
- `ToolStripSeparator`: A simple line that separates content
- `ToolStripTextBox`: An embedded `TextBox`

Programmatically speaking, the `MenuStrip` control contains a strongly typed collection named `ToolStripItemCollection`. Like other collection types, this object supports members such as `Add()`, `AddRange()`, `Remove()`, and the `Count` property. While this collection is typically populated indirectly using various design-time tools, you are able to manually manipulate this collection if you so choose.

To illustrate the process of working with the `MenuStrip` control, create a new Windows Forms application named MenuStripApp. Using the Forms designer, place a `MenuStrip` control named `mainMenuStrip` onto your Form. When you do so, your `*.Designer.cs` file is updated with a new `MenuStrip` member variable:

```
private System.Windows.Forms.MenuStrip mainMenuStrip;
```

`MenuStrip`s can be highly customized using the Visual Studio 2005 Forms designer. For example, if you look at the extreme upper-left of the control, you will notice a small arrow icon. After you select this icon, you are presented with a context-sensitive "inline editor," as shown in Figure 19-11.

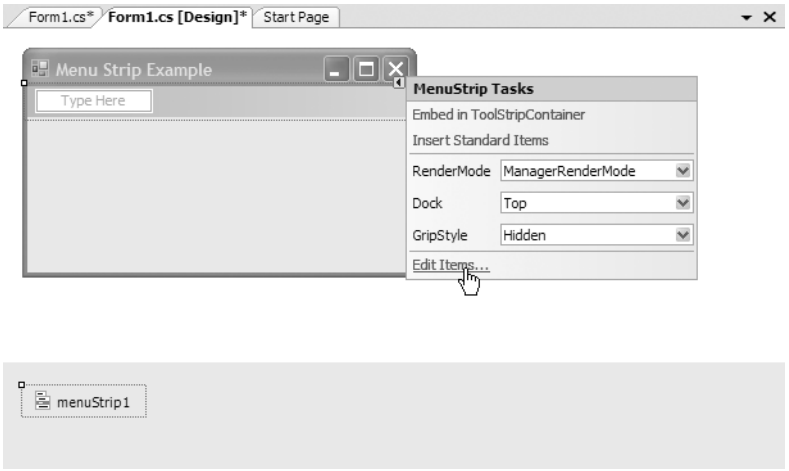

Figure 19-11. *The inline* `MenuStrip` *editor*

Many Windows Forms controls support such context-sensitive inline editors. As far as `MenuStrip` is concerned, the editor allows you to quickly do the following:

- Insert a "standard" menu system (File, Save, Tools, Help, etc.) using the Insert Standard Items link.
- Change the docking and gripping behaviors of the `MenuStrip`.
- Edit each item in the `MenuStrip` (this is simply a shortcut to selecting a specific item in the Properties window).

For this example, you'll ignore the options of the inline editor and stay focused on the design of the menu system. To begin, select the `MenuStrip` control on the designer and define a standard File ➤ Exit menu by typing in the names within the Type Here prompts (see Figure 19-12).

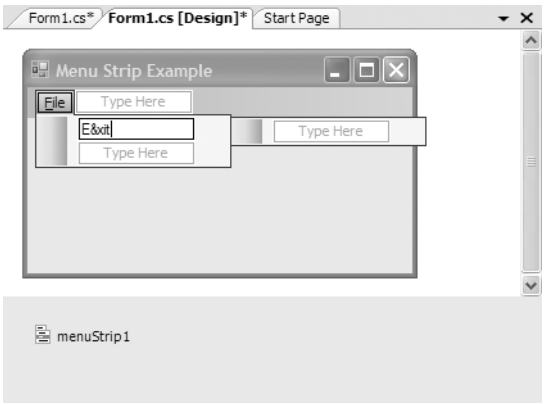

Figure 19-12. *Designing a menu system*

■ **Note** As you may know, when the ampersand character (&) is placed before a letter in a menu item, it denotes the item's shortcut key. In this example, you are creating &File ➤ E&xit; therefore, the user may activate the Exit menu by pressing Alt+f, and then x.

Each menu item you type into the designer is represented by the `ToolStripMenuItem` class type. If you open your `*.Designer.cs` file, you will find two new member variables for each item:

```
partial class MainWindow
{
...
    private System.Windows.Forms.MenuStrip mainMenuStrip;
    private System.Windows.Forms.ToolStripMenuItem fileToolStripMenuItem;
    private System.Windows.Forms.ToolStripMenuItem exitToolStripMenuItem;
}
```

When you use the menu editor, the `InitializeComponent()` method is updated accordingly. Notice that the `MenuStrip`'s internal `ToolStripItemCollection` has been updated to contain the new topmost menu item (`fileToolStripMenuItem`). In a similar fashion, the `fileToolStripMenuItem` variable has been updated to insert the `exitToolStripMenuItem` variable into its `ToolStripItemCollection` collection via the `DropDownItems` property:

```
private void InitializeComponent()
{
...
    //
    // menuStrip1
    //
    this.menuStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
      this.fileToolStripMenuItem});
   ...
    //
    // fileToolStripMenuItem
    //
    this.fileToolStripMenuItem.DropDownItems.AddRange(new
      System.Windows.Forms.ToolStripItem[] {
       this.exitToolStripMenuItem});
...
    //
    // MainWindow
    //
    this.Controls.Add(this.menuStrip1);
}
```

Last but not least, notice that the `MenuStrip` control is inserted to the Form's controls collection. This collection will be examined in greater detail in Chapter 21, but for the time being, just know that in order for a control to be visible at runtime, it must be a member of this collection.

To finish the initial code of this example, return to the designer and handle the `Click` event for the Exit menu item using the events button of the Properties window. Within the generated event handler, make a call to `Application.Exit`:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

At this point, you should be able to compile and run your program. Verify that you can terminate the application via File ➤ Exit as well as pressing Alt+f and then x on the keyboard.

Adding a TextBox to the MenuStrip

Now, let's create a new topmost menu item named Change Background Color. The subitem in this case will not be a menu item, but a `ToolStripTextBox` (see Figure 19-13). Once you have added the new control, rename this control to `toolStripTextBoxColor` using the Properties window.

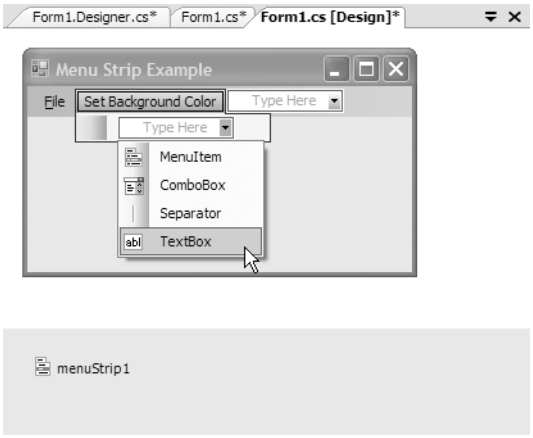

Figure 19-13. *Adding* `TextBox`*es to a* `MenuStrip`

The goal here is to allow the user to enter the name of a color (red, green, pink, etc.) that will be used to set the `BackColor` property of the Form. First, handle the `LostFocus` event on the new `ToolStripTextBox` member variable within the Form's constructor (as you would guess, this event fires when the `TextBox` within the `ToolStrip` is no longer the active UI element):

```
public MainWindow()
{
...
    toolStripTextBoxColor.LostFocus
        += new EventHandler(toolStripTextBoxColor_LostFocus);
}
```

Within the generated event handler, you will extract the string data entered within the `ToolStripTextBox` (via the `Text` property) and make use of the `System.Drawing.Color.FromName()` method. This static method will return a `Color` type based on a known string value. To account for the possibility that the user enters an unknown color (or types bogus data), you will make use of some simple `try/catch` logic:

```
void toolStripTextBoxColor_LostFocus(object sender, EventArgs e)
{
    try
    {
        BackColor = Color.FromName(toolStripTextBoxColor.Text);
    } catch { } // Just do nothing if the user provides bad data.
}
```

Go ahead and take your updated application out for another test drive and try entering in the names of various colors. Once you do, you should see your Form's background color change. If you are interested in checking out some valid color names, look up the `System.Drawing.Color` type using the Visual Studio 2005 Object Browser or the .NET Framework 2.0 SDK documentation.

Creating a Context Menu

Let's now examine the process of building a context-sensitive pop-up (i.e., right-click) menu. Under .NET 1.1, the `ContextMenu` type was the class of choice for building context menus, but under .NET 2.0 the preferred type is `ContextMenuStrip`. Like the `MenuStrip` type, `ContextMenuStrip` maintains a `ToolStripItemCollection` to represent the possible subitems (such as `ToolStripMenuItem`, `ToolStripComboBox`, `ToolStripSeperator`, `ToolStripTextBox`, etc.).

Drag a new `ContextMenuStrip` control from the Toolbox onto the Forms designer and rename the control to `fontSizeContextStrip` using the Properties window. Notice that you are able to populate the subitems graphically in much the same way you would edit the Form's main `MenuStrip` (a welcome change from the method used in Visual Studio .NET 2003). For this example, add three `ToolStripMenuItems` named `Huge`, `Normal`, and `Tiny` (see Figure 19-14).

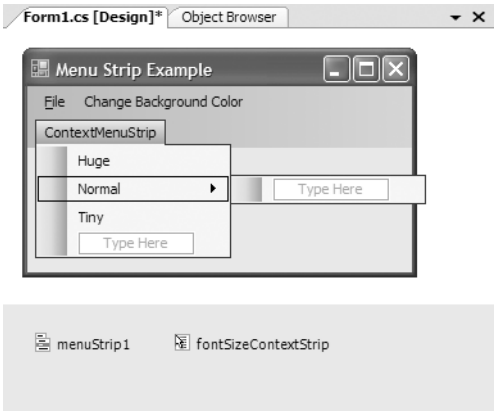

Figure 19-14. *Designing a* `ContextMenuStrip`

This context menu will be used to allow the user to select the size to render a message within the Form's client area. To facilitate this endeavor, create an enum type named `TextFontSize` within the `MenuStripApp` namespace and declare a new member variable of this type within your `Form` type (set to `TextFontSize.FontSizeNormal`):

```
namespace MainForm
{
     // Helper enum for font size.
     enum TextFontSize
     {
          FontSizeHuge = 30,
          FontSizeNormal = 20,
          FontSizeTiny = 8
     }

     public class MainForm : Form
     {
```

```
        // Current size of font.
        private TextFontSize currFontSize
             = TextFontSize.FontSizeNormal;
...
    }
}
```

The next step is to handle the Form's `Paint` event using the Properties window. As described in greater detail in the next chapter, the `Paint` event allows you to render graphical data (including stylized text) onto a Form's client area. Here, you are going to draw a textual message using a font of user-specified size. Don't sweat the details at this point, but do update your `Paint` event handler as so:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawString("Right click on me...",
            new Font("Times New Roman", (float)currFontSize),
            new SolidBrush(Color.Black), 50, 50);
}
```

Last but not least, you need to handle the `Click` events for each of the `ToolStripMenuItem` types maintained by the `ContextMenuStrip`. While you could have a separate `Click` event handler for each, you will simply specify a single event handler that will be called when any of the three `ToolStripMenuItems` have been clicked. Using the Properties window, specify the name of the `Click` event handler as `ContextMenuItemSelection_Clicked` for each of the three `ToolStripMenuItems` and implement this method as so:

```
private void ContextMenuItemSelection_Clicked(object sender, EventArgs e)
{
    // Obtain the currently clicked ToolStripMenuItem.
    ToolStripMenuItem miClicked =
        miClicked = (ToolStripMenuItem)sender;

    // Figure out which item was clicked using its Name.
    if (miClicked.Name == "hugeToolStripMenuItem")
        currFontSize = TextFontSize.FontSizeHuge;
    if (miClicked.Name == "normalToolStripMenuItem")
        currFontSize = TextFontSize.FontSizeNormal;
    if (miClicked.Name == "tinyToolStripMenuItem")
        currFontSize = TextFontSize.FontSizeTiny;

    // Tell the Form to repaint itself.
    Invalidate();
}
```

Notice that using the "sender" argument, you are able to determine the name of the `ToolStripMenuItem` member variable in order to set the current text size. Once you have done so, the call to `Invalidate()` fires the `Paint` event, which will cause your `Paint` event handler to execute.

The final step is to inform the Form which `ContextMenuStrip` it should display when the right mouse button is clicked in its client area. To do so, simply use the Properties window to set the `ContextMenuStrip` property equal to the name of your context menu item. Once you have done so, you will find the following line within `InitializeComponent()`:

```
this.ContextMenuStrip = this.fontSizeContextStrip;
```

■ **Note** Be aware that *any* control can be assigned a context menu via the `ContextMenuStrip` property. For example, you could create a `Button` object on a dialog box that responds to a particular context menu. In this way, the menu would be displayed only if the mouse button were right-clicked within the bounding rectangle of the button.

If you now run the application, you should be able to change the size of the rendered text message via a right-click of your mouse.

Checking Menu Items

`ToolStripMenuItem` defines a number of members that allow you to check, enable, and hide a given item. Table 19-11 gives a rundown of some (but not all) of the interesting properties.

Table 19-11. *Members of the* `ToolStripMenuItem` *Type*

Member	Meaning in Life
`Checked`	Gets or sets a value indicating whether a check mark appears beside the text of the `ToolStripMenuItem`
`CheckOnClick`	Gets or sets a value indicating whether the `ToolStripMenuItem` should automatically appear checked/unchecked when clicked
`Enabled`	Gets or sets a value indicating whether the `ToolStripMenuItem` is enabled

Let's extend the previous pop-up menu to display a check mark next to the currently selected menu item. Setting a check mark on a given menu item is not at all difficult (just set the `Checked` property to true). However, tracking which menu item should be checked does require some additional logic. One possible approach is to define a distinct `ToolStripMenuItem` member variable that represents the currently checked item:

```
public class MainWindow : Form
{
...
    // Marks the item checked.
    private ToolStripMenuItem currentCheckedItem;
}
```

Recall that the default text size is `TextFontSize.FontSizeNormal`. Given this, the initial item to be checked is the `normalToolStripMenuItem` `ToolStripMenuItem` member variable. Update your Form's constructor as so:

```
public MainWindow()
{
    // Inherited method to center the Form.
    CenterToScreen();
    InitializeComponent();

    // Now check the 'Normal' menu item.
    currentCheckedItem = normalToolStripMenuItem;
    currentCheckedItem.Checked = true;
}
```

Now that you have a way to programmatically identify the currently checked item, the last step is to update the `ContextMenuItemSelection_Clicked()` event handler to uncheck the previous item and check the new current `ToolStripMenuItem` object in response to the user selection:

```
private void ContextMenuItemSelection_Clicked(object sender, EventArgs e)
{
    // Uncheck the currently checked item.
    currentCheckedItem.Checked = false;
...
    if (miClicked.Name == "hugeToolStripMenuItem")
    {
        currFontSize = TextFontSize.FontSizeHuge;
        currentCheckedItem = hugeToolStripMenuItem;
    }
    if (miClicked.Name == "normalToolStripMenuItem")
    {
        currFontSize = TextFontSize.FontSizeNormal;
        currentCheckedItem = normalToolStripMenuItem;
    }
    if (miClicked.Name == "tinyToolStripMenuItem")
    {
        currFontSize = TextFontSize.FontSizeTiny;
        currentCheckedItem = tinyToolStripMenuItem;
    }
    // Check new item.
    currentCheckedItem.Checked = true;
...
}
```

Figure 19-15 shows the completed MenuStripApp project in action.

Figure 19-15. *Checking/unchecking* `ToolStripMenuItem`*s*

Source Code The MenuStripApp application is located under the Chapter 19 subdirectory.

Working with StatusStrips

In addition to a menu system, many Forms also maintain a *status bar* that is typically mounted at the bottom of the Form. A status bar may be divided into any number of "panes" that hold some textual (or graphical) information such as menu help strings, the current time, or other application-specific information.

Although status bars have been supported since the release of the .NET platform (via the `System.Windows.Forms.StatusBar` type), as of .NET 2.0 the simple `StatusBar` has been ousted by the new `StatusStrip` type. Like a status bar, a `StatusStrip` can consist of any number of panes to hold textual/graphical data using a `ToolStripStatusLabel` type. However, status strips have the ability to contain additional tool strip items such as the following:

- `ToolStripProgressBar`: An embedded progress bar.
- `ToolStripDropDownButton`: An embedded button that displays a drop-down list of choices when clicked.
- `ToolStripSplitButton`: This is similar to the `ToolStripDropDownButton`, but the items of the drop-down list are displayed only if the user clicks directly on the drop-down area of the control. The `ToolStripSplitButton` also has normal buttonlike behavior and can thus support the `Click` event.

In this example, you will build a new `MainWindow` that supports a simple menu (File ➤ Exit and Help ➤ About) as well as a `StatusStrip`. The leftmost pane of the status strip will be used to display help string data regarding the currently selected menu subitem (e.g., if the user selects the Exit menu, the pane will display "Exits the app").

The far right pane will display one of two dynamically created strings that will show either the current time or the current date. Finally, the middle pane will be a `ToolStripDropDownButton` type that allows the user to toggle the date/time display (with a happy face icon to boot!). Figure 19-16 shows the application in its completed form.

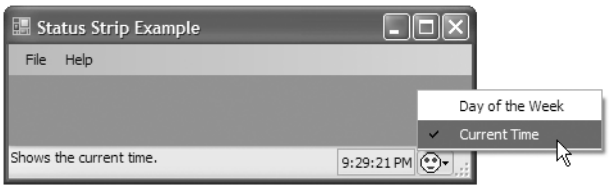

Figure 19-16. *The* `StatusStrip` *application*

Designing the Menu System

To begin, create a new Windows Forms application project named StatusStripApp. Place a `MenuStrip` control onto the Forms designer and build the two menu items (File ➤ Exit and Help ➤ About). Once you have done so, handle the `Click` and `MouseHover` events for each subitem (Exit and About) using the Properties window.

The implementation of the File ➤ Exit `Click` event handler will simply terminate the application, while the Help ➤ About `Click` event handler shows a friendly `MessageBox`.

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{ Application.Exit(); }

private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{ MessageBox.Show("My StatusStripApp!"); }
```

You will update the `MouseHover` event handler to display the correct prompt in the leftmost pane of the `StatusStrip` in just a bit, so leave them empty for the time being.

Designing the StatusStrip

Next, place a `StatusStrip` control onto the designer and rename this control to `mainStatusStrip`. Understand that by default a `StatusStrip` contains no panes whatsoever. To add the three panes, you may take various approaches:

- Author the code by hand without designer support (perhaps using a helper method named `CreateStatusStrip()` that is called in the Form's constructor).
- Add the items via a dialog box activated using the Edit Items link using the `StatusStrip` context-sensitive inline editor (see Figure 19-17).
- Add the items one by one via the new item drop-down editor mounted on the `StatusStrip` (see Figure 19-18).

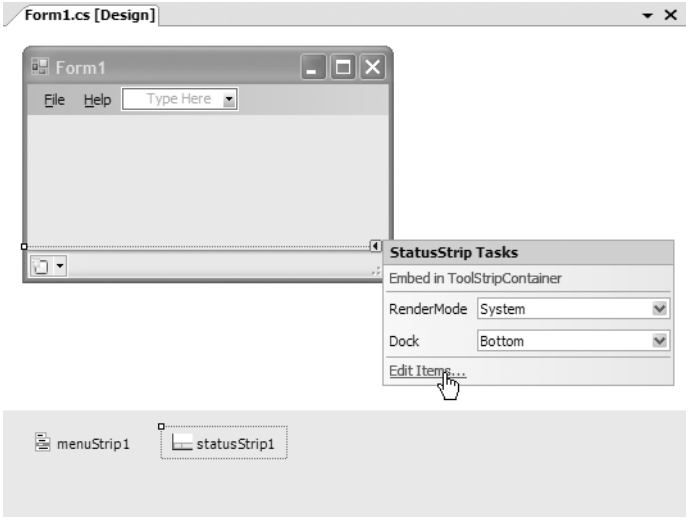

Figure 19-17. *The* `StatusStrip` *context editor*

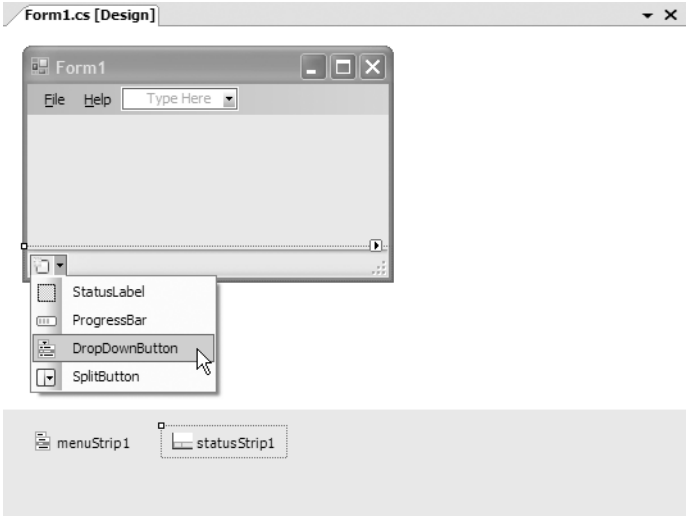

Figure 19-18. *Adding items via the* `StatusStrip` *new item drop-down editor*

For this example, you will leverage the new item drop-down editor. Add two new `ToolStripStatusLabel` types named `toolStripStatusLabelMenuState` and `toolStripStatusLabelClock`, and a `ToolStripDropDownButton` named `toolStripDropDownButtonDateTime`. As you would expect, this will add new member variables in the `*.Designer.cs` file and update `InitializeComponent()` accordingly. Note that the `StatusStrip` maintains an internal collection to hold each of the panes:

```
partial class MainForm
{
    private void InitializeComponent()
    {
...
    //
    // mainStatusStrip
    //
    this.mainStatusStrip.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
        this.toolStripStatusLabelMenuState,
        this.toolStripStatusLabelClock,
        this.toolStripDropDownButtonDateTime});
...
    }

    private System.Windows.Forms.StatusStrip mainStatusStrip;
    private System.Windows.Forms.ToolStripStatusLabel
        toolStripStatusLabelMenuState;
    private System.Windows.Forms.ToolStripStatusLabel
        toolStripStatusLabelClock;
    private System.Windows.Forms.ToolStripDropDownButton
        toolStripDropDownButtonDateTime;
...
}
```

Now, select the `ToolStripDropDownButton` on the designer and add two new menu items named `currentTimeToolStripMenuItem` and `dayoftheWeekToolStripMenuItem` (see Figure 19-19).

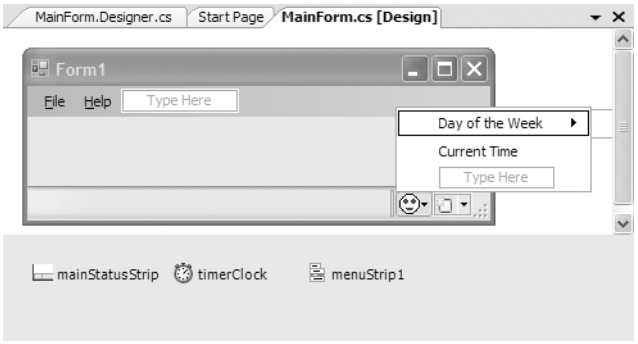

Figure 19-19. *Adding menu items to the* `ToolStripDropDownButton`

To configure your panes to reflect the look and feel shown in Figure 19-19, you will need to set several properties, which you do using the Visual Studio 2005 Properties window. Table 19-12 documents the necessary properties to set and events to handle for each item on your `StatusStrip` (of course, feel free to stylize the panes with additional settings as you see fit).

Table 19-12. `StatusStrip` *Pane Configuration*

Pane Member Variable	Properties to Set	Events to Handle

`toolStripStatusLabelMenuState`	`Spring` = `true`
	`Text` = (empty)
	`TextAlign` = `TopLeft`
		None

`toolStripStatusLabelClock`	`BorderSides` = `All` `Text` = (empty)	None
`toolStripDropDownButtonDateTime`	`Image` = (see text that follows)	None
`dayoftheWeekToolStripMenuItem`	`Text` = "Day of the Week"	`MouseHover` `Click`
`currentTimeToolStripMenuItem`	`Text` = "Current Time"	`MouseHover` `Click`

The `Image` property of the `toolStripDropDownButtonDateTime` member can be set to any image file on your machine (of course, extremely large image files will be quite skewed). For this example, you may wish to use the `happyDude.bmp` file included with this book's downloadable source code (please visit the Downloads section of the Apress website, `http://www.apress.com`).

So at this point, the GUI design is complete! Before you implement the remaining event handlers, you need to get to know the role of the `Timer` component.

Working with the Timer Type

Recall that the second pane should display the current time or current date based on user preference. The first step to take to achieve this design goal is to add a `Timer` member variable to the Form. A `Timer` is a component that calls some method (specified using the `Tick` event) at a given interval (specified by the `Interval` property).

Drag a `Timer` component onto your Forms designer and rename it to `timerDateTimeUpdate`. Using the Properties window, set the `Interval` property to 1,000 (the value in milliseconds) and set the `Enabled` property to true. Finally, handle the `Tick` event. Before implementing the `Tick` event handler, define a new enum type in your project named `DateTimeFormat`. This enum will be used to determine whether the second `ToolStripStatusLabel` should display the current time or the current day of the week:

```
enum DateTimeFormat
{
    ShowClock,
    ShowDay
}
```

With this enum in place, update your `MainWindow` with the following code:

```
public partial class MainWindow : Form
{
    // Which format to display?
    DateTimeFormat dtFormat = DateTimeFormat.ShowClock;
...
    private void timerDateTimeUpdate_Tick(object sender, EventArgs e)
    {
        string panelInfo = "";

        // Create current format.
        if (dtFormat == DateTimeFormat.ShowClock)
            panelInfo = DateTime.Now.ToLongTimeString();
```

```
        else
            panelInfo = DateTime.Now.ToLongDateString();

        // Set text on pane.
        toolStripStatusLabelClock.Text = panelInfo;
    }
}
```

Notice that the `Timer` event handler makes use of the `DateTime` type. Here, you simply find the current system time or date using the `Now` property and use it to set the `Text` property of the `toolStripStatusLabelClock` member variable.

Toggling the Display

At this point, the `Tick` event handler should be displaying the current time within the `toolStripStatusLabelClock` pane, given that the default value of your `DateTimeFormat` member variable as been set to `DateTimeFormat.ShowClock`. To allow the user to toggle between the date and time display, update your `MainWindow` as so (note you are also toggling which of the two menu items in the `ToolStripDropDownButton` should be checked):

```
public partial class MainWindow : Form
{
    // Which format to display?
    DateTimeFormat dtFormat = DateTimeFormat.ShowClock;

    // Marks the item checked.
    private ToolStripMenuItem currentCheckedItem;

    public MainWindow()
    {
        InitializeComponent();

        // These properties can also be set
        // with the Properties window.
        Text = "Status Strip Example";
        CenterToScreen();
        BackColor = Color.CadetBlue;
        currentCheckedItem = currentTimeToolStripMenuItem;
        currentCheckedItem.Checked = true;
    }
...
    private void currentTimeToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // Toggle check mark and set pane format to time.
        currentCheckedItem.Checked = false;
        dtFormat = DateTimeFormat.ShowClock;
        currentCheckedItem = currentTimeToolStripMenuItem;
        currentCheckedItem.Checked = true;
    }
    private void dayoftheWeekToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // Toggle check mark and set pane format to date.
        currentCheckedItem.Checked = false;
        dtFormat = DateTimeFormat.ShowDay;
        currentCheckedItem = dayoftheWeekToolStripMenuItem;
        currentCheckedItem.Checked = true;
    }
}
```

Displaying the Menu Selection Prompts

Finally, you need to configure the first pane to hold menu help strings. As you know, most applications send a small bit of text information to the first pane of a status bar whenever the end user selects a menu item (e.g., "This terminates the application"). Given that you have already handled the `MouseHover` events for each submenu on the `MenuStrip` and `TooStripDropDownButton`, all you need to do is assign a proper value to the `Text` property for the `toolStripStatusLabelMenuState` member variable, for example:

```
private void exitToolStripMenuItem_MouseHover(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Exits the app."; }

private void aboutToolStripMenuItem_MouseHover(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Shows about box."; }

private void dayoftheWeekToolStripMenuItem_MouseHover(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Shows the day of the week."; }

private void currentTimeToolStripMenuItem_MouseHover(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Shows the current time."; }
```

Take your updated project out for a test drive. You should now be able to find these informational help strings in the first pane of your `StatusStrip` as you select each menu item.

Establishing a "Ready" State

The final thing to do for this example is ensure that when the user deselects a menu item, the first text pane is set to a default message (e.g., "Ready"). With the current design, the previously selected menu prompt remains on the leftmost text pane, which is confusing at best. To rectify this issue, handle the `MouseLeave` event for the Exit, About, Day of the Week, and Current Time menu items. *However*, rather than generating a new event handler for each item, have them all call a method named `SetReadyPrompt()`:

```
private void SetReadyPrompt(object sender, EventArgs e)
{  toolStripStatusLabelMenuState.Text = "Ready."; }
```

With this, you should find that the first pane resets to this default message as soon as the mouse cursor leaves any of your four menu items.

■ **Source Code** The StatusBarApp project is included under the Chapter 19 subdirectory.

Working with ToolStrips

The next Form-level GUI item to examine in this chapter is the .NET 2.0 `ToolStrip` type, which overshadows the functionality found within the depreciated .NET 1.*x* `ToolBar` class. As you know, toolbars typically provide an alternate means to activate a given menu item. Thus, if the user clicks a Save button, this has the same effect as selecting File ➤ Save. Much like `MenuStrip` and `StatusStrip`, the `ToolStrip` type can contain numerous toolbar items, some of which you have already encountered in previous examples:

- `ToolStripButton`
- `ToolStripLabel`
- `ToolStripSplitButton`

- ToolStripDropDownButton
- ToolStripSeparator
- ToolStripComboBox
- ToolStripTextBox
- ToolStripProgressBar

Like other Windows Forms controls, the ToolStrip supports an inline editor that allows you to quickly add standard button types (File, Exit, Help, Copy, Paste, etc.) to a ToolStrip, change the docking position, and embed the ToolStrip in a ToolStripContainer (more details in just a bit). Figure 19-20 illustrates the designer support for ToolStrips.

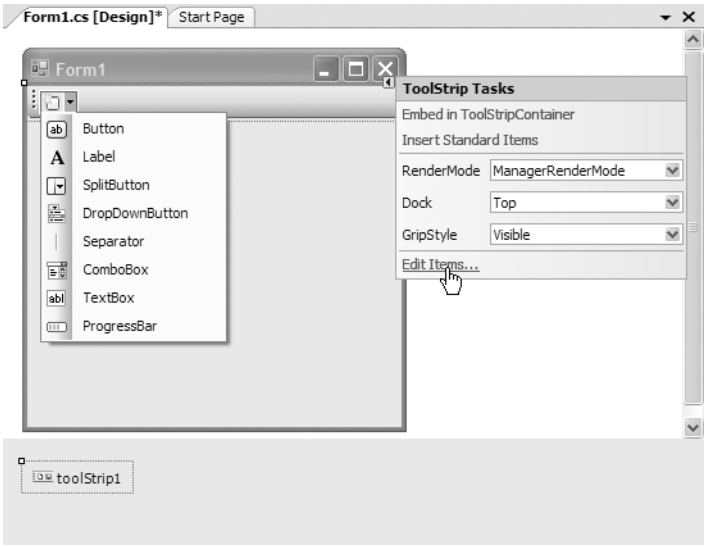

Figure 19-20. *Designing* ToolStrips

Like MenuStrips and StatusStrips, individual ToolStrip controls are added to the ToolStrip's internal collection via the Items property. If you click the Insert Standard Items link on the inline ToolStrip editor, your InitializeComponent() method is updated to insert an array of ToolStripItem-derived types that represent each item:

```
private void InitializeComponent()
{
...
    // Autogenerated code to prep a ToolStrip.
    this.toolStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
        this.newToolStripButton, this.openToolStripButton,
        this.saveToolStripButton, this.printToolStripButton,
        this.toolStripSeparator, this.cutToolStripButton,
        this.copyToolStripButton, this.pasteToolStripButton,
        this.toolStripSeparator1, this.helpToolStripButton});
...
}
```

To illustrate working with `ToolStrips`, the following Windows Forms application creates a `ToolStrip` containing two `ToolStripButton` types (named `toolStripButtonGrowFont` and `toolStripButtonShrinkFont`), a `ToolBarSeparator`, and a `ToolBarTextBox` (named `toolStripTextBoxMessage`).

The end user is able to enter a message to be rendered on the Form via the `ToolBarTextBox`, and the two `ToolBarButton` types will be used to increase or decrease the font size. Figure 19-21 shows the end result of the project you will construct.

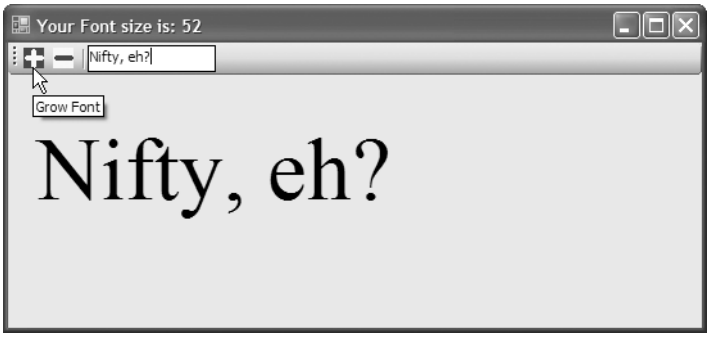

Figure 19-21. *ToolStripApp in action*

By now I'd guess you have a handle on working with the Visual Studio 2005 Forms designer, so I won't belabor the point of building the `ToolStrip`. Do note, however, that each `ToolStripButton` has a custom (albeit poorly drawn by yours truly) icon that was created using the Visual Studio 2005 image editor. If you wish to create image files for your project, simply select the Project ➤ Add New Item menu option, and from the resulting dialog box add a new icon file (see Figure 19-22).

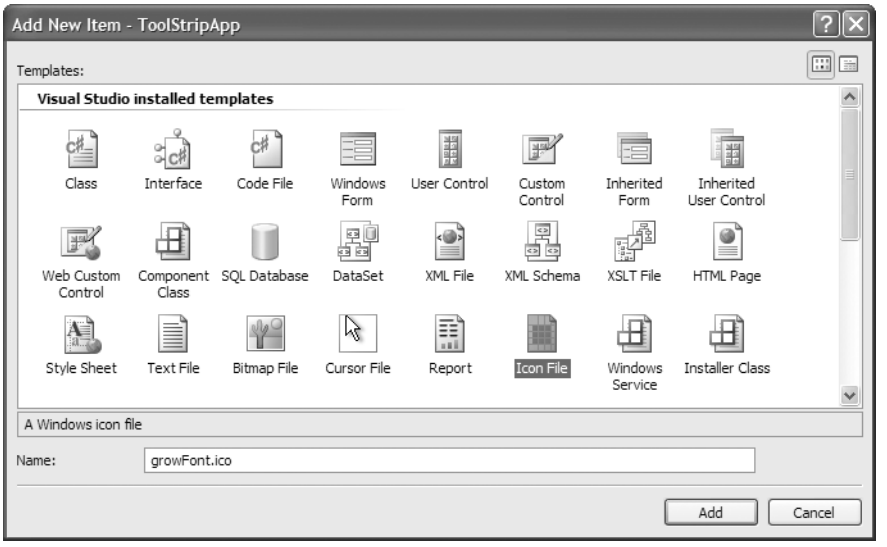

Figure 19-22. *Inserting new image files*

Once you have done so, you are able to edit your images using the Colors tab on the Toolbox and the Image Editor toolbox. In any case, once you have designed your icons, you are able to associate them with the `ToolStripButton` types via the `Image` property in the Properties window. Once you are happy with the `ToolStrip`'s look and feel, handle the `Click` event for each `ToolStripButton`.

Here is the relevant code in the `InitializeComponent()` method for the first `ToolStripButton` type (the second `ToolStripButton` will look just about the same):

```
private void InitializeComponent()
{
...
    // toolStripButtonGrowFont
    //
    this.toolStripButtonGrowFont.DisplayStyle =
        System.Windows.Forms.ToolStripItemDisplayStyle.Image;
    this.toolStripButtonGrowFont.Image =
    ((System.Drawing.Image)
    (resources.GetObject("toolStripButtonGrowFont.Image")));
    this.toolStripButtonGrowFont.ImageTransparentColor =
        System.Drawing.Color.Magenta;
    this.toolStripButtonGrowFont.Name = "toolStripButtonGrowFont";
    this.toolStripButtonGrowFont.Text = "toolStripButton2";
    this.toolStripButtonGrowFont.ToolTipText = "Grow Font";
    this.toolStripButtonGrowFont.Click += new
        System.EventHandler(this.toolStripButtonGrowFont_Click);
...
}
```

■ **Note** Notice that the value assigned to the `Image` of a `ToolStripButton` is obtained using a method named `GetObject()`. As explained in the next chapter, this method is used to extract embedded resources used by your assembly.

The remaining code is extremely straightforward. In the following updated `MainWindow`, notice that the current font size is constrained between 12 and 70:

```
public partial class MainWindow : Form
{
    // The current, max and min font sizes.
    int currFontSize = 12;
    const int MinFontSize = 12;
    const int MaxFontSize = 70;

    public MainWindow()
    {
        InitializeComponent();
        CenterToScreen();
        Text = string.Format("Your Font size is: {0}", currFontSize);
    }

    private void toolStripButtonShrinkFont_Click(object sender, EventArgs e)
    {
        // Reduce font size by 5 and refresh display.
        currFontSize -= 5;
        if (currFontSize <= MinFontSize)
            currFontSize = MinFontSize;
```

```
        Text = string.Format("Your Font size is: {0}", currFontSize);
        Invalidate();
    }

    private void toolStripButtonGrowFont_Click(object sender, EventArgs e)
    {
        // Increase font size by 5 and refresh display.
        currFontSize += 5;
        if (currFontSize >= MaxFontSize)
            currFontSize = MaxFontSize;
        Text = string.Format("Your Font size is: {0}", currFontSize);
        Invalidate();
    }

    private void MainWindow_Paint(object sender, PaintEventArgs e)
    {
        // Paint the user-defined message.
        Graphics g = e.Graphics;
        g.DrawString(toolStripTextBoxMessage.Text,
            new Font("Times New Roman", currFontSize),
            Brushes.Black, 10, 60);
    }
}
```

As a final enhancement, if you wish to ensure that the user message is updated as soon as the `ToolStripTextBox` loses focus, you can handle the `LostFocus` event and `Invalidate()` your Form within the generated event handler:

```
public partial class MainWindow : Form
{
...
    public MainWindow()
    {
...
        this.toolStripTextBoxMessage.LostFocus
            += new EventHandler(toolStripTextBoxMessage_LostFocus);
    }
    void toolStripTextBoxMessage_LostFocus(object sender, EventArgs e)
    {
        Invalidate();
    }
...
}
```

Working with ToolStripContainers

`ToolStrips`, if required, can be configured to be "dockable" against any or all sides of the Form that contains it. To illustrate how you can accomplish this, right-click your current `ToolStrip` using the designer and select the Embed in ToolStripContainer menu option. Once you have done so, you will find that the `ToolStrip` has been contained within a `ToolStripContainer`. For this example, select the Dock Fill in Form option (see Figure 19-23).

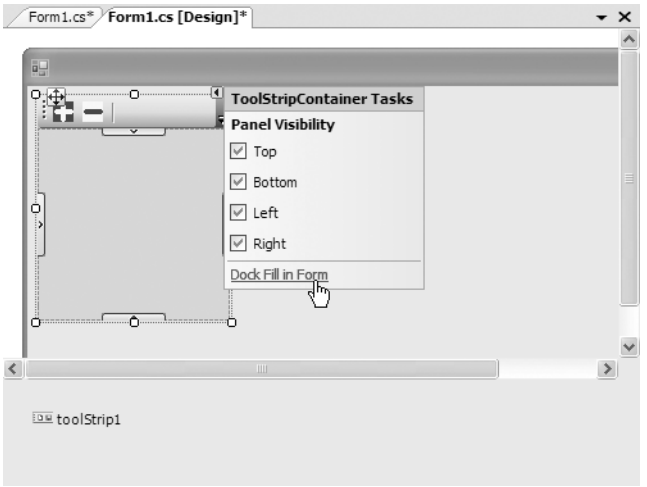

Figure 19-23. *Docking the* `ToolStripContainer` *within the entire Form*

If you run your current update, you will find that the `ToolStrip` can be moved and docked to each side of the container. However, your custom message has now vanished. The reason for this is that `ToolStripContainer`s are actually *child controls* of the Form. Therefore, the graphical render is still taking place, but the output is being hidden by the container that now sits on top of the Form's client area.

To fix this problem, you will need to handle the `Paint` event on the `ToolStripContainer` rather than on the Form. First, locate the Form's `Paint` event within the Properties window and right-click the current event handler. From the context menu, select Reset (see Figure 19-24).

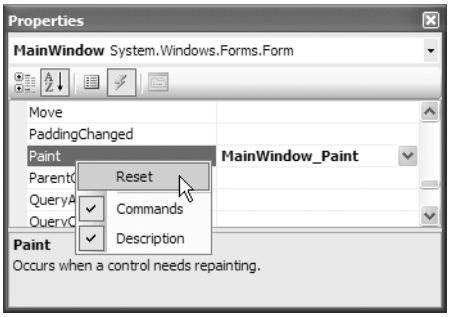

Figure 19-24. *Resetting an event*

This will remove the event handling logic in `InitializeComponent()`, but it will leave the event handler in place (just to ensure you don't lose code you would like to maintain).

Now, handle the `Paint` event for the `ToolStripContainer` and move the rendering code from the existing Form's `Paint` event handler into the container's `Paint` event handler. Once you have done so, you can delete the (now empty) `MainWindow_Paint()` method. Finally, you will need to replace each occurrence of the call to the Form's `Invalidate()` method to the container's `Invalidate()` method. Here are the relevant code updates:

```
public partial class MainWindow : Form
{
...
    void toolStripTextBoxMessage_LostFocus(object sender, EventArgs e)
    {
        toolStripContainer1.Invalidate(true);
    }

    private void toolStripButtonShrinkFont_Click(object sender, EventArgs e)
    {
...
        toolStripContainer1.Invalidate(true);
    }
    private void toolStripButtonGrowFont_Click(object sender, EventArgs e)
    {
...
        toolStripContainer1.Invalidate(true);
    }
    // We are now painting on the container, not the form!
    private void ContentPanel_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawString(toolStripTextBoxMessage.Text,
            new Font("Times New Roman", currFontSize),
            Brushes.Black, 10, 60);
    }
}
```

Of course, the `ToolStripContainer` can be configured in various ways to tweak how it operates. I leave it to you to check out the .NET Framework 2.0 SDK documentation for complete details. Figure 19-25 shows the completed project.

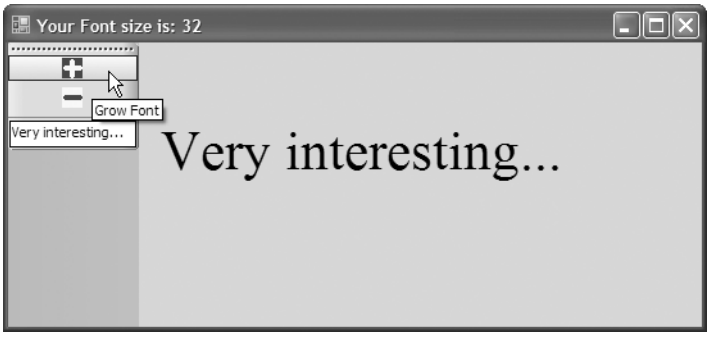

Figure 19-25. *ToolStripApp, now with a dockable* `ToolStrip`

Source Code The ToolStripApp project is included under the Chapter 19 subdirectory.

Building an MDI Application

To wrap up our initial look at Windows Forms, I'll close this chapter by discussing how to configure a Form to function as a parent to any number of child windows (i.e., an MDI container). MDI applications allow users to have multiple child windows open at the same time within the same topmost window. In the world of MDIs, each window represents a given "document" of the application. For example, Visual Studio 2005 is an MDI application in that you are able to have multiple documents open from within an instance of the application.

When you are building MDI applications using Windows Forms, your first task is to (of course) create a brand-new Windows application. The initial Form of the application typically hosts a menu system that allows you to create new documents (such as File ➤ New) as well as arrange existing open windows (cascade, vertical tile, and horizontal tile).

Creating the child windows is interesting, as you typically define a prototypical Form that functions as a basis for each child window. Given that Forms are class types, any private data defined in the child Form will be unique to a particular instance. For example, if you were to create an MDI word processing application, you might create a child Form that maintains a `StringBuilder` to represent the text. If a user created five new child windows, each Form would maintain its own `StringBuilder` instance, which could be individually manipulated.

Additionally, MDI applications allow you to "merge menus." As mentioned previously, parent windows typically have a menu system that allows the user to spawn and organize additional child windows. However, what if the child window also maintains a menuing system? If the user maximizes a particular child window, you need to merge the child's menu system within the parent Form to allow the user to activate items from each menu system. The Windows Forms namespace defines a number of properties, methods, and events that allow you to programmatically merge menu systems. In addition, there is a "default merge" system, which works in a good number of cases.

Building the Parent Form

To illustrate the basics of building an MDI application, begin by creating a brand-new Windows application named SimpleMdiApp. Almost all of the MDI infrastructure can be assigned to your initial Form using various design-time tools. To begin, locate the `IsMdiContainer` property in the Properties window and set it to true. If you look at the design-time Form, you'll see that the client area has been modified to visually represent a container of child windows.

Next, place a new `MenuStrip` control on your main Form. This menu specifies three topmost items named File, Window, and Arrange Windows. The File menu contains two subitems named New and Exit. The Window menu does not contain any subitems, because you will programmatically add new items as the user creates additional child windows. Finally, the Arrange Window menu defines three subitems named Cascade, Vertical, and Horizontal.

Once you have created the menu UI, handle the `Click` event for the Exit, New, Cascade, Vertical, and Horizontal menu items (remember, the Window menu does not have any subitems just yet). You'll implement the File ➤ New handler in the next section, but for now here is the code behind the remaining menu selections:

```
// Handle File | Exit event and arrange all child windows.
private void cascadeToolStripMenuItem_Click(object sender, EventArgs e)
{
    LayoutMdi(MdiLayout.Cascade);
}
private void verticalToolStripMenuItem_Click(object sender, EventArgs e)
{
    LayoutMdi(MdiLayout.TileVertical);
}
private void horizontalToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{
    LayoutMdi(MdiLayout.TileHorizontal);
}
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

The main point of interest here is the use of the `LayoutMdi()` method and the corresponding `MdiLayout` enumeration. The code behind each menu select handler should be quite clear. When the user selects a given arrangement, you tell the parent Form to automatically reposition any and all child windows.

Before you move on to the construction of the child Form, you need to set one additional property of the `MenuStrip`. The `MdiWindowListItem` property is used to establish which topmost menu item should be used to automatically list the name of each child window as a possible menu selection. Set this property to the `windowToolStripMenuItem` member variable. By default, this list is the value of the child's `Text` property followed by a numerical suffix (i.e., `Form1`, `Form2`, `Form3`, etc.).

Building the Child Form

Now that you have the shell of an MDI container, you need to create an additional Form that functions as the prototype for a given child window. Begin by inserting a new `Form` type into your current project (using Project ➤ Add Windows Form) named `ChildPrototypeForm` and handle the `Click` event for this Form. In the generated event handler, randomly set the background color of the client area. In addition, print out the "stringified" value of the new `Color` object into the child's caption bar. The following logic should do the trick:

```
private void ChildPrototypeForm_Click(object sender, EventArgs e)
{
    // Get three random numbers
    int r, g, b;
    Random ran = new Random();
    r = ran.Next(0, 255);
    g = ran.Next(0, 255);
    b = ran.Next(0, 255);

    // Now create a color for the background.
    Color currColor = Color.FromArgb(r, g, b);
    this.BackColor = currColor;
    this.Text = currColor.ToString();
}
```

Spawning Child Windows

Your final order of business is to flesh out the details behind the parent Form's File ➤ New event handler. Now that you have defined a child Form, the logic is simple: create and show a new instance of the `ChildPrototypeForm` type. As well, you need to set the value of the child Form's `MdiParent` property to point to the containing Form (in this case, your main window). Here is the update:

```
private void newToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Make a new child window.
    ChildPrototypeForm newChild = new ChildPrototypeForm();

    // Set the Parent Form of the Child window.
    newChild.MdiParent = this;
```

```
    // Display the new form.
    newChild.Show();
}
```

■ **Note** A child Form may access the `MdiParent` property directly whenever it needs to manipulate (or communicate with) its parent window.

To take this application out for a test drive, begin by creating a set of new child windows and click each one to establish a unique background color. If you examine the subitems under the Windows menu, you should see each child Form present and accounted for. As well, if you access the Arrange Window menu items, you can instruct the parent Form to vertically tile, horizontally tile, or cascade the child Forms. Figure 19-26 shows the completed application.

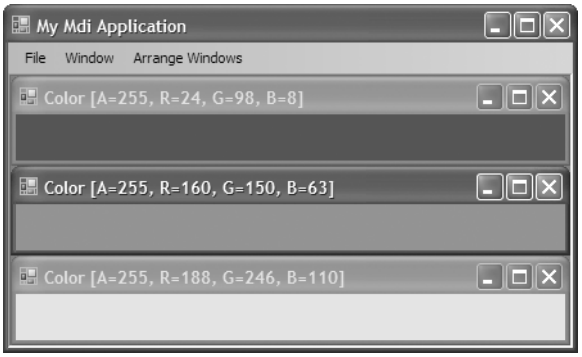

Figure 19-26. *An MDI application*

■ **Source Code** The SimpleMdiApp project can be found under the Chapter 19 subdirectory.

Summary

This chapter introduced the fine art of building a UI with the types contained in the `System.Windows.Forms` namespace. You began by building a number of applications by hand, and you learned along the way that at a minimum, a GUI application needs a class that derives from `Form` and a `Main()` method that invokes `Application.Run()`.

During the course of this chapter, you learned how to build topmost menus (and pop-up menus) and how to respond to a number of menu events. You also came to understand how to further enhance your `Form` types using toolbars and status bars. As you have seen, .NET 2.0 prefers to build such UI elements using `MenuStrips`, `ToolStrips`, and `StatusStrips` rather than the older .NET 1.*x* `MainMenu`, `ToolBar`, and `StatusBar` types (although these deprecated types are still supported). Finally, this chapter wrapped up by illustrating how to construct MDI applications using Windows Forms.

CHAPTER 20

Rendering Graphical Data with GDI+

The previous chapter introduced you to the process of building a GUI-based desktop application using `System.Windows.Forms`. The point of this chapter is to examine the details of rendering graphics (including stylized text and image data) onto a Form's surface area. We'll begin by taking a high-level look at the numerous drawing-related namespaces, and we'll examine the role of the `Paint` event,and the almighty Graphics object.

The remainder of this chapter covers how to manipulate colors, fonts, geometric shapes, and graphical images. This chapter also explores a number of rendering-centric programming techniques, such as nonrectangular hit testing, drag-and-drop logic, and the .NET resource format. While *technically* not part of GDI+ proper, resources often involve the manipulation of graphical data (which, in my opinion, is "GDI+-enough" to be presented here).

Note If you are a web programmer by trade, you may think that GDI+ is of no use to you. However, GDI+ is not limited to traditional desktop applications and is extremely relevant for web applications.

A Survey of the GDI+ Namespaces

The .NET platform provides a number of namespaces devoted to two-dimensional graphical rendering. In addition to the basic functionality you would expect to find in a graphics toolkit (colors, fonts, pens, brushes, etc.), you also find types that enable geometric transformations, antialiasing, palette blending, and document printing support. Collectively speaking, these namespaces make up the .NET facility we call *GDI+*, which is a managed alternative to the Win32 Graphical Device Interface (GDI) API. Table 20-1 gives a high-level view of the core GDI+ namespaces.

Table 20-1. *Core GDI+ Namespaces*

Namespace	Meaning in Life
`System.Drawing`	This is the core GDI+ namespace that defines numerous types for basic rendering (fonts, pens, basic brushes, etc.) as well as the almighty `Graphics` type.
`System.Drawing.Drawing2D`	This namespace provides types used for more advanced two-dimensional/vector graphics functionality (e.g., gradient brushes, pen caps, geometric transforms, etc.).
`System.Drawing.Imaging`	This namespace defines types that allow you to manipulate graphical images (e.g., change the palette, extract image metadata, manipulate metafiles, etc.).
`System.Drawing.Printing`	This namespace defines types that allow you to render images to the printed page, interact with the printer itself, and format the overall appearance of a given print job.
`System.Drawing.Text`	This namespace allows you to manipulate collections of fonts.

■ **Note** All of the GDI+ namespaces are defined within the `System.Drawing.dll` assembly. While many Visual Studio 2005 project types automatically set a reference to this code library, you can manually reference `System.Drawing.dll` using the Add References dialog box if necessary.

An Overview of the System.Drawing Namespace

The vast majority of the types you'll use when programming GDI+ applications are found within the `System.Drawing` namespace. As you would expect, there are classes that represent images, brushes, pens, and fonts. Furthermore, `System.Drawing` defines a number of related utility types such as `Color`, `Point`, and `Rectangle`. Table 20-2 lists some (but not all) of the core types.

Table 20-2. *Core Types of the* `System.Drawing` *Namespace*

Type	Meaning in Life
`Bitmap`	This type encapsulates image data (`*.bmp` or otherwise).

`Brush`
`Brushes`
`SolidBrush`
`SystemBrushes`
`TextureBrush`
	`Brush` objects are used to fill the interiors of graphical shapes such as rectangles, ellipses, and polygons.

`BufferedGraphics`	This new .NET 2.0 type provides a graphics buffer for double buffering, which is used to reduce or eliminate flicker caused by redrawing a display surface.
`Color` `SystemColors`	The `Color` and `SystemColors` types define a number of static read-only properties used to obtain specific colors for the construction of various pens/brushes.
`Font` `FontFamily`	The `Font` type encapsulates the characteristics of a given font (i.e., type name, bold, italic, point size, etc.). `FontFamily` provides an abstraction for a group of fonts having a similar design but with certain variations in style.
`Graphics`	This core class represents a valid drawing surface, as well as a number of methods to render text, images, and geometric patterns.

Type	Meaning in Life
`Icon` `SystemIcons`	These classes represent custom icons, as well as the set of standard system-supplied icons.
`Image` `ImageAnimator`	`Image` is an abstract base class that provides functionality for the `Bitmap`, `Icon`, and `Cursor` types. `ImageAnimator` provides a way to iterate over a number of `Image`-derived types at some specified interval.

`Pen`
`Pens`
`SystemPens`
	`Pens` are objects used to draw lines and curves. The `Pens` type defines a number of static properties that return a new `Pen` of a given color.

`Point` `PointF`	These structures represent an (*x*, *y*) coordinate mapping to an underlying integer or float, respectively.
`Rectangle` `RectangleF`	These structures represent a rectangular dimension (again mapping to an underlying integer or float).
`Size` `SizeF`	These structures represent a given height/width (again mapping to an underlying integer or float).
`StringFormat`	This type is used to encapsulate various features of textual layout (i.e., alignment, line spacing, etc.).
`Region`	This type describes the interior of a geometric image composed of rectangles and paths.

The System.Drawing Utility Types

Many of the drawing methods defined by the `System.Drawing.Graphics` object require you to specify the position or area in which you wish to render a given item. For example, the `DrawString()` method requires you to specify the location to render the text string on the `Control`-derived type. Given that `DrawString()` has been overloaded a number of times, this positional parameter may be specified using an (*x*, *y*) coordinate or the dimensions of a "box" to draw within. Other GDI+ type methods may require you to specify the width and height of a given item, or the internal bounds of a geometric image.

To specify such information, the `System.Drawing` namespace defines the `Point`, `Rectangle`, `Region`, and `Size` types. Obviously, a `Point` represents an (*x*, *y*) coordinate. `Rectangle` types capture a pair of points representing the upper-left and bottom-right bounds of a rectangular region. `Size` types are similar to `Rectangle`s, but this structure represent a particular dimension using a given length and width. Finally, `Region`s provide a way to represent and qualify nonrectangular surfaces.

The member variables used by the `Point`, `Rectangle`, and `Size` types are internally represented as an integer data type. If you need a finer level of granularity, you are free to make use of the corresponding `PointF`, `RectangleF`, and `SizeF` types, which (as you might guess) map to an underlying float. Regardless of the underlying data representation, each type has an identical set of members, including a number of overloaded operators.

The Point(F) Type

The first utility type you should be aware of is `System.Drawing.Point(F)`. Unlike the illustrative `Point` types created in previous chapters, the GDI+ `Point(F)` type supports a number of helpful members, including

- `+`, `-`, `==`, `!=`: The `Point` type overloads various C# operators.
- `X`, `Y`: These members provide access to the underlying (*x*, *y*) values of the `Point`.
- `IsEmpty`: This member returns true if *x* and *y* are both set to 0.

To illustrate working with the GDI+ utility types, here is a console application (named UtilTypes) that makes use of the `System.Drawing.Point` type (be sure to set a reference to `System.Drawing.dll`).

```
using System;
using System.Drawing;

namespace UtilTypes
{
     public class Program
     {
          static void Main(string[] args)
          {
               // Create and offset a point.
               Point pt = new Point(100, 72);
               Console.WriteLine(pt);
               pt.Offset(20, 20);
               Console.WriteLine(pt);

               // Overloaded Point operators.
               Point pt2 = pt;
               if(pt == pt2)
                    WriteLine("Points are the same");
               else
                    WriteLine("Different points");

               // Change pt2's X value.
               pt2.X = 4000;

               // Now show each X value:
               Console.WriteLine("First point: {0} ", pt);
               Console.WriteLine("Second point: {0} ", pt2);
               Console.ReadLine();
          }
     }
}
```

The Rectangle(F) Type

`Rectangles`, like `Points`, are useful in many applications (GUI-based or otherwise). One of the more useful methods of the `Rectangle` type is `Contains()`. This method allows you to determine if a given `Point` or `Rectangle` is within the current bounds of another object. Later in this chapter, you'll see how to make use of this method to perform hit testing of GDI+ images. Until then, here is a simple example:

```
static void Main(string[] args)
{
     ...
     // Point is initially outside of rectangle's bounds.
     Rectangle r1 = new Rectangle(0, 0, 100, 100);
     Point pt3 = new Point(101, 101);
     if(r1.Contains(pt3))
          Console.WriteLine("Point is within the rect!");
     else
          Console.WriteLine("Point is not within the rect!");

     // Now place point in rectangle's area.
     pt3.X = 50;
     pt3.Y = 30;
```

```
    if(r1.Contains(pt3))
        Console.WriteLine("Point is within the rect!");
    else
        Console.WriteLine("Point is not within the rect!");
    Console.ReadLine();
}
```

The Region Class

The `Region` type represents the interior of a geometric shape. Given this last statement, it should make sense that the constructors of the `Region` class require you to send an instance of some existing geometric pattern. For example, assume you have created a 100×100 pixel rectangle. If you wish to gain access to the rectangle's interior region, you could write the following:

```
// Get the interior of this rectangle.
Rectangle r = new Rectangle(0, 0, 100, 100);
Region rgn = new Region(r);
```

Once you have the interior dimensions of a given shape, you may manipulate it using various members such as the following:

- `Complement()`: Updates this `Region` to the portion of the specified graphics object that does not intersect with this `Region`
- `Exclude()`: Updates this `Region` to the portion of its interior that does not intersect with the specified graphics object
- `GetBounds()`: Returns a `Rectangle(F)` that represents a rectangular region that bounds this `Region`
- `Intersect()`: Updates this `Region` to the intersection of itself with the specified graphics object
- `Transform()`: Transforms a `Region` by the specified `Matrix` object
- `Union()`: Updates this `Region` to the union of itself and the specified graphics object
- `Translate()`: Offsets the coordinates of this `Region` by a specified amount

I'm sure you get the general idea behind these coordinate primitives; please consult the .NET Framework 2.0 SDK documentation if you require further details.

■ **Note** The `Size` and `SizeF` types require little comment. These types each define `Height` and `Width` properties and a handful of overloaded operators.

■ **Source Code** The UtilTypes project is included under the Chapter 20 subdirectory.

Understanding the Graphics Class

The `System.Drawing.Graphics` class is the gateway to GDI+ rendering functionality. This class not only represents the surface you wish to draw upon (such as a Form's surface, a control's surface, or region of memory), but also defines dozens of members that allow you to render text, images (icons, bitmaps, etc.), and numerous geometric patterns. Table 20-3 gives a partial list of members.

Table 20-3. *Members of the* `Graphics` *Class*

Methods	Meaning in Life

`FromHdc()`
`FromHwnd()`
`FromImage()`
	These static methods provide a way to obtain a valid `Graphics` object from a given image (e.g., icon, bitmap, etc.) or GUI widget.

`Clear()`	Fills a `Graphics` object with a specified color, erasing the current drawing surface in the process.

`DrawArc()`
`DrawBezier()`
`DrawBeziers()`
`DrawCurve()`
`DrawEllipse()`
`DrawIcon()`
`DrawLine()`
`DrawLines()`
`DrawPie()`
`DrawPath()`
`DrawRectangle()`
`DrawRectangles()`
`DrawString()`
	These methods are used to render a given image or geometric pattern. As you will see, `DrawXXX()` methods require the use of GDI+ `Pen` objects.

`FillEllipse()`
`FillPath()`
`FillPie()`
`FillPolygon()`
`FillRectangle()`
	These methods are used to fill the interior of a given geometric shape. As you will see, `FillXXX()` methods require the use of GDI+ `Brush` objects.

As well as providing a number of rendering methods, the `Graphics` class defines additional members that allow you to configure the "state" of the `Graphics` object. By assigning values to the properties shown in Table 20-4, you are able to alter the current rendering operation.

Table 20-4. *Stateful Properties of the* `Graphics` *Class*

Properties	Meaning in Life

`Clip`
`ClipBounds`
`VisibleClipBounds`
`IsClipEmpty`
`IsVisibleClipEmpty`
	These properties allow you to set the clipping options used with the current `Graphics` object.

`Transform`	This property allows you to transform "world coordinates" (more details on this later).

`PageUnit`
`PageScale`
`DpiX`
`DpiY`
	These properties allow you to configure the point of origin for your rendering operations, as well as the unit of measurement.

`SmoothingMode`
`PixelOffsetMode`
`TextRenderingHint`
	These properties allow you to configure the smoothness of geometric objects and text.

`CompositingMode` `CompositingQuality`	The `CompositingMode` property determines whether drawing overwrites the background or is blended with the background.
`InterpolationMode`	This property specifies how data is interpolated between endpoints.

■ **Note** As of .NET 2.0, the `System.Drawing` namespace provides a `BufferedGraphics` type that allows you to render graphics using a double-buffering system to minimize or eliminate the flickering that can occur during a rendering operation. Consult the .NET Framework 2.0 SDK documentation for full details.

Now, despite what you may be thinking, the `Graphics` class is not directly creatable via the `new` keyword, as there are no publicly defined constructors. How, then, do you obtain a valid `Graphics` object? Glad you asked.

Understanding Paint Sessions

The most common way to obtain a `Graphics` object is to interact with the `Paint` event. Recall from the previous chapter that the `Control` class defines a virtual method named `OnPaint()`. When you want a Form to render graphical data to its surface, you may override this method and extract a `Graphics` object from the incoming `PaintEventArgs` parameter. To illustrate, create a new Windows Forms application named BasicPaintForm, and update the `Form`-derived class as so:

```
public partial class MainForm : Form
{
     public MainForm()
     {
          InitializeComponent();
          CenterToScreen();
          this.Text = "Basic Paint Form";
     }

     protected override void OnPaint(PaintEventArgs e)
     {
          // If overriding OnPaint(), be sure to call base class implementation.
          base.OnPaint(e);

          // Obtain a Graphics object from the incoming
          // PaintEventArgs.
          Graphics g = e.Graphics;

          // Render a textual message in a given font and color.
          g.DrawString("Hello GDI+", new Font("Times New Roman", 20),
                               Brushes.Green, 0, 0);
     }
}
```

While overriding `OnPaint()` is permissible, it is more common to handle the `Paint` event using the associated `PaintEventHandler` delegate (in fact, this is the default behavior taken by Visual Studio 2005 when handling events with the Properties window). This delegate can point to any method taking a `System.Object` as the first parameter and a `PaintEventArgs` as the second. Assuming you have handled the `Paint` event (via the Visual Studio 2005 designers or manually in code), you are once again able to extract a `Graphics` object from the incoming `PaintEventArgs`. Here is the update:

```
public partial class MainForm : Form
{
     public MainForm()
     {
          InitializeComponent();
          CenterToScreen();
```

```
        this.Text = "Basic Paint Form";

        // Visual Studio 2005 places this
        // code within InitializeComponent().
        this.Paint += new PaintEventHandler(MainForm_Paint);
    }

    private void MainForm_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawString("Hello GDI+", new Font("Times New Roman", 20),
            Brushes.Green, 0, 0);
    }
}
```

Regardless of how you respond to the `Paint` event, be aware that whenever a window becomes "dirty," the `Paint` event will fire. As you may be aware, a window is considered "dirty" whenever it is resized, uncovered by another window (partially or completely), or minimized and then restored. In all these cases, the .NET platform ensures that when your Form needs to be redrawn, the `Paint` event handler (or overridden `OnPaint()` method) is called automatically.

Invalidating the Form's Client Area

During the flow of a GDI+ application, you may need to explicitly fire the `Paint` event, rather than waiting for the window to become "naturally dirty." For example, you may be building a program that allows the user to select from a number of bitmap images using a custom dialog box. Once the dialog box is dismissed, you need to draw the newly selected image onto the Form's client area. Obviously, if you waited for the window to become "naturally dirty," the user would not see the change take place until the window was resized or uncovered by another window. To force a window to repaint itself programmatically, simply call the inherited `Invalidate()` method:

```
public partial class MainForm: Form
{
...
    private void MainForm_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        // Render a bitmap here.
    }

    private void GetNewBitmap()
    {
        // Show dialog box and get new image.
        // Repaint the entire client area.
        Invalidate();
    }
}
```

The `Invalidate()` method has been overloaded a number of times to allow you to specify a specific rectangular region to repaint, rather than repainting the entire client area (which is the default). If you wish to only update the extreme upper-left rectangle of the client area, you could write the following:

```
// Repaint a given rectangular area of the Form.
private void UpdateUpperArea()
{
```

```
    Rectangle myRect = new Rectangle(0, 0, 75, 150);
    Invalidate(myRect);
}
```

Obtaining a Graphics Object Outside of a Paint Event Handler

In some rare cases, you may need to access a `Graphics` object *outside* the scope of a `Paint` event handler. For example, assume you wish to draw a small circle at the (*x*, *y*) position where the mouse has been clicked. To obtain a valid `Graphics` object from within the scope of a `MouseDown` event handler, one approach is to call the static `Graphics.FromHwnd()` method. Based on your background in Win32 development, you may know that an `HWND` is a data structure that represents a given Win32 window. Under the .NET platform, the inherited `Handle` property extracts the underlying `HWND`, which can be used as a parameter to `Graphics.FromHwnd()`:

```
private void MainForm_MouseDown(object sender, MouseEventArgs e)
{
    // Grab a Graphics object via Hwnd.
    Graphics g = Graphics.FromHwnd(this.Handle);

    // Now draw a 10*10 circle at mouse click.
    g.FillEllipse(Brushes.Firebrick, e.X, e.Y, 10, 10);

    // Dispose of all Graphics objects you create directly.
    g.Dispose();
}
```

While this logic renders a circle outside an `OnPaint()` event handler, it is very important to understand that when the form is invalidated (and thus redrawn), each of the circles is erased! This should make sense, given that this rendering happens only within the context of a `MouseDown` event. A far better approach is to have the `MouseDown` event handler create a new `Point` type, which is then added to an internal collection (such as a `List<T>`), followed by a call to `Invalidate()`. At this point, the `Paint` event handler can simply iterate over the collection and draw each `Point`:

```
public partial class MainForm : Form
{
    // Used to hold all the points.
    private List<Point> myPts = new List<Point>();

    public MainForm()
    {
...
        this.MouseDown += new MouseEventHandler(MainForm_MouseDown);
    }

    private void MainForm_MouseDown(object sender, MouseEventArgs e)
    {
        // Add to points collection.
        myPts.Add(new Point(e.X, e.Y));
        Invalidate();
    }

    private void MainForm_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawString("Hello GDI+", new Font("Times New Roman", 20),
                     new SolidBrush(Color.Black), 0, 0);
      foreach(Point p in myPts)
```

```
                g.FillEllipse(Brushes.Firebrick, p.X, p.Y, 10, 10);
        }
}
```

Using this approach, the rendered circles are always present and accounted for, as the graphical rendering has been handled within the `Paint` event. Figure 20-1 shows a test run of this initial GDI+ application.

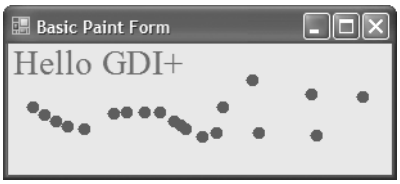

Figure 20-1. *A simple painting application*

■ **Source Code** The BasicPaintForm project is included under the Chapter 20 subdirectory.

Regarding the Disposal of a Graphics Object

If you were reading closely over the last several pages, you may have noticed that *some* of the sample code directly called the `Dispose()` method of the `Graphics` object, while other sample code did not. Given that a `Graphics` type is manipulating various underlying unmanaged resources, it should make sense that it would be advantageous to release said resources via `Dispose()` as soon as possible (rather than via the garbage collector in the finalization process). The same can be said for any type that supports the `IDisposable` interface. When working with GDI+ `Graphics` objects, remember the following rules of thumb:

- If you directly create a `Graphics` object, dispose of it when you are finished.
- If you reference an existing `Graphics` object, do *not* dispose of it.

To clarify, consider the following `Paint` event handler:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    // Load a local *.jpg file.
    Image myImageFile = Image.FromFile("landscape.jpg");

    // Create new Graphics object based on the image.
    Graphics imgGraphics = Graphics.FromImage(myImageFile);

    // Render new data onto the image.
    imgGraphics.FillEllipse(Brushes.DarkOrange, 50, 50, 150, 150);

    // Draw image to Form.
    Graphics g = e.Graphics;
    g.DrawImage(myImageFile, new PointF(0.0F, 0.0F));

    // Release Graphics object we created.
    imgGraphics.Dispose();
}
```

Now at this point in the chapter, don't become concerned if some of this GDI+ logic looks a bit foreign. However, notice that you are obtaining a `Graphics` object from a `*.jpg` file loaded from the local application directory (via the static `Graphics.FromImage()` method). Because you have explicitly created this `Graphics` object, best practice states that you should `Dispose()` of the object when you have finished making use of it, to free up the internal resources for use by other parts of the system.

However, notice that you did not explicitly call `Dispose()` on the `Graphics` object you obtained from the incoming `PaintEventArgs`. This is due to the fact that you did not directly create the object and cannot ensure other parts of the program are making use of it. Clearly, it would be a problem if you released a `Graphics` object used elsewhere!

On a related note, recall from our examination of the .NET garbage collector in Chapter 5 that if you do forget to call `Dispose()` on a method implementing `IDisposable`, the internal resources will eventually be freed when the object is garbage-collected at a later time. In this light, the manual disposal of the `imgGraphics` object is not technically necessary. Although explicitly disposing of GDI+ objects you directly created is smart programming, in order to keep the code examples in this chapter crisp, I will not manually dispose of each GDI+ type.

The GDI+ Coordinate Systems

Our next task is to examine the underlying coordinate system. GDI+ defines three distinct coordinate systems, which are used by the runtime to determine the location and size of the content to be rendered. First we have what are known as *world coordinates*. World coordinates represent an abstraction of the size of a given GDI+ type, irrespective of the unit of measurement. For example, if you draw a rectangle using the dimensions (0, 0, 100, 100), you have specified a rectangle 100×100 "things" in size. As you may guess, the default "thing" is a pixel; however, it can be configured to be another unit of measure (inch, centimeter, etc.).

Next, we have *page coordinates*. Page coordinates represent an offset applied to the original world coordinates. This is helpful in that you are not the one in charge of manually applying offsets in your code (should you need them). For example, if you have a Form that needs to maintain a 100×100 pixel border, you can specify a (100*100) page coordinate to allow all rending to begin at point (100*100). In your code base, however, you are able to specify simple world coordinates (thereby avoiding the need to manually calculate the offset).

Finally, we have *device coordinates*. Device coordinates represent the result of applying page coordinates to the original world coordinates. This coordinate system is used to determine exactly where the GDI+ type will be rendered. When you are programming with GDI+, you will typically think in terms of world coordinates, which are the baseline used to determine the size and location of a GDI+ type. To render in world coordinates requires no special coding actions—simply pass in the dimensions for the current rendering operation:

```
void MainForm_Paint(object sender, PaintEventArgs e)
{
     // Render a rectangle in world coordinates.
     Graphics g = e.Graphics;
     g.DrawRectangle(Pens.Black, 10, 10, 100, 100);
}
```

Under the hood, your world coordinates are automatically mapped in terms of page coordinates, which are then mapped into device coordinates. In many cases, you will never directly make use of page or device coordinates unless you wish to apply some sort of graphical transformation. Given that the previous code did not specify any transformational logic, the world, page, and device coordinates are identical.

If you do wish to apply various transformations before rendering your GDI+ logic, you will make use of various members of the `Graphics` type (such as the `TranslateTransform()` method) to

specify various "page coordinates" to your existing world coordinate system before the rendering operation. The result is the set of device coordinates that will be used to render the GDI+ type to the target device:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     // Specify page coordinate offsets (10 * 10).
     Graphics g = e.Graphics;
     g.TranslateTransform(10, 10);
     g.DrawRectangle(10, 10, 100, 100);
}
```

In this case, the rectangle is actually rendered with a top-left point of (20, 20), given that the world coordinates have been offset by the call to `TranslateTransform()`.

The Default Unit of Measure

Under GDI+, the default unit of measure is pixel-based. The origin begins in the upper-left corner with the *x*-axis increasing to the right and the *y*-axis increasing downward (see Figure 20-2).

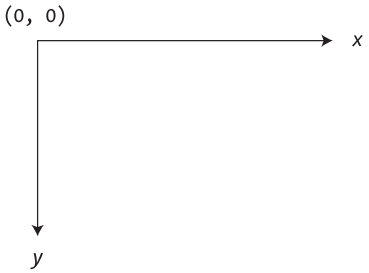

Figure 20-2. *The default coordinate system of GDI+*

Thus, if you render a `Rectangle` using a 5-pixel thick red pen as follows:

```
void MainForm_Paint(object sender, PaintEventArgs e)
{
     // Set up world coordinates using the default unit of measure.
     Graphics g = e.Graphics;
     g.DrawRectangle(new Pen(Color.Red, 5), 10, 10, 100, 100);
}
```

you would see a square rendered 10 pixels down and in from the top-left client edge of the Form, as shown in Figure 20-3.

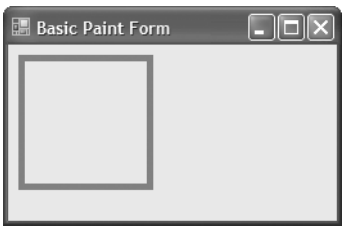

Figure 20-3. *Rendering via pixel units*

Specifying an Alternative Unit of Measure

If you do not wish to render images using a pixel-based unit of measure, you are able to change this default setting by setting the `PageUnit` property of the `Graphics` object to alter the units used by the page coordinate system. The `PageUnit` property can be assigned any member of the `GraphicsUnit` enumeration:

```
public enum GraphicsUnit
{
    // Specifies world coordinates.
    World,
    // Pixels for video displays and 1/100 inch for printers.
    Display,
    // Specifies a pixel.
    Pixel,
    // Specifies a printer's point (1/72 inch).
    Point,
    // Specifies an inch.
    Inch,
    // Specifies a document unit (1/300 inch).
    Document,
    // Specifies a millimeter.
    Millimeter
}
```

To illustrate how to change the underlying `GraphicsUnit`, update the previous rendering code as follows:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     // Draw a rectangle in inches...not pixels.
     Graphics g = e.Graphics;
     g.PageUnit = GraphicsUnit.Inch;
     g.DrawRectangle(new Pen(Color.Red, 5), 0, 0, 100, 100);
}
```

You would find a *radically* different rectangle, as shown in Figure 20-4.

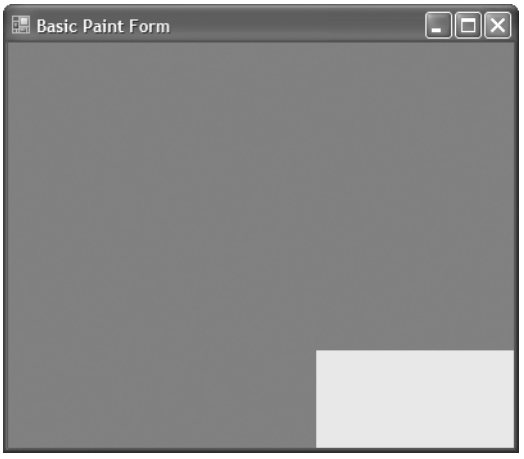

Figure 20-4. *Rendering using inch units*

The reason that 95 percent (or so) of the Form's client area is now filled with red is because you have configured a `Pen` with a 5-*inch* nib! The rectangle itself is 100×100 *inches* in size. In fact, the small gray box you see located in the lower-right corner is the upper-left interior of the rectangle.

Specifying an Alternative Point of Origin

Recall that when you make use of the default coordinate and measurement system, point (0, 0) is at the extreme upper left of the surface area. While this is often what you desire, what if you wish to alter the location where rendering begins? For example, let's assume that your application always needs to reserve a 100-pixel boundary around the Form's client area (for whatever reason). You need to ensure that all GDI+ operations take place somewhere within this internal region.

One approach you could take is to offset all your rendering code manually. This, of course, would be bothersome, as you would need to constantly apply some offset value to each and every rendering operation. It would be far better (and simpler) if you could set a property that says in effect, "Although *I* might say render a rectangle with a point of origin at (0, 0), make sure *you* begin at point (100, 100)." This would simplify your life a great deal, as you could continue to specify your plotting points without modification.

In GDI+, you can adjust the point of origin by setting the transformation value using the `TranslateTransform()` method of the `Graphics` class, which allows you to specify a page coordinate system that will be applied to your original world coordinate specifications, for example:

```
void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
     // Set page coordinate to (100, 100).
     g.TranslateTransform(100, 100);

     // World origin is still (0, 0, 100, 100),
     // however, device origin is now (100, 100, 200, 200).
     g.DrawRectangle(new Pen(Color.Red, 5), 0, 0, 100, 100);
}
```

Here, you have set the world coordinate values (0, 0, 100, 100). However, the page coordinate values have specified an offset of (100, 100). Given this, the device coordinates map to (100, 100, 200, 200). Thus, although the call to `DrawRectangle()` looks as if you are rendering a rectangle on the upper left of the Form, the rendering shown in Figure 20-5 has taken place.

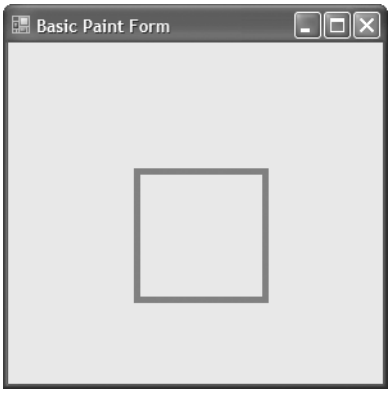

Figure 20-5. *The result of applying page offsets*

To help you experiment with some of the ways to alter the GDI+ coordinate system, this book's downloadable source code (visit the Downloads section of the Apress website at `www.apress.com`) provides a sample application named CoorSystem. Using two menu items, you are able to alter the point of origin as well as the unit of measurement (see Figure 20-6).

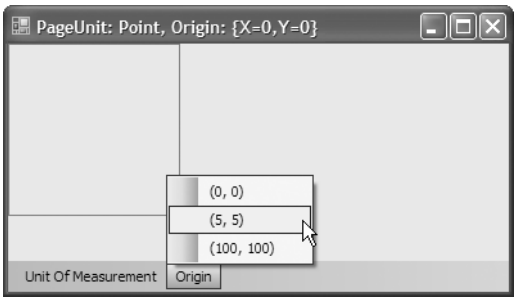

Figure 20-6. *Altering coordinate and measurement modes*

Now that you have a better understanding of the underlying transformations used to determine where to render a given GDI+ type onto a target device, the next order of business is to examine details of color manipulation.

■ **Source Code** The CoorSystem project is included under the Chapter 20 subdirectory.

Defining a Color Value

Many of the rendering methods defined by the `Graphics` class require you to specify the color that should be used during the drawing process. The `System.Drawing.Color` structure represents an alpha-red-green-blue (ARGB) color constant. Most of the `Color` type's functionality comes by way of a number of static read-only properties, which return a specific `Color` type:

```
// One of many predefined colors...
Color c = Color.PapayaWhip;
```

If the default color values do not fit the bill, you are also able to create a new `Color` type and specify the `A`, `R`, `G`, and `B` values using the `FromArgb()` method:

```
// Specify ARGB manually.
Color myColor = Color.FromArgb(0, 255, 128, 64);
```

As well, using the `FromName()` method, you are able to generate a `Color` type given a string value. The characters in the string parameter must match one of the members in the `KnownColor` enumeration (which includes values for various Windows color elements such as `KnownColor.WindowFrame` and `KnownColor.WindowText`):

```
// Get Color from a known name.
Color myColor = Color.FromName("Red");
```

Regardless of the method you use, the `Color` type can be interacted with using a variety of members:

- `GetBrightness()`: Returns the brightness of the `Color` type based on hue-saturation-brightness (HSB) measurements
- `GetSaturation()`: Returns the saturation of the `Color` type based on HSB measurements
- `GetHue()`: Returns the hue of the `Color` type based on HSB measurements
- `IsSystemColor`: Determines if the `Color` type is a registered system color
- `A`, `R`, `G`, `B`: Returns the value assigned to the alpha, red, green, and blue aspects of a `Color` type

The ColorDialog Class

If you wish to provide a way for the end user of your application to configure a `Color` type, the `System.Windows.Forms` namespace provides a predefined dialog box class named `ColorDialog` (see Figure 20-7).

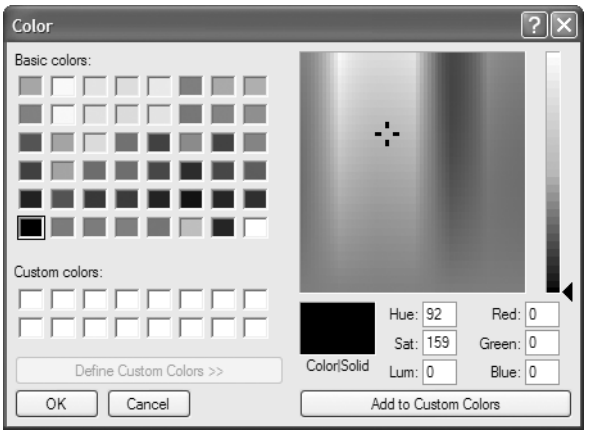

Figure 20-7. *The Windows Forms color dialog box*

Working with this dialog box is quite simple. Using a valid instance of the `ColorDialog` type, call `ShowDialog()` to display the dialog box modally. Once the user has closed the dialog box, you can extract the corresponding `Color` object using the `ColorDialog.Color` property.

Assume you wish to allow the user to configure the background color of the Form's client area using the `ColorDialog`. To keep things simple, you will display the `ColorDialog` when the user clicks anywhere on the client area:

```
public partial class MainForm : Form
{
    private ColorDialog colorDlg;
    private Color currColor = Color.DimGray;

    public MainForm()
    {
        InitializeComponent();
        colorDlg = new ColorDialog();
        Text = "Click on me to change the color";
        this.MouseDown += new MouseEventHandler(MainForm_MouseDown);
    }
```

```
    private void MainForm_MouseDown(object sender, MouseEventArgs e)
    {
        if (colorDlg.ShowDialog() != DialogResult.Cancel)
        {
            currColor = colorDlg.Color;
            this.BackColor = currColor;
            string strARGB = colorDlg.Color.ToString();
            MessageBox.Show(strARGB, "Color is:");
        }
    }
}
```

Source Code The ColorDlg application is included under the Chapter 20 subdirectory.

Manipulating Fonts

Next, let's examine how to programmatically manipulate fonts. The `System.Drawing.Font` type represents a given font installed on the user's machine. Font types can be defined using any number of overloaded constructors. Here are a few examples:

```
// Create a Font of a given type name and size.
Font f = new Font("Times New Roman", 12);

// Create a Font with a given name, size, and style set.
Font f2 = new Font("WingDings", 50, FontStyle.Bold | FontStyle.Underline);
```

Here, `f2` has been created by OR-ing together a set of values from the `FontStyle` enumeration:

```
public enum FontStyle
{
    Regular, Bold,
    Italic, Underline, Strikeout
}
```

Once you have configured the look and feel of your `Font` object, the next task is to pass it as a parameter to the `Graphics.DrawString()` method. Although `DrawString()` has also been overloaded a number of times, each variation typically requires the same basic information: the text to draw, the font to draw it in, a brush used for rendering, and a location in which to place it.

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;

     // Specify (String, Font, Brush, Point) as args.
     g.DrawString("My string", new Font("WingDings", 25),
          Brushes.Black, new Point(0,0));

     // Specify (String, Font, Brush, int, int)
     g .DrawString("Another string", new Font("Times New Roman", 16),
          Brushes.Red, 40, 40);
}
```

Working with Font Families

The `System.Drawing` namespace also defines the `FontFamily` type, which abstracts a group of typefaces having a similar basic design but with certain style variations. A family of fonts, such as Verdana, can include several fonts that differ in style and size. For example, Verdana 12-point bold and Verdana 24-point italic are different fonts within the Verdana font family.

The constructor of the `FontFamily` type takes a string representing the name of the font family you are attempting to capture. Once you create the "generic family," you are then able to create a more specific `Font` object:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;

     // Make a family of fonts.
     FontFamily myFamily = new FontFamily("Verdana");

     // Pass family into ctor of Font.
     Font myFont = new Font(myFamily, 12);
     g.DrawString("Hello!", myFont, Brushes.Blue, 10, 10);
}
```

Of greater interest is the ability to gather statistics regarding a given family of fonts. For example, say you are building a text-processing application and wish to determine the average width of a character in a particular `FontFamily`. What if you wish to know the ascending and descending values for a given character? To answer such questions, the `FontFamily` type defines the key members shown in Table 20-5.

Table 20-5. *Members of the* `FontFamily` *Type*

Member	Meaning in Life
`GetCellAscent()`	Returns the ascender metric for the members in this family
`GetCellDescent()`	Returns the descender metric for members in this family
`GetLineSpacing()`	Returns the distance between two consecutive lines of text for this `FontFamily` with the specified `FontStyle`
`GetName()`	Returns the name of this `FontFamily` in the specified language
`IsStyleAvailable()`	Indicates whether the specified `FontStyle` is available

To illustrate, here is a `Paint` event handler that prints a number of characteristics of the Verdana font family:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
     FontFamily myFamily = new FontFamily("Verdana");
     Font myFont = new Font(myFamily, 12);
     int y = 0;
     int fontHeight = myFont.Height;

     // Show units of measurement for FontFamily members.
     this.Text = "Measurements are in GraphicsUnit." + myFont.Unit;
     g.DrawString("The Verdana family.", myFont, Brushes.Blue, 10, y);
     y += 20;
```

```
    // Print our family ties...
    g.DrawString("Ascent for bold Verdana: " +
                 myFamily.GetCellAscent(FontStyle.Bold),
                 myFont, Brushes.Black, 10, y + fontHeight);
    y += 20;
    g.DrawString("Descent for bold Verdana: " +
                 myFamily.GetCellDescent(FontStyle.Bold),
                 myFont, Brushes.Black, 10, y + fontHeight);
    y += 20;
    g.DrawString("Line spacing for bold Verdana: " +
                 myFamily.GetLineSpacing(FontStyle.Bold),
                 myFont, Brushes.Black, 10, y + fontHeight);
    y += 20;
    g.DrawString("Height for bold Verdana: " +
                 myFamily.GetEmHeight(FontStyle.Bold),
                 myFont, Brushes.Black, 10, y + fontHeight);
    y += 20;
}
```

Figure 20-8 shows the result.

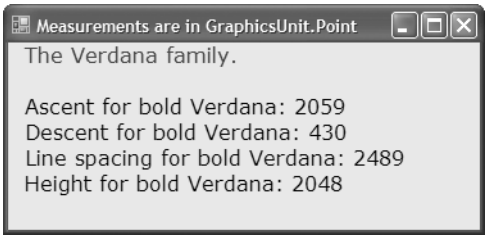

Figure 20-8. *Gathering statistics of the Verdana font family*

Note that these members of the `FontFamily` type return values using `GraphicsUnit.Point` (not `Pixel`) as the unit of measure, which corresponds to 1/72 inch. You are free to transform these values to other units of measure as you see fit.

■ **Source Code** The FontFamilyApp application is included under the Chapter 20 subdirectory.

Working with Font Faces and Font Sizes

Next, you'll build a more complex application that allows the user to manipulate a `Font` object maintained by a Form. The application will allow the user to select the current font face from a predefined set using the Configure ➤ Font Face menu selection. You'll also allow the user to indirectly control the size of the `Font` object using a Windows Forms `Timer` object. If the user activates the `Timer` using the Configure ➤ Swell? menu item, the size of the `Font` object increases at a regular interval (to a maximum upper limit). In this way, the text appears to swell and thus provides an animation of "breathing" text. Finally, you'll use a final menu item under the Configure menu named List All Fonts, which will be used to list all fonts installed on the end user's machine. Figure 20-9 shows the menu UI logic.

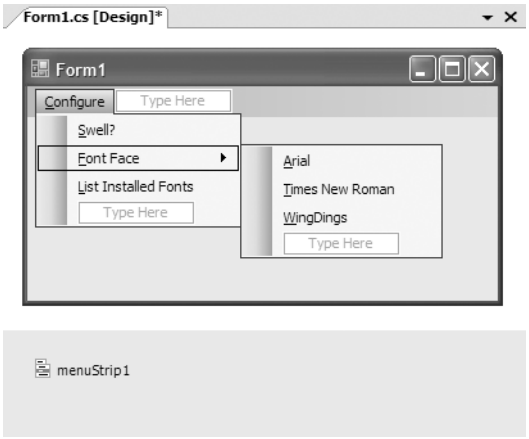

Figure 20-9. *Menu layout of the FontApp project*

To begin implementing the application, update the Form with a `Timer` member variable (named `swellTimer`), a string (`strFontFace`) to represent the current font face, and an integer (`swellValue`) to represent the amount to adjust the font size. Within the Form's constructor, configure the `Timer` to emit a `Tick` event every 100 milliseconds:

```
public partial class MainForm : Form
{
     private Timer swellTimer = new Timer();
     private int swellValue;
     private string strFontFace  = "WingDings";

     public MainForm()
     {
          InitializeComponent();
          BackColor = Color.Honeydew;
          CenterToScreen();

          // Configure the Timer.
          swellTimer.Enabled = true;
          swellTimer.Interval = 100;
          swellTimer.Tick += new EventHandler(swellTimer_Tick);
     }
}
```

In the `Tick` event handler, increase the value of the `swellValue` data member by 5. Recall that the `swellValue` integer will be added to the current font size to provide a simple animation (assume `swellValue` has a maximum upper limit of 50). To help reduce the flicker that can occur when redrawing the entire client area, notice how the call to `Invalidate()` is only refreshing the upper rectangular area of the Form:

```
private void swellTimer_Tick(object sender, EventArgs e)
{
     // Increase current swellValue by 5.
     swellValue += 5;
     // If this value is greater than or equal to 50, reset to zero.
     if(swellValue >= 50)
          swellValue = 0;
```

```
    // Just invalidate the minimal dirty rectangle to help reduce flicker.
    Invalidate(new Rectangle(0, 0, ClientRectangle.Width, 100));
}
```

Now that the upper 100 pixels of your client area are refreshed with each tick of the `Timer`, you had better have something to render! In the Form's `Paint` handler, create a `Font` object based on the user-defined font face (as selected from the appropriate menu item) and current `swellValue` (as dictated by the `Timer`). Once you have your `Font` object fully configured, render a message into the center of the dirty rectangle:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    // Our font size can be between 12 and 62,
    // based on the current swellValue.
    Font theFont = new Font(strFontFace, 12 + swellValue);
    string message = "Hello GDI+";

    // Display message in the center of the rect.
    float windowCenter = this.DisplayRectangle.Width/2;
    SizeF stringSize = g.MeasureString(message, theFont);
    float startPos = windowCenter - (stringSize.Width/2);
    g.DrawString(message, theFont, new SolidBrush(Color.Blue), startPos, 10);
}
```

As you would guess, if a user selects a specific font face, the `Clicked` handler for each menu selection is in charge of updating the `fontFace` string variable and invalidating the client area, for example:

```
private void arialToolStripMenuItem_Click(object sender, EventArgs e)
{
    strFontFace = "Arial";
    Invalidate();
}
```

The `Click` menu handler for the Swell menu item will be used to allow the user to stop or start the swelling of the text (i.e., enable or disable the animation). To do so, toggle the `Enabled` property of the `Timer` as follows:

```
private void swellToolStripMenuItem_Click(object sender, EventArgs e)
{
    swellTimer.Enabled = !swellTimer.Enabled;
}
```

Enumerating Installed Fonts

Next, let's expand this program to display the set of installed fonts on the target machine using types within `System.Drawing.Text`. This namespace contains a handful of types that can be used to discover and manipulate the set of fonts installed on the target machine. For our purposes, we are only concerned with the `InstalledFontCollection` class.

When the user selects the Configure ➤ List Installed Fonts menu item, the corresponding `Clicked` handler creates an instance of the `InstalledFontCollection` class. This class maintains an array named `FontFamily`, which represents the set of all fonts on the target machine and may be obtained using the `InstalledFontCollection.Families` property. Using the `FontFamily.Name` property, you are able to extract the font face (e.g., Times New Roman, Arial, etc.) for each font.

Add a private string data member to your Form named `installedFonts` to hold each font face. The logic in the List Installed Fonts menu handler creates an instance of the `InstalledFontCollection` type, reads the name of each string, and adds the new font face to the private `installedFonts` data member:

```
public partial class MainForm : Form
{
     // Holds the list of fonts.
     private string installedFonts;

     // Menu handler to get the list of installed fonts.
     private void mnuConfigShowFonts_Clicked(object sender, EventArgs e)
     {
          InstalledFontCollection fonts = new InstalledFontCollection();
          for(int i = 0; i < fonts.Families.Length; i++)
               installedFonts += fonts.Families[i].Name + "  ";

          // This time, we need to invalidate the entire client area,
          // as we will paint the installedFonts string on the lower half
          // of the client rectangle.
          Invalidate();
     }
...
}
```

The final task is to render the `installedFonts` string to the client area, directly below the screen real estate that is used for your swelling text:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
     Font theFont = new Font(strFontFace, 12 + swellValue);
     string message = "Hello GDI+";

     // Display message in the center of the window!
     float windowCenter = this.DisplayRectangle.Width/2;
     SizeF stringSize = e.Graphics.MeasureString(message, theFont);
     float startPos = windowCenter - (stringSize.Width/2);
     g.DrawString(message, theFont, Brushes.Blue, startPos, 10);

     // Show installed fonts in the rectangle below the swell area.
     Rectangle myRect = new Rectangle(0, 100,
          ClientRectangle.Width, ClientRectangle.Height);

     // Paint this area of the Form black.
     g.FillRectangle(new SolidBrush(Color.Black), myRect);
     g.DrawString(installedFonts, new Font("Arial", 12),
          Brushes.White, myRect);
}
```

Recall that the size of the "dirty rectangle" has been mapped to the upper 100 pixels of the client rectangle. Because your `Tick` handler invalidates only a portion of the Form, the remaining area is not redrawn when the `Tick` event has been sent (to help optimize the rendering of the client area).

As a final touch to ensure proper redrawing, let's handle the Form's `Resize` event to ensure that if the user resizes the Form, the lower part of client rectangle is redrawn correctly:

```
private void MainForm_Resize(object sender, System.EventArgs e)
{
     Rectangle myRect = new Rectangle(0, 100,
          ClientRectangle.Width, ClientRectangle.Height);
     Invalidate(myRect);
}
```

Figure 20-10 shows the result (with the text rendered in Wingdings!).

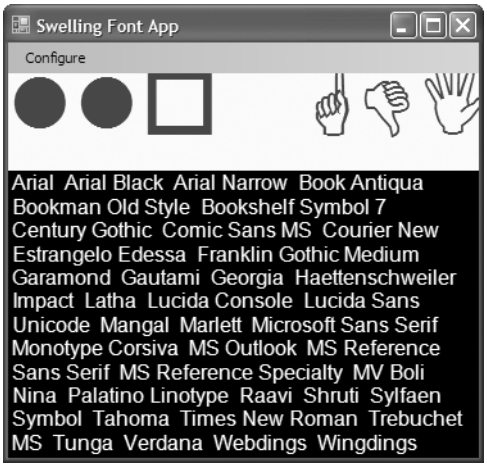

Figure 20-10. *The FontApp application in action*

■ **Source Code** The SwellingFontApp project is included under the Chapter 20 subdirectory.

The FontDialog Class

As you might assume, there is an existing font dialog box (`FontDialog`), as shown in Figure 20-11.

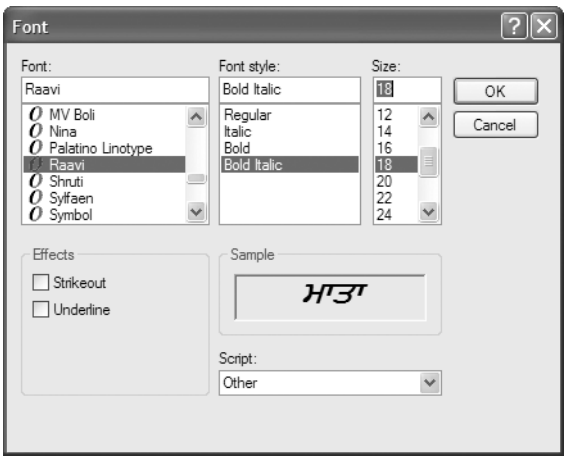

Figure 20-11. *The Windows Forms Font dialog box*

Like the `ColorDialog` type examined earlier in this chapter, when you wish to work with the `FontDialog`, simply call the `ShowDialog()` method. Using the `Font` property, you may extract the

characteristics of the current selection for use in the application. To illustrate, here is a Form that mimics the logic of the previous ColorDlg project. When the user clicks anywhere on the Form, the Font dialog box displays and renders a message with the current selection:

```
public partial class MainForm : Form
{
    private FontDialog fontDlg = new FontDialog();
    private Font currFont = new Font("Times New Roman", 12);
    public MainForm()
    {
        InitializeComponent();
        CenterToScreen();
    }
    private void MainForm_MouseDown(object sender, MouseEventArgs e)
    {
        if (fontDlg.ShowDialog() != DialogResult.Cancel)
        {
            currFont = fontDlg.Font;
            this.Text = string.Format("Selected Font: {0}", currFont);
            Invalidate();
        }
    }
    private void MainForm_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawString("Testing...", currFont, Brushes.Black, 0, 0);
    }
}
```

■ **Source Code** The FontDlgForm application is included under the Chapter 20 subdirectory.

Survey of the System.Drawing.Drawing2D Namespace

Now that you have manipulated Font types, the next task is to examine how to manipulate Pen and Brush objects to render geometric patterns. While you could do so making use of nothing more than Brushes and Pens helper types to obtain preconfigured types in a solid color, you should be aware that many of the more "exotic" pens and brushes are found within the System.Drawing.Drawing2D namespace.

This additional GDI+ namespace provides a number of classes that allow you to modify the end cap (triangle, diamond, etc.) used for a given pen, build textured brushes, and work with vector graphic manipulations. Some core types to be aware of (grouped by related functionality) are shown in Table 20-6.

Table 20-6. *Classes of* System.Drawing.Drawing2D

Classes	Meaning in Life
AdjustableArrowCap CustomLineCap	Pen caps are used to paint the beginning and end points of a given line. These types represent an adjustable arrow-shaped and user-defined cap.
Blend ColorBlend	These classes are used to define a blend pattern (and colors) used in conjunction with a LinearGradientBrush.

Classes	Meaning in Life

`GraphicsPath`
`GraphicsPathIterator`
`PathData`
	A `GraphicsPath` object represents a series of lines and curves. This class allows you to insert just about any type of geometrical pattern (arcs, rectangles, lines, strings, polygons, etc.) into the path. `PathData` holds the graphical data that makes up a path.

`HatchBrush`
`LinearGradientBrush`
`PathGradientBrush`
	These are exotic brush types.

Also be aware that the `System.Drawing.Drawing2D` namespace defines another set of enumerations (`DashStyle`, `FillMode`, `HatchStyle`, `LineCap`, and so forth) that are used in conjunction with these core types.

Working with Pens

GDI+ `Pen` types are used to draw lines between two end points. However, a `Pen` in and of itself is of little value. When you need to render a geometric shape onto a `Control`-derived type, you send a valid `Pen` type to any number of render methods defined by the `Graphics` class. In general, the `DrawXXX()` methods are used to render some set of lines to a graphics surface and are typically used with `Pen` objects.

The `Pen` type defines a small set of constructors that allow you to determine the initial color and width of the pen nib. Most of a `Pen`'s functionality comes by way of its supported properties. Table 20-7 gives a partial list.

Table 20-7. `Pen` *Properties*

Properties	Meaning in Life
`Brush`	Determines the `Brush` used by this `Pen`.
`Color`	Determines the `Color` type used by this `Pen`.
`CustomStartCap` `CustomEndCap`	Gets or sets a custom cap style to use at the beginning or end of lines drawn with this `Pen`. *Cap style* is simply the term used to describe how the initial and final stroke of the `Pen` should look and feel. These properties allow you to build custom caps for your `Pen` types.
`DashCap`	Gets or sets the cap style used at the beginning or end of dashed lines drawn with this `Pen`.
`DashPattern`	Gets or sets an array of custom dashes and spaces. The dashes are made up of line segments.
`DashStyle`	Gets or sets the style used for dashed lines drawn with this `Pen`.
`StartCap` `EndCap`	Gets or sets the predefined cap style used at the beginning or end of lines drawn with this `Pen`. Set the cap of your `Pen` using the `LineCap` enumeration defined in the `System.Drawing.Drawing2D` namespace.
`Width`	Gets or sets the width of this `Pen`.
`DashOffset`	Gets or sets the distance from the start of a line to the beginning of a dash pattern.

Remember that in addition to the `Pen` type, GDI+ provides a `Pens` collection. Using a number of static properties, you are able to retrieve a `Pen` (or a given color) on the fly, rather than creating a custom `Pen` by hand. Be aware, however, that the `Pen` types returned will always have a width of 1.

If you require a more exotic pen, you will need to build a Pen type by hand. This being said, let's render some geometric images using simple Pen types. Assume you have a main Form object that is capable of responding to paint requests. The implementation is as follows:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
     // Make a big blue pen.
     Pen bluePen = new Pen(Color.Blue, 20);

     // Get a stock pen from the Pens type.
     Pen pen2 = Pens.Firebrick;

     // Render some shapes with the pens.
     g.DrawEllipse(bluePen, 10, 10, 100, 100);
     g.DrawLine(pen2, 10, 130, 110, 130);
     g.DrawPie(Pens.Black, 150, 10, 120, 150, 90, 80);

     // Draw a purple dashed polygon as well...
     Pen pen3 = new Pen(Color.Purple, 5);
     pen3.DashStyle = DashStyle.DashDotDot;
     g.DrawPolygon(pen3, new Point[]{new Point(30, 140),
          new Point(265, 200), new Point(100, 225),
          new Point(190, 190), new Point(50, 330),
          new Point(20, 180)});

     // And a rectangle containing some text...
     Rectangle r = new Rectangle(150, 10, 130, 60);
     g.DrawRectangle(Pens.Blue, r);
     g.DrawString("Hello out there...How are ya?",
                  new Font("Arial", 12), Brushes.Black, r);
}
```

Notice that the Pen used to render your polygon makes use of the DashStyle enumeration (defined in System.Drawing.Drawing2D):

```
public enum DashStyle
{
     Solid, Dash, Dot,
     DashDot, DashDotDot, Custom
}
```

In addition to the preconfigured DashStyles, you are able to define custom patterns using the DashPattern property of the Pen type:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
...
     // Draw custom dash pattern all around the border of the form.
     Pen customDashPen = new Pen(Color.BlueViolet, 10);
     float[] myDashes = { 5.0f, 2.0f, 1.0f, 3.0f };
     customDashPen.DashPattern = myDashes;
     g.DrawRectangle(customDashPen, ClientRectangle);
}
```

Figure 20-12 shows the final output of this Paint event handler.

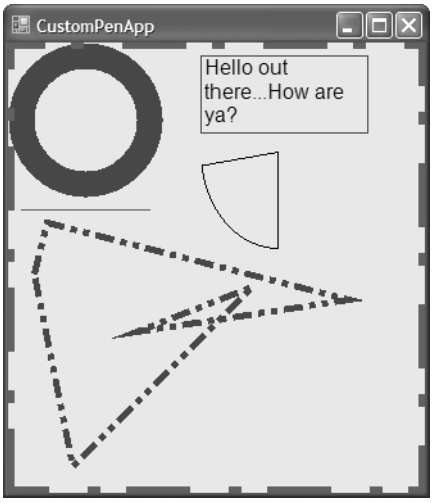

Figure 20-12. *Working with* `Pen` *types*

■ **Source Code** The CustomPenApp project is included under the Chapter 20 subdirectory.

Working with Pen Caps

If you examine the output of the previous pen example, you should notice that the beginning and end of each line was rendered using a standard pen protocol (an end cap composed of 90 degree angles). Using the `LineCap` enumeration, however, you are able to build `Pens` that exhibit a bit more flair:

```
public enum LineCap
{
     Flat, Square, Round,
     Triangle, NoAnchor,
     SquareAnchor, RoundAnchor,
     DiamondAnchor, ArrowAnchor,
     AnchorMask, Custom
}
```

To illustrate, the following Pens application draws a series of lines using each of the `LineCap` styles. The end result can be seen in Figure 20-13.

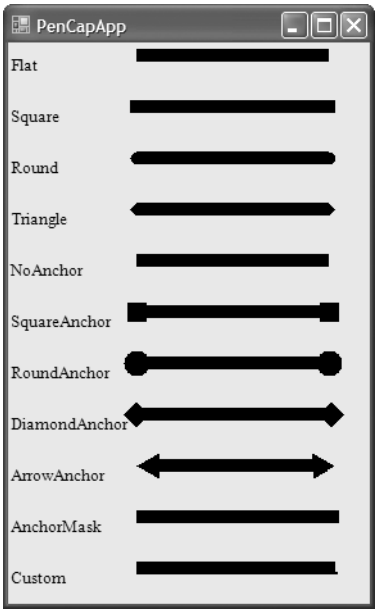

Figure 20-13. *Working with pen caps*

The code simply loops through each member of the `LineCap` enumeration and prints out the name of the item (e.g., `ArrowAnchor`). It then configures and draws a line with the current cap:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
     Pen thePen = new Pen(Color.Black, 10);
     int yOffSet = 10;
     // Get all members of the LineCap enum.
     Array obj = Enum.GetValues(typeof(LineCap));

     // Draw a line with a LineCap member.
     for(int x = 0; x < obj.Length; x++)
     {
          // Get next cap and configure pen.
          LineCap temp = (LineCap)obj.GetValue(x);
          thePen.StartCap = temp;
          thePen.EndCap = temp;

          // Print name of LineCap enum.
          g.DrawString(temp.ToString(), new Font("Times New Roman", 10),
                       new SolidBrush(Color.Black), 0, yOffSet);

          // Draw a line with the correct cap.
          g.DrawLine(thePen, 100, yOffSet, Width - 50, yOffSet);
          yOffSet += 40;
     }
}
```

■ **Source Code** The PenCapApp project is included under the Chapter 20 subdirectory.

Working with Brushes

`System.Drawing.Brush`-derived types are used to fill a region with a given color, pattern, or image. The `Brush` class itself is an abstract type and cannot be directly created. However, `Brush` serves as a base class to the other related brush types (e.g., `SolidBrush`, `HatchBrush`, `LinearGradientBrush`, and so forth). In addition to specific `Brush`-derived types, the `System.Drawing` namespace also defines two helper classes that return a configured brush using a number of static properties: `Brushes` and `SystemBrushes`. In any case, once you obtain a brush, you are able to call any number of the `FillXXX()` methods of the `Graphics` type.

Interestingly enough, you are also able to build a custom `Pen` type based on a given brush. In this way, you are able to build some brush of interest (e.g., a brush that paints a bitmap image) and render geometric patterns with configured `Pen`. To illustrate, here is a small sample program that makes use of various `Brushes`:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;

     // Make a blue SolidBrush.
     SolidBrush blueBrush = new SolidBrush(Color.Blue);
     // Get a stock brush from the Brushes type.
     SolidBrush pen2 = (SolidBrush)Brushes.Firebrick;
     // Render some shapes with the brushes.
     g.FillEllipse(blueBrush, 10, 10, 100, 100);
     g.FillPie(Brushes.Black, 150, 10, 120, 150, 90, 80);
     // Draw a purple polygon as well...
     SolidBrush brush3= new SolidBrush(Color.Purple);
     g.FillPolygon(brush3, new Point[]{ new Point(30, 140),
          new Point(265, 200), new Point(100, 225),
          new Point(190, 190), new Point(50, 330),
          new Point(20, 180)} );
     // And a rectangle with some text...
     Rectangle r = new Rectangle(150, 10, 130, 60);
     g.FillRectangle(Brushes.Blue, r);
     g.DrawString("Hello out there...How are ya?",
                  new Font("Arial", 12), Brushes.White, r);
}
```

If you can't tell, this application is little more than the CustomPenApp program, this time making use of the `FillXXX()` methods and `SolidBrush` types, rather than pens and the related `DrawXXX()` methods. Figure 20-14 shows the output.

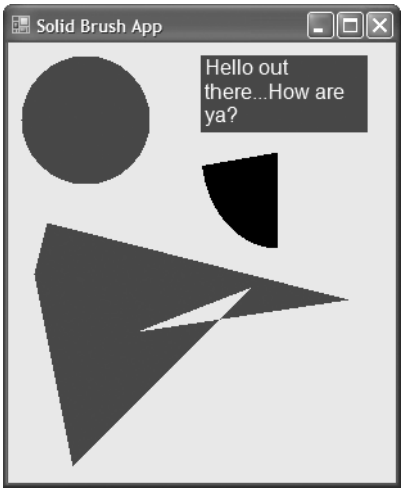

Figure 20-14. *Working with* `Brush` *types*

■ **Source Code** The SolidBrushApp project is included under the Chapter 20 subdirectory.

Working with HatchBrushes

The `System.Drawing.Drawing2D` namespace defines a `Brush`-derived type named `HatchBrush`. This type allows you to fill a region using a (very large) number of predefined patterns, represented by the `HatchStyle` enumeration. Here is a partial list of names:

```
public enum HatchStyle
{
    Horizontal, Vertical, ForwardDiagonal,
    BackwardDiagonal, Cross, DiagonalCross,
    LightUpwardDiagonal, DarkDownwardDiagonal,
    DarkUpwardDiagonal, LightVertical,
    NarrowHorizontal, DashedDownwardDiagonal,
    SmallConfetti, LargeConfetti, ZigZag,
    Wave, DiagonalBrick, Divot, DottedGrid, Sphere,
    OutlinedDiamond, SolidDiamond,
...
}
```

When constructing a `HatchBrush`, you need to specify the foreground and background colors to use during the fill operation. To illustrate, let's rework the logic seen previously in the PenCapApp example:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    int yOffSet = 10;
```

```
    // Get all members of the HatchStyle enum.
    Array obj = Enum.GetValues(typeof(HatchStyle));
    // Draw an oval with first 5 HatchStyle values.
    for (int x = 0; x < 5; x++)
    {
        // Configure Brush.
        HatchStyle temp = (HatchStyle)obj.GetValue(x);
        HatchBrush theBrush = new HatchBrush(temp,
                Color.White, Color.Black);
        // Print name of HatchStyle enum.
        g.DrawString(temp.ToString(), new Font("Times New Roman", 10),
             Brushes.Black, 0, yOffSet);
        // Fill a rectangle with the correct brush.
        g.FillEllipse(theBrush, 150, yOffSet, 200, 25);
        yOffSet += 40;
    }
}
```

The output renders a filled oval for the first five hatch values (see Figure 20-15).

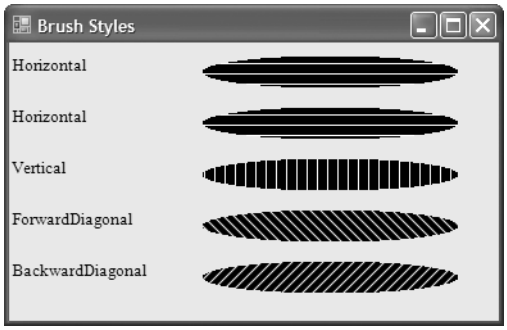

Figure 20-15. *Select hatch styles*

■ **Source Code** The BrushStyles application is included under the Chapter 20 subdirectory.

Working with TextureBrushes

The `TextureBrush` type allows you to attach a bitmap image to a brush, which can then be used in conjunction with a fill operation. In just a few pages, you will learn about the details of the GDI+ `Image` class. For the time being, understand that a `TextureBrush` is assigned an `Image` reference for use during its lifetime. The image itself is typically found stored in some local file (`*.bmp`, `*.gif`, `*.jpg`) or embedded into a .NET assembly.

Let's build a sample application that makes use of the `TextureBrush` type. One brush is used to paint the entire client area with the image found in a file named `clouds.bmp`, while the other brush is used to paint text with the image found within `soap bubbles.bmp`. The output is shown in Figure 20-16.

Figure 20-16. *Bitmaps as brushes*

To begin, your `Form`-derived class maintains two `Brush` member variables, which are assigned to a new `TextureBrush` in the constructor. Notice that the constructor of the `TextureBrush` type requires a type derived from `Image`:

```
public partial class MainForm : Form
{
     // Data for the image brush.
     private Brush texturedTextBrush;
     private Brush texturedBGroundBrush;

     public MainForm()
     {
          ...
          // Load image for background brush.
          Image bGroundBrushImage = new Bitmap("Clouds.bmp");
          texturedBGroundBrush = new TextureBrush(bGroundBrushImage);

          // Now load image for text brush.
          Image textBrushImage = new Bitmap("Soap Bubbles.bmp");
          texturedTextBrush = new TextureBrush(textBrushImage);
     }
}
```

■ **Note** The `*.bmp` files used in this example must be in the same folder as the application (or specified using hard-coded paths). We'll address this limitation later in this chapter.

Now that you have two `TextureBrush` types to render with, the `Paint` event handler is quite straightforward:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
     Rectangle r = ClientRectangle;
     // Paint the clouds on the client area.
     g.FillRectangle(texturedBGroundBrush, r);
     // Some big bold text with a textured brush.
     g.DrawString("Bitmaps as brushes!  Way cool...",
          new Font("Arial", 30,
          FontStyle.Bold | FontStyle.Italic),  texturedTextBrush, r);
}
```

■ **Source Code** The TexturedBrushes application is included under the Chapter 20 subdirectory.

Working with LinearGradientBrushes

Last but not least is the `LinearGradientBrush` type, which you can use whenever you want to blend two colors together in a gradient pattern. Working with this type is just as simple as working with the other brush types. The only point of interest is that when you build a `LinearGradientBrush`, you need to specify a pair of `Color` types and the direction of the blend via the `LinearGradientMode` enumeration:

```
public enum LinearGradientMode
{
     Horizontal, Vertical,
     ForwardDiagonal, BackwardDiagonal
}
```

To test each value, let's render a series of rectangles using a `LinearGradientBrush`:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
     Rectangle r = new Rectangle(10, 10, 100, 100);
     // A gradient brush.
     LinearGradientBrush theBrush = null;
     int yOffSet = 10;
     // Get all members of the LinearGradientMode enum.
     Array obj = Enum.GetValues(typeof(LinearGradientMode));
     // Draw an oval with a LinearGradientMode member.
     for(int x = 0; x < obj.Length; x++)
     {
          // Configure Brush.
          LinearGradientMode temp = (LinearGradientMode)obj.GetValue(x);
          theBrush = new LinearGradientBrush(r, Color.GreenYellow,
                                             Color.Blue, temp);
          // Print name of LinearGradientMode enum.
          g.DrawString(temp.ToString(), new Font("Times New Roman", 10),
                       new SolidBrush(Color.Black), 0, yOffSet);
          // Fill a rectangle with the correct brush.
          g. FillRectangle(theBrush, 150, yOffSet, 200, 50);
          yOffSet += 80;
     }
}
```

Figure 20-17 shows the end result.

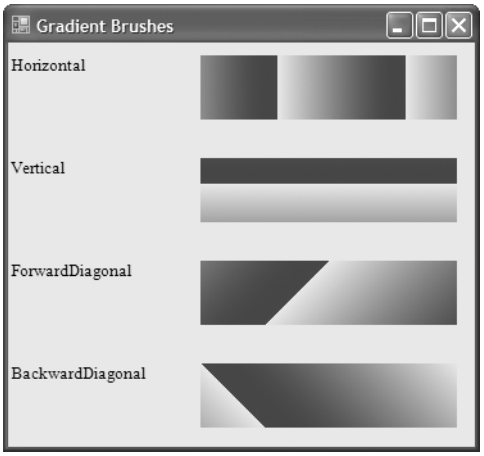

Figure 20-17. *Gradient brushes at work*

■ **Source Code** The GradientBrushes application is included under the Chapter 20 subdirectory.

Rendering Images

At this point, you have examined how to manipulate three of the four major GDI+ types: fonts, pens, and brushes. The final type you'll examine in this chapter is the `Image` class and related subtypes. The abstract `System.Drawing.Image` type defines a number of methods and properties that hold various bits of information regarding the underlying image data it represents. For example, the `Image` class supplies the `Width`, `Height`, and `Size` properties to retrieve the dimensions of the image. Other properties allow you to gain access to the underlying palette. The `Image` class defines the core members shown in Table 20-8.

Table 20-8. *Members of the* `Image` *Type*

Members	Meaning in Life
`FromFile()`	This static method creates an `Image` from the specified file.
`FromStream()`	This static method creates an `Image` from the specified data stream.

`Height`
`Width`
`Size`
`HorizontalResolution`
`VerticalResolution`
	These properties return information regarding the dimensions of this `Image`.

`Palette`	This property returns a `ColorPalette` data type that represents the underlying palette used for this `Image`.
`GetBounds()`	This method returns a `Rectangle` that represents the current size of this `Image`.
`Save()`	This method saves the data held in an `Image`-derived type to file.

Given that the abstract `Image` class cannot be directly created, you typically make a direct instance of the `Bitmap` type. Assume you have some `Form`-derived class that renders three bitmaps

into the client area. Once you fill the `Bitmap` types with the correct image file, simply render each one within your `Paint` event handler using the `Graphics.DrawImage()` method:

```
public partial class MainForm : Form
{
    private Bitmap[] myImages = new Bitmap[3];
    public MainForm()
    {
        // Load some local images.
        myImages[0] = new Bitmap("imageA.bmp");
        myImages[1] = new Bitmap("imageB.bmp");
        myImages[2] = new Bitmap("imageC.bmp");
        CenterToScreen();
        InitializeComponent();
    }

    private void MainForm_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        // Render all three images.
        int yOffset = 10;
        foreach (Bitmap b in myImages)
        {
            g.DrawImage(b, 10, yOffset, 90, 90);
            yOffset += 100;
        }
    }
}
```

■ **Note** The `*.bmp` files used in this example must be in the same folder as the application (or specified using hard-coded paths). We'll resolve this limitation later in this chapter.

Figure 20-18 shows the output.

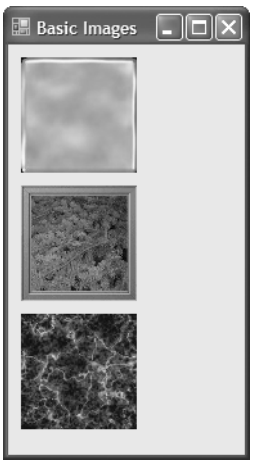

Figure 20-18. *Rendering images*

Finally, be aware that regardless of its name, the `Bitmap` class can contain image data stored in any number of file formats (`*.tif`, `*.gif`, `*.bmp`, etc.).

■ **Source Code** The BasicImages application is included under the Chapter 20 subdirectory.

Dragging and Hit Testing the PictureBox Control

While you are free to render `Bitmap` images directly onto any `Control`-derived class, you will find that you gain far greater control and functionality if you instead choose to make use of a `PictureBox` type to contain your image. For example, because the `PictureBox` type "is-a" `Control`, you inherit a great deal of functionality, such as the ability to handle various events, assign a tool tip or context menu, and so forth. While you could achieve similar behaviors using a raw `Bitmap`, you would be required to author a fair amount of boilerplate code.

To showcase the usefulness of the `PictureBox` type, let's create a simple "game" that illustrates the ability to capture mouse activity over a graphical image. If the user clicks the mouse somewhere within the bounds of the image, he is in "dragging" mode and can move the image around the Form. To make things more interesting, let's monitor where the user releases the image. If it is within the bounds of a GDI+-rendered rectangle, you'll take some additional course of action (seen shortly). As you may know, the process of testing for mouse click events within a specific region is termed *hit testing*.

The `PictureBox` type gains most of its functionality from the `Control` base class. You've already explored a number of `Control`'s members in the previous chapter, so let's quickly turn your attention to the process of assigning an image to the `PictureBox` member variable using the `Image` property (again, the `happyDude.bmp` file must be in the application directory):

```
public partial class MainForm : Form
{
     // This holds an image of a smiley face.
     private PictureBox happyBox = new PictureBox();

     public MainForm()
     {
          // Configure the PictureBox.
          happyBox.SizeMode = PictureBoxSizeMode.StretchImage;
          happyBox.Location = new System.Drawing.Point(64, 32);
          happyBox.Size = new System.Drawing.Size(50, 50);
          happyBox.Cursor = Cursors.Hand;
          happyBox.Image = new Bitmap("happyDude.bmp");

          // Now add to the Form's Controls collection.
          Controls.Add(happyBox);
     }
}
```

Beyond the `Image` property, the only other property of interest is `SizeMode`, which makes use of the `PictureBoxSizeMode` enumeration. This type is used to control how the associated image should be rendered within the bounding rectangle of the `PictureBox`. Here, you assigned `PictureBoxSizeMode.StretchImage`, indicating that you wish to skew the image over the entire area of the `PictureBox` type (which is set to 50×50 pixels).

The next task is to handle the `MouseMove`, `MouseUp`, and `MouseDown` events for the `PictureBox` member variable using the expected C# event syntax:

```
public MainForm()
{
...
    // Add handlers for the following events.
    happyBox.MouseDown += new MouseEventHandler(happyBox_MouseDown);
    happyBox.MouseUp += new MouseEventHandler(happyBox_MouseUp);
    happyBox.MouseMove += new MouseEventHandler(happyBox_MouseMove);
    Controls.Add(happyBox);
    InitializeComponent();
}
```

The `MouseDown` event handler is in charge of storing the incoming (*x*, *y*) location of the cursor within two `System.Int32` member variables (`oldX` and `oldY`) for later use, as well as setting a `System.Boolean` member variable (`isDragging`) to true, to indicate that a drag operation is in process. Add these member variables to your Form and implement the `MouseDown` event handler as so:

```
private void happyBox_MouseDown(object sender, MouseEventArgs e)
{
     isDragging = true;
     oldX = e.X;
     oldY = e.Y;
}
```

The `MouseMove` event handler simply relocates the position of the `PictureBox` (using the `Top` and `Left` properties) by offsetting the current cursor location with the integer data captured during the `MouseDown` event:

```
private void happyBox_MouseMove(object sender, MouseEventArgs e)
{
     if (isDragging)
     {
          // Need to figure new Y value based on where the mouse
          // down click happened.
          happyBox.Top = happyBox.Top + (e.Y - oldY);
          // Same process for X (use oldX as a baseline).
          happyBox.Left = happyBox.Left + (e.X - oldX);
     }
}
```

The `MouseUp` event handler sets the `isDragging` Boolean to false, to signal the end of the drag operation. As well, if the `MouseUp` event occurs when the `PictureBox` is contained within our GDI+-rendered `Rectangle` image, you can assume the user has won the (albeit rather simplistic) game. First, add a `Rectangle` member variable (named `dropRect`) to your `Form` class set to a given size:

```
public partial class MainForm : Form
{
     private PictureBox happyBox = new PictureBox();
     private int oldX, oldY;
     private bool isDragging;
     private Rectangle dropRect = new Rectangle(100, 100, 140, 170);
...
}
```

The `MouseUp` event handler can now be implemented as so:

```
private void happyBox_MouseUp(object sender, MouseEventArgs e)
{
     isDragging = false;

     // Is the mouse within the area of the drop rect?
```

```
     if(dropRect.Contains(happyBox.Bounds))
          MessageBox.Show("You win!", "What an amazing test of skill...");
}
```

Finally, you need to render the rectangular area (maintained by the `dropRect` member variable) on the Form within a `Paint` event handler:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     // Draw the drop box.
     Graphics g = e.Graphics;
     g.FillRectangle(Brushes.AntiqueWhite, dropRect);
     // Display instructions.
     g.DrawString("Drag the happy guy in here...",
          new Font("Times New Roman", 25), Brushes.Red, dropRect);
}
```

When you run the application, you are presented with what appears in Figure 20-19.

Figure 20-19. *The amazing happy-dude game*

If you have what it takes to win the game, you are rewarded with the kudos shown in Figure 20-20.

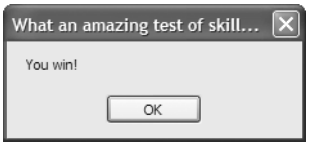

Figure 20-20. *You have nerves of steel!*

■ **Source Code** The DraggingImages application is included under the Chapter 20 subdirectory.

Hit Testing Rendered Images

Validating a hit test against a `Control`-derived type (such as the `PictureBox`) is very simple, as it can respond directly to mouse events. However, what if you wish to perform a hit test on a geometric shape rendered directly on the surface of a Form?

To illustrate the process, let's revisit the previous `BasicImages` application and add some new functionality. The goal is to determine when the user clicks one of the three images. Once you discover which image was clicked, you'll adjust the `Text` property of the Form and highlight the image with a 5-pixel outline.

The first step is to define a new set of member variables in the `Form` type that represents the `Rectangles` you will be testing against in the `MouseDown` event. When this event occurs, you need to programmatically figure out if the incoming (*x*, *y*) coordinate is somewhere within the bounds of the `Rectangles` used to represent the dimension of each `Image`. If the user does click a given image, you set a private Boolean member variable (`isImageClicked`) to true and indicate which image was selected via another member variable of a custom enumeration named `ClickedImage`, defined as so:

```
enum ClickedImage
{
    ImageA, ImageB, ImageC
}
```

With this, here is the initial update to the `Form`-derived class:

```
public partial class MainForm : Form
{
    private Bitmap[] myImages = new Bitmap[3];
    private Rectangle[] imageRects = new Rectangle[3];
    private bool isImageClicked = false;
    ClickedImage imageClicked = ClickedImage.ImageA;

    public MainForm()
    {
...
        // Set up the rectangles.
        imageRects[0] = new Rectangle(10, 10, 90, 90);
        imageRects[1] = new Rectangle(10, 110, 90, 90);
        imageRects[2] = new Rectangle(10, 210, 90, 90);
    }

    private void MainForm_MouseDown(object sender, MouseEventArgs e)
    {
        // Get (x, y) of mouse click.
        Point mousePt = new Point(e.X, e.Y);

        // See if the mouse is anywhere in the 3 Rectangles.
        if (imageRects[0].Contains(mousePt))
        {
            isImageClicked = true;
            imageClicked = ClickedImage.ImageA;
            this.Text = "You clicked image A";
        }
        else if (imageRects[1].Contains(mousePt))
        {
            isImageClicked = true;
            imageClicked = ClickedImage.ImageB;
            this.Text = "You clicked image B";
        }
        else if (imageRects[2].Contains(mousePt))
```

```
        {
            isImageClicked = true;
            imageClicked = ClickedImage.ImageC;
            this.Text = "You clicked image C";
        }
        else     // Not in any shape, set defaults.
        {
            isImageClicked = false;
            this.Text = "Hit Testing Images";
        }
        // Redraw the client area.
        Invalidate();
    }
}
```

Notice that the final conditional check sets the `isImageClicked` member variable to false, indicating that the user did not click one of the three images. This is important, as you want to erase the outline of the previously selected image. Once all items have been checked, invalidate the client area. Here is the updated `Paint` handler:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;

    // Render all three images.
...
    // Draw outline (if clicked)
    if (isImageClicked == true)
    {
        Pen outline = new Pen(Color.Tomato, 5);
        switch (imageClicked)
        {
            case ClickedImage.ImageA:
                g.DrawRectangle(outline, imageRects[0]);
                break;
            case ClickedImage.ImageB:
                g.DrawRectangle(outline, imageRects[1]);
                break;
            case ClickedImage.ImageC:
                g.DrawRectangle(outline, imageRects[2]);
                break;
            default:
                break;
        }
    }
}
```

At this point, you should be able to run your application and validate that an outline appears around each image that has been clicked (and that no outline is present when you click outside the bounds of said images).

Hit Testing Nonrectangular Images

Now, what if you wish to perform a hit test in a nonrectangular region, rather than a simple square? Assume you updated your application to render an oddball geometric shape that will also sport an outline when clicked (see Figure 20-21).

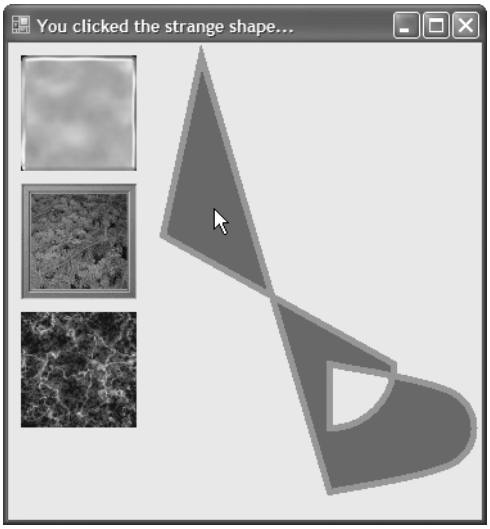

Figure 20-21. *Hit-testing polygons*

This geometric image was rendered on the Form using the `FillPath()` method of the `Graphics` type. This method takes an instance of a `GraphicsPath` object, which encapsulates a series of connected lines, curves, and strings. Adding new items to a `GraphicsPath` instance is achieved using a number of related `Add` methods, as described in Table 20-9.

Table 20-9. `Add`-*Centric Methods of the* `GraphicsPath` *Class*

Methods	Meaning in Life
`AddArc()`	Appends an elliptical arc to the current figure
`AddBezier()` `AddBeziers()`	Adds a cubic Bezier curve (or set of Bezier curves) to the current figure
`AddClosedCurve()`	Adds a closed curve to the current figure
`AddCurve()`	Adds a curve to the current figure
`AddEllipse()`	Adds an ellipse to the current figure
`AddLine()` `AddLines()`	Appends a line segment to the current figure
`AddPath()`	Appends the specified `GraphicsPath` to the current figure
`AddPie()`	Adds the outline of a pie shape to the current figure
`AddPolygon()`	Adds a polygon to the current figure
`AddRectangle()` `AddRectangles()`	Adds one (or more) rectangle to the current figure
`AddString()`	Adds a text string to the current figure

Specify that you are "using" `System.Drawing.Drawing2D` and add a new `GraphicsPath` member variable to your `Form`-derived class. In the Form's constructor, build the set of items that represent your path as follows:

```
public partial class MainForm : Form
{
     GraphicsPath myPath = new GraphicsPath();
...
     public MainForm()
     {
          // Create an interesting path.
          myPath.StartFigure();
          myPath.AddLine(new Point(150, 10), new Point(120, 150));
          myPath.AddArc(200, 200, 100, 100, 0, 90);
          Point point1 = new Point(250, 250);
          Point point2 = new Point(350, 275);
          Point point3 = new Point(350, 325);
          Point point4 = new Point(250, 350);
          Point[] points = { point1, point2, point3, point4} ;
          myPath.AddCurve(points);
          myPath.CloseFigure();
     ...
     }
}
```

Notice the calls to `StartFigure()` and `CloseFigure()`. When you call `StartFigure()`, you are able to insert a new item into the current path you are building. A call to `CloseFigure()` closes the current figure and begins a new figure (if you require one). Also know that if the figure contains a sequence of connected lines and curves (as in the case of the `myPath` instance), the loop is closed by connecting a line from the endpoint to the starting point. First, add an additional name to the `ImageClicked` enumeration named `StrangePath`:

```
enum ClickedImage
{
    ImageA, ImageB,
    ImageC, StrangePath
}
```

Next, update your existing `MouseDown` event handler to test for the presence of the cursor's (*x*, *y*) position within the bounds of the `GraphicsPath`. Like a `Region` type, this can be discovered using the `IsVisible()` member:

```
protected void OnMouseDown (object sender, MouseEventArgs e)
{
     // Get (x, y) of mouse click.
     Point mousePt = new Point(e.X, e.Y);
     ...
     else if(myPath.IsVisible(mousePt))
     {
          isImageClicked = true;
          imageClicked = ClickedImage.StrangePath;
          this.Text = "You clicked the strange shape...";
     }
...
}
```

Finally, update the `Paint` handler as follows:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
     Graphics g = e.Graphics;
     ...
     // Draw the graphics path.
     g.FillPath(Brushes.Sienna, myPath);
```

```
    // Draw outline (if clicked)
    if(isImageClicked == true)
    {
        Pen outline = new Pen(Color.Red, 5);
        switch(imageClicked)
        {
            ...
            case ClickedImage.StrangePath:
                g.DrawPath(outline, myPath);
                break;
            default:
                break;
        }
    }
}
```

Source Code The HitTestingImages project is included under the Chapter 20 subdirectory.

Understanding the .NET Resource Format

Up to this point in the chapter, each application that made use of external resources (such as bitmap files) demanded that the image files be within the client's application directory. Given this, you loaded your `*.bmp` files using an absolute name:

```
// Fill the images with bitmaps.
bMapImageA = new Bitmap("imageA.bmp");
bMapImageB = new Bitmap("imageB.bmp");
bMapImageC = new Bitmap("imageC.bmp");
```

This logic, of course, demands that the application directory does indeed contain three files named `imageA.bmp`, `imageB.bmp`, and `imageC.bmp`; otherwise, you will receive a runtime exception.

As you may recall from Chapter 11, an assembly is a collection of types and *optional resources*. Given this, your final task of the chapter is to learn how to bundle external resources (such as image files and strings) into the assembly itself. In this way, your .NET binary is truly self-contained. At the lowest level, bundling external resources into a .NET assembly involves the following steps:

1. Create an `*.resx` file that establishes name/value pairs for each resource in your application via XML data representation.
2. Use the `resgen.exe` command-line utility to convert your XML-based `*.resx` file into a binary equivalent (a `*.resources` file).
3. Using the `/resource` flag of the C# compiler, embed the binary `*.resources` file into your assembly.

As you might suspect, these steps are automated when using Visual Studio 2005. You'll examine how this IDE can assist you in just a moment. For the time being, let's check out how to generate and embed .NET resources at the command line.

The System.Resources Namespace

The key to understanding the .NET resource format is to know the types defined within the `System.Resources` namespace. This set of types provides the programmatic means to read and write `*.resx` (XML-based) and `*.resources` (binary) files, as well as obtain resources embedded in a given assembly. Table 20-10 provides a rundown of the core types.

Table 20-10. *Members of the* `System.Resources` *Namespace*

Members	Meaning in Life
`ResourceReader` `ResourceWriter`	These types allow you to read from and write to binary `*.resources` files.
`ResXResourceReader` `ResXResourceWriter`	These types allow you to read from and write to XML-based `*.resx` files.
`ResourceManager`	This type allows you to programmatically obtain embedded resources from a given assembly.

Programmatically Creating an *.resx File

As mentioned, an `*.resx` file is a block of XML data that assigns name/value pairs for each resource in your application. The `ResXResourceWriter` class provides a set of members that allow you to create the `*.resx` file, add binary and string-based resources, and commit them to storage. To illustrate, let's create a simple application (`ResXWriter`) that will generate an `*.resx` file containing an entry for the `happyDude.bmp` file (first seen in the `DraggingImages` example) and a single string resource. The GUI consists of a single `Button` type (see Figure 20-22).

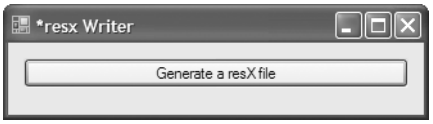

Figure 20-22. *The ResX application*

The `Click` event handler for the `Button` adds the `happyDude.bmp` and string resource to the `*.resx` file, which is saved on the local C drive:

```
private void btnGenResX_Click(object sender, EventArgs e)
{
     // Make an resx writer and specify the file to write to.
     ResXResourceWriter w =
          new ResXResourceWriter(@"C:\ResXForm.resx");
     // Add happy dude and string.
     Image i = new Bitmap("happyDude.bmp");
     w.AddResource("happyDude", i);
     w.AddResource("welcomeString", "Hello new resource format!");
     // Commit file.
     w.Generate();
     w.Close();
}
```

The member of interest is `ResXResourceWriter.AddResource()`. This method has been overloaded a few times to allow you to insert binary data (as you did with the `happyDude.bmp` image), as well as textual data (as you have done for your test string). Notice that each version takes two parameters: the name of a given resource in the `*.resx` file and the data itself. The `Generate()` method commits the information to file. At this point, you have an XML description of the image and string resources. To verify, open the new `ResXForm.resx` file using a text editor (see Figure 20-23).

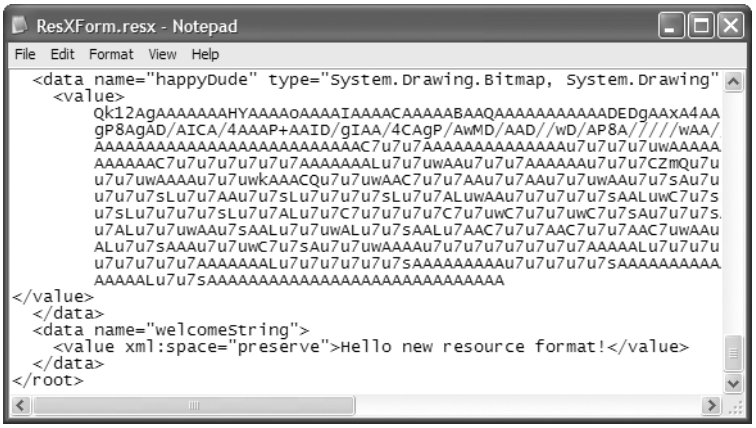

Figure 20-23. `*.resx` *expressed as XML*

Building the *.resources File

Now that you have an `*.resx` file, you can make use of the `resgen.exe` utility to produce the binary equivalent. To do so, open a Visual Studio 2005 command prompt, navigate to your C drive, and issue the following command:

```
resgen resxform.resx resxform.resources
```

You can now open the new `*.resources` file using Visual Studio 2005 and view the binary format (see Figure 20-24).

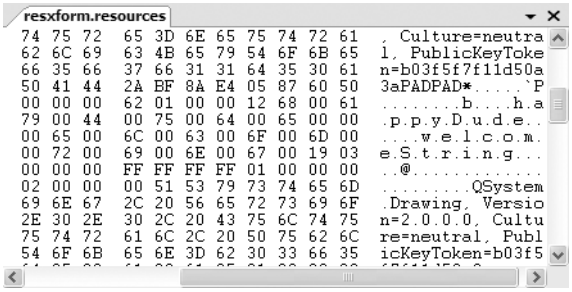

Figure 20-24. *The binary* `*.resources` *file*

Binding the *.resources File into a .NET Assembly

At this point, you are able to embed the `*.resources` file into a .NET assembly using the `/resources` command-line argument of the C# compiler. To illustrate, copy the `Program.cs`, `Form1.cs`, and `Form1.Designer.cs` files to your C drive, open a Visual Studio 2005 command prompt, and issue the following command:

```
csc /resource:resxform.resources /r:System.Drawing.dll *.cs
```

If you were to now open your new assembly using `ildasm.exe`, you would find the manifest has been updated as shown in Figure 20-25.

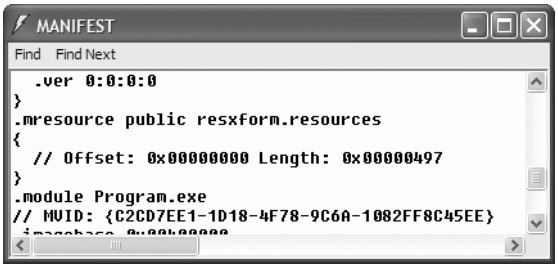

Figure 20-25. *The embedded resources*

Working with ResourceWriters

The previous example made use of the `ResXResourceWriter` types to generate an XML file that contains name/value pairs for each application resource. The resulting `*.resx` file was then run through the `resgen.exe` utility. Finally, the `*.resources` file was embedded into the assembly using the `/resource` flag of the C# compiler. The truth of the matter is that you do not need to build an `*.resx` file (although having an XML representation of your resources can come in handy and is readable). If you do not require an `*.resx` file, you can make use of the `ResourceWriter` type to directly create a binary `*.resources` file:

```
private void GenerateResourceFile()
{
     // Make a new *.resources file.
     ResourceWriter rw;
     rw = new ResourceWriter(@"C:\myResources.resources");

     // Add 1 image and 1 string.
     rw.AddResource("happyDude", new Bitmap("happyDude.bmp"));
     rw.AddResource("welcomeString", "Hello new resource format!");
     rw.Generate();
     rw.Close();
}
```

At this point, the `*.resources` file can be bundled into an assembly using the `/resources` option:

```
csc /resource:myresources.resources *.cs
```

■ **Source Code** The ResXWriter project is included under the Chapter 20 subdirectory.

Generating Resources using Visual Studio 2005

Although it is possible to work with `*.resx`/`*.resources` files manually at the command line, the good news is that Visual Studio 2005 automates the creation and embedding of your project's resources. To illustrate, create a new Windows Forms application named MyResourcesWinApp. Now, if you open Solution Explorer, you will notice that each Form in your application has an associated `*.resx` file in place automatically (see Figure 20-26).

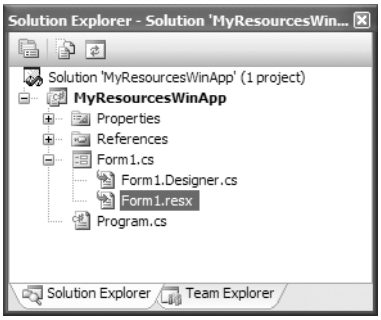

Figure 20-26. *The autogenerated* `*.resx` *files of Visual Studio 2005*

This `*.resx` file will be maintained automatically while you naturally add resources (such as an image in a `PictureBox` widget) using the visual designers. Now, despite what you may be thinking, you should *not* manually update this file to specify your custom resources as Visual Studio 2005 regenerates this file with each compilation. To be sure, you will do well if you allow the IDE to manage a Form's `*.resx` file on your behalf.

When you want to maintain a custom set of resources that are not directly mapped to a given Form, simply insert a new `*.resx` file (named `MyCustomResources.resx` in this example) using the Project ➤ Add New Item menu item (see Figure 20-27).

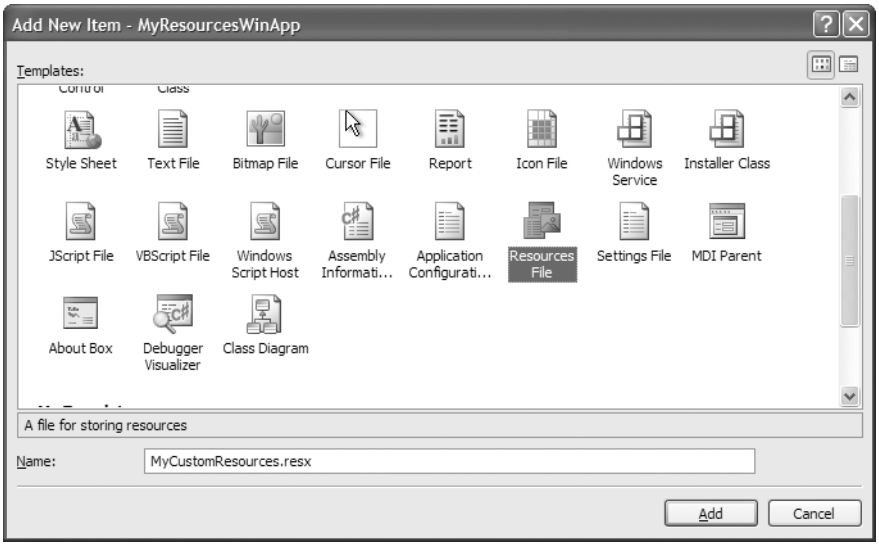

Figure 20-27. *Inserting a new* `*.resx` *file*

If you open your new `*.resx` file, a friendly GUI editor appears that allows you to insert string data, image files, sound clips, and other resources. The leftmost drop-down menu item allows you to select the type of resource you wish to add. First, add a new string resource named WelcomeString that is set to a message of your liking (see Figure 20-28).

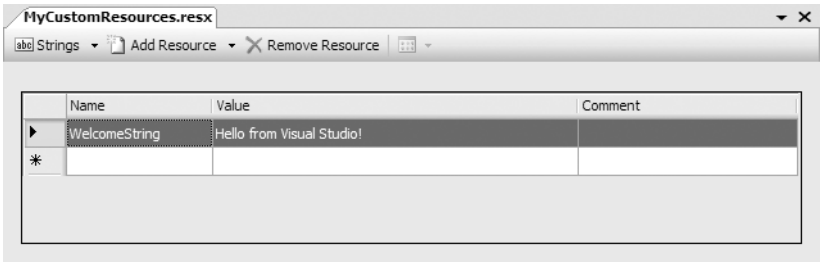

Figure 20-28. *Inserting new string resources with the* `*.resx` *editor*

Next, add the `happyDude.bmp` image file by selecting Images from the leftmost drop-down, choosing the Add Existing File option (see Figure 20-29), and navigating to the `happyDude.bmp` file.

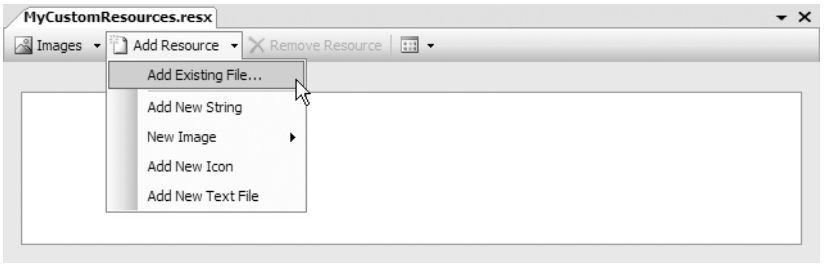

Figure 20-29. *Inserting new* `*.bmp` *resources with the* `*.resx` *editor*

At this point, you will find that the `*.bmp` file has been copied into your application directory. If you select the happyDude icon from the `*.resx` editor, you can now specify that this image should be embedded directly into the assembly (rather than linked as an external stand-alone file) by adjusting the Persistence property (see Figure 20-30).

Figure 20-30. *Embedding specified resources*

Furthermore, Solution Explorer now has a new folder named Resources that contains each item to be embedded into the assembly. As you would guess, if you open a given resource, Visual Studio 2005 launches an associated editor. In any case, if you were to now compile your application, the string and image data will be embedded within your assembly.

Programmatically Reading Resources

Now that you understand the process of embedding resources into your assembly (using `csc.exe` or Visual Studio 2005), you'll need to learn how to programmatically read them for use in your program using the `ResourceManager` type. To illustrate, add a `Button` and `PictureBox` widget on your Form type (see Figure 20-31).

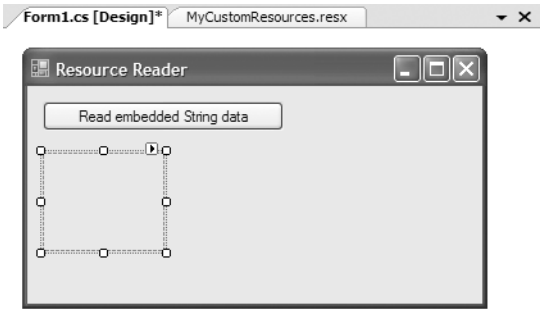

Figure 20-31. *The updated UI*

Next, handle the `Button`'s `Click` event. Update the event handler with the following code:

```
// Be sure to 'use' System.Resources and System.Reflection!
private void btnGetStringData_Click(object sender, EventArgs e)
{
    // Make a resource manager.
    ResourceManager rm =
        new ResourceManager("MyResourcesWinApp.MyCustomResources",
        Assembly.GetExecutingAssembly());
    // Get the embedded string (case sensitive!)
    MessageBox.Show(rm.GetString("WelcomeString"));
    // Get the embedded bitmap (case sensitive!)
    myPictureBox.Image = (Bitmap)rm.GetObject("HappyDude");
    // Clean up.
    rm.ReleaseAllResources();
}
```

Notice that the first constructor argument to the `ResourceManager` is the fully qualified name of your `*.resx` file (minus the file extension). The second parameter is a reference to the assembly that contains the embedded resource (which is the current assembly in this case). Once you have created the `ResourceManager`, you can call `GetString()` or `GetObject()` to extract the embedded data. If you were to run the application and click the button, you would find that the string data is displayed in the `MessageBox` and the image data has been extracted from the assembly and placed into the `PictureBox`.

■ **Source Code** The MyResourcesWinApp project is included under the Chapter 20 subdirectory.

Well, that wraps up our examination of GDI+ and the `System.Drawing` namespaces. If you are interested in exploring GDI+ further (including printing support), be sure to check out *GDI+ Programming in C# and VB .NET* by Nick Symmonds (Apress, 2002).

Summary

GDI+ is the name given to a number of related .NET namespaces, each of which is used to render graphic images to a `Control`-derived type. The bulk of this chapter was spent examining how to work with core GDI+ object types such as colors, fonts, graphics images, pens, and brushes in conjunction with the almighty `Graphics` type. Along the way, you examined some GDI+-centric details such as hit testing and how to drag and drop images.

This chapter wrapped up by examining the new .NET resource format. As shown, a `*.resx` denotes resources using a set of name/value pairs describes as XML. This file can be fed into the `resgen.exe` utility, resulting in a binary format (`*.resources`) that can then be embedded into a related assembly. Finally, the `ResourceManager` type provides a simple way to programmatically retrieve embedded resources at runtime.

CHAPTER 21

Programming with Windows Forms Controls

This chapter is concerned with providing a road map of the controls defined in the `System.Windows.Forms` namespace. Chapter 19 already gave you a chance to work with some controls mounted onto a main Form such as `MenuStrip`, `ToolStrip`, and `StatusStrip`. In this chapter, however, you will examine various types that tend to exist within the boundaries of a Form's client area (e.g., `Button`, `MaskedTextBox`, `WebBrowser`, `MonthCalendar`, `TreeView`, and the like). Once you look at the core UI widgets, you will then cover the process of building custom Windows Forms controls that integrate into the Visual Studio 2005 IDE.

The chapter then investigates the process of building custom dialog boxes and the role of *form inheritance*, which allows you to build hierarchies of related Form types. The chapter wraps up with a discussion of how to establish the *docking* and *anchoring* behaviors for your family of GUI types, and the role of the `FlowControlPanel` and `TableControlPanel` types supplied by .NET 2.0.

The World of Windows Forms Controls

The `System.Windows.Forms` namespace contains a number of types that represent common GUI widgets typically used to allow you to respond to user input in a Windows Forms application. Many of the controls you will work with on a day-to-day basis (such as `Button`, `TextBox`, and `Label`) are quite intuitive to work with. Other, more exotic controls and components (such as `TreeView`, `ErrorProvider`, and `TabControl`) require a bit more explanation.

As you learned in Chapter 19, the `System.Windows.Forms.Control` type is the base class for all derived widgets. Recall that `Control` provides the ability to process mouse and keyboard events, establish the physical dimensions and position of the widget using various properties (`Height`, `Width`, `Left`, `Right`, `Location`, etc.), manipulate background and foreground colors, establish the active font/cursor, and so forth. As well, the `Control` base type defines members that control a widget's anchoring and docking behaviors (explained at the conclusion of this chapter).

As you read through this chapter, remember that the widgets you examine here gain a good deal of their functionality from the `Control` base class. Thus, we'll focus (more or less) on the unique members of a given widget. Do understand that this chapter does not attempt to fully describe each and every member of each and every control (that is a task for the .NET Framework 2.0 SDK documentation). Rest assured, though, that once you complete this chapter, you will have no problem understanding the widgets I have not directly described.

■ **Note** Windows Forms provide a number of controls that allow you to display relational data (`DataGridView`, `BindingSource`, etc.). Some of these data-centric controls are examined in Chapter 22 during our discussion of ADO.NET.

Adding Controls to Forms by Hand

Regardless of which type of control you choose to place on a Form, you will follow an identical set of steps to do so. First of all, you must define member variables that represent the controls themselves. Next, inside the Form's constructor (or within a helper method called by the constructor), you'll configure the look and feel of each control using the exposed properties, methods, and events. Finally (and most important), once you've set the control to its initial state, you must add it into the Form's internal controls collection using the inherited `Controls` property. If you forget this final step, your widgets will *not* be visible at runtime.

To illustrate the process of adding controls to a Form, let's begin by building a `Form` type "wizard-free" using your text editor of choice and the C# command-line compiler. Create a new C# file named `ControlsByHand.cs` and code a new `MainWindow` class as so:

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace ControlsByHand
{
    class MainWindow : Form
    {
        // Form widget member variables.
        private TextBox firstNameBox = new TextBox();
        private Button btnShowControls = new Button();

        public MainWindow()
        {
            // Configure Form.
            this.Text = "Simple Controls";
            this.Width = 300;
            this.Height = 200;
            CenterToScreen();

            // Add a new textbox to the Form.
            firstNameBox.Text = "Hello";
            firstNameBox.Size = new Size(150, 50);
            firstNameBox.Location = new Point(10, 10);
            this.Controls.Add(firstNameBox);

            // Add a new button to the Form.
            btnShowControls.Text = "Click Me";
            btnShowControls.Size = new Size(90, 30);
            btnShowControls.Location = new Point(10, 70);
            btnShowControls.BackColor = Color.DodgerBlue;
            btnShowControls.Click +=
                new EventHandler(btnShowControls_Clicked);
            this.Controls.Add(btnShowControls);
        }
```

```
        // Handle Button's Click event.
        private void btnShowControls_Clicked(object sender, EventArgs e)
        {
            // Call ToString() on each control in the
            // Form's Controls collection
            string ctrlInfo= "";
            foreach (Control c in this.Controls)
            {
                ctrlInfo += string.Format("Control: {0}\n",
                    c.ToString());
            }
            MessageBox.Show(ctrlInfo, "Controls on Form");
        }
    }
}
```

Now, add a second class to the `ControlsByHand` namespace that implements the program's `Main()` method:

```
class Program
{
    public static void Main(string[] args)
    {
        Application.Run(new MainWindow());
    }
}
```

At this point, compile your C# file at the command line using the following command:

```
csc /target:winexe *.cs
```

When you run your program and click the Form's button, you will find a message box that lists each item on the Form (see Figure 21-1).

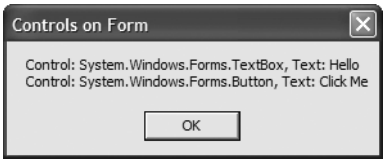

Figure 21-1. *Interacting with the Form's controls collection*

The Control.ControlCollection Type

While the process of adding a new widget to a Form is quite simple, I'd like to discuss the `Controls` property in a bit more detail. This property returns a reference to a nested class named `ControlCollection` defined within the `Control` class. The nested `ControlCollection` type maintains an entry for each widget placed on the Form. You can obtain a reference to this collection anytime you wish to "walk the list" of child widgets:

```
// Get access to the nested ControlCollection for this Form.
Control.ControlCollection coll = this.Controls;
```

Once you have a reference to this collection, you can manipulate its contents using the members shown in Table 21-1.

Table 21-1. `ControlCollection` *Members*

Member	Meaning in Life
`Add()` `AddRange()`	Used to insert a new `Control`-derived type (or array of types) in the collection
`Clear()`	Removes all entries in the collection
`Count`	Returns the number of items in the collection
`GetEnumerator()`	Returns the `IEnumerator` interface for this collection
`Remove()` `RemoveAt()`	Used to remove a control from the collection

Given that a Form maintains a collection of its controls, it is very simple under Windows Forms to dynamically create, remove, or otherwise manipulate visual elements. For example, assume you wish to disable all `Button` types on a given Form (or some such similar operation, such as change the background color of all `TextBoxes`). To do so, you can leverage the C# `is` keyword to determine who's who and change the state of the widgets accordingly:

```
private void DisableAllButtons()
{
    foreach (Control c in this.Controls)
    {
        if (c is Button)
            ((Button)c).Enabled = false;
    }
}
```

■ **Source Code** The ControlsByHand project is included under the Chapter 21 subdirectory.

Adding Controls to Forms Using Visual Studio 2005

Now that you understand the process of adding controls to a Form by hand, let's see how Visual Studio 2005 can automate the process. Create a new Windows Application project for testing purposes named whatever you choose. Similar to the process of designing menu, toolbar, or status bar controls, when you drop a control from the Toolbox onto the Forms designer, the IDE responds by automatically adding the correct member variable to the `*.Designer.cs` file. As well, when you design the look and feel of the widget using the IDE's Properties window, the related code changes are added to the `InitializeComponent()` member function (also located within the `*.Designer.cs` file).

■ **Note** Recall that the Properties window also allows you handle events for a given control when you click the lightning bolt icon. Simply select the widget from the drop-down list and type in the name of the method to be called for the events you are interested in responding to (or just double-click the event to generate a default event handler name).

Assume you have added a `TextBox` and `Button` type to the Forms designer. Notice that when you reposition a control on the designer, the Visual Studio 2005 IDE provides visual hints regarding the spacing and alignment of the current widget (see Figure 21-2).

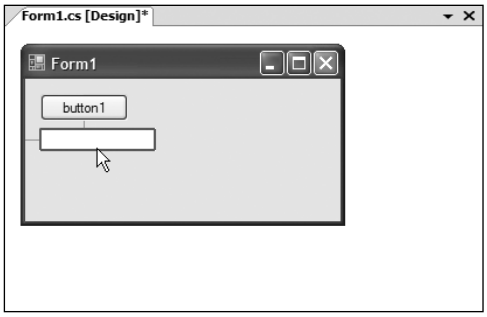

Figure 21-2. *Alignment and spacing hints*

Once you have placed the `Button` and `TextBox` on the designer, examine the code generated in the `InitializeComponent()` method. Here you will find that the types have been `new`-ed and inserted into the Form's `ControlCollection` automatically (in addition to any settings you may have made using the Properties window):

```
private void InitializeComponent()
{
    this.btnMyButton = new System.Windows.Forms.Button();
    this.txtMyTextBox = new System.Windows.Forms.TextBox();
...
    // MainWindow
    //
   ...
    this.Controls.Add(this.txtMyTextBox);
    this.Controls.Add(this.btnMyButton);
...
}
```

As you can see, a tool such as Visual Studio 2005 simply saves you some typing time (and helps you avoid hand cramps). Although `InitializeComponent()` is maintained on your behalf, do understand that you are free to configure a given control directly in code anywhere you see necessary (constructors, event handlers, helper functions, etc.). The role of `InitializeComponent()` is simply to establish the initial state of your UI elements. If you want to keep your life simple, I suggest allowing Visual Studio 2005 to maintain `InitializeComponent()` on your behalf, given that the designers may ignore or overwrite edits you make within this method.

Working with the Basic Controls

The `System.Windows.Forms` namespace defines numerous "basic controls" that are commonplace to any windowing framework (buttons, labels, text boxes, check boxes, etc.). Although I would guess you are familiar with the basic operations of such types, let's examine some of the more interesting aspects of the following basic UI elements:

- `Label`, `TextBox`, and `MaskedTextBox`
- `Button`
- `CheckBox`, `RadioButton`, and `GroupBox`
- `CheckedListBox`, `ListBox`, and `ComboBox`

Once you have become comfortable with these basic `Control`-derived types, we will turn our attention to more exotic widgets such as `MonthCalendar`, `TabControl`, `TrackBar`, `WebBrowser`, and so forth.

Fun with Labels

The `Label` control is capable of holding read-only information (text or image based) that explains the role of the other controls to help the user along. Assume you have created a new Visual Studio 2005 Windows Forms project named LabelsAndTextBoxes. Define a method called `CreateLabelControl` in your `Form`-derived type that creates and configures a `Label` type, and then adds it to the Form's controls collection:

```
private void CreateLabelControl()
{
    // Create and configure a Label.
    Label lblInstructions = new Label();
    lblInstructions.Name = "lblInstructions";
    lblInstructions.Text = "Please enter values in all the text boxes";
    lblInstructions.Font = new Font("Times New Roman", 9.75F, FontStyle.Bold);
    lblInstructions.AutoSize = true;
    lblInstructions.Location = new System.Drawing.Point(16, 13);
    lblInstructions.Size = new System.Drawing.Size(240, 16);

    // Add to Form's controls collection.
    Controls.Add(lblInstructions);
}
```

If you were to call this helper function within your Form's constructor, you would find your prompt displayed in the upper portion of the main window:

```
public MainWindow()
{
    InitializeComponent();
    CreateLabelControl();
    CenterToScreen();
}
```

Unlike most other widgets, `Label` controls cannot receive focus via a Tab keypress. However, under .NET 2.0, it is now possible to create *mnemonic keys* for any `Label` by setting the `UseMnemonic` property to true (which happens to be the default setting). Once you have done so, a `Label`'s `Text` property can define a shortcut key (via the ampersand symbol, &), which is used to tab to the control that follows it in the tab order.

■ **Note** You'll learn more about configuring tab order later in this chapter, but for the time being, understand that a control's tab order is established via the `TabIndex` property. By default, a control's `TabIndex` is set based on the order in which it was added to the Forms designer. Thus, if you add a `Label` followed by a `TextBox`, the `Label` is set to `TabIndex 0` while the `TextBox` is set to `TabIndex 1`.

To illustrate, let's now leverage the Forms designer to build a UI containing a set of three `Label`s and three `TextBox`es (be sure to leave room on the upper part of the Form to display the `Label` dynamically created in the `CreateLabelControl()` method). In Figure 21-3, note that each label has an underlined letter that was identified using the & character in the value assigned to the `Text` property (as you might know, &-specified characters allow the user to activate an item using the Alt+<*some key*> keystroke).

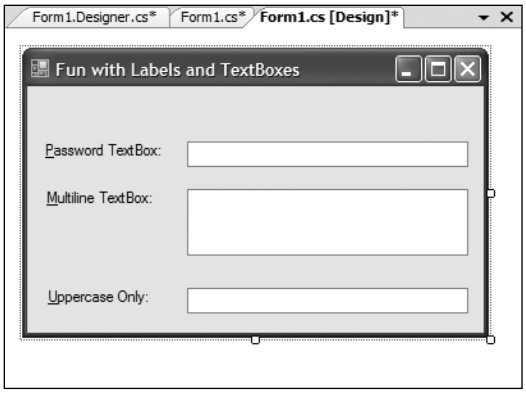

Figure 21-3. *Assigning mnemonics to* `Label` *controls*

If you now run your project, you will be able to tab between each `TextBox` using the Alt+p, Alt+m, or Alt+u keystrokes.

Fun with TextBoxes

Unlike the `Label` control, the `TextBox` control is typically not read-only (although it could be if you set the `ReadOnly` property to true), and it is commonly used to allow the user to enter textual data for processing. The `TextBox` type can be configured to hold a single line or multiple lines of text, it can be configured with a *password character* (such as an asterisk, *), and it may support scroll bars in the case of multiline text boxes. In addition to the behavior inherited by its base classes, `TextBox` defines a few particular properties of interest (see Table 21-2).

Table 21-2. `TextBox` *Properties*

Property	Meaning in Life
`AcceptsReturn`	Gets or sets a value indicating whether pressing Enter in a multiline `TextBox` control creates a new line of text in the control or activates the "default button" for the Form
`CharacterCasing`	Gets or sets whether the `TextBox` control modifies the case of characters as they are typed
`PasswordChar`	Gets or sets the character used to mask characters in a single-line `TextBox` control used to enter passwords
`ScrollBars`	Gets or sets which scroll bars should appear in a multiline `TextBox` control
`TextAlign`	Gets or sets how text is aligned in a `TextBox` control, using the `HorizontalAlignment` enumeration

To illustrate some aspects of the `TextBox`, let's configure the three `TextBox` controls on the current Form. The first `TextBox` (named `txtPassword`) should be configured as a password text box, meaning the characters typed into the `TextBox` should not be directly visible, but are instead masked with a predefined password character via the `PasswordChar` property.

The second `TextBox` (named `txtMultiline`) will be a multiline text area that has been configured to accept return key processing and displays a vertical scroll bar when the text entered exceeds the space of the `TextBox` area. Finally, the third `TextBox` (named `txtUppercase`) will be configured to translate the entered character data into uppercase.

Configure each `TextBox` accordingly via the Properties window and use the following (partial) `InitializeComponent()` implementation as a guide:

```
private void InitializeComponent()
{
...
    // txtPassword
    //
    this.txtPassword.PasswordChar = '*';
...
    // txtMultiline
    //
    this.txtMultiline.Multiline = true;
    this.txtMultiline.ScrollBars = System.Windows.Forms.ScrollBars.Vertical;
...
    // txtUpperCase
    //
    this.txtUpperCase.CharacterCasing =
        System.Windows.Forms.CharacterCasing.Upper;
...
}
```

Notice that the `ScrollBars` property is assigned a value from the `ScrollBars` enumeration, which defines the following values:

```
public enum System.Windows.Forms.ScrollBars
{
    Both, Horizontal, None, Vertical
}
```

The `CharacterCasing` property works in conjunction with the `CharacterCasing` enum, which is defined as so:

```
public enum System.Windows.Forms.CharacterCasing
{
    Normal, Upper, Lower
}
```

Now assume you have placed a `Button` on the Form (named `btnDisplayData`) and added an event handler for the `Button`'s `Click` event. The implementation of this method simply displays the value in each `TextBox` within a message box:

```
private void btnDisplayData_Click(object sender, EventArgs e)
{
    // Get data from all the text boxes.
    string textBoxData = "";
    textBoxData += string.Format("MultiLine:  {0}\n", txtMultiline.Text);
    textBoxData += string.Format("\nPassword:  {0}\n", txtPassword.Text);
    textBoxData += string.Format("\nUppercase:  {0}\n", txtUpperCase.Text);

    // Display all the data.
    MessageBox.Show(textBoxData, "Here is the data in your TextBoxes");
}
```

Figure 21-4 shows one possible input session (note that you need to hold down the Alt key to see the label mnemonics).

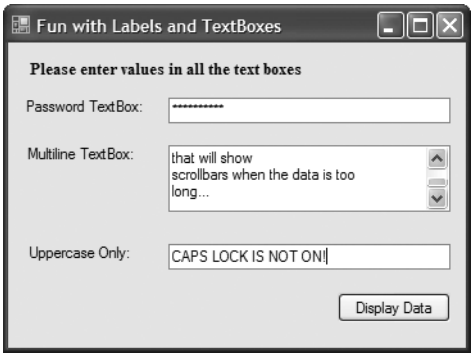

Figure 21-4. *The many faces of the* `TextBox` *type*

Figure 21-5 shows the result of clicking the `Button` type.

Figure 21-5. *Extracting values from* `TextBox` *types*

Fun with MaskedTextBoxes

As of .NET 2.0, we now have a *masked* text box that allows us to specify a valid sequence of characters that will be accepted by the input area (Social Security number, phone number with area code, zip code, or whatnot). The mask to test against (termed a *mask expression*) is established using specific tokens embedded into a string literal. Once you have created a mask expression, this value is assigned to the `Mask` property. Table 21-3 documents some (but not all) valid masking tokens.

Table 21-3. *Mask Tokens of* `MaskedTextBox`

Mask Token	Meaning in Life
`0`	Represents a mandatory digit with the value 0–9
`9`	Represents an optional digit or a space
`L`	Required letter (in uppercase or lowercase), A–Z
`?`	Optional letter (in uppercase or lowercase), A–Z
`,`	Represents a thousands separator placeholder
`:`	Represents a time placeholder
`/`	Represents a date placeholder
`$`	Represents a currency symbol

■ **Note** The characters understood by the `MaskedTextBox` *do not* directly map to the syntax of regular expressions. Although .NET provides namespaces to work with proper regular expressions (`System.Text.RegularExpressions` and `System.Web.RegularExpressions`), the `MaskedTextBox` uses syntax based on the legacy `MaskedEdit` VB6 COM control.

In addition to the `Mask` property, the `MaskedTextBox` has additional members that determine how this control should respond if the user enters incorrect data. For example, `BeepOnError` will cause the control to (obviously) issue a beep when the mask is not honored, and it prevents the illegal character from being processed.

To illustrate the use of the `MaskedTextBox`, add an additional `Label` and `MaskedTextBox` to your current Form. Although you are free to build a mask pattern directly in code, the Properties window provides an ellipsis button for the `Mask` property that will launch a dialog box with a number of predefined masks (see Figure 21-6).

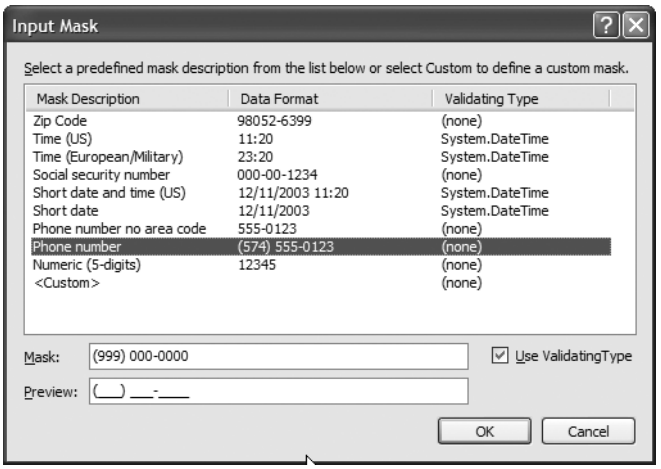

Figure 21-6. *Predefined mask values of the* `Mask` *property*

Find a masking pattern (such as Phone number), enable the `BeepOnError` property, and take your program out for another test run. You should find that you are unable to enter any alphabetic characters (in the case of the Phone number mask).

As you would expect, the `MaskedTextBox` will send out various events during its lifetime, one of which is `MaskInputRejected`, which is fired when the end user enters erroneous input. Handle this event using the Properties window and notice that the second incoming argument of the generated event handler is of type `MaskInputRejectedEventArgs`. This type has a property named `RejectionHint` that contains a brief description of the input error. For testing purposes, simply display the error on the Form's caption.

```
private void txtMaskedTextBox_MaskInputRejected(object sender,
    MaskInputRejectedEventArgs e)
{
    this.Text = string.Format("Error: {0}", e.RejectionHint);
}
```

To ensure that this error is not displayed when the user enters valid data, handle the `KeyDown` event on the `MaskedTextBox` and implement the event handler to reset the Form's caption to a default value:

```
private void txtMaskedTextBox_KeyDown(object sender, KeyEventArgs e)
{
    this.Text = "Fun with Labels and TextBoxes";
}
```

■ **Source Code** The LabelsAndTextBoxes project is included under the Chapter 21 subdirectory.

Fun with Buttons

The role of the `System.Windows.Forms.Button` type is to provide a vehicle for user confirmation, typically in response to a mouse click or keypress. The `Button` class immediately derives from an abstract type named `ButtonBase`, which provides a number of key behaviors for all derived types (such as `CheckBox`, `RadioButton`, and `Button`). Table 21-4 describes some (but by no means all) of the core properties of `ButtonBase`.

Table 21-4. `ButtonBase` *Properties*

Property	Meaning in Life
`FlatStyle`	Gets or sets the flat style appearance of the `Button` control, using members of the `FlatStyle` enumeration.
`Image`	Configures which (optional) image is displayed somewhere within the bounds of a `ButtonBase`-derived type. Recall that the `Control` class also defines a `BackgroundImage` property, which is used to render an image over the entire surface area of a widget.
`ImageAlign`	Sets the alignment of the image on the `Button` control, using the `ContentAlignment` enumeration.
`TextAlign`	Gets or sets the alignment of the text on the `Button` control, using the `ContentAlignment` enumeration.

The `TextAlign` property of `ButtonBase` makes it extremely simple to position text at just about any location. To set the position of your `Button`'s caption, use the `ContentAlignment` enumeration (defined in the `System.Drawing` namespace). As you will see, this same enumeration can be used to place an optional image on the `Button` type:

```
public enum System.Drawing.ContentAlignment
{
    BottomCenter, BottomLeft, BottomRight,
    MiddleCenter, MiddleLeft, MiddleRight,
    TopCenter, TopLeft, TopRight
}
```

`FlatStyle` is another property of interest. It is used to control the general look and feel of the `Button` control, and it can be assigned any value from the `FlatStyle` enumeration (defined in the `System.Windows.Forms` namespace):

```
public enum System.Windows.Forms.FlatStyle
{
    Flat, Popup, Standard, System
}
```

To illustrate working with the Button type, create a new Windows Forms application named Buttons. On the Forms designer, add three Button types (named btnFlat, btnPopup, and btnStandard) and set each Button's FlatStyle property value accordingly (e.g., FlatStyle.Flat, FlatStyle.Popup, or FlatStyle.Standard). As well, set the Text value of each Button to a fitting value and handle the Click event for the btnStandard Button. As you will see in just a moment, when the user clicks this button, you will reposition the button's text using the TextAlign property.

Now, add a final fourth Button (named btnImage) that supports a background image (set via the BackgroundImage property) and a small bull's-eye icon (set via the Image property), which will also be dynamically relocated when the btnStandard Button is clicked. Feel free to use any image files to assign to the BackgroundImage and Image properties, but do note that the downloadable source code contains the images used here.

Given that the designer has authored all the necessary UI prep code within InitializeComponent(), the remaining code makes use of the ContentAlignment enumeration to reposition the location of the text on btnStandard and the icon on btnImage. In the following code, notice that you are calling the static Enum.GetValues() method to obtain the list of names from the ContentAlignment enumeration:

```
partial class MainWindow : Form
{
    // Used to hold the current text alignment value.
    ContentAlignment currAlignment = ContentAlignment.MiddleCenter;
    int currEnumPos = 0;

    public MainWindow()
    {
        InitializeComponent();
        CenterToScreen();
    }

    private void btnStandard_Click (object sender, EventArgs e)
    {
        // Get all possible values of the ContentAlignment enum.
        Array values = Enum.GetValues(currAlignment.GetType());

        // Bump the current position in the enum.
        // and check for wraparound.
        currEnumPos++;
        if(currEnumPos >= values.Length)
            currEnumPos = 0;

        // Bump the current enum value.
        currAlignment = (ContentAlignment)Enum.Parse(currAlignment.GetType(),
            values.GetValue(currEnumPos).ToString());

        // Paint enum value and align text on btnStandard.
        btnStandard.TextAlign = currAlignment;
            btnStandard.Text = currAlignment.ToString();

        // Now assign the location of the icon on btnImage
        btnImage.ImageAlign = currAlignment;
    }
}
```

Now run your program. As you click the middle button, you will see its text is set to the current name and position of the currAlignment member variable. As well, the icon within the btnImage is repositioned based on the same value. Figure 21-7 shows the output.

Figure 21-7. *The many faces of the* `Button` *type*

■ **Source Code** The Buttons project is included under the Chapter 21 directory.

Fun with CheckBoxes, RadioButtons, and GroupBoxes

The `System.Windows.Forms` namespace defines a number of other types that extend `ButtonBase`, specifically `CheckBox` (which can support up to three possible states) and `RadioButton` (which can be either selected or not selected). Like the `Button`, these types also receive most of their functionality from the `Control` base class. However, each class defines some additional functionality. First, consider the core properties of the `CheckBox` widget described in Table 21-5.

Table 21-5. `CheckBox` *Properties*

Property	Meaning in Life
`Appearance`	Configures the appearance of a `CheckBox` control, using the `Appearance` enumeration.
`AutoCheck`	Gets or sets a value indicating if the `Checked` or `CheckState` value and the `CheckBox`'s appearance are automatically changed when it is clicked.
`CheckAlign`	Gets or sets the horizontal and vertical alignment of a `CheckBox` on a `CheckBox` control, using the `ContentAlignment` enumeration (much like the `Button` type).
`Checked`	Returns a Boolean value representing the state of the `CheckBox` (checked or unchecked). If the `ThreeState` property is set to true, the `Checked` property returns true for either checked or indeterminately checked values.
`CheckState`	Gets or sets a value indicating whether the `CheckBox` is checked, using a `CheckState` enumeration rather than a Boolean value.
`ThreeState`	Configures whether the `CheckBox` supports three states of selection (as specified by the `CheckState` enumeration) rather than two.

The `RadioButton` type requires little comment, given that it is (more or less) just a slightly redesigned `CheckBox`. In fact, the members of a `RadioButton` are almost identical to those of the `CheckBox` type. The only notable difference is the `CheckedChanged` event, which (not surprisingly) is fired when the `Checked` value changes. Also, the `RadioButton` type does not support the `ThreeState` property, as a `RadioButton` must be on or off.

Typically, multiple `RadioButton` objects are logically and physically grouped together to function as a whole. For example, if you have a set of four `RadioButton` types representing the color choice of a given automobile, you may wish to ensure that only one of the four types can be checked at a time. Rather than writing code programmatically to do so, simply use the `GroupBox` control to ensure all `RadioButtons` are mutually exclusive.

To illustrate working with the `CheckBox`, `RadioButton`, and `GroupBox` types, let's create a new Windows Forms application named CarConfig, which you will extend over the next few sections. The main Form allows users to enter (and confirm) information about a new vehicle they intend to purchase. The order summary is displayed in a `Label` type once the Confirm Order button has been clicked. Figure 21-8 shows the initial UI.

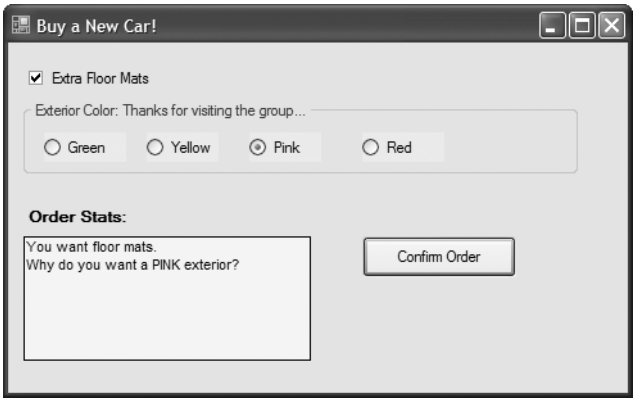

Figure 21-8. *The initial UI of the CarConfig Form*

Assuming you have leveraged the Forms designer to build your UI, you will now have numerous member variables representing each GUI widget. As well, the `InitializeComponent()` method will be updated accordingly. The first point of interest is the construction of the `CheckBox` type. As with any `Control`-derived type, once the look and feel has been established, it must be inserted into the Form's internal collection of controls:

```
private void InitializeComponent()
{
...
    // checkFloorMats
    //
    this.checkFloorMats.Name = "checkFloorMats";
    this.checkFloorMats.TabIndex = 0;
    this.checkFloorMats.Text = "Extra Floor Mats";
...
    this.Controls.Add(this.checkFloorMats);
}
```

Next, you have the configuration of the `GroupBox` and its contained `RadioButton` types. When you wish to place a control under the ownership of a `GroupBox`, you want to add each item to the `GroupBox`'s `Controls` collection (in the same way you add widgets to the Form's `Controls` collection). To make things a bit more interesting, use the Properties window to handle the `Enter` and `Leave` events sent by the `GroupBox` object, as shown here:

```
private void InitializeComponent()
{
...
    // radioRed
    //
    this.radioRed.Name = "radioRed";
    this.radioRed.Size = new System.Drawing.Size(64, 23);
    this.radioRed.Text = "Red";
    //
    // groupBoxColor
    //
...
    this.groupBoxColor.Controls.Add(this.radioRed);
    this.groupBoxColor.Text = "Exterior Color";
    this.groupBoxColor.Enter += new System.EventHandler(this.groupBoxColor_Enter);
    this.groupBoxColor.Leave += new System.EventHandler(this.groupBoxColor_Leave);
...
}
```

Understand, of course, that you do not need to capture the `Enter` or `Leave` event for a `GroupBox`. However, to illustrate, the event handlers update the caption text of the `GroupBox` as shown here:

```
// Figure out when the focus is in your group.
private void groupBoxColor_Leave(object sender, EventArgs e)
{
    groupBoxColor.Text = "Exterior Color: Thanks for visiting the group...";
}

private void groupBoxColor_Enter(object sender, EventArgs e)
{
    groupBoxColor.Text = "Exterior Color: You are in the group...";
}
```

The final GUI widgets on this Form (the `Label` and `Button` types) will also be configured and inserted in the Form's `Controls` collection via `InitializeComponent()`. The `Label` is used to display the order confirmation, which is formatted in the `Click` event handler of the order `Button`, as shown here:

```
private void btnOrder_Click (object sender, System.EventArgs e)
{
     // Build a string to display information.
     string orderInfo = "";
     if(checkFloorMats.Checked)
          orderInfo += "You want floor mats.\n";
     if(radioRed.Checked)
          orderInfo += "You want a red exterior.\n";
     if(radioYellow.Checked)
          orderInfo += "You want a yellow exterior.\n";
     if(radioGreen.Checked)
          orderInfo += "You want a green exterior.\n";
     if(radioPink.Checked)
          orderInfo += "Why do you want a PINK exterior?\n";
     // Send this string to the Label.
     infoLabel.Text = orderInfo;
}
```

Notice that both the `CheckBox` and `RadioButton` support the `Checked` property, which allows you to investigate the state of the widget. Finally, recall that if you have configured a tri-state `CheckBox`, you will need to check the state of the widget using the `CheckState` property.

Fun with CheckedListBoxes

Now that you have explored the basic `Button`-centric widgets, let's move on to the set of list selection–centric types, specifically `CheckedListBox`, `ListBox`, and `ComboBox`. The `CheckedListBox` widget allows you to group related `CheckBox` options in a scrollable list control. Assume you have added such a control to your CarConfig Form that allows users to configure a number of options regarding an automobile's sound system (see Figure 21-9).

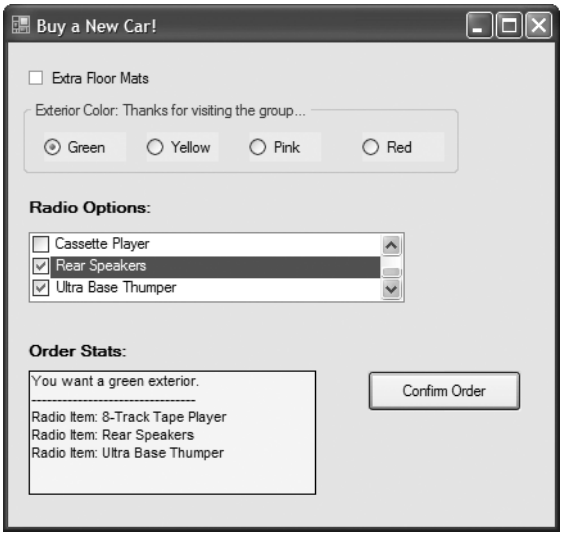

Figure 21-9. *The* `CheckedListBox` *type*

To insert new items in a `CheckedListBox`, call `Add()` for each item, or use the `AddRange()` method and send in an array of objects (strings, to be exact) that represent the full set of checkable items. Be aware that you can fill any of the list types at design time using the `Items` property located on the Properties window (just click the ellipsis button and type the string values). Here is the relevant code within `InitializeComponent()` that configures the `CheckedListBox`:

```
private void InitializeComponent()
{
...
    // checkedBoxRadioOptions
    //
    this.checkedBoxRadioOptions.Items.AddRange(new object[] {
        "Front Speakers", "8-Track Tape Player",
        "CD Player", "Cassette Player",
        "Rear Speakers", "Ultra Base Thumper"});
...
    this.Controls.Add (this.checkedBoxRadioOptions);
}
```

Now update the logic behind the `Click` event for the Confirm Order button. Ask the `CheckedListBox` which of its items are currently selected and add them to the `orderInfo` string. Here are the relevant code updates:

```
private void btnOrder_Click (object sender, EventArgs e)
{
    // Build a string to display information.
```

```
    string orderInfo = "";
    ...
    orderInfo += "--------------------------------\n";

    // For each item in the CheckedListBox:
    for(int i = 0; i < checkedBoxRadioOptions.Items.Count; i++)
    {
        // Is the current item checked?
        if(checkedBoxRadioOptions.GetItemChecked(i))
        {
            // Get text of checked item and append to orderinfo string.
            orderInfo += "Radio Item: ";
            orderInfo += checkedBoxRadioOptions.Items[i].ToString();
            orderInfo += "\n";
        }
    }
    ...
}
```

The final note regarding the `CheckedListBox` type is that it supports the use of multiple columns through the inherited `MultiColumn` property. Thus, if you make the following update:

```
checkedBoxRadioOptions.MultiColumn = true;
```

you see the multicolumn `CheckedListBox` shown in Figure 21-10.

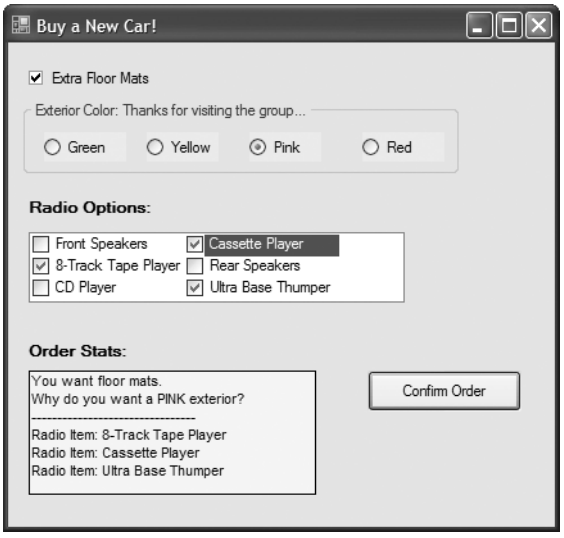

Figure 21-10. *Multicolumn* `CheckedListBox` *type*

Fun with ListBoxes

As mentioned earlier, the `CheckedListBox` type inherits most of its functionality from the `ListBox` type. To illustrate using the `ListBox` type, let's add another feature to the current CarConfig application: the ability to select the make (BMW, Yugo, etc.) of the automobile. Figure 21-11 shows the desired UI.

Figure 21-11. *The* `ListBox` *type*

As always, begin by creating a member variable to manipulate your type (in this case, a `ListBox` type). Next, configure the look and feel using the following snapshot from `InitializeComponent()`:

```
private void InitializeComponent()
{
...
    // carMakeList
    //
    this.carMakeList.Items.AddRange(new object[] {
        "BMW", "Caravan", "Ford", "Grand Am",
        "Jeep", "Jetta", "Saab", "Viper", "Yugo"});
...
    this.Controls.Add (this.carMakeList);
}
```

The update to the `btnOrder_Click()` event handler is also simple:

```
private void btnOrder_Click (object sender, EventArgs e)
{
     // Build a string to display information.
     string orderInfo = "";
     ...
     // Get the currently selected item (not index of the item).
     if(carMakeList.SelectedItem != null)
          orderInfo += "Make: " + carMakeList.SelectedItem + "\n";
     ...
}
```

Fun with ComboBoxes

Like a `ListBox`, a `ComboBox` allows users to make a selection from a well-defined set of possibilities. However, the `ComboBox` type is unique in that users can also insert additional items. Recall that `ComboBox` derives from `ListBox` (which then derives from `Control`). To illustrate its use, add yet another GUI

widget to the CarConfig Form that allows a user to enter the name of a preferred salesperson. If the salesperson in question is not on the list, the user can enter a custom name. One possible UI update is shown in Figure 21-12 (feel free to add your own salesperson monikers).

Figure 21-12. *The* `ComboBox` *type*

This modification begins with configuring the `ComboBox` itself. As you can see here, the logic looks identical to that for the `ListBox`:

```
private void InitializeComponent()
{
...
    // comboSalesPerson
    //
    this.comboSalesPerson.Items.AddRange(new object[] {
        "Baby Ry-Ry", "Dan \'the Machine\'",
        "Danny Boy", "Tommy Boy"});
...
    this.Controls.Add (this.comboSalesPerson);
}
```

The update to the `btnOrder_Click()` event handler is again simple, as shown here:

```
private void btnOrder_Click (object sender, EventArgs e)
{
     // Build a string to display information.
     string orderInfo = "";
     ...
     // Use the Text property to figure out the user's salesperson.
     if(comboSalesPerson.Text != "")
          orderInfo += "Sales Person: " + comboSalesPerson.Text + "\n";
     else
          orderInfo += "You did not select a sales person!" + "\n";
     ...
}
```

Configuring the Tab Order

Now that you have created a somewhat interesting Form, let's formalize the issue of tab order. As you may know, when a Form contains multiple GUI widgets, users expect to be able to shift focus using the Tab key. Configuring the tab order for your set of controls requires that you understand two key properties: `TabStop` and `TabIndex`.

The `TabStop` property can be set to `true` or `false`, based on whether or not you wish this GUI item to be reachable using the Tab key. Assuming the `TabStop` property has been set to true for a given widget, the `TabOrder` property is then set to establish its order of activation in the tabbing sequence (which is zero based). Consider this example:

```
// Configure tabbing properties.
radioRed.TabIndex = 2;
radioRed.TabStop = true;
```

The Tab Order Wizard

The Visual Studio 2005 IDE supplies a Tab Order Wizard, which you access by choosing View ➤ Tab Order (be aware that you will not find this menu option unless the Forms designer is active). Once activated, your design-time Form displays the current `TabIndex` value for each widget. To change these values, click each item in the order you choose (see Figure 21-13).

To exit the Tab Order Wizard, simply press the Esc key.

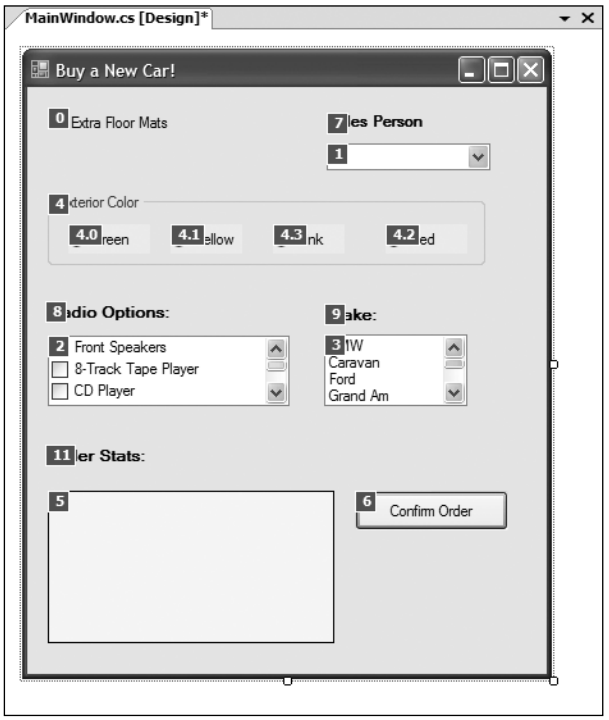

Figure 21-13. *The Tab Order Wizard*

Setting the Form's Default Input Button

Many user-input forms (especially dialog boxes) have a particular `Button` that will automatically respond to the user pressing the Enter key. For the current Form, if you wish to ensure that when the user presses the Enter key, the `Click` event handler for `btnOrder` is invoked, simply set the Form's `AcceptButton` property as so:

```
// When the Enter key is pressed, it is as if
// the user clicked the btnOrder button.
this.AcceptButton = btnOrder;
```

■ **Note** Some Forms require the ability to simulate clicking the Form's Cancel button when the user presses the Esc key. This can be done by assigning the `CancelButton` property to the `Button` object representing the Cancel button.

Working with More Exotic Controls

At this point, you have seen how to work most of the basic Windows Forms controls (`Labels`, `TextBoxes`, and the like). The next task is to examine some GUI widgets, which are a bit more high-powered in their functionality. Thankfully, just because a control may seem "more exotic" does not mean it is hard to work with, only that it requires a bit more elaboration from the outset. Over the next several pages, we will examine the following GUI elements:

- `MonthCalendar`
- `ToolTip`
- `TabControl`
- `TrackBar`
- `Panel`
- `UpDown` controls
- `ErrorProvider`
- `TreeView`
- `WebBrower`

To begin, let's wrap up the CarConfig project by examining the `MonthCalendar` and `ToolTip` controls.

Fun with MonthCalendars

The `System.Windows.Forms` namespace provides an extremely useful widget, the `MonthCalendar` control, that allows the user to select a date (or range of dates) using a friendly UI. To showcase this new control, update the existing CarConfig application to allow the user to enter in the new vehicle's delivery date. Figure 21-14 shows the updated (and slightly rearranged) Form.

Figure 21-14. *The* `MonthCalendar` *type*

Although the `MonthCalendar` control offers a fair bit of functionality, it is very simple to programmatically capture the range of dates selected by the user. The default behavior of this type is to always select (and mark) today's date automatically. To obtain the currently selected date programmatically, you can update the `Click` event handler for the order `Button`, as shown here:

```
private void btnOrder_Click (object sender, EventArgs e)
{
    // Build a string to display information.
    string orderInfo = "";
    ...
    // Get ship date.
    DateTime d = monthCalendar.SelectionStart;
    string dateStr = string.Format("{0}/{1}/{2}", d.Month, d.Day, d.Year);
    orderInfo += "Car will be sent: " + dateStr;
    ...
}
```

Notice that you can ask the `MonthCalendar` control for the currently selected date by using the `SelectionStart` property. This property returns a `DateTime` reference, which you store in a local variable. Using a handful of properties of the `DateTime` type, you can extract the information you need in a custom format.

At this point, I assume the user will specify exactly one day on which to deliver the new automobile. However, what if you want to allow the user to select a range of possible shipping dates? In that case, all the user needs to do is drag the cursor across the range of possible shipping dates. You already have seen that you can obtain the start of the selection using the `SelectionStart` property. The end of the selection can be determined using the `SelectionEnd` property. Here is the code update:

```
private void btnOrder_Click (object sender, EventArgs e)
{
     // Build a string to display information.
     string orderInfo = "";
     ...
     // Get ship date range....
     DateTime startD = monthCalendar.SelectionStart;
     DateTime endD = monthCalendar.SelectionEnd;
     string dateStartStr =
         string.Format("{0}/{1}/{2}", startD.Month, startD.Day, startD.Year);
     string dateEndStr =
        string.Format("{0}/{1}/{2}", endD.Month, endD.Day, endD.Year);

     // The DateTime type supports overloaded operators!
     if(dateStartStr != dateEndStr)
     {
          orderInfo += "Car will be sent between "
          + dateStartStr + " and\ n" + dateEndStr;
     }
     else   // They picked a single date.
          orderInfo += "Car will be sent on "  + dateStartStr;
     ...
}
```

■ **Note** The Windows Forms toolkit also provides the `DateTimePicker` control, which exposes a `MonthCalendar` from a `DropDown` control.

Fun with ToolTips

As far as the CarConfig Form is concerned, we have one final point of interest. Most modern UIs support *tool tips*. In the `System.Windows.Forms` namespace, the `ToolTip` type represents this functionality. These widgets are simply small floating windows that display a helpful message when the cursor hovers over a given item.

To illustrate, add a tool tip to the CarConfig's `Calendar` type. Begin by dragging a new `ToolTip` control from the Toolbox onto your Forms designer, and rename it to `calendarTip`. Using the Properties window, you are able to establish the overall look and feel of the `ToolTip` widget, for example:

```
private void InitializeComponent()
{
...
    // calendarTip
    //
    this.calendarTip.IsBalloon = true;
    this.calendarTip.ShowAlways = true;
    this.calendarTip.ToolTipIcon = System.Windows.Forms.ToolTipIcon.Info;
...
}
```

To associate a `ToolTip` with a given control, select the control that should activate the `ToolTip` and set the "ToolTip on" property (see Figure 21-15).

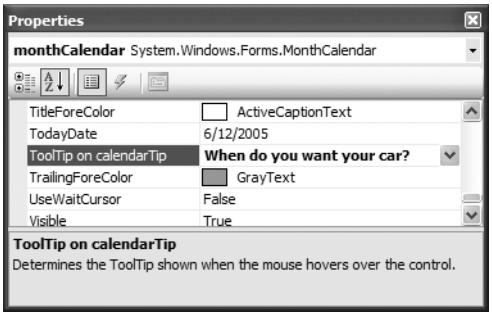

Figure 21-15. *Associating a* `ToolTip` *to a given widget*

At this point, the CarConfig project is complete. Figure 21-16 shows the `ToolTip` in action.

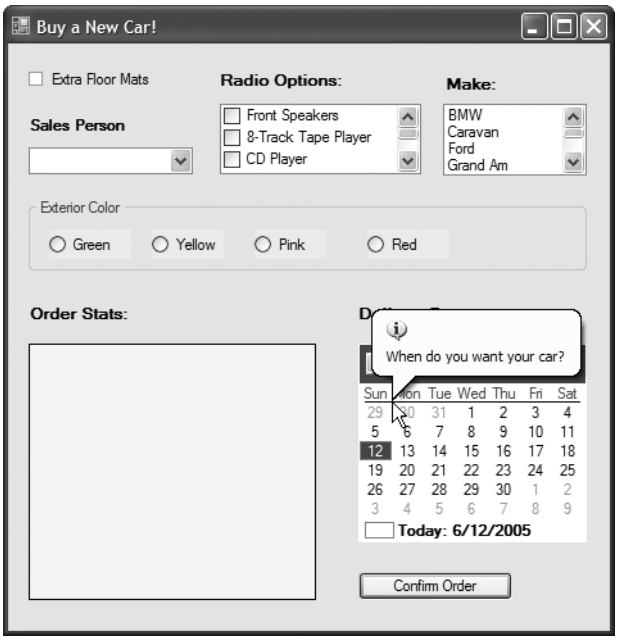

Figure 21-16. *The* `ToolTip` *in action*

■ **Source Code** The CarConfig project is included under the Chapter 21 directory.

Fun with TabControls

To illustrate the remaining "exotic" controls, you will build a new Form that maintains a `TabControl`. As you may know, `TabControls` allow you to selectively hide or show pages of related GUI content via clicking a given tab. To begin, create a new Windows Forms application named `ExoticControls` and rename your initial Form to `MainWindow`.

Next, add a `TabControl` onto the Forms designer and, using the Properties window, open the page editor via the `TabPages` collection (just click the ellipsis button on the Properties window). A dialog configuration tool displays. Add a total of six pages, setting each page's `Text` and `Name` properties based on the completed `TabControl` shown in Figure 21-17.

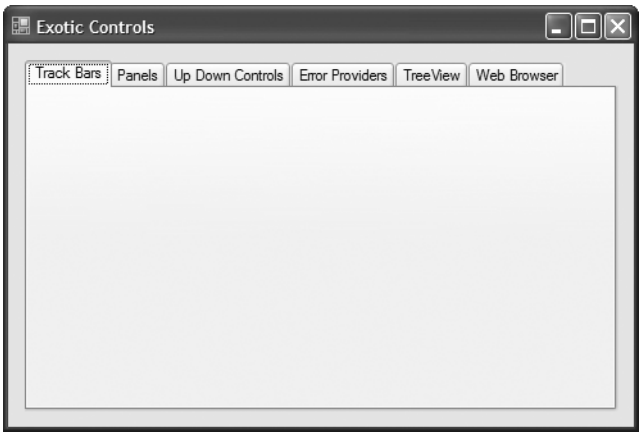

Figure 21-17. *A multipage* `TabControl`

As you are designing your `TabControl`, be aware that each page is represented by a `TabPage` object, which is inserted into the `TabControl`'s internal collection of pages. Once the `TabControl` has been configured, this object (like any other GUI widget within a Form) is inserted into the Form's `Controls` collection. Consider the following partial `InitializeComponent()` method:

```
private void InitializeComponent()
{
...
    // tabControlExoticControls
    //
    this.tabControlExoticControls.Controls.Add(this.pageTrackBars);
    this.tabControlExoticControls.Controls.Add(this.pagePanels);
    this.tabControlExoticControls.Controls.Add(this.pageUpDown);
    this.tabControlExoticControls.Controls.Add(this.pageErrorProvider);
    this.tabControlExoticControls.Controls.Add(this.pageTreeView);
    this.tabControlExoticControls.Controls.Add(this.pageWebBrowser);
    this.tabControlExoticControls.Location = new System.Drawing.Point(13, 13);
    this.tabControlExoticControls.Name = "tabControlExoticControls";
    this.tabControlExoticControls.SelectedIndex = 0;
    this.tabControlExoticControls.Size = new System.Drawing.Size(463, 274);
    this.tabControlExoticControls.TabIndex = 0;
...
    this.Controls.Add(this.tabControlExoticControls);
}
```

Now that you have a basic Form supporting multiple tabs, you can build each page to illustrate the remaining exotic controls. First up, let's check out the role of the `TrackBar`.

■ **Note** The `TabControl` widget supports `Selected`, `Selecting`, `Deselected`, and `Deselecting` events. These can prove helpful when you need to dynamically generate the elements within a given page.

Fun with TrackBars

The `TrackBar` control allows users to select from a range of values, using a scroll bar–like input mechanism. When working with this type, you need to set the minimum and maximum range, the minimum and maximum change increments, and the starting location of the slider's thumb. Each of these aspects can be set using the properties described in Table 21-6.

Table 21-6. `TrackBar` *Properties*

Properties	Meaning in Life
`LargeChange`	The number of ticks by which the `TrackBar` changes when an event considered a large change occurs (e.g., clicking the mouse button while the cursor is on the sliding range and using the Page Up or Page Down key).
`Maximum` `Minimum`	Configure the upper and lower bounds of the `TrackBar`'s range.
`Orientation`	The orientation for this `TrackBar`. Valid values are from the `Orientation` enumeration (i.e., horizontally or vertically).
`SmallChange`	The number of ticks by which the `TrackBar` changes when an event considered a small change occurs (e.g., using the arrow keys).
`TickFrequency`	Indicates how many ticks are drawn. For a `TrackBar` with an upper limit of 200, it is impractical to draw all 200 ticks on a control 2 inches long. If you set the `TickFrequency` property to 5, the `TrackBar` draws 20 total ticks (each tick represents 5 units).
`TickStyle`	Indicates how the `TrackBar` control draws itself. This affects both where the ticks are drawn in relation to the movable thumb and how the thumb itself is drawn (using the `TickStyle` enumeration).
`Value`	Gets or sets the current location of the `TrackBar`. Use this property to obtain the numeric value contained by the `TrackBar` for use in your application.

To illustrate, you'll update the first tab of your `TabControl` with three `TrackBars`, each of which has an upper range of 255 and a lower range of 0. As the user slides each thumb, the application intercepts the `Scroll` event and dynamically builds a new `System.Drawing.Color` type based on the value of each slider. This `Color` type will be used to display the color within a `PictureBox` widget (named `colorBox`) and the RGB values within a `Label` type (named `lblCurrColor`). Figure 21-18 shows the (completed) first page in action.

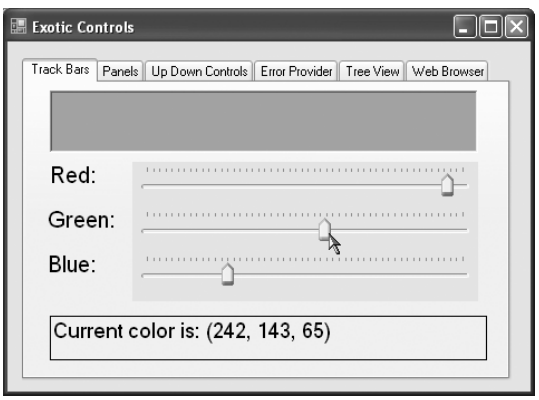

Figure 21-18. *The* `TrackBar` *page*

First, place three TrackBars onto the first tab using the Forms designer and rename your member variables with an appropriate value (redTrackBar, greenTrackBar, and blueTrackBar). Next, handle the Scroll event for each of your TrackBar controls. Here is the relevant code within InitializeComponent() for blueTrackBar (the remaining bars look almost identical, with the exception of the name of the Scroll event handler):

```
private void InitializeComponent()
{
...
    //
    // blueTrackBar
    //
    this.blueTrackBar.Maximum = 255;
    this.blueTrackBar.Name = "blueTrackBar";
    this.blueTrackBar.TickFrequency = 5;
    this.blueTrackBar.TickStyle = System.Windows.Forms.TickStyle.TopLeft;
    this.blueTrackBar.Scroll += new System.EventHandler(this.blueTrackBar_Scroll);
...
}
```

Note that the default minimum value of the TrackBar is 0 and thus does not need to be explicitly set. In the Scroll event handlers for each TrackBar, you make a call to a yet-to-be-written helper function named UpdateColor():

```
private void blueTrackBar_Scroll (object sender, EventArgs e)
{
     UpdateColor();
}
```

UpdateColor() is responsible for two major tasks. First, you read the current value of each TrackBar and use this data to build a new Color variable using Color.FromArgb(). Once you have the newly configured color, update the PictureBox member variable (again, named colorBox) with the current background color. Finally, UpdateColor() formats the thumb values in a string placed on the Label (lblCurrColor), as shown here:

```
private void UpdateColor()
{
     // Get the new color based on track bars.
     Color c = Color.FromArgb(redTrackBar.Value,
          greenTrackBar.Value, blueTrackBar.Value);
     // Change the color in the PictureBox.
     colorBox.BackColor = c;
     // Set color label.
     lblCurrColor.Text =
          string.Format("Current color is: (R:{0}, G:{1}, B:{2})",
          redTrackBar.Value, greenTrackBar.Value,
          blueTrackBar.Value);
}
```

The final detail is to set the initial values of each slider when the Form comes to life and render the current color, as shown here:

```
public MainWindow()
{
     InitializeComponent();
     CenterToScreen();
     // Set initial position of each slider.
     redTrackBar.Value = 100;
     greenTrackBar.Value = 255;
```

```
    blueTrackBar.Value = 0;
    UpdateColor();
}
```

Fun with Panels

As you saw earlier in this chapter, the `GroupBox` control can be used to logically bind a number of controls (such as `RadioButtons`) to function as a collective. Closely related to the `GroupBox` is the `Panel` control. `Panels` are also used to group related controls in a logical unit. One difference is that the `Panel` type derives from the `ScrollableControl` class, thus it can support scroll bars, which is not possible with a `GroupBox`.

`Panels` can also be used to conserve screen real estate. For example, if you have a group of controls that takes up the entire bottom half of a Form, you can contain the group in a `Panel` that is half the size and set the `AutoScroll` property to true. In this way, the user can use the scroll bar(s) to view the full set of items. Furthermore, if a `Panel`'s `BorderStyle` property is set to `None`, you can use this type to simply group a set of elements that can be easily shown or hidden from view in a manner transparent to the end user.

To illustrate, let's update the second page of the `TabControl` with two `Button` types (`btnShowPanel` and `btnHidePanel`) and a single `Panel` that contains a pair of text boxes (`txtNormalText` and `txtUpperText`) and an instructional `Label`. (Mind you, the widgets on the `Panel` are not terribly important for this example.) Figure 21-19 shows the final GUI.

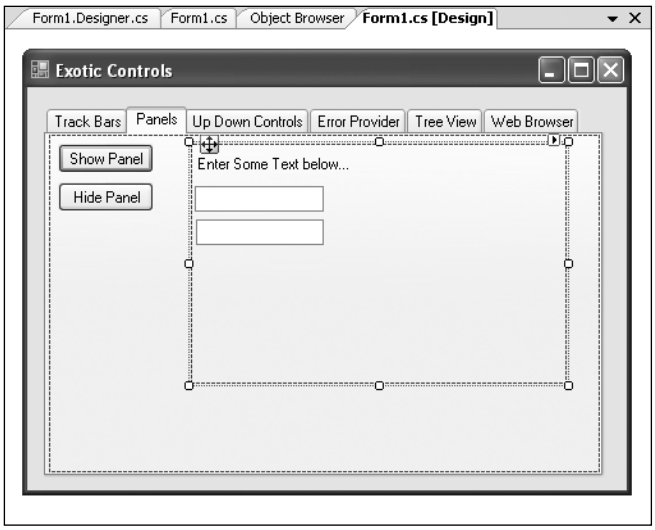

Figure 21-19. *The* `TrackBar` *page*

Using the Properties window, handle the `TextChanged` event for the first `TextBox`, and within the generated event handler, place an uppercase version of the text entered within `txtNormalText` into `txtUpperText`:

```
private void txtNormalText_TextChanged(object sender, EventArgs e)
{
    txtUpperText.Text = txtNormalText.Text.ToUpper();
}
```

Now, handle the `Click` event for each button. As you might suspect, you will simply hide or show the `Panel` (and all of its contained UI elements):

```
private void btnShowPanel_Click(object sender, EventArgs e)
{
    panelTextBoxes.Visible = true;
}
private void btnHidePanel_Click(object sender, EventArgs e)
{
    panelTextBoxes.Visible = false;
}
```

If you now run your program and click either button, you will find that the `Panel`'s contents are shown and hidden accordingly. While this example is hardly fascinating, I am sure you can see the possibilities. For example, you may have a menu option (or security setting) that allows the user to see a "simple" or "complex" view. Rather than having to manually set the `Visible` property to false for multiple widgets, you can group them all within a `Panel` and set its `Visible` property accordingly.

Fun with the UpDown Controls

Windows Forms provide two widgets that function as *spin controls* (also known as *up/down controls*). Like the `ComboBox` and `ListBox` types, these new items also allow the user to choose an item from a range of possible selections. The difference is that when you're using a `DomainUpDown` or `NumericUpDown` control, the information is selected using a pair of small up and down arrows. For example, check out Figure 21-20.

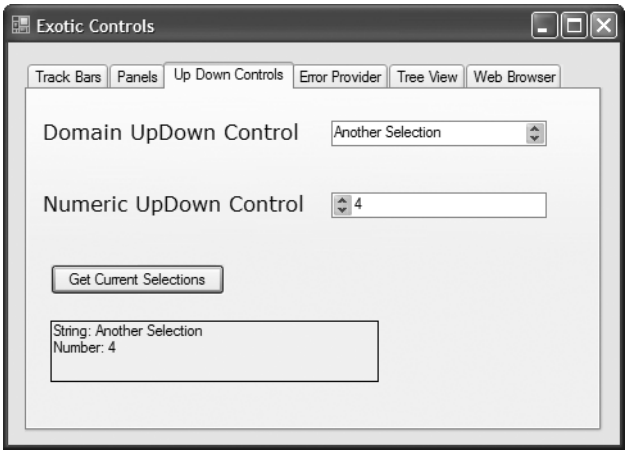

Figure 21-20. *Working with* `UpDown` *types*

Given your work with similar types, you should find working with the `UpDown` widgets painless. The `DomainUpDown` widget allows the user to select from a set of string data. `NumericUpDown` allows selections from a range of numeric data points. Each widget derives from a common direct base class, `UpDownBase`. Table 21-7 describes some important properties of this class.

Table 21-7. `UpDownBase` *Properties*

Property	Meaning in Life
`InterceptArrowKeys`	Gets or sets a value indicating whether the user can use the up arrow and down arrow keys to select values
`ReadOnly`	Gets or sets a value indicating whether the text can only be changed by the use of the up and down arrows and not by typing in the control to locate a given string
`Text`	Gets or sets the current text displayed in the spin control
`TextAlign`	Gets or sets the alignment of the text in the spin control
`UpDownAlign`	Gets or sets the alignment of the up and down arrows on the spin control, using the `LeftRightAlignment` enumeration

The `DomainUpDown` control adds a small set of properties (see Table 21-8) that allow you to configure and manipulate the textual data in the widget.

Table 21-8. `DomainUpDown` *Properties*

Property	Meaning in Life
`Items`	Allows you to gain access to the set of items stored in the widget
`SelectedIndex`	Returns the zero-based index of the currently selected item (a value of –1 indicates no selection)
`SelectedItem`	Returns the selected item itself (not its index)
`Sorted`	Configures whether or not the strings should be alphabetized
`Wrap`	Controls if the collection of items continues to the first or last item if the user continues past the end of the list

The `NumericUpDown` type is just as simple (see Table 21-9).

Table 21-9. `NumericUpDown` *Properties*

Property	Meaning in Life

`DecimalPlaces`
`ThousandsSeparator`
`Hexadecimal`
	Used to configure how the numerical data is to be displayed.

`Increment`	Sets the numerical value to increment the value in the control when the up or down arrow is clicked. The default is to advance the value by 1.
`Minimum` `Maximum`	Sets the upper and lower limits of the value in the control.
`Value`	Returns the current value in the control.

Here is a partial `InitializeComponent()` that configures this page's `NumericUpDown` and `DomainUpDown` widgets:

```
private void InitializeComponent()
{
...
    //
    // numericUpDown
    //
```

```
...
    this.numericUpDown.Maximum = new decimal(new int[] {
        5000, 0, 0, 0});
    this.numericUpDown.Name = "numericUpDown";
    this.numericUpDown.ThousandsSeparator = true;
    //
    // domainUpDown
    //
    this.domainUpDown.Items.Add("Another Selection");
    this.domainUpDown.Items.Add("Final Selection");
    this.domainUpDown.Items.Add("Selection One");
    this.domainUpDown.Items.Add("Third Selection");
    this.domainUpDown.Name = "domainUpDown";
    this.domainUpDown.Sorted = true;
...
}
```

The `Click` event handler for this page's `Button` type simply asks each type for its current value and places it in the appropriate `Label` (`lblCurrSel`) as a formatted string, as shown here:

```
private void btnGetSelections_Click (object sender, EventArgs e)
{
     // Get info from updowns...
     lblCurrSel.Text =
          string.Format("String: {0}\nNumber: {1}",
          domainUpDown.Text, numericUpDown.Value);
 }
```

Fun with ErrorProviders

Most Windows Forms applications will need to validate user input in one way or another. This is especially true with dialog boxes, as you should inform users if they make a processing error before continuing forward. The `ErrorProvider` type can be used to provide a visual cue of user input error. For example, assume you have a Form containing a `TextBox` and `Button` widget. If the user enters more than five characters in the `TextBox` and the `TextBox` loses focus, the error information shown in Figure 21-21 could be displayed.

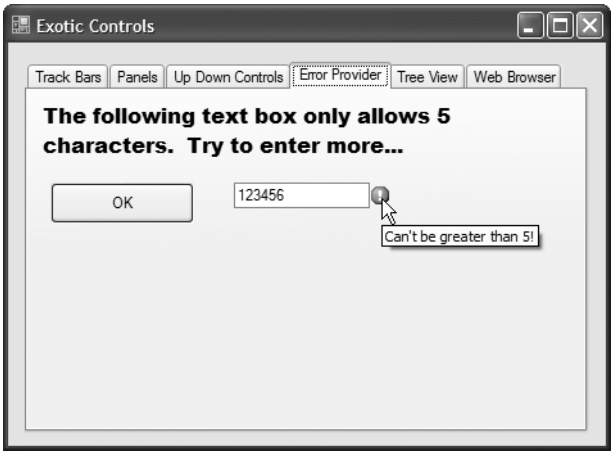

Figure 21-21. *The* `ErrorProvider` *in action*

Here, you have detected that the user entered more than five characters and responded by placing a small error icon (!) next to the `TextBox` object. When the user places his cursor over this icon, the descriptive error text appears as a pop-up. Also, this `ErrorProvider` is configured to cause the icon to blink a number of times to strengthen the visual cue (which, of course, you can't see without running the application).

If you wish to support this type of input validation, the first step is to understand the properties of the `Control` class shown in Table 21-10.

Table 21-10. `Control` *Properties*

Property	Meaning in Life
`CausesValidation`	Indicates whether selecting this control causes validation on the controls requiring validation
`Validated`	Occurs when the control is finished performing its validation logic
`Validating`	Occurs when the control is validating user input (e.g., when the control loses focus)

Every GUI widget can set the `CausesValidation` property to true or false (the default is true). If you set this bit of state data to true, the control forces the other controls on the Form to validate themselves when it receives focus. Once a validating control has received focus, the `Validating` and `Validated` events are fired for each control. In the scope of the `Validating` event handler, you configure a corresponding `ErrorProvider`. Optionally, the `Validated` event can be handled to determine when the control has finished its validation cycle.

The `ErrorProvider` type has a small set of members. The most important item for your purposes is the `BlinkStyle` property, which can be set t any of the values of the `ErrorBlinkStyle` enumeration described in Table 21-11.

Table 21-11. `ErrorBlinkStyle` *Properties*

Property	Meaning in Life
`AlwaysBlink`	Causes the error icon to blink when the error is first displayed or when a new error description string is set for the control and the error icon is already displayed
`BlinkIfDifferentError`	Causes the error icon to blink only if the error icon is already displayed, but a new error string is set for the control
`NeverBlink`	Indicates the error icon never blinks

To illustrate, update the UI of the Error Provider page with a `Button`, `TextBox`, and `Label` as shown in Figure 20-21. Next, drag an `ErrorProvider` widget named `tooManyCharactersErrorProvider` onto the designer. Here is the configuration code within `InitializeComponent()`:

```
private void InitializeComponent()
{
...
    //
    // tooManyCharactersErrorProvider
    //
    this.tooManyCharactersErrorProvider.BlinkRate = 500;
    this.tooManyCharactersErrorProvider.BlinkStyle =
        System.Windows.Forms.ErrorBlinkStyle.AlwaysBlink;
```

```
    this.tooManyCharactersErrorProvider.ContainerControl = this;
...
}
```

Once you have configured how the `ErrorProvider` looks and feels, you bind the error to the `TextBox` within the scope of its `Validating` event handler, as shown here:

```
private void txtInput_Validating (object sender, CancelEventArgs e)
{
     // Check if the text length is greater than 5.
     if(txtInput.Text.Length > 5)
     {
          errorProvider1.SetError( txtInput, "Can't be greater than 5!");
     }
     else // Things are OK, don't show anything.
          errorProvider1.SetError(txtInput, "");
}
```

Fun with TreeViews

`TreeView` controls are very helpful types in that they allow you to visually display hierarchical data (such as a directory structure or any other type of parent/child relationship). As you would expect, the Window Forms `TreeView` control can be highly customized. If you wish, you can add custom images, node colors, node subcontrols, and other visual enhancements. (I'll assume interested readers will consult the .NET Framework 2.0 SDK documentation for full details of this widget.)

To illustrate the basic use of the `TreeView`, the next page of your `TabControl` will programmatically construct a `TreeView` defining a series of topmost nodes that represent a set of `Car` types. Each Car node has two subnodes that represent the selected car's current speed and favorite radio station. In Figure 21-22, notice that the selected item will be highlighted. Also note that if the selected node has a parent (or sibling), its name is presented in a `Label` widget.

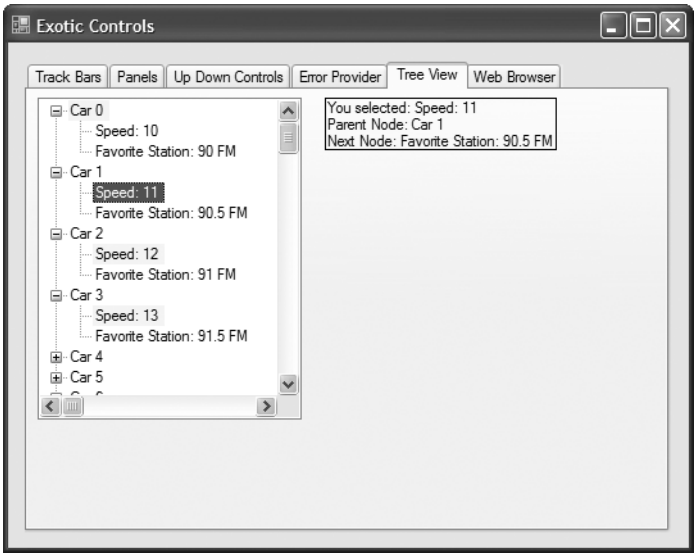

Figure 21-22. *The* `TreeView` *in action*

Assuming your Tree View UI is composed of a `TreeView` control (named `treeViewCars`) and a `Label` (named `lblNodeInfo`), insert a new C# file into your ExoticControls project that models a trivial `Car` that has-a `Radio`:

```
namespace ExoticControls
{
    class Car
    {
        public Car(string pn, int cs)
        {
            petName = pn;
            currSp = cs;
        }
        public string petName;
        public int currSp;
        public Radio r;
    }

    class Radio
    {
        public double favoriteStation;
        public Radio(double station)
        { favoriteStation = station; }
    }
}
```

The `Form`-derived type will maintain a generic `List<>` (named `listCars`) of 100 `Car` types, which will be populated in the default constructor of the `MainForm` type. As well, the constructor will call a new helper method named `BuildCarTreeView()`, which takes no arguments and returns `void`. Here is the initial update:

```
public partial class MainWindow : Form
{
    // Create a new generic List to hold the Car objects.
    private List<Car> listCars = new List<Car>();

    public MainWindow()
    {
...
        // Fill List<> and build TreeView.
        double offset = 0.5;
        for (int x = 0; x < 100; x++)
        {
            listCars.Add(new Car(string.Format("Car {0}", x), 10 + x));
            offset += 0.5;
            listCars[x].r = new Radio(89.0 + offset);
        }
        BuildCarTreeView();
    }
    ...
}
```

Note that the `petName` of each car is based on the current value of `x` (Car 0, Car 1, Car 2, etc.). As well, the current speed is set by offsetting `x` by 10 (10 mph to 109 mph), while the favorite radio station is established by offsetting the value 89.0 by 0.5 (90, 90.5, 91, 91.5, etc.).

Now that you have a list of Cars, you need to map these values to nodes of the TreeView control. The most important aspect to understand when working with the TreeView widget is that each topmost node and subnode is represented by a System.Windows.Forms.TreeNode object, derived directly from MarshalByRefObject. Here are some of the interesting properties of TreeNode:

```
public class TreeNode : MarshalByRefObject,
  ICloneable, ISerializable
{
...
    public Color BackColor { get; set; }
    public bool Checked { get; set; }
    public virtual ContextMenu ContextMenu { get; set; }
    public virtual ContextMenuStrip ContextMenuStrip { get; set; }
    public Color ForeColor { get; set; }
    public int ImageIndex { get; set; }
    public bool IsExpanded { get; }
    public bool IsSelected { get; }
    public bool IsVisible { get; }
    public string Name { get; set; }
    public TreeNode NextNode { get; }
    public Font NodeFont { get; set; }
    public TreeNodeCollection Nodes { get; }
    public TreeNode PrevNode { get; }
    public string Text { get; set; }
    public string ToolTipText { get; set; }
...
}
```

As you can see, each node of a TreeView can be assigned images, colors, fonts, tool tips, and context menus. As well, the TreeNode provides members to navigate to the next (or previous) TreeNode. Given this, consider the initial implementation of BuildCarTreeView():

```
private void BuildCarTreeView()
{
    // Don't paint the TreeView until all the nodes have been created.
    treeViewCars.BeginUpdate();

    // Clear the TreeView of any current nodes.
    treeViewCars.Nodes.Clear();

    // Add a TreeNode for each Car object in the List<>.
    foreach (Car c in listCars)
    {
        // Add the current Car as a topmost node.
        treeViewCars.Nodes.Add(new TreeNode(c.petName));

        // Now, get the Car you just added to build
        // two subnodes based on the speed and
        // internal Radio object.
        treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(
            new TreeNode(string.Format("Speed: {0}",
            c.currSp.ToString())));
        treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(
            new TreeNode(string.Format("Favorite Station: {0} FM",
            c.r.favoriteStation)));
    }
```

```
    // Now paint the TreeView.
    treeViewCars.EndUpdate();
}
```

As you can see, the construction of the `TreeView` nodes are sandwiched between a call to `BeginUpdate()` and `EndUpdate()`. This can be helpful when you are populating a massive `TreeView` with a great many nodes, given that the widget will wait to display the items until you have finished filling the `Nodes` collection. In this way, the end user does not see the gradual rendering of the `TreeView`'s elements.

The topmost nodes are added to the `TreeView` simply by iterating over the generic `List<>` type and inserting a new `TreeNode` object into the `TreeView`'s `Nodes` collection. Once a topmost node has been added, you pluck it from the `Nodes` collection (via the type indexer) to add its subnodes (which are also represented by `TreeNode` objects). As you might guess, if you wish to add subnodes to a current subnode, simply populate its internal collection of nodes via the `Nodes` property.

The next task for this page of the `TabControl` is to highlight the currently selected node (via the `BackColor` property) and display the selected item (as well as any parent or subnodes) within the `Label` widget. All of this can be accomplished by handling the `TreeView` control's `AfterSelect` event via the Properties window. This event fires after the user has selected a node via a mouse click or keyboard navigation. Here is the complete implementation of the `AfterSelect` event handler:

```
private void treeViewCars_AfterSelect(object sender, TreeViewEventArgs e)
{
    string nodeInfo = "";

    // Build info about selected node.
    nodeInfo = string.Format("You selected: {0}\n", e.Node.Text);
    if (e.Node.Parent != null)
        nodeInfo += string.Format("Parent Node: {0}\n", e.Node.Parent.Text);
    if (e.Node.NextNode != null)
        nodeInfo += string.Format("Next Node: {0}", e.Node.NextNode.Text);

    // Show info and highlight node.
    lblNodeInfo.Text = nodeInfo;
    e.Node.BackColor = Color.AliceBlue;
}
```

The incoming `TreeViewEventArgs` object contains a property named `Node`, which returns a `TreeNode` object representing the current selection. From here, you are able to extract the node's name (via the `Text` property) as well as the parent and next node (via the `Parent/NextNode` properties). Note you are explicitly checking the `TreeNode` objects returned from `Parent/NextNode` for null, in case the user has selected the first topmost node or the very last subnode (if you did not do this, you might trigger a `NullReferenceException`).

Adding Node Images

To wrap up our examination of the `TreeView` type, let's spruce up the current example by defining three new `*.bmp` images that will be assigned to each node type. To do so, add a new `ImageList` component (named `imageListTreeView`) to the designer of the `MainForm` type. Next, add three new bitmap images to your project via the Project ➤ Add New Item menu selection (or make use of the supplied `*.bmp` files within this book's downloadable code) that represent (or at least closely approximate) a car, radio, and "speed" image. Do note that each of these `*.bmp` files is 16×16 pixels (set via the Properties window) so that they have a decent appearance within the `TreeView`.

Once you have created these image files, select the `ImageList` on your designer and populate the `Images` property with each of these three images, ordered as shown in Figure 21-23, to ensure you can assign the correct `ImageIndex` (0, 1, or 2) to each node.

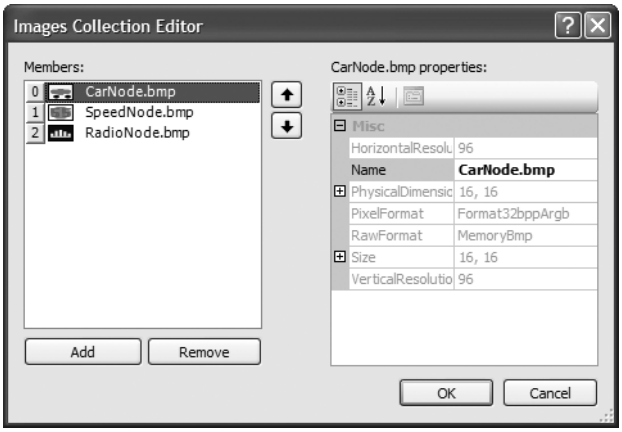

Figure 21-23. *Populating the* `ImageList`

As you recall from Chapter 20, when you incorporate resources (such as bitmaps) into your Visual Studio 2005 solutions, the underlying `*.resx` file is automatically updated. Therefore, these images will be embedded into your assembly with no extra work on your part. Now, using the Properties window, set the `TreeView` control's `ImageList` property to your `ImageList` member variable (see Figure 21-24).

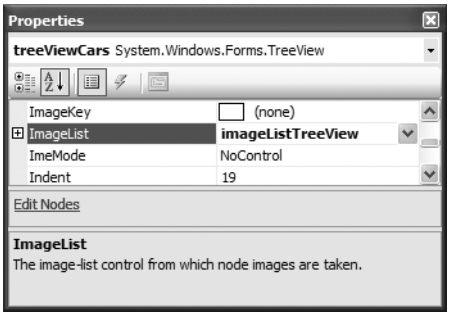

Figure 21-24. *Associating the* `ImageList` *to the* `TreeView`

Last but not least, update your `BuildCarTreeView()` method to specify the correct `ImageIndex` (via constructor arguments) when creating each `TreeNode`:

```
private void BuildCarTreeView()
{
...
    foreach (Car c in listCars)
    {
        treeViewCars.Nodes.Add(new TreeNode(c.petName, 0, 0));

        treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(
            new TreeNode(string.Format("Speed: {0}",
            c.currSp.ToString()), 1, 1));
```

```
            treeViewCars.Nodes[listCars.IndexOf(c)].Nodes.Add(
                new TreeNode(string.Format("Favorite Station: {0} FM",
                c.r.favoriteStation), 2, 2));
        }
...
}
```

Notice that you are specifying each `ImageIndex` twice. The reason for this is that a given `TreeNode` can have two unique images assigned to it: one to display when unselected and another to display when selected. To keep things simple, you are specifying the same image for both possibilities. In any case, Figure 21-25 shows the updated `TreeView` type.

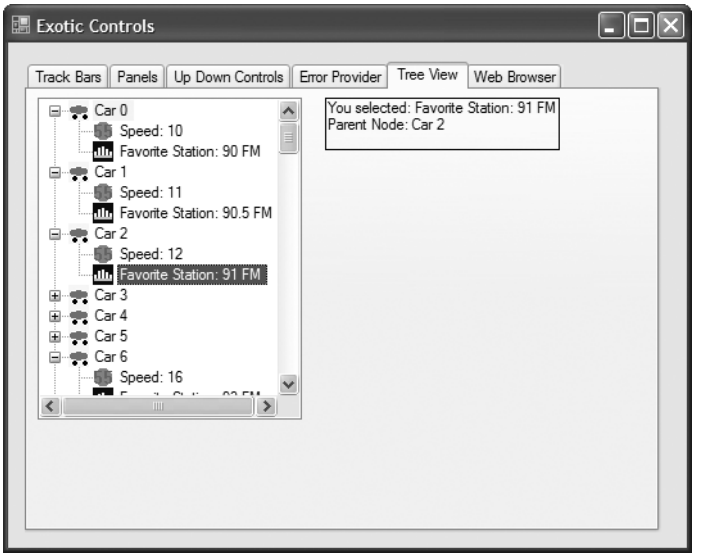

Figure 21-25. *The* `TreeView` *with images*

Fun with WebBrowsers

The final page of this example will make use of the `System.Windows.Forms.WebBrowser` widget, which is new to .NET 2.0. This widget is a highly configurable mini web browser that may be embedded into any `Form`-derived type. As you would expect, this control defines a `Url` property that can be set to any valid URI, formally represented by the `System.Uri` type. On the Web Browser page, add a `WebBrowser` (configured to your liking), a `TextBox` (to enter the URL), and a `Button` (to perform the HTTP request). Figure 21-26 shows the runtime behavior of assigning the `Url` property to `http://www.intertechtraining.com` (yes, a shameless promotion for the company I am employed with).

Figure 21-26. *The* `WebBrowser` *showing the home page of Intertech Training*

The only necessary code to instruct the `WebBrowser` to display the incoming HTTP request form data is to assign the `Url` property, as shown in the following `Button` `Click` event handler:

```
private void btnGO_Click(object sender, EventArgs e)
{
    // Set URL based on value within page's TextBox control.
    myWebBrowser.Url = new System.Uri(txtUrl.Text);
}
```

That wraps up our examination of the widgets of the `System.Windows.Forms` namespace. Although I have not commented on each possible UI element, you should have no problem investigating the others further on your own time. Next up, let's look at the process of building *custom* Windows Forms controls.

■ **Source Code** The ExoticControls project is included under the Chapter 21 directory.

Building Custom Windows Forms Controls

The .NET platform provides a very simple way for developers to build custom UI elements. Unlike (the now legacy) ActiveX controls, Windows Forms controls do not require vast amounts of COM infrastructure or complex memory management. Rather, .NET developers simply build a new class deriving from `UserControl` and populate the type with any number of properties, methods, and events. To demonstrate this process, during the next several pages you'll construct a custom control named `CarControl` using Visual Studio 2005.

■ **Note** As with any .NET application, you are always free to build a custom Windows Forms control using nothing more than the command-line compiler and a simple text editor. As you will see, custom controls reside in a `*.dll` assembly; therefore, you may specify the `/target:dll` option of `csc.exe`.

To begin, fire up Visual Studio 2005 and select a new Windows Control Library workspace named CarControlLibrary (see Figure 21-27).

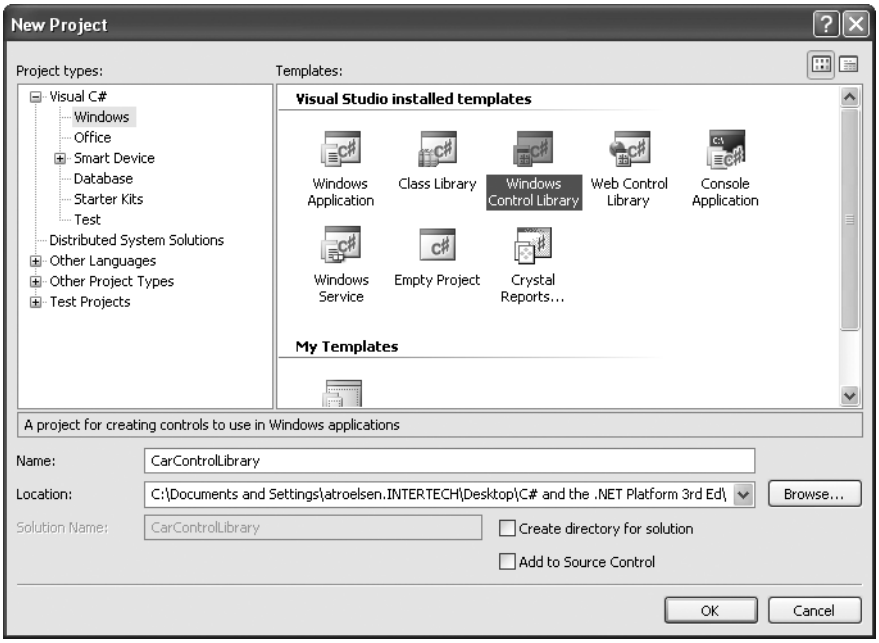

Figure 21-27. *Creating a new Windows Control Library workspace*

When you are finished, rename the initial C# class to `CarControl`. Like a Windows Application project workspace, your custom control is composed of two partial classes. The `*.Designer.cs` file contains all of the designer-generated code, while your primary partial class definition defines a type deriving from `System.Windows.Forms.UserControl`:

```
namespace CarControlLibrary
{
    public partial class CarControl : UserControl
    {
        public CarControl()
        {
            InitializeComponent();
        }
    }
}
```

Before we get too far along, let's establish the big picture of where you are going with this example. The `CarControl` type is responsible for animating through a series of bitmaps that will change based on the internal state of the automobile. If the car's current speed is safely under the car's maximum speed limit, the `CarControl` loops through three bitmap images that render an automobile driving safely along. If the current speed is 10 mph below the maximum speed, the `CarControl` loops through four images, with the fourth image showing the car slowly breaking down. Finally, if the car has surpassed its maximum speed, the `CarControl` loops over five images, where the fifth image represents a doomed automobile.

Creating the Images

Given the preceding design notes, the first order of business is to create a set of five `*.bmp` files for use by the animation loop. If you wish to create custom images, begin by activating the Project ➤ Add New Item menu selection and insert five new bitmap files. If you would rather not showcase your artistic abilities, feel free to use the images that accompany this sample application (keep in mind that I in *no way* consider myself a graphic artist!). The first of these three images (`Lemon1.bmp`, `Lemon2.bmp`, and `Lemon3.bmp`) illustrates a car navigating down the road in a safe and orderly fashion. The final two bitmap images (`AboutToBlow.bmp` and `EngineBlown.bmp`) represent a car approaching its maximum upper limit and its ultimate demise.

Building the Design-Time UI

The next step is to leverage the design-time editor for the `CarControl` type. As you can see, you are presented with a Form-like designer that represents the client area of the control under construction. Using the Toolbox window, add an `ImageList` type to hold each of the bitmaps (named `carImages`), a `Timer` type to control the animation cycle (named `imageTimer`), and a `PictureBox` to hold the current image (named `currentImage`). Don't worry about configuring the size or location of the `PictureBox` type, as you will programmatically position this widget within the bounds of the `CarControl`. However, be sure to set the `SizeMode` property of the `PictureBox` to `StretchImage` via the Properties window. Figure 21-28 shows the story thus far.

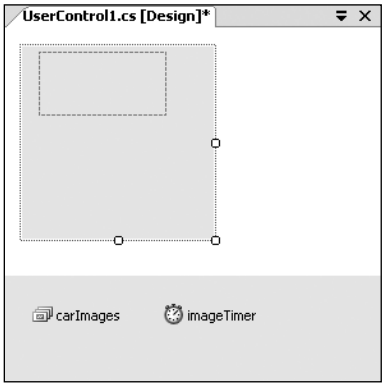

Figure 21-28. *Creating the design-time GUI*

Now, using the Properties window, configure the `ImageList`'s `Images` collection by adding each bitmap to the list. Be aware that you will want to add these items sequentially (`Lemon1.bmp`, `Lemon2.bmp`, `Lemon3.bmp`, `AboutToBlow.bmp`, and `EngineBlown.bmp`) to ensure a linear animation cycle. Also be aware that the default width and height of `*.bmp` files inserted by Visual Studio 2005 is 47×47 pixels. Thus, the `ImageSize` of the `ImageList` should also be set to 47×47 (or else you will have with some skewed rendering). Finally, configure the state of your `Timer` type such that the `Interval` property is set to 200 and is initially disabled.

Implementing the Core CarControl

With this UI prep work out of the way, you can now turn to implementation of the type members. To begin, create a new public enumeration named `AnimFrames`, which has a member representing each item maintained by the `ImageList`. You will make use of this enumeration to determine the current frame to render into the `PictureBox`:

```
// Helper enum for images.
public enum AnimFrames
{
    Lemon1, Lemon2, Lemon3,
    AboutToBlow, EngineBlown
}
```

The `CarControl` type maintains a good number of private data points to represent the animation logic. Here is the rundown of each member:

```
public partial class CarControl : UserControl
{
    // State data.
    private AnimFrames currFrame = AnimFrames.Lemon1;
    private AnimFrames currMaxFrame = AnimFrames.Lemon3;
    private bool IsAnim;
    private int currSp = 50;
    private int maxSp = 100;
    private string carPetName= "Lemon";
    private Rectangle bottomRect = new Rectangle();

    public CarControl()
    {
        InitializeComponent();
    }
}
```

As you can see, you have data points that represent the current and maximum speed, the pet name of the automobile, and two members of type `AnimFrames`. The `currFrame` variable is used to specify which member of the `ImageList` is to be rendered. The `currMaxFrame` variable is used to mark the current upper limit in the `ImageList` (recall that the `CarControl` loops through three to five images based on the current speed). The `IsAnim` data point is used to determine if the car is currently in animation mode. Finally, you have a `Rectangle` member (`bottomRect`), which is used to represent the bottom region of the `CarControl` type. Later, you render the pet name of the automobile into this piece of control real estate.

To divide the `CarControl` into two rectangular regions, create a private helper function named `StretchBox()`. The role of this member is to calculate the correct size of the `bottomRect` member and to ensure that the `PictureBox` widget is stretched out over the upper two-thirds (or so) of the `CarControl` type.

```
private void StretchBox()
{
    // Configure picture box.
    currentImage.Top = 0;
    currentImage.Left = 0;
    currentImage.Height = this.Height - 50;
    currentImage.Width = this.Width;
    currentImage.Image =
        carImages.Images[(int)AnimFrames.Lemon1];
    // Figure out size of bottom rect.
    bottomRect.X = 0;
    bottomRect.Y = this.Height - 50;
    bottomRect.Height = this.Height - currentImage.Height;
    bottomRect.Width = this.Width;
}
```

Once you have carved out the dimensions of each rectangle, call `StretchBox()` from the default constructor:

```
public CarControl()
{
    InitializeComponent();
    StretchBox();
}
```

Defining the Custom Events

The `CarControl` type supports two events that are fired back to the host Form based on the current speed of the automobile. The first event, `AboutToBlow`, is sent out when the `CarControl`'s speed approaches the upper limit. `BlewUp` is sent to the container when the current speed is greater than the allowed maximum. Each of these events leverages a custom delegate (`CarEventHandler`) that can hold the address of any method returning `void` and taking a single `System.String` as its parameter. You'll fire these events in just a moment, but for the time being, add the following members to the public sector of the `CarControl`:

```
// Car events/custom delegate.
public delegate void CarEventHandler(string msg);
public event CarEventHandler AboutToBlow;
public event CarEventHandler BlewUp;
```

■ **Note** Recall that a "prim and proper" delegate (see Chapter 8) would specify two arguments, the first of which is a `System.Object` (to represent the sender), and the second of which is a `System.EventArgs`-derived type. For this example, however, your delegate fits the bill.

Defining the Custom Properties

Like any class type, custom controls may define a set of properties to allow the outside world to interact with the state of the widget. For your current purposes, you are interested only in defining three properties. First, you have `Animate`. This property enables or disables the `Timer` type:

```
// Used to configure the internal Timer type.
public bool Animate
{
    get {return IsAnim;}
```

```
    set
    {
        IsAnim = value;
        imageTimer.Enabled = IsAnim;
    }
}
```

The PetName property is what you would expect and requires little comment. Do notice, however, that when the user sets the pet name, you make a call to Invalidate() to render the name of the CarControl into the bottom rectangular area of the widget (you'll do this step in just a moment):

```
// Configure pet name.
public string PetName
{
    get{return carPetName;}
    set
    {
        carPetName = value;
        Invalidate();
    }
}
```

Next, you have the Speed property. In addition to simply modifying the currSp data member, Speed is the entity that fires the AboutToBlow and BlewUp events based on the current speed of the CarControl. Here is the complete logic:

```
// Adjust currSp and currMaxFrame, and fire our events.
public int Speed
{
    get { return currSp; }
    set
    {
        // Within safe speed?
        if (currSp <= maxSp)
        {
            currSp = value;
            currMaxFrame = AnimFrames.Lemon3;
        }
        // About to explode?
        if ((maxSp - currSp) <= 10)
        {
            if (AboutToBlow != null)
            {
                AboutToBlow("Slow down dude!");
                currMaxFrame = AnimFrames.AboutToBlow;
            }
        }
        // Maxed out?
        if (currSp >= maxSp)
        {
            currSp = maxSp;
            if (BlewUp != null)
            {
                BlewUp("Ug...you're toast...");
                currMaxFrame = AnimFrames.EngineBlown;
            }
        }
    }
}
```

As you can see, if the current speed is 10 mph below the maximum upper speed, you fire the `AboutToBlow` event and adjust the upper frame limit to `AnimFrames.AboutToBlow`. If the user has pushed the limits of your automobile, you fire the `BlewUp` event and set the upper frame limit to `AnimFrames.EngineBlown`. If the speed is below the maximum speed, the upper frame limit remains as `AnimFrames.Lemon3`.

Controlling the Animation

The next detail to attend to is ensuring that the `Timer` type advances the current frame to render within the `PictureBox`. Again, recall that the number of frames to loop through depends on the current speed of the automobile. You only want to bother adjusting the image in the `PictureBox` if the `Animate` property has been set to true. Begin by handling the `Tick` event for the `Timer` type, and flesh out the details as follows:

```
private void imageTimer_Tick(object sender, EventArgs e)
{
    if(IsAnim)
        currentImage.Image = carImages.Images[(int)currFrame];
    // Bump frame.
    int nextFrame = ((int)currFrame) + 1;
    currFrame = (AnimFrames)nextFrame;
    if (currFrame > currMaxFrame)
        currFrame = AnimFrames.Lemon1;
}
```

Rendering the Pet Name

Before you can take your control out for a spin, you have one final detail to attend to: rendering the car's moniker. To do this, handle the `Paint` event for your `CarControl`, and within the handler, render the `CarControl`'s pet name into the bottom rectangular region of the client area:

```
private void CarControl_Paint(object sender, PaintEventArgs e)
{
    // Render the pet name on the bottom of the control.
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.GreenYellow, bottomRect);
    g.DrawString(PetName, new Font("Times New Roman", 15),
        Brushes.Black, bottomRect);
}
```

At this point, your initial crack at the `CarControl` is complete. Go ahead and build your project.

Testing the CarControl Type

When you run or debug a Windows Control Library project within Visual Studio 2005, the UserControl Test Container (a managed replacement for the now legacy ActiveX Control Test Container) automatically loads your control into its designer test bed. As you can see from Figure 21-29, this tool allows you to set each custom property (as well as all inherited properties) for testing purposes.

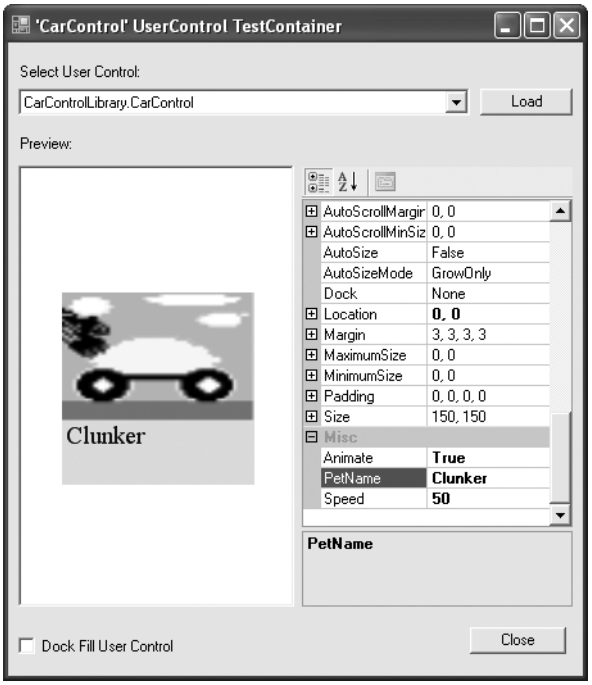

Figure 21-29. *Testing the* `CarControl` *with the UserControl Test Container*

If you set the `Animate` property to true, you should see the `CarControl` cycle through the first three `*.bmp` files. What you are unable to do with this testing utility, however, is handle events. To test this aspect of your UI widget, you need to build a custom Form.

Building a Custom CarControl Form Host

As with all .NET types, you are now able to make use of your custom control from any language targeting the CLR. Begin by closing down the current workspace and creating a new C# Windows Application project named `CarControlTestForm`. To reference your custom controls from within the Visual Studio 2005 IDE, right-click anywhere within the Toolbox window and select the Choose Item menu selection. Using the Browse button on the .NET Framework Components tab, navigate to your `CarControlLibrary.dll` library. Once you click OK, you will find a new icon on the Toolbox named, of course, `CarControl`.

Next, place a new `CarControl` widget onto the Forms designer. Notice that the `Animate`, `PetName`, and `Speed` properties are all exposed through the Properties window. Again, like the UserControl Test Container, the control is "alive" at design time. Thus, if you set the `Animate` property to true, you will find your car is animating on the Forms designer.

Once you have configured the initial state of your `CarControl`, add additional GUI widgets that allow the user to increase and decrease the speed of the automobile, and view the string data sent by the incoming events as well as the car's current speed (`Label` controls will do nicely for these purposes). One possible GUI design is shown in Figure 21-30.

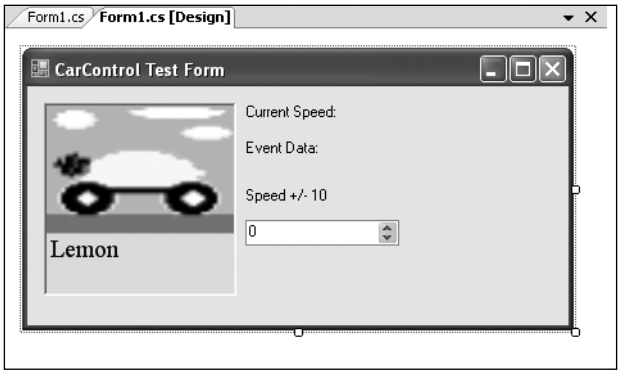

Figure 21-30. *The client-side GUI*

Provided you have created a GUI identical to mine, the code within the `Form`-derived type is quite straightforward (here I am assuming you have handled each of the `CarControl` events using the Properties window):

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
        lblCurrentSpeed.Text = string.Format("Current Speed: {0}",
            this.myCarControl.Speed.ToString());
        numericUpDownCarSpeed.Value = myCarControl.Speed;
    }
    private void numericUpDownCarSpeed_ValueChanged(object sender, EventArgs e)
    {
        // Assume the min of this NumericUpDown is 0 and max is 300.
        this.myCarControl.Speed = (int)numericUpDownCarSpeed.Value;
        lblCurrentSpeed.Text = string.Format("Current Speed: {0}",
            this.myCarControl.Speed.ToString());
    }
    private void myCarControl_AboutToBlow(string msg)
    { lblEventData.Text = string.Format("Event Data: {0}", msg); }

    private void myCarControl_BlewUp(string msg)
    { lblEventData.Text = string.Format("Event Data: {0}", msg); }
}
```

At this point, you are able to run your client application and interact with the `CarControl`. As you can see, building and using custom controls is a fairly straightforward task, given what you already know about OOP, the .NET type system, GDI+ (aka `System.Drawing.dll`), and Windows Forms.

While you now have enough information to continue exploring the process of .NET Windows controls development, there is one additional programmatic aspect you have to contend with: design-time functionality. Before I describe exactly what this boils down to, you'll need to understand the role of the `System.ComponentModel` namespace.

The Role of the System.ComponentModel Namespace

The `System.ComponentModel` namespace defines a number of attributes (among other types) that allow you to describe how your custom controls should behave at design time. For example, you can opt to supply a textual description of each property, define a default event, or group related properties or events into a custom category for display purposes within the Visual Studio 2005 Properties window. When you are interested in making the sorts of modifications previously mentioned, you will want to make use of the core attributes shown in Table 21-12.

Table 21-12. *Select Members of* `System.ComponentModel`

Attribute	Applied To	Meaning in Life
`BrowsableAttribute`	Properties and events	Specifies whether a property or an event should be displayed in the property browser. By default, all custom properties and events can be browsed.
`CategoryAttribute`	Properties and events	Specifies the name of the category in which to group a property or event.
`DescriptionAttribute`	Properties and events	Defines a small block of text to be displayed at the bottom of the property browser when the user selects a property or event.
`DefaultPropertyAttribute`	Properties	Specifies the default property for the component. This property is selected in the property browser when a user selects the control.
`DefaultValueAttribute`	Properties	Defines a default value for a property that will be applied when the control is "reset" within the IDE.
`DefaultEventAttribute`	Events	Specifies the default event for the component. When a programmer double-clicks the control, stub code is automatically written for the default event.

Enhancing the Design-Time Appearance of CarControl

To illustrate the use of some of these new attributes, close down the CarControlTestForm project and reopen your CarControlLibrary project. Let's create a custom category called "Car Configuration" to which each property and event of the `CarControl` belongs. Also, let's supply a friendly description for each member and default value for each property. To do so, simply update each of the properties and events of the `CarControl` type to support the `[Category]`, `[DefaultValue]`, and `[Description]` attributes as required:

```
public partial class CarControl : UserControl
{
    ...
    [Category("Car Configuration"),
    Description("Sent when the car is approaching terminal speed.")]
    public event CarEventHandler AboutToBlow;
    ...
    [Category("Car Configuration"),
    Description("Name your car!"),
    DefaultValue("Lemon")]
```

```
    public string PetName {...}
...
}
```

Now, let me make a comment on what it means to assign a *default value* to a property, because I can guarantee you it is not what you would (naturally) assume. Simply put, the `[DefaultValue]` attribute does *not* ensure that the underlying value of the data point wrapped by a given property will be automatically initialized to the default value. Thus, although you specified a default value of "No Name" for the `PetName` property, the `carPetName` member variable will not be set to "Lemon" unless you do so via the type's constructor or via member initialization syntax (as you have already done):

```
private string carPetName= "Lemon";
```

Rather, the `[DefaultValue]` attribute comes into play when the programmer "resets" the value of a given property using the Properties window. To reset a property using Visual Studio 2005, select the property of interest, right-click it, and select Reset. In Figure 21-31, notice that the `[Description]` value appears in the bottom pane of the Properties window.

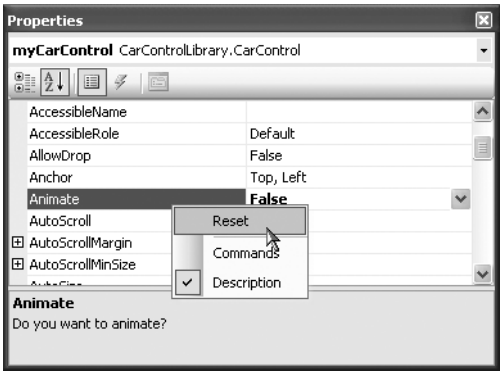

Figure 21-31. *Resetting a property to the default value*

The `[Category]` attribute will be realized only if the programmer selects the categorized view of the Properties window (as opposed to the default alphabetical view) as shown in Figure 21-32.

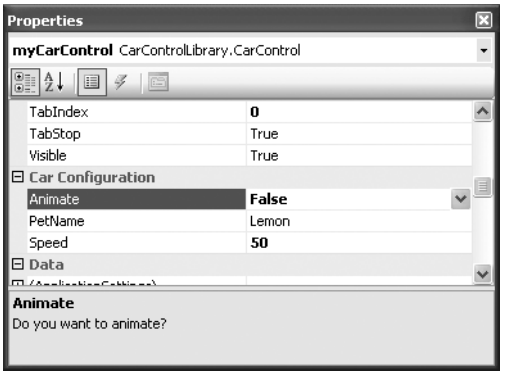

Figure 21-32. *The custom category*

Defining a Default Property and Default Event

In addition to describing and grouping like members into a common category, you may want to configure your controls to support default behaviors. A given control may support a default property. When you define the default property for a class using the `[DefaultProperty]` attribute as follows:

```
//Mark the default property for this control.
[DefaultProperty("Animate")]
public partial class CarControl : UserControl
{...}
```

you ensure that when the user selects this control at design time, the `Animate` property is automatically highlighted in the Properties window. Likewise, if you configure your control to have a default event as follows:

```
// Mark the default event and property for this control.
[DefaultEvent("AboutToBlow"),
DefaultProperty("Animate")]
public partial class CarControl : UserControl
{...}
```

you ensure that when the user double-clicks the widget at design time, stub code is automatically written for the default event (which explains why when you double-click a `Button`, the `Click` event is automatically handled; when you double-click a `Form`, the `Load` event is automatically handled; and so on).

Specifying a Custom Toolbox Bitmap

A final design-time bell-and-whistle any polished custom control should sport is a custom toolbox bitmap image. Currently, when the user selects the `CarControl`, the IDE will show this type within the Toolbox using the default "gear" icon. If you wish to specify a custom image, your first step is to insert a new `*.bmp` file into your project (`CarControl.bmp`) that is configured to be 16×16 pixels in size (established via the `Width` and `Height` properties). Here, I simply reused the `Car` image used in the `TreeView` example.

Once you have created the image as you see fit, use the `[ToolboxBitmap]` attribute (which is applied at the type level) to assign this image to your control. The first argument to the attribute's constructor is the type information for the control itself, while the second argument is the friendly name of the `*.bmp` file.

```
[DefaultEvent("AboutToBlow"),
DefaultProperty("Animate"),
ToolboxBitmap(typeof(CarControl), "CarControl")]
public partial class CarControl : UserControl
{...}
```

The final step is to make sure you set the Build Action value of the control's icon image to Embedded Resource (via the Properties window) to ensure the image data is embedded within your assembly (see Figure 21-33).

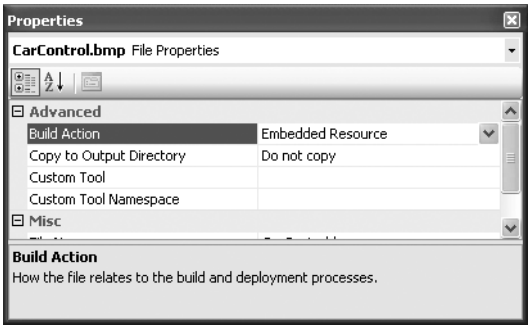

Figure 21-33. *Embedding the image resource*

■ **Note** The reason you are manually embedding the `*.bmp` file (in contrast to when you make use of the `ImageList` type) is that you are not assigning the `CarControl.bmp` file to a UI element at design time, therefore the underlying `*.resx` file will not automatically update.

Once you recompile your Windows Controls library, you can now load your previous CarControlTestForm project. Right-click the current `CarControl` icon within the Toolbox and select Delete. Next, re-add the `CarControl` widget to the Toolbox (by right-clicking and selecting Choose Items). This time, you should see your custom toolbox bitmap (see Figure 21-34).

Figure 21-34. *The custom toolbox bitmap*

So, that wraps up our examination of the process of building custom Windows Forms controls. I hope this example sparked your interest in custom control development. Here, I stuck with the book's automobile theme. Imagine, though, the usefulness of a custom control that will render a pie chart based on the current inventory of a given table in a given database, or a control that extends the functionality of standard UI widgets.

■ **Note** If you are interested in learning more about developing custom Windows Forms controls, pick up a copy of *User Interfaces in C#: Windows Forms and Custom Controls,* by Matthew MacDonald (Apress, 2002).

■ **Source Code** The CarControlLibrary and CarControlTestForm projects are included under the Chapter 21 directory.

Building Custom Dialog Boxes

Now that you have a solid understanding of the core Windows Forms controls and the process of building custom controls, let's examine the construction of custom dialog boxes. The good news is that everything you have already learned about Windows Forms applies directly to dialog box programming. By and large, creating (and showing) a dialog box is no more difficult than inserting a new Form into your current project.

There is no "Dialog" base class in the `System.Windows.Forms` namespace. Rather, a dialog box is simply a stylized Form. For example, many dialog boxes are intended to be nonsizable, therefore you will typically want to set the `FormBorderStyle` property to `FormBorderStyle.FixedDialog`. As well, dialog boxes typically set the `MinimizeBox` and `MaximizeBox` properties to false. In this way, the dialog box is configured to be a fixed constant. Finally, if you set the `ShowInTaskbar` property to false, you will prevent the Form from being visible in the Windows XP task bar.

To illustrate the process of working with dialog boxes, create a new Windows application named SimpleModalDialog. The main `Form` type supports a `MenuStrip` that contains a File ➤ Exit menu item as well as Tools ➤ Configure. Build this UI now, and handle the `Click` event for the Exit and Enter Message menu items. As well, define a string member variable in your main `Form` type (named `userMessage`), and render this data within a `Paint` event handler of your main Form. Here is the current code within the `MainForm.cs` file:

```
public partial class MainWindow : Form
{
    private string userMessage = "Default Message";

    public MainWindow()
    {
        InitializeComponent();
    }

    private void exitToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Application.Exit();
    }

    private void configureToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // We will implement this method in just a bit...
    }

    private void MainWindow_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawString(userMessage, new Font("Times New Roman", 24), Brushes.DarkBlue,
            50, 50);
    }
}
```

Now add a new Form to your current project using the Project ➤ Add Windows Form menu item named `UserMessageDialog.cs`. Set the `ShowInTaskbar`, `MinimizeBox`, and `MaximizeBox` properties to false. Next, build a UI that consists of two `Button` types (for the OK and Cancel buttons), a single

TextBox (to allow the user to enter her message), and an instructive Label. Figure 21-35 shows one possible UI.

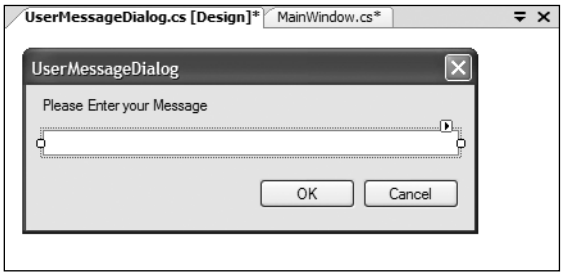

Figure 21-35. *A custom dialog box*

Finally, expose the Text value of the Form's TextBox using a custom property named Message:

```
public partial class UserMessageDialog : Form
{
    public UserMessageDialog()
    {
        InitializeComponent();
    }

    public string Message
    {
        set { txtUserInput.Text = value; }
        get { return txtUserInput.Text; }
    }
}
```

The DialogResult Property

As a final UI task, select the OK button on the Forms designer and find the DialogResult property. Assign DialogResult.OK to your OK button and DialogResult.Cancel to your Cancel button. Formally, you can assign the DialogResult property to any value from the DialogResult enumeration:

```
public enum System.Windows.Forms.DialogResult
{
    Abort, Cancel, Ignore, No,
    None, OK, Retry, Yes
}
```

So, what exactly does it mean to assign a Button's DialogResult value? This property can be assigned to any Button type (as well as the Form itself) and allows the parent Form to determine which button the end user selected. To illustrate, update the Tools ➤ Configure menu handler on the MainForm type as so:

```
private void configureToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Create an instance of UserMessageDialog.
    UserMessageDialog dlg = new UserMessageDialog();

    // Place the current message in the TextBox.
    dlg.Message = userMessage;
```

```
    // If user clicked OK button, render his message.
    if (DialogResult.OK == dlg.ShowDialog())
    {
        userMessage = dlg.Message;
        Invalidate();
    }

    // Have dialog clean up internal widgets now, rather
    // than when the GC destroys the object.
    dlg.Dispose();
}
```

Here, you are showing the `UserMessageDialog` via a call to `ShowDialog()`. This method will launch the Form as a *modal* dialog box which, as you may know, means the user is unable to activate the main form until she dismisses the dialog box. Once the user does dismiss the dialog box (by clicking the OK or Cancel button), the Form is no longer visible, but it is still in memory. Therefore, you are able to ask the `UserMessageDialog` instance (`dlg`) for its new `Message` value in the event the user has clicked the OK button. If so, you render the new message. If not, you do nothing.

■ **Note** If you wish to show a modeless dialog box (which allows the user to navigate between the parent and dialog Forms), call `Show()` rather than `ShowDialog()`.

Understanding Form Inheritance

One very appealing aspect of building dialog boxes under Windows Forms is *form inheritance*. As you are no doubt aware, inheritance is the pillar of OOP that allows one class to extend the functionality of another class. Typically, when you speak of inheritance, you envision one non-GUI type (e.g., `SportsCar`) deriving from another non-GUI type (e.g., `Car`). However, in the world of Windows Forms, it is possible for one Form to derive from another Form and in the process inherit the base class's widgets and implementation.

Form-level inheritance is a very powerful technique, as it allows you to build a base Form that provides core-level functionality for a family of related dialog boxes. If you were to bundle these base-level Forms into a .NET assembly, other members of your team could extend these types using the .NET language of their choice.

For the sake of illustration, assume you wish to subclass the `UserMessageDialog` to build a new dialog box that also allows the user to specify if the message should be rendered in italics. To do so, active the Project ➤ Add Windows Form menu item, but this time add a new Inherited Form named `ItalicUserMessageDialog.cs` (see Figure 21-36).

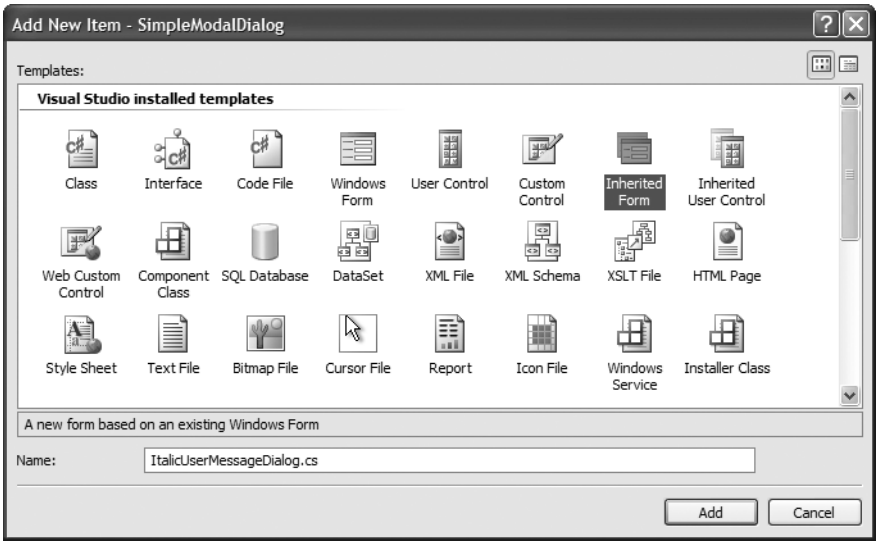

Figure 21-36. *A derived Form*

Once you select Add, you will be shown the *inheritance picker* utility, which allows you to choose from a Form in your current project or select a Form in an external assembly via the Browse button. For this example, select your existing `UserMessageDialog` type. You will find that your new Form type extends your current dialog type rather than directly from Form. At this point, you are free to extend this derived Form any way you choose. For test purposes, simply add a new `CheckBox` control (named `checkBoxItalic`) that is exposed through a property named `Italic`:

```
public partial class ItalicUserMessageDialog :
    SimpleModalDialog.UserMessageDialog
{
    public ItalicUserMessageDialog()
    {
        InitializeComponent();
    }
    public bool Italic
    {
        set { checkBoxItalic.Checked = value; }
        get { return checkBoxItalic.Checked; }
    }
}
```

Now that you have subclassed the basic `UserMessageDialog` type, update your `MainForm` to leverage the new `Italic` property. Simply add a new Boolean member variable that will be used to build an italic `Font` object, and update your Tools ➤ Configure Click menu handler to make use of `ItalicUserMessageDialog`. Here is the complete update:

```
public partial class MainWindow : Form
{
    private string userMessage = "Default Message";
    private bool textIsItalic = false;
...
    private void configureToolStripMenuItem_Click(object sender, EventArgs e)
```

```
    {
        ItalicUserMessageDialog dlg = new ItalicUserMessageDialog();
        dlg.Message = userMessage;
        dlg.Italic = textIsItalic;

        // If user clicked OK button, render his message.
        if (DialogResult.OK == dlg.ShowDialog())
        {
            userMessage = dlg.Message;
            textIsItalic = dlg.Italic;
            Invalidate();
        }
        // Have dialog clean up internal widgets now, rather
        // than when the GC destroys the object.
        dlg.Dispose();
    }

    private void MainWindow_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        Font f = null;
        if(textIsItalic)
            f = new Font("Times New Roman", 24, FontStyle.Italic);
        else
            f = new Font("Times New Roman", 24);
        g.DrawString(userMessage, f, Brushes.DarkBlue,
            50, 50);
    }
}
```

■ **Source Code** The SimpleModalDialog application is included under the Chapter 21 directory.

Dynamically Positioning Windows Forms Controls

To wrap up this chapter, let's examine a few techniques you can use to control the layout of widgets on a Form. By and large, when you build a `Form` type, the assumption is that the controls are rendered using *absolute position*, meaning that if you placed a `Button` on your Forms designer 10 pixels down and 10 pixels over from the upper left portion of the Form, you expect the `Button` to stay put during its lifetime.

On a related note, when you are creating a Form that contains UI controls, you need to decide whether the Form should be resizable. Typically speaking, main windows are resizable, whereas dialog boxes are not. Recall that the resizability of a Form is controlled by the `FormBorderStyle` property, which can be set to any value of the `FormBorderStyle` enum.

```
public enum System.Windows.Forms.FormBorderStyle
{
    None, FixedSingle, Fixed3D,
    FixedDialog, Sizable,
    FixedToolWindow, SizableToolWindow
}
```

Assume that you have allowed your Form to be resizable. This brings up some interesting questions regarding the contained controls. For example, if the user makes the Form smaller than the rectangle needed to display each control, should the controls adjust their size (and possibly location) to morph correctly with the Form?

The Anchor Property

In Windows Forms, the `Anchor` property is used to define a relative fixed position in which the control should always be rendered. Every `Control`-derived type has an `Anchor` property, which can be set to any of the values from the `AnchorStyles` enumeration described in Table 21-13.

Table 21-13. `AnchorStyles` *Values*

Value	Meaning in Life
`Bottom`	The control's bottom edge is anchored to the bottom edge of its container.
`Left`	The control's left edge is anchored to the left edge of its container.
`None`	The control is not anchored to any edges of its container.
`Right`	The control's right edge is anchored to the right edge of its container.
`Top`	The control's top edge is anchored to the top edge of its container.

To anchor a widget at the upper-left corner, you are free to OR styles together (e.g., `AnchorStyles.Top ➤ AnchorStyles.Left`). Again, the idea behind the `Anchor` property is to configure which edges of the control are anchored to the edges of its container. For example, if you configure a `Button` with the following `Anchor` value:

```
// Anchor this widget relative to the right position.
myButton.Anchor = AnchorStyles.Right;
```

you are ensured that as the Form is resized, this `Button` maintains its position relative to the right side of the Form.

The Dock Property

Another aspect of Windows Forms programming is establishing the *docking behavior* of your controls. If you so choose, you can set a widget's `Dock` property to configure which side (or sides) of a Form the widget should be attached to. The value you assign to a control's `Dock` property is honored, regardless of the Form's current dimensions. Table 21-14 describes possible options.

Table 21-14. `DockStyle` *Values*

Value	Meaning in Life
`Bottom`	The control's bottom edge is docked to the bottom of its containing control.
`Fill`	All the control's edges are docked to all the edges of its containing control and sized appropriately.
`Left`	The control's left edge is docked to the left edge of its containing control.
`None`	The control is not docked.
`Right`	The control's right edge is docked to the right edge of its containing control.
`Top`	The control's top edge is docked to the top of its containing control.

So, for example, if you want to ensure that a given widget is always docked on the left side of a Form, you would write the following:

```
// This item is always located on the left of the Form, regardless
// of the Form's current size.
myButton.Dock = DockStyle.Left;
```

To help you understand the implications of setting the `Anchor` and `Dock` properties, the downloadable code for this book contains a project named AnchoringControls. Once you build and run this application, you can make use of the Form's menu system to set various `AnchorStyles` and `DockStyle` values and observe the change in behavior of the `Button` type (see Figure 21-37).

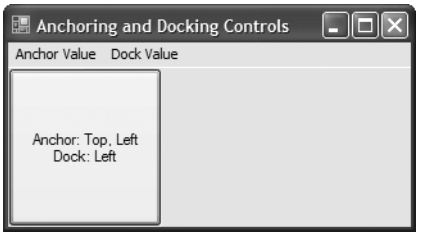

Figure 21-37. *The AnchoringControls application*

Be sure to resize the Form when changing the `Anchor` property to observe how the `Button` responds.

■ **Source Code** The AnchoringControls application is included under the Chapter 21 directory.

Table and Flow Layout

.NET 2.0 offers an additional way to control the layout of a Form's widgets using one of two layout managers. The `TableLayoutPanel` and `FlowLayoutPanel` types can be docked into a Form's client area to arrange the internal controls. For example, assume you place a new `FlowLayoutPanel` widget onto the Forms designer and configure it to dock fully within the parent Form (see Figure 21-38).

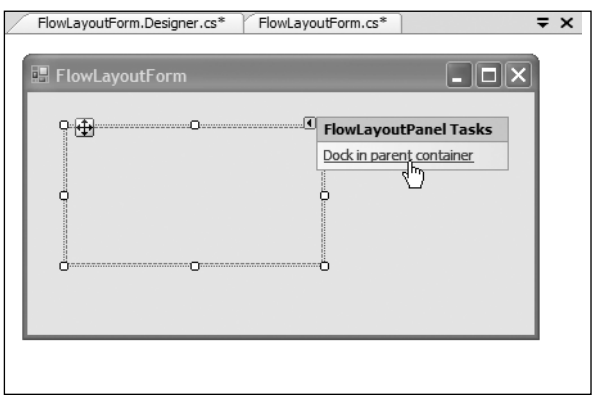

Figure 21-38. *Docking a* `FlowLayoutPanel` *into a Form*

Now, add ten new `Button` types within the `FlowLayoutPanel` using the Forms designer. If you now run your application, you will notice that the ten `Button`s automatically rearrange themselves in a manner very close to standard HTML.

On the other hand, if you create a Form that contains a `TableLayoutPanel`, you are able to build a UI that is partitioned into various "cells" (see Figure 21-39).

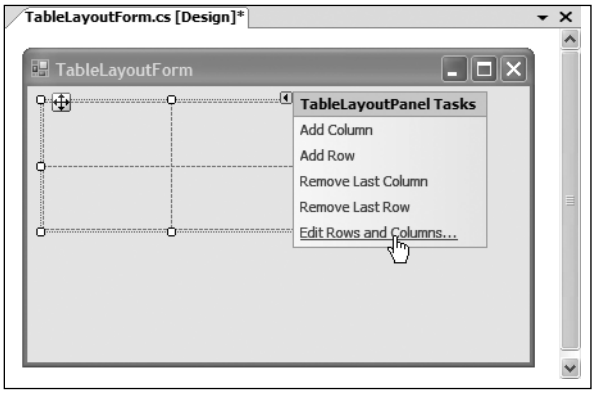

Figure 21-39. *The* `TableLayoutPanel` *type*

If you select the Edit Rows and Columns inline menu option using the Forms designer (as shown in Figure 21-39), you are able to control the overall format of the `TableLayoutPanel` on a cell-by-cell basis (see Figure 21-40).

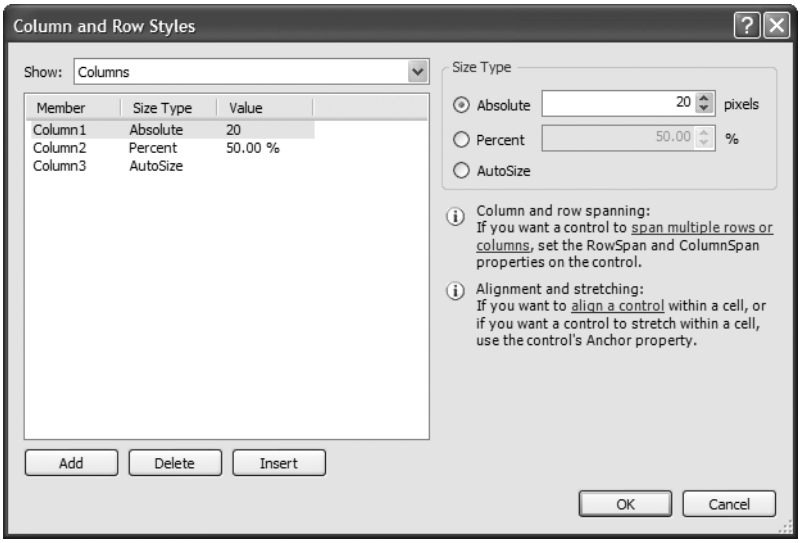

Figure 21-40. *Configuring the cells of the* `TableLayoutPanel` *type*

Truth be told, the only way to see the effects of the `TableLayoutPanel` type is to do so in a hands-on manner. I'll let interested readers handle that task.

Summary

This chapter rounded off your understanding of the Windows Forms namespace by examining the programming of numerous GUI widgets, from the simple (e.g., `Label`) to the more exotic (e.g., `TreeView`). After examining numerous control types, you moved on to cover the construction of custom controls, including the topic of design-time integration.

In the latter half of this chapter, you learned how to build custom dialog boxes and how to derive a new Form from an existing `Form` type using form inheritance. This chapter concluded by briefly exploring the various anchoring and docking behaviors you can use to enforce a specific layout of your GUI types, as well as the new .NET 2.0 layout managers.