APPENDIX A

Programming with Windows Forms

Since the release of the .NET platform (circa 2001), the base class libraries have included a particular API named Windows Forms, represented primarily by the `System.Windows.Forms.dll` assembly. The Windows Forms toolkit provides the types necessary to build desktop graphical user interfaces (GUIs), create custom controls, manage resources (e.g., string tables and icons), and perform other desktop-centric programming tasks. In addition, a separate API named GDI+ (represented by the `System.Drawing.dll` assembly) provides additional types that allow programmers to generate 2D graphics, interact with networked printers, and manipulate image data.

The Windows Forms (and GDI+) APIs remain alive and well within the .NET 4.0 platform, and they will exist within the base class library for quite some time (arguably forever). However, Microsoft has shipped a brand new GUI toolkit called Windows Presentation Foundation (WPF) since the release of .NET 3.0. As you saw in Chapters 27-31, WPF provides a massive amount of horsepower that you can use to build bleeding-edge user interfaces, and it has become the preferred desktop API for today's .NET graphical user interfaces.

The point of this appendix, however, is to provide a tour of the traditional Windows Forms API. One reason it is helpful to understand the original programming model: you can find many existing Windows Forms applications out there that will need to be maintained for some time to come. Also, many desktop GUIs simply might not require the horsepower offered by WPF. When you need to create more traditional business UIs that do not require an assortment of bells and whistles, the Windows Forms API can often fit the bill.

In this appendix, you will learn the Windows Forms programming model, work with the integrated designers of Visual Studio 2010, experiment with numerous Windows Forms controls, and receive an overview of graphics programming using GDI+. You will also pull this information together in a cohesive whole by wrapping things up in a (semi-capable) painting application.

■ **Note** Here's one proof that Windows Forms is not disappearing anytime soon: .NET 4.0 ships with a brand new Windows Forms assembly, `System.Windows.Forms.DataVisualization.dll`. You can use this library to incorporate charting functionality into your programs, complete with annotations; 3D rendering; and hit-testing support. This appendix will not cover this new .NET 4.0 Windows Forms API; however, you can look up the `System.Windows.Forms.DataVisualization.Charting` namespace if you want more information.

The Windows Forms Namespaces

The Windows Forms API consists of hundreds of types (e.g., classes, interfaces, structures, enums, and delegates), most of which are organized within various namespaces of the `System.Windows.Forms.dll` assembly. Figure A-1 shows these namespaces displayed in the Visual Studio 2010 object browser.

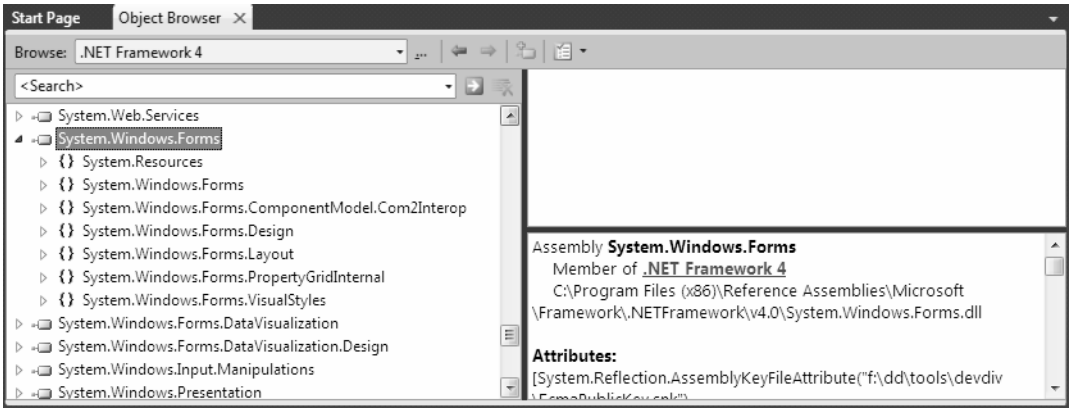

Figure A-1. *The namespaces of System.Windows.Forms.dll*

Far and away the most important Windows Forms namespace is `System.Windows.Forms`. At a high level, you can group the types within this namespace into the following broad categories:

- *Core infrastructure*: These are types that represent the core operations of a Windows Forms program (e.g., `Form` and `Application`) and various types to facilitate interoperability with legacy ActiveX controls, as well as interoperability with new WPF custom controls.
- *Controls*: These are types used to create graphical UIs (e.g., `Button`, `MenuStrip`, `ProgressBar`, and `DataGridView`), 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 may provide visual features to a Windows Forms program (e.g., `ToolTip` and `ErrorProvider`). Many components (e.g., the `Timer` and `System.ComponentModel.BackgroundWorker`) are not visible at runtime, but can be configured visually at design time.
- *Common dialog boxes*: Windows Forms provides several canned dialog boxes for common operations (e.g., `OpenFileDialog`, `PrintDialog`, and `ColorDialog`). 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. As you work through this appendix, however, you will gain a firm foundation that you can build on. In any case, be sure to check out the .NET Framework 4.0 SDK documentation for additional details.

Building a Simple Windows Forms Application

As you might expect, modern .NET IDEs (e.g., Visual Studio 2010, C# 2010 Express, and SharpDevelop) provide numerous form designers, visual editors, and integrated code-generation tools (wizards) to facilitate the construction of Windows Forms applications. These tools are extremely useful, but they can also hinder the process of learning Windows Forms, because these same tools tend to generate a good deal of boilerplate code that can obscure the core object model. Given this, you will create your first Windows Forms example using a Console Application project as a starting point.

Begin by creating a Console Application named SimpleWinFormsApp. Next, use the Project ➤ Add Reference menu option to set a reference to the `System.Windows.Forms.dll` and `System.Drawing.dll` assemblies through the .NET tab of the resulting dialog box. Next, update your `Program.cs` file with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

// The minimum required windows forms namespaces.
using System.Windows.Forms;

namespace SimpleWinFormsApp
{
  // This is the application object.
  class Program
  {
    static void Main(string[] args)
    {
      Application.Run(new MainWindow());
    }
  }

  // This is the main window.
  class MainWindow : Form {}
}
```

■ **Note** When Visual Studio 2010 finds a class that extends `System.Windows.Forms.Form`, it attempts to open the related GUI designer (provided this class is the first class in the C# code file). Double-clicking the `Program.cs` file from the Solution Explorer opens the designer, but don't do that yet! You will work with the Windows Forms designer in the next example; for now, be sure you right-click on the C# file containing your code within the Solution Explorer and select the View Code option.

This code represents the absolute simplest Windows Forms application you can build. At a bare minimum, you need a class that extends the `Form` base class and a `Main()` method to call the static `Application.Run()` method (you can find more details on `Form` and `Application` later in this chapter). Running your application now reveals that you have a resizable, minimizable, maximizable, and closable topmost window (see Figure A-2).

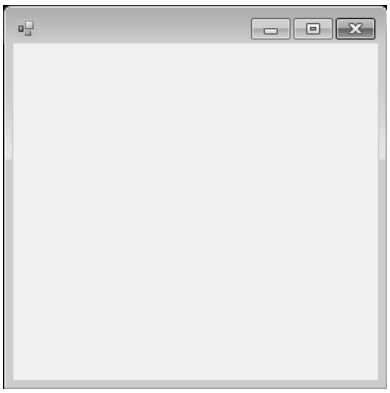

Figure A-2. *A simple Windows Forms application*

■ **Note** When you run this program, you will notice a command prompt looming in the background of your topmost window. This is because, when you create a Console Application, the `/target` flag sent to the C# compiler defaults to `/target:exe`. You can change this to `/target:winexe` (preventing the display of the command prompt) by double-clicking the Properties icon in the Solution Explorer and changing the Output Type setting to Windows Application using the Application tab.

Granted, the current application is not especially exciting, but it does illustrate how simple a Windows Forms application can be. To spruce things up a bit, you can add a custom constructor to your `MainWindow` class, which allows the caller to set various properties on the window to be displayed:

```
// This is the main window.
class MainWindow : Form
{
  public MainWindow() {}
  public MainWindow(string title, int height, int width)
  {
    // Set various properties from the parent classes.
    Text = title;
    Width = width;
    Height = height;
```

```
    // Inherited method to center the form on the screen.
    CenterToScreen();
  }
}
```

You can now update the call to `Application.Run()`, as follows:

```
static void Main(string[] args)
{
  Application.Run(new MainWindow("My Window", 200, 300));
}
```

This is a step in the right direction, but any window worth its salt requires various user interface elements (e.g., menu systems, status bars, and buttons) to allow for input. To understand how a `Form`-derived type can contain such elements, you must understand the role of the `Controls` property and the underlying controls collection.

Populating the Controls Collection

The `System.Windows.Forms.Control` base class (which is the inheritance chain of the `Form` type) defines a property named `Controls`. This property wraps a custom collection nested in the `Control` class named `ControlsCollection`. This collection (as the name suggests) references each UI element maintained by the derived type. Like other containers, this type supports several methods for inserting, removing, and finding a given UI widget (see Table A-1).

Table A-1. *ControlCollection Members*

Member	Meaning in Life
`Add()` `AddRange()`	You use these members to insert a new `Control`-derived type (or array of types) in the collection.
`Clear()`	This member removes all entries in the collection.
`Count`	This member returns the number of items in the collection.
`Remove()` `RemoveAt()`	You use these members to remove a control from the collection.

When you wish to populate the UI of a `Form`-derived type, you typically follow a predictable series of steps:

- Define a member variable of a given UI element within the `Form`-derived class.
- Configure the look and feel of the UI element.
- Add the UI element to the form's ControlsCollection container using a call to Controls.Add().

Assume you wish to update your `MainWindow` class to support a File ➤ Exit menu system. Here are the relevant updates, with code analysis to follow:

```
class MainWindow : Form
{
  // Members for a simple menu system.
  private MenuStrip mnuMainMenu = new MenuStrip();
  private ToolStripMenuItem mnuFile = new ToolStripMenuItem();
  private ToolStripMenuItem mnuFileExit = new ToolStripMenuItem();

  public MainWindow(string title, int height, int width)
  {
    ...
    // Method to create the menu system.
    BuildMenuSystem();
  }

  private void BuildMenuSystem()
  {
    // Add the File menu item to the main menu.
    mnuFile.Text = "&File";
    mnuMainMenu.Items.Add(mnuFile);

    // Now add the Exit menu to the File menu.
    mnuFileExit.Text = "E&xit";
    mnuFile.DropDownItems.Add(mnuFileExit);
    mnuFileExit.Click += (o, s) => Application.Exit();

    // Finally, set the menu for this Form.
    Controls.Add(this.mnuMainMenu);
    MainMenuStrip = this.mnuMainMenu;
  }
}
```

Notice that the `MainWindow` type now maintains three new member variables. The `MenuStrip` type represents the entirety of the menu system, while a given `ToolStripMenuItem` represents any topmost menu item (e.g., File) or submenu item (e.g., Exit) supported by the host.

You configure the menu system within the `BuildMenuSystem()` helper function. Notice that the text of each `ToolStripMenuItem` is controlled through the `Text` property; each menu item has been assigned a string literal that contains an embedded ampersand symbol. As you might already know, this syntax sets the Alt key shortcut. Thus, selecting Alt+F activates the File menu, while selecting Alt+X activates the Exit menu. Also notice that the File `ToolStripMenuItem` object (`mnuFile`) adds subitems using the `DropDownItems` property. The `MenuStrip` object itself adds a topmost menu item using the `Items` property.

Once you establish the menu system, you can add it to the controls collection (through the `Controls` property). Next, you assign your `MenuStrip` object to the form's `MainMenuStrip` property. This step might seem redundant, but having a specific property such as `MainMenuStrip` makes it possible to dynamically establish which menu system to show a user. You might change the menu displayed based on user preferences or security settings.

The only other point of interest is the fact that you handle the `Click` event of the File ➤ Exit menu; this helps you capture when the user selects this submenu. The `Click` event works in conjunction with a standard delegate type named `System.EventHandler`. This event can only call methods that take a

`System.Object` as the first parameter and a `System.EventArgs` as the second. Here, you use a lambda expression to terminate the entire application with the static `Application.Exit()` method.

Once you recompile and execute this application, you will find your simple window sports a custom menu system (see Figure A-3).

Figure A-3. *A simple window, with a simple menu system*

The Role of System.EventArgs and System.EventHandler

`System.EventHandler` is one of many delegate types used within the Windows Forms (and ASP.NET) APIs during the event-handling process. As you have seen, this delegate can only point to methods where the first argument is of type `System.Object`, which is a reference to the object that sent the event. For example, assume you want to update the implementation of the lambda expression, as follows:

```
mnuFileExit.Click += (o, s) =>
  {
    MessageBox.Show(string.Format("{0} sent this event", o.ToString()));
    Application.Exit();
  };
```

You can verify that the `mnuFileExit` type sent the event because the string is displayed within the message box:

```
"E&xit sent this event"
```

You might be wondering what purpose the second argument, `System.EventArgs`, serves. In reality, the `System.EventArgs` type brings little to the table because it simply extends `Object` and provides practically nothing by way of addition functionality:

```
public class EventArgs
{
  public static readonly EventArgs Empty;
  static EventArgs();
  public EventArgs();
}
```

However, this type is useful in the overall scheme of .NET event handling because it is the parent to many (useful) derived types. For example, the `MouseEventArgs` type extends `EventArgs` to provide details regarding the current state of the mouse. `KeyEventArgs` also extends `EventArgs` to provide details of the

state of the keyboard (such as which key was pressed); `PaintEventArgs` extends `EventArgs` to yield graphically relevant data; and so forth. You can also see numerous `EventArgs` descendents (and the delegates that make use of them) not only when working with Windows Forms, but when working with the WPF and ASP.NET APIs, as well.

While you could continue to build more functionality into your `MainWindow` (e.g., status bars and dialog boxes) using a simple text editor, you would eventually end up with hand cramps because you have to author all the grungy control configuration logic manually. Thankfully, Visual Studio 2010 provides numerous integrated designers that take care of these details on your behalf. As you use these tools during the remainder of this chapter, always remember that these tools authoring everyday C# code; there is nothing magical about them whatsoever.

■ **Source Code** You can find the `SimpleWinFormsApp` project under the Appendix A subdirectory.

The Visual Studio Windows Forms Project Template

When you wish to leverage the Windows Forms designer tools of Visual Studio 2010, your typically begin by selecting the Windows Forms Application project template using the File ➤ New Project menu option. To get comfortable with the core Windows Forms designer tools, create a new application named SimpleVSWinFormsApp (see Figure A-4).

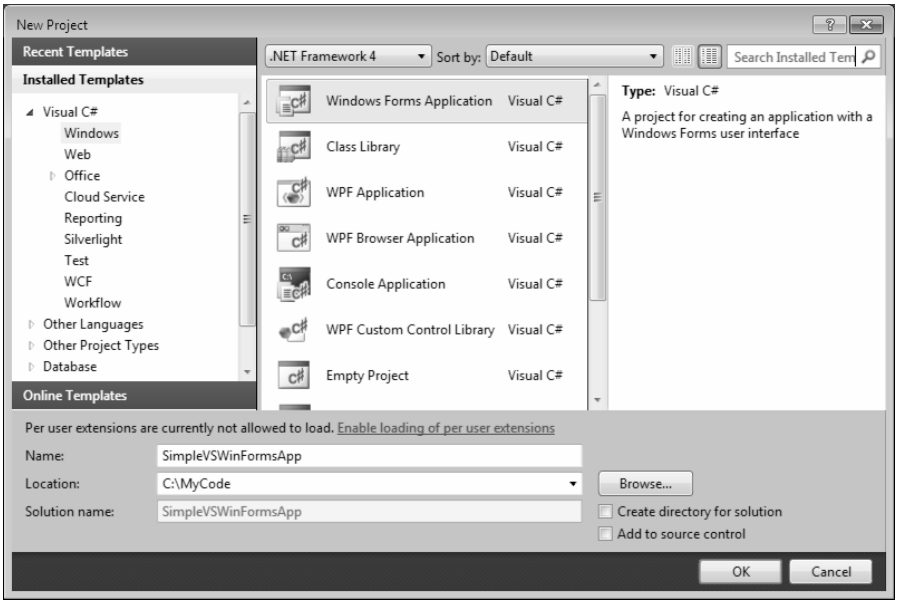

Figure A-4. *The Visual Studio Windows Forms Project Template*

The Visual Designer Surface

Before you begin to build more interesting Windows applications, you will re-create the previous example leveraging the designer tools. Once you create a new Windows Forms project, you will notice that Visual Studio 2010 presents a designer surface to which you can drag-and-drop any number of controls. You can use this same designer to configure the initial size of the window simply by resizing the form itself using the supplied grab handles (see Figure A-5).

Figure A-5. *The visual forms designer*

When you wish to configure the look-and-feel of your window (as well as any control placed on a form designer), you do so using the Properties window. Similar to a Windows Presentation Foundation project, this window can be used to assign values to properties, as well as to establish event handlers for the currently selected item on the designer (you select a configuration using the drop-down list box mounted on the top of the Properties window).

Currently, your form is devoid of content, so you see only a listing for the initial `Form`, which has been given a default name of `Form1`, as shown in the read-only `Name` property of Figure A-6.

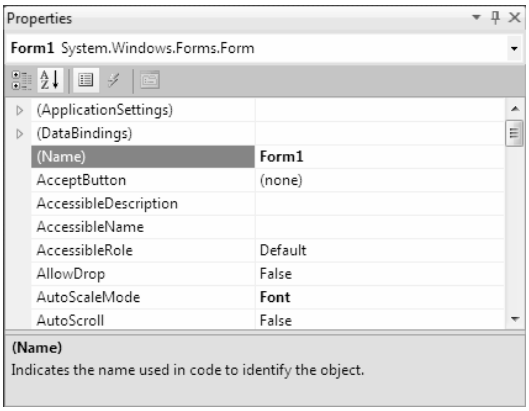

Figure A-6. *The Properties window for setting properties and handling events*

■ **Note** You can configure the Properties window to display its content by category or alphabetically using the first two buttons mounted beneath the drop-down list box. I'd suggest that you sort the items alphabetically to find a given property or event quickly.

The next designer element to be aware of is the Solution Explorer window. All Visual Studio 2010 projects support this window, but it is especially helpful when building Windows Forms applications to be able to (1) change the name of the file and related class for any window quickly, and (2) view the file that contains the designer-maintained code (you'll learn more information on this tidbit in just a moment). For now, right-click the `Form1.cs` icon and select the Rename option. Name this initial window to something more fitting: `MainWindow.cs`. The IDE will ask you if you wish to change the name of your initial class; it's fine to do this.

Dissecting the Initial Form

Before you build your menu system, you need to examine exactly what Visual Studio 2010 has created by default. Right-click the `MainWindow.cs` icon from the Solution Explorer window and select View Code. Notice that the form has been defined as a partial type, which allows a single type to be defined within multiple code files (see Chapter 5 for more information about this). Also, note that the form's constructor makes a call to a method named `InitializeComponent()` and your type *is-a* `Form`:

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

As you might be expecting, `InitializeComponent()` is defined in a separate file that completes the partial class definition. As a naming convention, this file always ends in `.Designer.cs`, preceded by the name of the related C# file containing the `Form`-derived type. Using the Solution Explorer window, open your `MainWindow.Designer.cs` file. Now, ponder the following code (this snippet strips out the code comments for simplicity; your code might differ slightly, based on the configurations you did in the Properties window):

```
partial class MainWindow
{
  private System.ComponentModel.IContainer components = null;

  protected override void Dispose(bool disposing)
  {
    if (disposing && (components != null))
    {
```

```
      components.Dispose();
    }
    base.Dispose(disposing);
  }

  private void InitializeComponent()
  {
    this.SuspendLayout();
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(422, 114);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
  }
}
```

The `IContainer` member variable and `Dispose()` methods are little more than infrastructure used by the Visual Studio designer tools. However, notice that the `InitializeComponent()` is present and accounted for. Not only is this method invoked by a form's constructor at runtime, Visual Studio makes use of this same method at design time to render correctly the UI seen on the Forms designer. To see this in action, change the value assigned to the `Text` property of the window to `"My Main Window"`. Once you activate the designer, the form's caption will update accordingly.

When you use the visual design tools (e.g., the Properties window or the form designer), the IDE updates `InitializeComponent()` automatically. To illustrate this aspect of the Windows Forms designer tools, ensure that the Forms designer is the active window within the IDE and find the `Opacity` property listed in the Properties window. Change this value to `0.8` (80%); this gives your window a slightly transparent look-and-feel the next time you compile and run your program. Once you make this change, reexamine the implementation of `InitializeComponent()`:

```
private void InitializeComponent()
{
...
  this.Opacity = 0.8;
}
```

For all practical purposes, you should ignore the `*.Designer.cs` files and allow the IDE to maintain them on your behalf when you build a Windows Forms application using Visual Studio. If you were to author syntactically (or logically) incorrect code within `InitializeComponent()`, you might break the designer. Also, Visual Studio often reformats this method at design time. Thus, if you were to add custom code to `InitializeComponent()`, the IDE might delete it! In any case, remember that each window of a Windows Forms application is composed using partial classes.

Dissecting the Program Class

Beyond providing implementation code for an initial `Form`-derived type, the Windows Application project types also provide a static class (named `Program`) that defines your program's entry point, `Main()`:

```
static class Program
{
```

```
  [STAThread]
  static void Main()
  {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new MainWindow());
  }
}
```

The `Main()` method invokes `Application.Run()` and a few other calls on the `Application` type to establish some basic rendering options. Last but not least, note that the `Main()` method has been adorned with the `[STAThread]` attribute. This informs the runtime that if this thread happens to create any classic COM objects (including legacy ActiveX UI controls) during its lifetime, the runtime must place these objects in a COM-maintained area termed the *single-threaded apartment*. In a nutshell, this ensures that the COM objects are thread-safe, even if the author of a given COM object did not explicitly include code to ensure this is the case.

Visually Building a Menu System

To wrap up this look at the Windows Forms visual designer tools and move on to some more illustrative examples, activate the Forms designer window, locate the Toolbox window of Visual Studio 2010, and find the `MenuStrip` control within the Menus & Toolbars node (see Figure A-7).

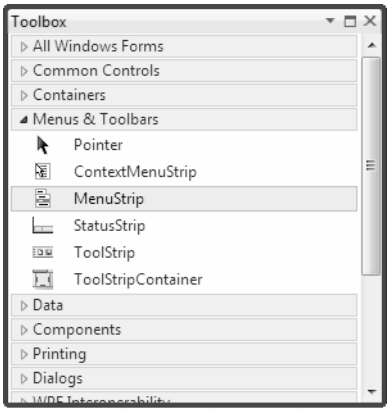

Figure A-7. *Windows Forms controls you can add to your designer surface*

Drag a `MenuStrip` control onto the top of your Forms designer. Notice that Visual Studio responds by activating the menu editor. If you look closely at this editor, you will notice a small triangle on the top-right of the control. Clicking this icon opens a context-sensitive inline editor that allows you to make numerous property settings at once (be aware that many Windows Forms controls have similar inline editors). For example, click the Insert Standard Items option, as shown in Figure A-8.

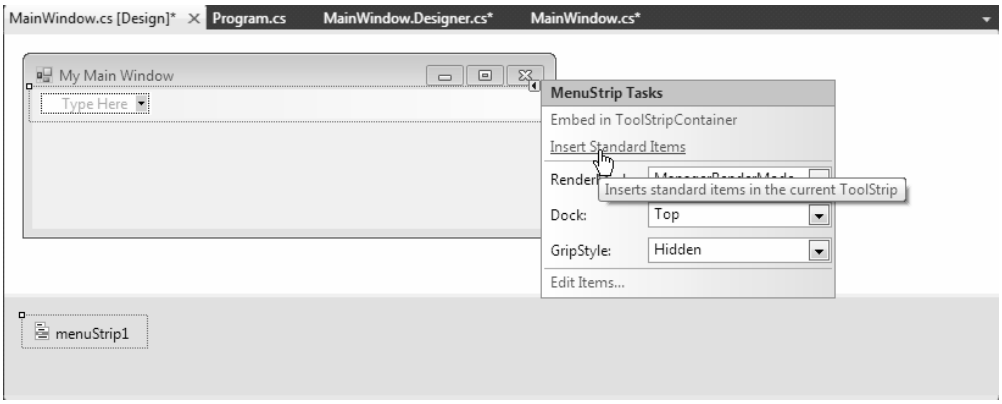

Figure A-8. *The inline menu editor*

In this example, Visual Studio was kind enough to establish an entire menu system on your behalf. Now open your designer-maintained file (`MainWindow.Designer.cs`) and note the numerous lines of code added to `InitializeComponent()`, as well as several new member variables that represent your menu system (designer tools are good things!). Finally, flip back to the designer and undo the previous operation by clicking the Ctrl+Z keyboard combination. This brings you back to the initial menu editor and removes the generated code. Using the menu designer, type in a topmost File menu item, followed by an Exit submenu (see Figure A-9).

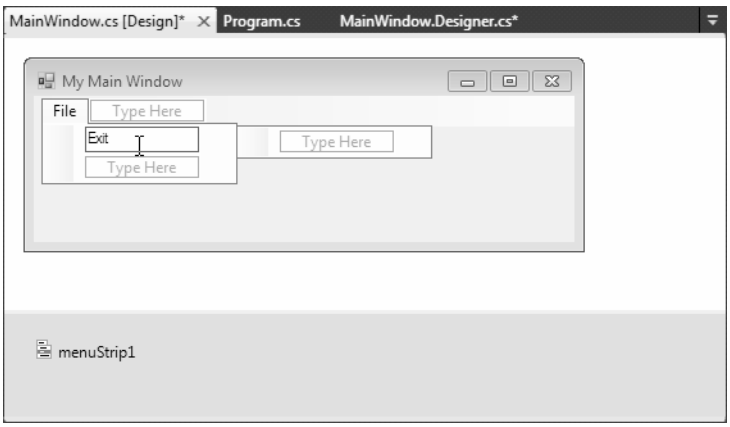

Figure A-9. *Manually building our menu system*

If you take a look at `InitializeComponent()`, you will find the same sort of code you authored by hand in the first example of this chapter. To complete this exercise, flip back to the Forms designer and click the lightning bolt button mounted on the Properties window. This shows you all of the events you can handle for the selected control. Make sure you select the Exit menu (named `exitToolStripMenuItem` by default), then locate the `Click` event (see Figure A-10).

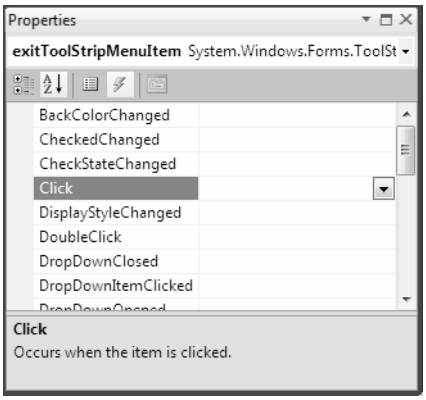

Figure A-10. *Establishing Events with the IDE*

At this point, you can enter the name of the method to be called when the item is clicked; or, if you feel lazy at this point, you can double-click the event listed in the Properties window. This lets the IDE pick the name of the event handler on your behalf (which follows the pattern, *NameOfControl_NameOfEvent()*).In either case, the IDE will create stub code, and you can fill in the implementation details:

```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    InitializeComponent();
    CenterToScreen();
  }
  private void exitToolStripMenuItem_Click(object sender, EventArgs e)
  {
    Application.Exit();
  }
}
```

If you so desire, you can take a quick peek at `InitializeComponent()`, where the necessary event riggings have also been accounted for:

```
this.exitToolStripMenuItem.Click +=
  new System.EventHandler(this.exitToolStripMenuItem_Click);
```

At this point, you probably feel more comfortable moving around the IDE when building Windows Forms applications. While there are obviously many additional shortcuts, editors, and integrated code wizards, this information is more than enough for you to press onward.

The Anatomy of a Form

So far you have examined how to build simple Windows Forms applications with (and without) the aid of Visual Studio; now it's time to examine the `Form` type in greater detail. In the world of Windows Forms, the `Form` type represents any window in the application, including the topmost main windows, child windows of a multiple document interface (MDI) application, as well as modal and modeless dialog boxes. As Figure A-11 shows, the `Form` type gathers a good deal of functionality from its parent classes and the numerous interfaces it implements.

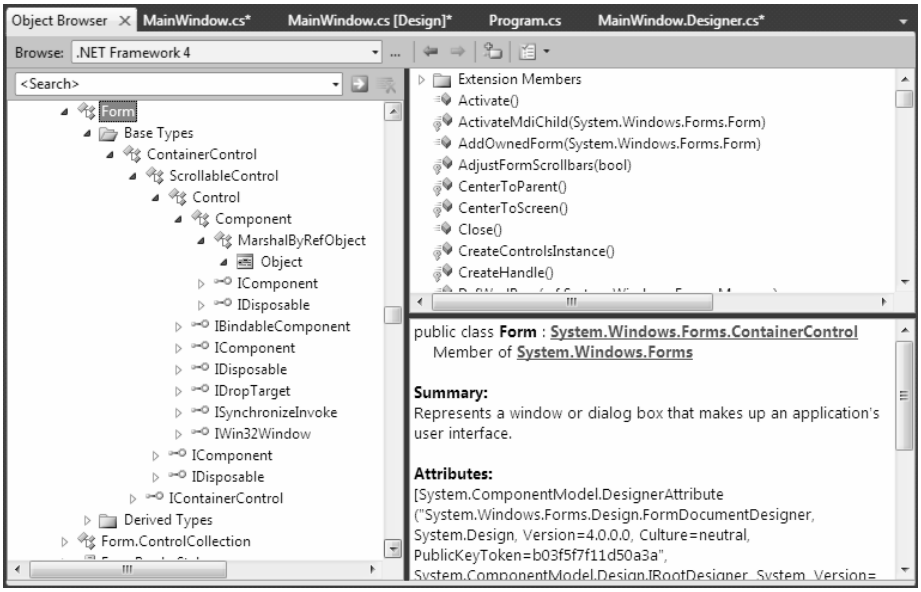

Figure A-11. *The inheritance chain of System.Windows.Forms.Form*

Table A-2 offers a high-level look at each parent class in the `Form`'s inheritance chain.

Table A-2. *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`	Types deriving from this class are accessed remotely through a *reference to* (not a local 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 it is not necessarily visible at runtime.

Table A-2. *Base Classes in the Form Inheritance Chain* (*continued*)

Parent Class	Meaning in Life
`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 horizontal and vertical scrollbars, as well as members, which allow you to manage the viewport shown within the scrollable region.
`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.

Although the complete derivation of a `Form` type involves numerous base classes and interfaces, you should keep in mind that you are *not* required to learn the role of each and every member of each and every parent class or implemented interface to be a proficient Windows Forms developer. In fact, you can easily set the majority of the members (specifically, properties and events) you use on a daily basis using the Visual Studio 2010 Properties window. That said, it is important that you understand the functionality provided by the `Control` and `Form` parent classes.

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 A-3 defines some properties of interest, which are grouped by related functionality.

Table A-3. *Core Properties of the Control Type*

Property	Meaning in Life

`BackColor`
`ForeColor`
`BackgroundImage`
`Font`
`Cursor`
	These properties define the core UI of the control (e.g., colors, font for text, and the mouse cursor to display when the mouse is over the widget).

`Anchor`
`Dock`
`AutoSize`
	These properties control how the control should be positioned within the container.

Table A-3. *Core Properties of the Control Type (continued)*

Property	Meaning in Life

`Top`
`Left`
`Bottom`
`Right`
`Bounds`
`ClientRectangle`
`Height`
`Width`
	These properties specify the current dimensions of the control.

`Enabled`
`Focused`
`Visible`
	These properties encapsulate a Boolean that specifies the state of the current control.

`ModifierKeys`	This static property checks the current state of the modifier keys (e.g., 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`	You use these properties to configure the tab order of the control.
`Opacity`	This property determines the opacity of the control (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 (e.g., `ControlsCollection`) that contains any child controls within the current control.

As you might 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 A-4 lists some events of interest, grouping them by related functionality.

Table A-4. *Events of the Control Type*

Event	Meaning in Life

`Click`
`DoubleClick`
`MouseEnter`
`MouseLeave`
`MouseDown`
`MouseUp`
`MouseMove`
`MouseHover`
`MouseWheel`
	These events that let you interact with the mouse.

`KeyPress`
`KeyUp`
`KeyDown`
	These events let you interact with the keyboard.

`DragDrop`
`DragEnter`
`DragLeave`
`DragOver`
	You use these events to monitor drag-and-drop activity.

`Paint`	This event lets you to interact with the graphical rendering services of GDI+.

Finally, the `Control` base class also defines a several methods that allow you to interact with any `Control`-derived type. As you examine the methods of the `Control` type, notice that a many of them have an `On` prefix, followed by the name of a specific event (e.g., `OnMouseMove`, `OnKeyUp`, and `OnPaint`). 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 post-processing of the event before (or after) invoking the parent's default implementation:

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

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

This can be helpful in some circumstances (especially if you want to build a custom control that derives from a standard control), but you will often handle events using the standard C# event syntax (in fact, this is the default behavior of the Visual Studio designers). When you handle events in this manner, the framework calls your custom event handler once the parent's implementation has completed. For example, this code lets you manually handle the `MouseDown` event:

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

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

You should also be aware of a few other methods, in addition to the just described `OnXXX()` methods:

- `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 (you learn more about graphical rendering using GDI+ later in this chapter).

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 A-5.

Table A-5. *Properties of the Form Type*

Property	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 that indicates whether the form has a control box (e.g., the minimize, maximize, and close icons in the upper right of a window).
`FormBorderStyle`	Gets or sets the border style of the form. You use this in conjunction with the `FormBorderStyle` enumeration.

Table A-5. *Properties of the Form Type (continued)*

Property	Meaning in Life
`Menu`	Gets or sets the menu to dock on the form.
`MaximizeBox` `MinimizeBox`	Used to determine whether this form will enable the maximize and minimize boxes.
`ShowInTaskbar`	Determines whether 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. You use this in conjunction with the `FormWindowState` enumeration.

In addition to numerous `On`-prefixed default event handlers, Table A-6 provides a list of some core methods defined by the `Form` type.

Table A-6. *Key Methods of the Form Type*

Method	Meaning in Life
`Activate()`	Activates a given form and gives it focus.
`Close()`	Closes the current form.
`CenterToScreen()`	Places the form in the center of the screen
`LayoutMdi()`	Arranges each child form (as specified by the `MdiLayout` enumeration) within the parent form.
`Show()`	Displays a form as a modeless window.
`ShowDialog()`	Displays a form as a modal dialog box.

Finally, the `Form` class defines a number of events, many of which fire during the form's lifetime (see Table A-7).

Table A-7. *Select Events of the Form Type*

Event	Meaning in Life
`Activated`	This event occurs whenever the form is *activated*, which means that the form has been given the current focus on the desktop.
`FormClosed` `FormClosing`	You use these events to determine when the form is about to close or has closed.
`Deactivate`	This event occurs whenever the form is *deactivated*, which means the form has lost the current focus on the desktop.
`Load`	This event occurs after the form has been allocated into memory, but is not yet visible on the screen.
`MdiChildActive`	This event is 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 WPF, you know that *window types* have many 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 class 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 (e.g., `ListBoxes` and `TreeViews`), or allocate resources used during the `Form`'s operation (e.g., database connections and proxies to remote objects).

Once the `Load` event fires, the next event to fire is `Activated`. This event fires when the form receives the 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 the 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` object as the user navigates between active windows and applications.

Two events fire when the user chooses to close a given form: `FormClosing` and `FormClosed`. The `FormClosing` event is fired first and is an ideal place to prompt the end user with the much hated (but useful) message: "Are you *sure* you wish to close this application?" This step gives the user a chance to save any application-centric data before terminating the program.

The `FormClosing` event works in conjunction with the `FormClosingEventHandler` delegate. If you set the `FormClosingEventArgs.Cancel` property to `true`, you prevent the window from being destroyed and instruct it to return to normal operation. If you set `FormClosingEventArgs.Cancel` to `false`, the `FormClosed` event fires, and the Windows Forms application exits, which unloads the AppDomain and terminates the process.

This snippet updates your form's constructor and handles the `Load`, `Activated`, `Deactivate`, `FormClosing`, and `FormClosed` events (you might recall from Chapter 11 that the IDE will autogenerate the correct delegate and event handler when you press the Tab key twice after typing `+=`):

```
public MainWindow()
{
```

```
  InitializeComponent();

  // Handle various lifetime events.
  FormClosing += new FormClosingEventHandler(MainWindow_Closing);
  Load += new EventHandler(MainWindow_Load);
  FormClosed += new FormClosedEventHandler(MainWindow_Closed);
  Activated += new EventHandler(MainWindow_Activated);
  Deactivate += new EventHandler(MainWindow_Deactivate);
}
```

Within the `Load`, `FormClosed`, `Activated`, and `Deactivate` event handlers, you must update the value of a new `Form`-level `string` member variable (named `lifeTimeInfo`) with a simple message that displays the name of the event that has just been intercepted. Begin by adding this member to your Form derived class:

```
public partial class MainWindow : Form
{
  private string lifeTimeInfo = "";
...
}
```

The next step is to implement the event handlers. Notice that you display the value of the `lifeTimeInfo` string within a message box in the `FormClosed` event handler:

```
private void MainWindow_Load(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Load event\n";
}

private void MainWindow_Activated(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Activate event\n";
}

private void MainWindow_Deactivate(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Deactivate event\n";
}

private void MainWindow_Closed(object sender, FormClosedEventArgs e)
{
  lifeTimeInfo += "FormClosed event\n";
  MessageBox.Show(lifeTimeInfo);
}
```

Within the `FormClosing` event handler, you prompt the user to ensure that she wishes to terminate the application using the incoming `FormClosingEventArgs`. In the following code, the `MessageBox.Show()` method returns a `DialogResult` type that contains a value representing which button has been selected by the end user. Here, you craft a message box that displays Yes and No buttons; therefore, you want to discover whether the return value from `Show()` is `DialogResult.No`:

```
private void MainWindow_Closing(object sender, FormClosingEventArgs e)
{
```

```
  lifeTimeInfo += "FormClosing event\n";

  // Show a message box with Yes and No buttons.
  DialogResult dr = MessageBox.Show("Do you REALLY want to close this app?",
    "Closing event!", MessageBoxButtons.YesNo);

  // Which button was clicked?
  if (dr == DialogResult.No)
    e.Cancel = true;
  else
    e.Cancel = false;
}
```

Let's make one final adjustment. Currently, the File ➤ Exit menu destroys the entire application, which is a bit aggressive. More often, the File ➤ Exit handler of a top-most window calls the inherited `Close()` method, which fires the close-centric events and then tears down the application:

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

Now run your application and shift the form into and out of focus a few times (to trigger the `Activated` and `Deactivate` events). When you eventually shut down the application, you will see a message box that looks something like the message shown in Figure A-12.

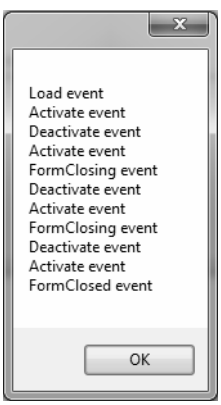

Figure A-12. *The life and times of a Form-derived type*

■ **Source Code** You can find the SimpleVSWinFormsApp project under the Appendix A subdirectory.

Responding to Mouse and Keyboard Activity

You might recall that the `Control` parent class defines a set of events that allow you to monitor mouse and keyboard activity in a variety of manners. To check this out firsthand, create a new Windows Forms Application project named MouseAndKeyboardEventsApp, rename the initial form to `MainWindow.cs` (using the Solution Explorer), and handle the `MouseMove` event using the Properties window. These steps generate the following event handler:

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

  // Generated via the Properties window.
  private void MainWindow_MouseMove(object sender, MouseEventArgs e)
  {
  }
}
```

The `MouseMove` event works in conjunction with the `System.Windows.Forms.MouseEventHandler` delegate. This delegate can only call methods where the first parameter is a `System.Object`, while the second is of type `MouseEventArgs`. This type contains various members that provide detailed information about the state of the event when mouse-centric events occur:

```
public class MouseEventArgs : EventArgs
{
  public MouseEventArgs(MouseButtons button, int clicks, int x,
    int y, int delta);

  public MouseButtons Button { get; }
  public int Clicks { get; }
  public int Delta { get; }
  public Point Location { get; }
  public int X { get; }
  public int Y { get; }
}
```

Most of the public properties are self-explanatory, but Table A-8 provides more specific details.

Table A-8. *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.

Table A-8. *Properties of the MouseEventArgs Type (continued)*

Property	Meaning in Life
`Delta`	Gets a signed count of the number of detents (which represents a single notch of the mouse wheel) for the current mouse rotation.
`Location`	Returns a `Point` that contains the current X and Y location of the mouse.
`X`	Gets the *x*-coordinate of a mouse click.
`Y`	Gets the *y*-coordinate of a mouse click.

Now it's time to implement your `MouseMove` handler to display the current X- and Y-position of the mouse on the `Form`'s caption; you do this using the `Location` property:

```
private void MainWindow_MouseMove(object sender, MouseEventArgs e)
{
  Text = string.Format("Mouse Position: {0}", e.Location);
}
```

When you run the application and move the mouse over the window, you find the position displayed on the title area of your `MainWindow` type (see Figure A-13).

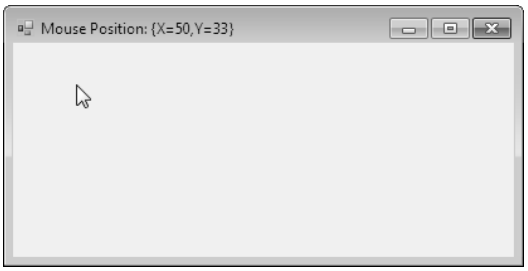

Figure A-13. *Intercepting mouse movements*

Determining Which Mouse Button Was Clicked

Another common mouse-centric detail to attend to is determining which button has been clicked when a `MouseUp`, `MouseDown`, `MouseClick`, or `MouseDoubleClick` event occurs. When you wish to determine exactly which button was clicked (whether 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:

```
public enum MouseButtons
{
  Left,
```

```
  Middle,
  None,
  Right,
  XButton1,
  XButton2
}
```

■ **Note** The `XButton1` and `XButton2` values allow you to capture forward and backwards navigation buttons that are supported on many mouse-controller devices.

You can see this in action by handling the `MouseDown` event on your `MainWindow` type using the Properties window. The following `MouseDown` event handler displays which mouse button was clicked inside a message box:

```
private void MainWindow_MouseDown (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!");
}
```

Determining Which Key Was Pressed

Windows applications typically define numerous input controls (e.g., the `TextBox`) where the user can enter information using the keyword. When you capture keyboard input in this manner, you do not need to handle keyboard events explicitly because you can extract the textual data from the control using various properties (e.g., the `Text` property of the `TextBox` type).

However, if you need to monitor keyboard input for more exotic purposes (e.g., filtering keystrokes on a control or capturing keypresses on the form itself), the base class libraries provide the `KeyUp` and `KeyDown` events. These 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. You define this type like this:

```
public class KeyEventArgs : EventArgs
{
  public KeyEventArgs(Keys keyData);

  public virtual bool Alt { get; }
  public bool Control { get; }
  public bool Handled { get; set; }
  public Keys KeyCode { get; }
  public Keys KeyData { get; }
```

```
  public int KeyValue { get; }
  public Keys Modifiers { get; }
  public virtual bool Shift { get; }
  public bool SuppressKeyPress { get; set; }
}
```

Table A-9 documents some of the more interesting properties supported by `KeyEventArgs`.

Table A-9. *Properties of the KeyEventArgs Type*

Property	Meaning in Life
`Alt`	Gets a value that indicates whether the Alt key was pressed.
`Control`	Gets a value that indicates whether the Ctrl key was pressed.
`Handled`	Gets or sets a value that indicates 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 (e.g., Ctrl, Shift, and/or Alt) were pressed.
`Shift`	Gets a value that indicates whether the Shift key was pressed.

You can see this in action by handling the `KeyDown` event as follows:

```
private void MainWindow_KeyDown(object sender, KeyEventArgs e)
{
  Text = string.Format("Key Pressed: {0} Modifiers: {1}",
                       e.KeyCode.ToString(), e.Modifiers.ToString());
}
```

Now compile and run your program. You should be able to determine which mouse button was clicked, as well as which keyboard key was pressed. For example, Figure A-14 shows the result of pressing the Ctrl and Shift keys simultaneously.

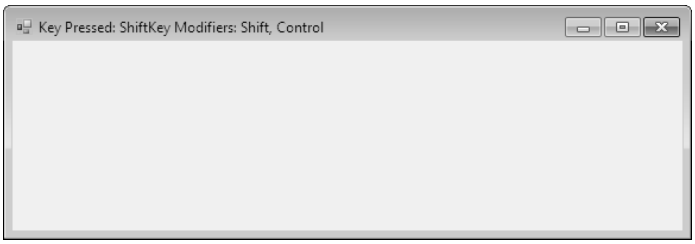

Figure A-14. *Intercepting keyboard activity*

■ **Source Code** You can find the `MouseAndKeyboardEventsApp` project under the Appendix A subdirectory.

Designing Dialog Boxes

Within a graphical user interface program, dialog boxes tend to be the primary way to capture user input for use within the application itself. Unlike other GUI APIs you might have used previously, there Windows Forms has no `Dialog` base class. Rather, dialog boxes under Windows Forms are simply types that derive from the `Form` class.

In addition, many dialog boxes are intended to be nonsizable; therefore, you typically want to set the `FormBorderStyle` property to `FormBorderStyle.FixedDialog`. Also, 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 taskbar.

Let's look at how to build and manipulate dialog boxes. Begin by creating a new Windows Forms Application project named CarOrderApp and rename the initial `Form1.cs` file to `MainWindow.cs` using Solution Explorer. Next, use the Forms designer to create a simple File ➤ Exit menu, as well as a Tool ➤ Order Automobile... menu item (remember: you create a menu by dragging a MenuStrip from the Toolbox and then configuring the menu items in the designer window). Once you do this, handle the `Click` event for the Exit and Order Automobile submenus using the Properties window.

You implement the File ➤ Exit menu handler so it terminates the application with a call to `Close()`:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
  Close();
}
```

Now use the Project menu of Visual Studio to select the Add Windows Forms menu option and name your new form `OrderAutoDialog.cs` (see Figure A-15).

For this example, design a dialog box that has the expected OK and Cancel buttons (named `btnOK` and `btnCancel`, respectively), as well as three `TextBox` controls named `txtMake`, `txtColor`, and `txtPrice`. Now use the Properties window to finalize the design of your dialog box, as follows:

- Set the `FormBorderStyle` property to `FixedDialog`.
- Set the `MinimizeBox` and `MaximizeBox` properties to `false`.
- Set the `StartPosition` property to `CenterParent`.
- Set the `ShowInTaskbar` property to `false`.

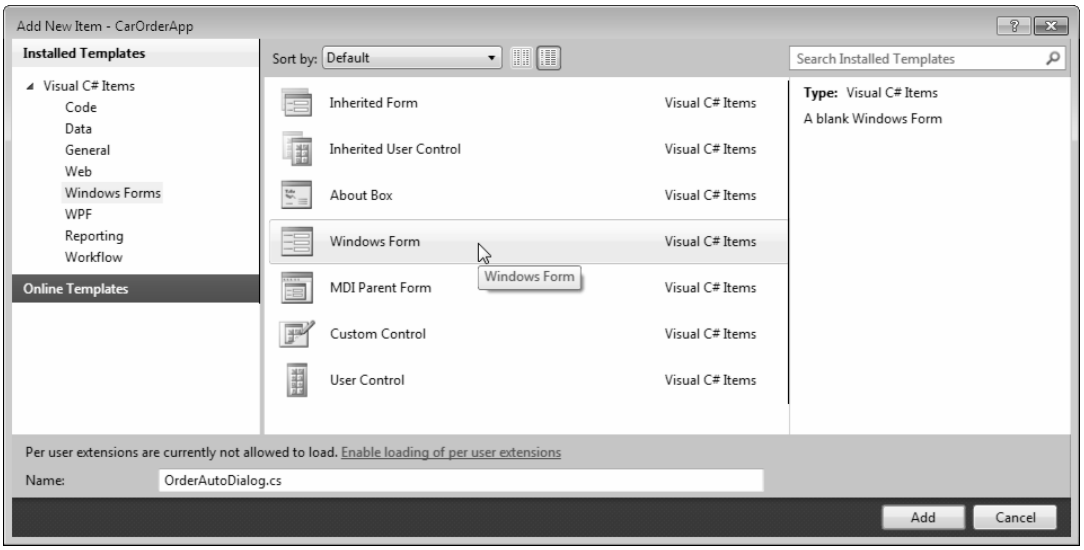

Figure A-15. *Inserting new dialog boxes using Visual Studio*

The DialogResult Property

Finally, select the OK button and use the Properties window to set the `DialogResult` property to `OK`. Similarly, you can set the `DialogResult` property of the Cancel button to (you guessed it) `Cancel`. As you will see in a moment, the `DialogResult` property is quite useful because it enables the launching form to determine quickly which button the user has clicked; this enables you to take the appropriate action. You can set the `DialogResult` property to any value from the related `DialogResult` enumeration:

```
public enum DialogResult
{
  Abort, Cancel, Ignore, No,
  None, OK, Retry, Yes
}
```

Figure A-16 shows one possible design of your dialog box; it even adds in a few descriptive `Label` controls.

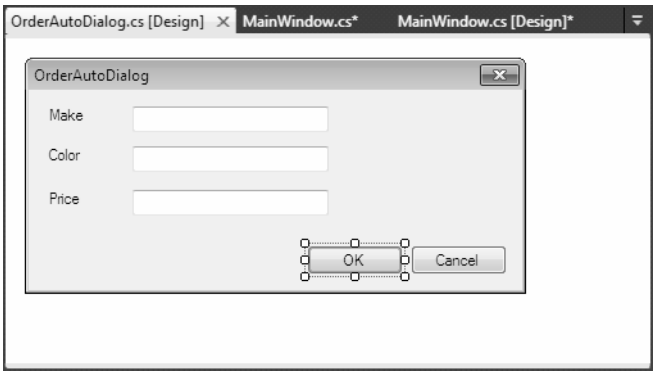

Figure A-16. *The OrderAutoDialog type*

Configuring the Tab Order

You have created a somewhat interesting dialog box; the next step is formalize the tab order. As you might know, users expect to be able to shift focus using the Tab key when a form contains multiple GUI widgets. Configuring the tab order for your set of controls requires that you understand two key properties: `TabStop` and `TabIndex`.

You can set the `TabStop` property to `true` or `false`, based on whether or not you wish this GUI item to be reachable using the Tab key. Assuming that you set the `TabStop` property to `true` for a given control, you can use the `TabOrder` property to establish the order of activation in the tabbing sequence (which is zero-based), as in this example:

```
// Configure tabbing properties.
txtMake.TabIndex = 0;
txtMake.TabStop = true;
```

The Tab Order Wizard

You can set the `TabStop` and `TabIndex` manually using the Properties window; however, the Visual Studio 2010 IDE supplies a Tab Order Wizard that you can 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 prefer the controls to tab (see Figure A-17).

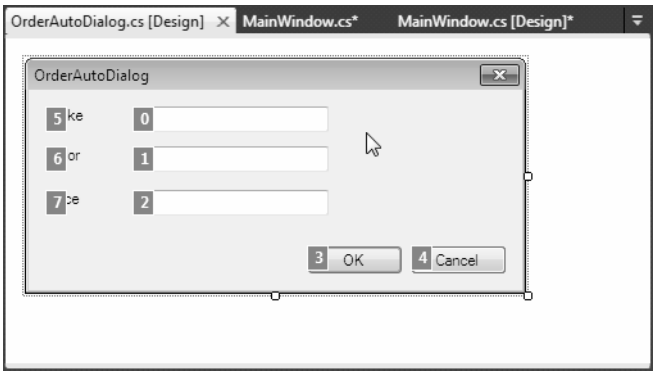

Figure A-17. *The Tab Order Wizard*

You can exit the Tab Order Wizard by pressing the Esc key.

Setting the Form's Default Input Button

Many user-input forms (especially dialog boxes) have a particular `button` that automatically responds to the user pressing the Enter key. Now assume that you want the `Click` event handler for `btnOK` invoked when the user presses the Enter key. Doing so is as simple as setting the form's `AcceptButton` property as follows (you can establish this same setting using the Properties window):

```
public partial class OrderAutoDialog : Form
{
  public OrderAutoDialog()
  {
    InitializeComponent();

    // When the Enter key is pressed, it is as if
    // the user clicked the btnOK button.
    this.AcceptButton = btnOK;
  }
}
```

■ **Note** Some forms require the ability to simulate clicking the form's Cancel button when the user presses the Esc key. You can accomplish this by assigning the `CancelButton` property of the `Form` to the `Button` object that represents the clicking of the Cancel button.

Displaying Dialog Boxes

When you wish to display a dialog box, you must first decide whether you wish to launch the dialog box in a *modal* or *modeless* fashion. As you might know, modal dialog boxes must be dismissed by the user before he can return to the window that launched the dialog box in the first place; for example, most About boxes are modal in nature. To show a modal dialog box, call `ShowDialog()` off your dialog box object. On the other hand, you can display a modeless dialog box by calling `Show()`, which allows the user to switch focus between the dialog box and the main window (e.g., a Find/Replace dialog box).

For this example, you want to update the Tools ➤ Order Automobile... menu handler of the `MainWindow` type to show the `OrderAutoDialog` object in a modal manner. Consider the following initial code:

```
private void orderAutomobileToolStripMenuItem_Click(object
    sender, EventArgs e)
{
  // Create your dialog object.
  OrderAutoDialog dlg = new OrderAutoDialog();

  // Show as modal dialog box, and figure out which button
  // was clicked using the DialogResult return value.
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    // They clicked OK, so do something...
  }
}
```

■ **Note** You can optionally call the `ShowDialog()` and `Show()` methods by specifying an object that represents the owner of the dialog box (which for the form loading the dialog box would be represented by `this`). Specifying the owner of a dialog box establishes the z-ordering of the form types and also ensures (in the case of a modeless dialog box) that all owned windows are also disposed of when the main window is destroyed.

Be aware that when you create an instance of a `Form`-derived type (`OrderAutoDialog` in this case), the dialog box is *not* visible on the screen, but simply allocated into memory. It is not until you call `Show()` or `ShowDialog()` that the form becomes visible. Also, notice that `ShowDialog()` returns the `DialogResult` value that has been assigned to the button (the `Show()` method simply returns `void`).

Once `ShowDialog()` returns, the form is no longer visible on the screen, but is still in memory. This means you can extract the values in each `TextBox`. However, you will receive compiler errors if you attempt to compile the following code:

```
private void orderAutomobileToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Create your dialog object.
  OrderAutoDialog dlg = new OrderAutoDialog();

  // Show as modal dialog box, and figure out which button
```

```
  // was clicked using the DialogResult return value.
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    // Get values in each text box? Compiler errors!
    string orderInfo = string.Format("Make: {0}, Color: {1}, Cost: {2}",
      dlg.txtMake.Text, dlg.txtColor.Text, dlg.txtPrice.Text);
    MessageBox.Show(orderInfo, "Information about your order!");
  }
}
```

The reason you get compiler errors is that Visual Studio 2010 declares the controls you add to the Forms designer as *private* member variables of the class! You can verify this fact by opening the `OrderAutoDialog.Designer.cs` file.

While a prim-and-proper dialog box might preserve encapsulation by adding public properties to get and set the values within these text boxes, you can take a shortcut and redefine them by using the `public` keyword. To do so, select each `TextBox` on the designer, and then set the `Modifiers` property of the control to `Public` using the Properties window. This changes the underlying designer code:

```
partial class OrderAutoDialog
{
...
  // Form member variables are defined within the designer-maintained file.
  public System.Windows.Forms.TextBox txtMake;
  public System.Windows.Forms.TextBox txtColor;
  public System.Windows.Forms.TextBox txtPrice;
}
```

At this point, you can compile and run your application. Once you launch your dialog box, you should be able to see the input data displayed within a message box (provided you click the OK button).

Understanding Form Inheritance

So far, each one of your custom windows/dialog boxes in this chapter has derived directly from `System.Windows.Forms.Form`. However, one intriguing aspect of Windows Forms development is the fact that `Form` types can function as the base class to derived `Forms`. For example, assume you create a .NET code library that contains each of your company's core dialog boxes. Later, you decide that your company's About box is a bit on the bland side, and you wish to add a 3D image of your company logo. Rather than having to re-create the entire About box, you can extend the basic About box, thereby inheriting the core look-and-feel:

```
// ThreeDAboutBox "is-a" AboutBox
public partial class ThreeDAboutBox : AboutBox
{
  // Add code to render company logo...
}
```

To see form inheritance in action, insert a new form into your project using the Project ➤ Add Windows Form menu option. This time, however, pick the Inherited Form icon, and name your new form `ImageOrderAutoDialog.cs` (see Figure A-18).

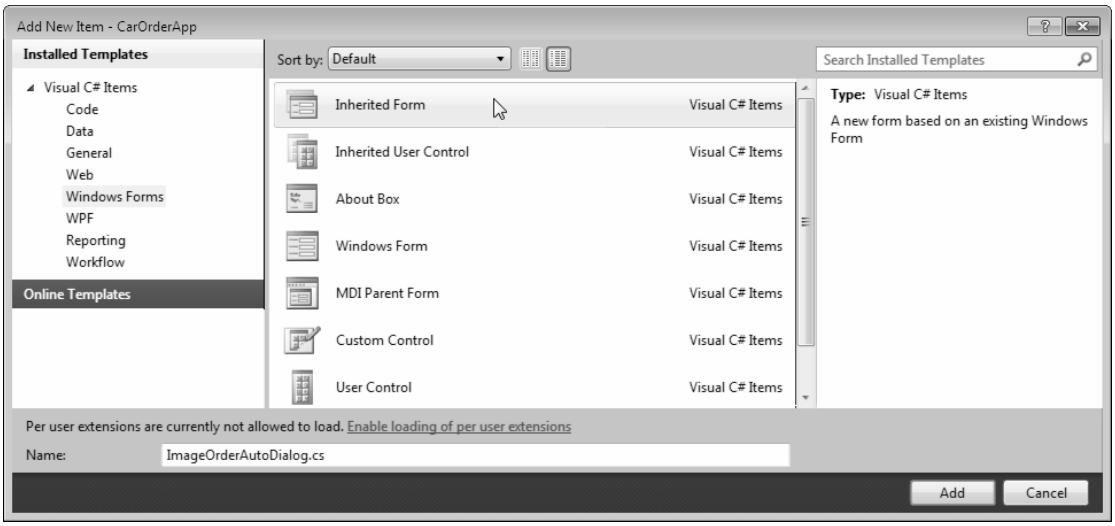

Figure A-18. *Adding a derived form to your project*

This option brings up the Inheritance Picker dialog box, which shows you each of the forms in your current project. Notice that the Browse button allows you to pick a form in an external .NET assembly. Here, simply pick your `OrderAutoDialog` class.

■ **Note** You must compile your project at least one time to see the forms of your project in the Inheritance Picker dialog box because this tool reads from the assembly metadata to show you your options.

Once you click the OK button, the visual designer tools show each of the base controls on your parent controls; each control has a small arrow icon mounted on the upper-left of the control (symbolizing inheritance). To complete your derived dialog box, locate the `PictureBox` control from the Common Controls section of the Toolbox and add one to your derived form. Next, use the `Image` property to select an image file of your choosing. Figure A-19 shows one possible UI, using a (crude) hand drawing of a junker automobile (a lemon!).

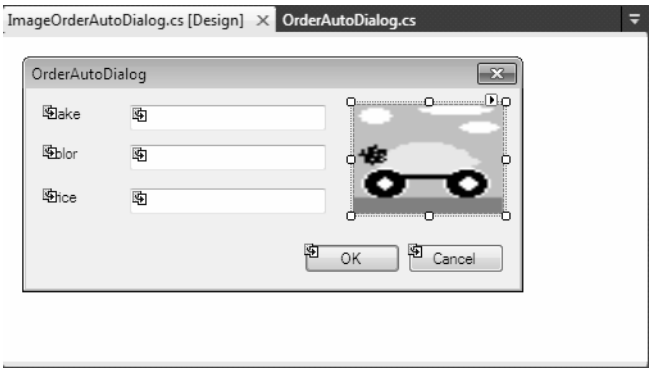

Figure A-19. *The UI of the ImageOrderAutoDialog class*

With this, you can now update the Tools ➤ Order Automobile... `Click` event handler to create an instance of your derived type, rather than the `OrderAutoDialog` base class:

```
private void orderAutomobileToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Create the derived dialog object.
  ImageOrderAutoDialog dlg = new ImageOrderAutoDialog();
...
}
```

■ **Source Code** You can find the CarOrderApp project under the Appendix A subdirectory.

Rendering Graphical Data Using GDI+

Many GUI applications require the ability to generate graphical data dynamically for display on the surface of a window. For example, perhaps you have selected a set of records from a relational database and wish to render a pie chart (or bar chart) that visually shows items in stock. Or, perhaps you want to re-create some old-school video game using the .NET platform. Regardless of your goal, GDI+ is the API to use when you need to render data graphically within a Windows Forms application. This technology is bundled within the `System.Drawing.dll` assembly, which defines a number of namespaces (see Figure A-20).

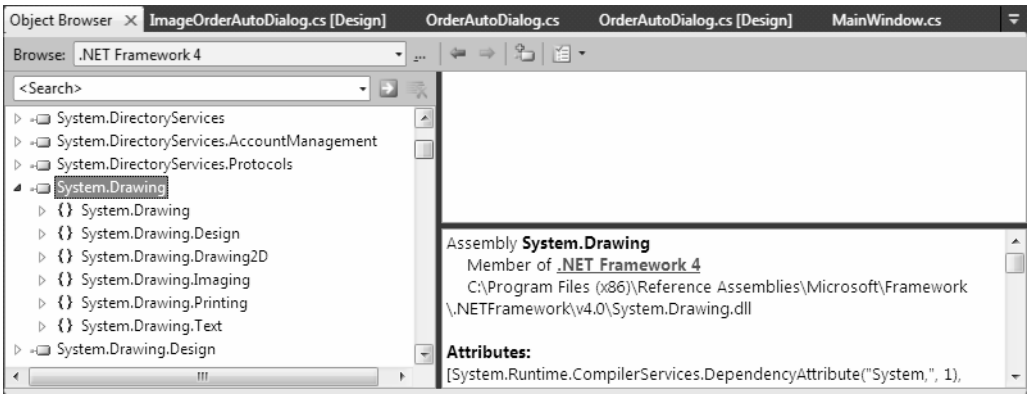

Figure A-20. *The namespaces of System.Drawing.dll*

■ **Note** A friendly reminder: WPF has its own graphical rendering subsystem and API; you use GDI+ only within a Windows Forms application.

Table A-10 documents the role of the key GDI+ namespaces at a high level.

Table A-10. *Core GDI+ Namespaces*

Namespace	Meaning in Life
`System.Drawing`	This is the core GDI+ namespace that defines numerous types for basic rendering (e.g., fonts, pens, and basic brushes), as well as the almighty `Graphics` type.
`System.Drawing.Drawing2D`	This namespace provides types used for more advanced 2D/vector graphics functionality (e.g., gradient brushes, pen caps, and geometric transforms).
`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.

The System.Drawing Namespace

You can find the vast majority of the types you'll use when programming GDI+ applications in the `System.Drawing` namespace. As you might expect, you can find classes that represent images, brushes, pens, and fonts. `System.Drawing` also defines a several related utility types, such as `Color`, `Point`, and `Rectangle`. Table A-11 lists some (but not all) of the core types.

Table A-11. *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`
	You use brush objects to fill the interiors of graphical shapes, such as rectangles, ellipses, and polygons.

`BufferedGraphics`	This type provides a graphics buffer for double buffering, which you use 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 you use to obtain specific colors for the construction of various pens/brushes.
`Font` `FontFamily`	The `Font` type encapsulates the characteristics of a given font (e.g., type name, bold, italic, and point size). `FontFamily` provides an abstraction for a group of fonts that have a similar design, but also certain variations in style.
`Graphics`	This core class represents a valid drawing surface, as well as several methods to render text, images, and geometric patterns.
`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 you use to draw lines and curves. The `Pens` type defines several 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.

Table A-11. *Core Types of the System.Drawing Namespace (continued)*

Type	Meaning in Life
`Rectangle` `RectangleF`	These structures represent a rectangular dimension (again, these map to an underlying integer or float).
`Size` `SizeF`	These structures represent a given height/width (again, these map to an underlying integer or float).
`StringFormat`	You use this type to encapsulate various features of textual layout (e.g., alignment and line spacing).
`Region`	This type describes the interior of a geometric image composed of rectangles and paths.

The Role of the Graphics Type

The `System.Drawing.Graphics` class serves as 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 a region of memory), but also defines dozens of members that allow you to render text, images (e.g., icons and bitmaps), and numerous geometric patterns. Table A-12 gives a partial list of members.

Table A-12. *Select Members of the Graphics Class*

Method	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 and bitmap) or GUI control.

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

`DrawArc()`
`DrawBeziers()`
`DrawCurve()`
`DrawEllipse()`
`DrawIcon()`
`DrawLine()`
`DrawLines()`
`DrawPie()`
`DrawPath()`
`DrawRectangle()`
`DrawRectangles()`
`DrawString()`
	You use these methods to render a given image or geometric pattern. All `DrawXXX()` methods require that you use GDI+ `Pen` objects.

Table A-12. *Select Members of the Graphics Class (continued)*

Method	Meaning in Life

`FillEllipse()`
`FillPie()`
`FillPolygon()`
`FillRectangle()`
`FillPath()`
	You use these methods to fill the interior of a given geometric shape. All `FillXXX()` methods require that you use GDI+ `Brush` objects.

Note that you cannot create the `Graphics` class directly using the `new` keyword because there are no publicly defined constructors. So, how do you obtain a valid `Graphics` object? I'm glad you asked!

Obtaining a Graphics Object with the Paint Event

The most common way to obtain a `Graphics` object is to use the Visual Studio 2010 Properties window to handle the `Paint` event on the window you want to render upon. This event is defined in terms of the `PaintEventHandler` delegate, which can point to any method taking a `System.Object` as the first parameter and a `PaintEventArgs` as the second.

The `PaintEventArgs` parameter contains the `Graphics` object you need to render onto the `Form`'s surface. For example, create a new Windows Application project named PaintEventApp. Next, use Solution Explorer to rename your initial `Form.cs` file to `MainWindow.cs`, and then handle the `Paint` event using the Properties window. This creates the following stub code:

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

  private void MainWindow_Paint(object sender, PaintEventArgs e)
  {
    // Add your painting code here!
  }
}
```

Now that you have handled the `Paint` event, you might wonder when it will fire. The `Paint` event fires whenever a window becomes *dirty*; 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 the `Paint` event handler is called automatically whenever your `Form` needs to be redrawn. Consider the following implementation of `MainWindow_Paint()`:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
  // Get the graphics object for this Form.
  Graphics g = e.Graphics;

  // Draw a circle.
  g.FillEllipse(Brushes.Blue, 10, 20, 150, 80);
```

```
  // Draw a string in a custom font.
  g.DrawString("Hello GDI+", new Font("Times New Roman", 30),
               Brushes.Red, 200, 200);

  // Draw a line with a custom pen.
  using (Pen p = new Pen(Color.YellowGreen, 10))
  {
    g.DrawLine(p, 80, 4, 200, 200);
  }
}
```

Once you obtain the `Graphics` object from the incoming `PaintEventArgs` parameter, you call `FillEllipse()`. Notice that this method (as well as any `Fill`-prefixed method) requires a `Brush`-derived type as the first parameter. While you could create any number of interesting brush objects from the `System.Drawing.Drawing2D` namespace (e.g., `HatchBrush` and `LinearGradientBrush`), the `Brushes` utility class provides handy access to a variety of solid-colored brush types.

Next, you make a call to `DrawString()`, which requires a `string` to render as its first parameter. Given this, GDI+ provides the `Font` type, which represents not only the name of the font to use when rendering the textual data, but also related characteristics, such as the point size (30, in this case). Also notice that `DrawString()` requires a `Brush` type; as far as GDI+ is concerned, "Hello GDI+" is nothing more than a collection of geometric patterns to fill on the screen. Finally, `DrawLine()` is called to render a line using a custom `Pen` type, 10 pixels wide. Figure A-21 shows the output of this rendering logic.

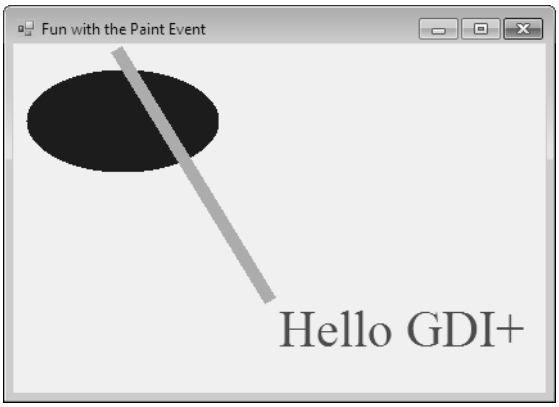

Figure A-21. *A simple GDI+ rendering operation*

■ **Note** In the preceding code, you explicitly dispose of the `Pen` object. As a rule, when you directly create a GDI+ type that implements `IDisposable`, you call the `Dispose()` method as soon as you are done with the object. Doing this lets you release the underlying resources as soon as possible. If you do not do this, the resources will eventually be freed by the garbage collector in a nondeterministic manner.

Invalidating the Form's Client Area

During the flow of a Windows Forms application, you might need to fire the `Paint` event in your code explicitly, rather than waiting for the window to become *naturally dirty* by the actions of the end user. For example, you might be building a program that allows the user to select from a number of predefined 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 were to wait 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, you call the inherited `Invalidate()` method:

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

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

The `Invalidate()` method has been overloaded a number of times. This allows you to specify a specific rectangular portion of the form to repaint, rather than having to repaint the entire client area (which is the default). If you wish to update only the extreme upper-left rectangle of the client area, you can write the following code:

```
// Repaint a given rectangular area of the Form.
private void UpdateUpperArea()
{
  Rectangle myRect = new Rectangle(0, 0, 75, 150);
  Invalidate(myRect);
}
```

■ **Source Code** You can find the PaintEventApp project under the Appendix A subdirectory.

Building a Complete Windows Forms Application

Let's conclude this introductory look at the Windows Forms and GDI+ APIs by building a complete GUI application that illustrates several of the techniques discussed in this chapter. The program you will create is a rudimentary painting program that allows users to select between two shape types (a circle or

rectangle) using the color of their choice to render data to the form. You will also allow end users to save their pictures to a local file on their hard drive for later use with object serialization services.

Building the Main Menu System

Begin by creating a new Windows Forms application named MyPaintProgram and rename your initial `Form1.cs` file to `MainWindow.cs`. Now design a menu system on this initial window that supports a topmost File menu that provides Save..., Load..., and Exit submenus (see Figure A-22).

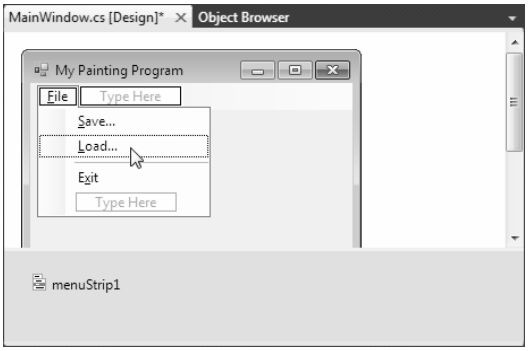

Figure A-22. *The File menu system*

■ **Note** If you specify a single dash (-) as a menu item, you can define separators within your menu system.

Next, create a second topmost Tools menu that provides options to select a shape and color to use for rendering, as well as an option to clear the form of all graphical data (see Figure A-23).

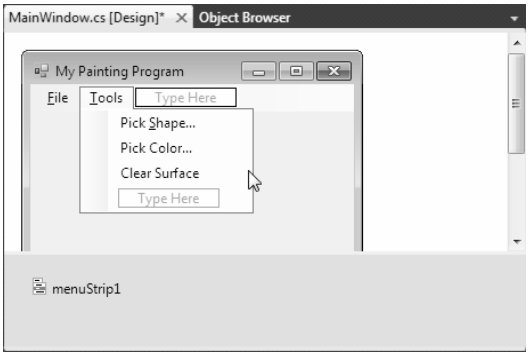

Figure A-23. *The Tools Menu System*

Finally, handle the `Click` event for each one of these subitems. You will implement each handler as you progress through the example; however, you can finish up the File ➤ Exit menu handler by calling `Close()`:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
  Close();
}
```

Defining the ShapeData Type

Recall that this application will allow end users to select from two predefined shapes in a given color. You will provide a way to allow users to save their graphical data to a file, so you want to define a custom class type that encapsulates each of these details; for the sake of simplicity, you do this using C# automatic properties (see Chapter 5 for more details on how to do this). Begin by adding a new class to your project named `ShapeData.cs` and implementing this type as follows:

```
[Serializable]
class ShapeData
{
  // The upper left of the shape to be drawn.
  public Point UpperLeftPoint { get; set; }

  // The current color of the shape to be drawn.
  public Color Color { get; set; }

  // The type of shape.
  public SelectedShape ShapeType { get; set; }
}
```

Here, `ShapeData` uses three automatic properties that encapsulates various types of data, two of which (`Point` and `Color`) are defined in the `System.Drawing` namespace, so be sure to import this namespace into your code file. Also, notice that this type has been adorned with the `[Serializable]` attribute. In an upcoming step, you will configure your `MainWindow` type to maintain a list of `ShapeData` types that you persist using object serialization services (see Chapter 20 for more details).

Defining the ShapePickerDialog Type

You can allow the user to choose between the circle or rectangle image type by creating a simple custom dialog box named `ShapePickerDialog` (insert this new `Form` now). Beyond adding the obligatory OK and Cancel buttons (each of which you should assign fitting `DialogResult` values), this dialog box uses of a single `GroupBox` that maintains two `RadioButton` objects: `radioButtonCircle` and `radioButtonRect`. Figure A-24 shows one possible design.

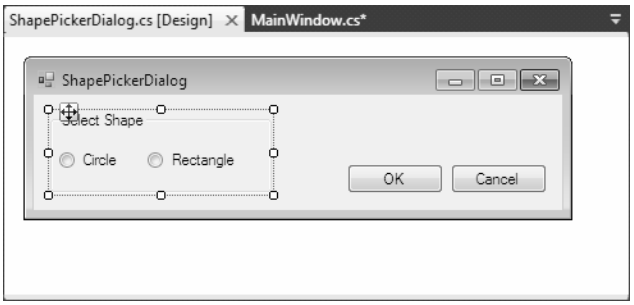

Figure A-24. *The ShapePickerDialog type*

Now, open the code window for your dialog box by right-clicking the Forms designer and selecting the View Code menu option. Within the `MyPaintProgram` namespace, define an enumeration (named `SelectedShape`) that defines names for each possible shape:

```
public enum SelectedShape
{
  Circle, Rectangle
}
```

Next, update your current `ShapePickerDialog` class type:

- Add an automatic property of type `SelectedShape`. The caller can use this property to determine which shape to render.
- Handle the `Click` event for the OK button using the Properties window.
- Implement this event handler to determine whether the circle radio button has been selected (through the `Checked` property). If so, set your `SelectedShape` property to `SelectedShape.Circle`; otherwise, set this property to `SelectedShape.Rectangle`.

Here is the complete code:

```
public partial class ShapePickerDialog : Form
{
  public SelectedShape SelectedShape { get; set; }

  public ShapePickerDialog()
  {
    InitializeComponent();
  }

  private void btnOK_Click(object sender, EventArgs e)
  {
    if (radioButtonCircle.Checked)
      SelectedShape = SelectedShape.Circle;
    else
```

```
      SelectedShape = SelectedShape.Rectangle;
  }
}
```

That wraps up the infrastructure of your program. Now all you need to do is implement the `Click` event handlers for the remaining menu items on the main window.

Adding Infrastructure to the MainWindow Type

Returning to the construction of the main window, add three new member variables to this `Form`. These member variables allow you to keep track of the selected shape (through a `SelectedShape` enum member variable), the selected color (represented by a `System.Drawing.Color` member variable), and through each of the rendered images held in a generic `List<T>` (where `T` is of type `ShapeData`):

```
public partial class MainWindow : Form
{
  // Current shape / color to draw.
  private SelectedShape currentShape;
  private Color currentColor = Color.DarkBlue;

  // This maintains each ShapeData.
  private List<ShapeData> shapes = new List<ShapeData>();
...
}
```

Next, you handle the `MouseDown` and `Paint` events for this `Form`-derived type using the Properties window. You will implement them in a later step; for the time being, however, you should find that the IDE has generated the following stub code:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
}

private void MainWindow_MouseDown(object sender, MouseEventArgs e)
{
}
```

Implementing the Tools Menu Functionality

You can allow a user to set the `currentShape` member variable by implementing the `Click` handler for the Tools ➤ Pick Shape... menu option. You use this to launch your custom dialog box; based on a user's selection, you assign this member variable accordingly:

```
private void pickShapeToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Load our dialog box and set the correct shape type.
  ShapePickerDialog dlg = new ShapePickerDialog();
  if (DialogResult.OK == dlg.ShowDialog())
  {
```

```
    currentShape = dlg.SelectedShape;
  }
}
```

You can let a user set the `currentColor` member variable by implementing the `Click` event handler for the Tools ➤ Pick Color... menu so it uses the `System.Windows.Forms.ColorDialog` type:

```
private void pickColorToolStripMenuItem_Click(object sender, EventArgs e)
{
  ColorDialog dlg = new ColorDialog();

  if (dlg.ShowDialog() == DialogResult.OK)
  {
    currentColor = dlg.Color;
  }
}
```

If you were to run your program as it now stands and select the Tools ➤ Pick Color menu option, you would get the dialog box shown in Figure A-25.

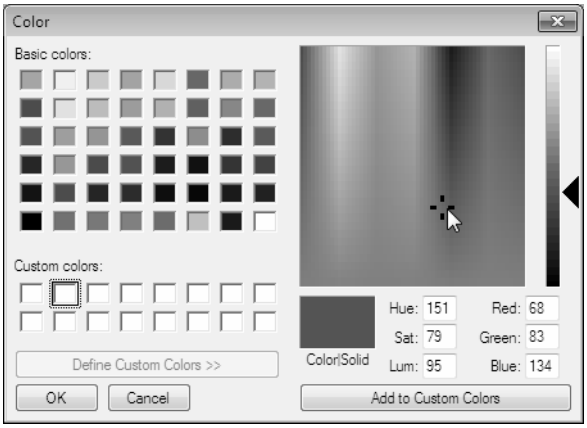

Figure A-25. *The stock* `ColorDialog` *type*

Finally, you implement the Tools ➤ Clear Surface menu handler so it empties the contents of the `List<T>` member variable and programmatically fires the `Paint` event using a call to `Invalidate()`:

```
private void clearSurfaceToolStripMenuItem_Click(object sender, EventArgs e)
{
  shapes.Clear();

  // This will fire the paint event.
  Invalidate();
}
```

Capturing and Rendering the Graphical Output

Given that a call to `Invalidate()` fires the `Paint` event, you need to author code within the `Paint` event handler. Your goal is to loop through each item in the (currently empty) `List<T>` member variable and render a circle or square at the current mouse location. The first step is to implement the `MouseDown` event handler and insert a new `ShapeData` type into the generic `List<T>` type, based on the user-selected color, shape type, and current location of the mouse:

```
private void MainWindow_MouseDown(object sender, MouseEventArgs e)
{
  // Make a ShapeData type based on current user
  // selections.
  ShapeData sd = new ShapeData();
  sd.ShapeType = currentShape;
  sd.Color = currentColor;
  sd.UpperLeftPoint = new Point(e.X, e.Y);

  // Add to the List<T> and force the form to repaint itself.
  shapes.Add(sd);
  Invalidate();
}
```

At this point, you can implement your `Paint` event handler:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
  // Get the Graphics object for this window.
  Graphics g = e.Graphics;

  // Render each shape in the selected color.
  foreach (ShapeData s in shapes)
  {
    // Render a rectangle or circle 20 x 20 pixels in size
    // using the correct color.
    if (s.ShapeType == SelectedShape.Rectangle)
      g.FillRectangle(new SolidBrush(s.Color),
        s.UpperLeftPoint.X, s.UpperLeftPoint.Y, 20, 20);
    else
      g.FillEllipse(new SolidBrush(s.Color),
        s.UpperLeftPoint.X, s.UpperLeftPoint.Y, 20, 20);
  }
}
```

If you run your application at this point, you should be able to render any number of shapes in a variety of colors (see Figure A-26).

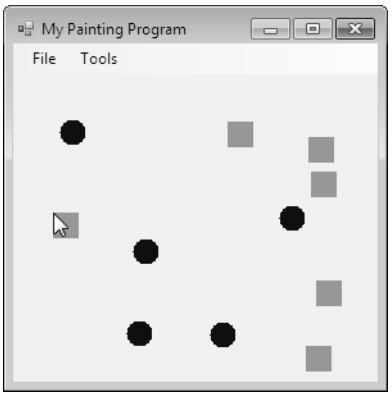

Figure A-26. *MyPaintProgram in action*

Implementing the Serialization Logic

The final aspect of this project involves implementing `Click` event handlers for the File ➤ Save... and File ➤ Load... menu items. Given that `ShapeData` has been marked with the `[Serializable]` attribute (and given that `List<T>` itself is serializable), you can quickly save out the current graphical data using the Windows Forms `SaveFileDialog` type. Begin by updating your `using` directives to specify you are using the `System.Runtime.Serialization.Formatters.Binary` and `System.IO` namespaces:

```
// For the binary formatter.
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
```

Now update your File ➤ Save... handler, as follows:

```
private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
  using (SaveFileDialog saveDlg = new SaveFileDialog())
  {
    // Configure the look and feel of the save dialog box.
    saveDlg.InitialDirectory = ".";
    saveDlg.Filter = "Shape files (*.shapes)|*.shapes";
    saveDlg.RestoreDirectory = true;
    saveDlg.FileName = "MyShapes";

    // If they click the OK button, open the new
    // file and serialize the List<T>.
    if (saveDlg.ShowDialog() == DialogResult.OK)
    {
      Stream myStream = saveDlg.OpenFile();
      if ((myStream != null))
      {
        // Save the shapes!
```

```
        BinaryFormatter myBinaryFormat = new BinaryFormatter();
        myBinaryFormat.Serialize(myStream, shapes);
        myStream.Close();
      }
    }
  }
}
```

The File ➤ Load event handler opens the selected file and deserializes the data back into the `List<T>` member variable with the help of the Windows Forms `OpenFileDialog` type:

```
private void loadToolStripMenuItem_Click(object sender, EventArgs e)
{
  using (OpenFileDialog openDlg = new OpenFileDialog())
  {
    openDlg.InitialDirectory = ".";
    openDlg.Filter = "Shape files (*.shapes)|*.shapes";
    openDlg.RestoreDirectory = true;
    openDlg.FileName = "MyShapes";

    if (openDlg.ShowDialog() == DialogResult.OK)
    {
      Stream myStream = openDlg.OpenFile();
      if ((myStream != null))
      {
        // Get the shapes!
        BinaryFormatter myBinaryFormat = new BinaryFormatter();
        shapes = (List<ShapeData>)myBinaryFormat.Deserialize(myStream);
        myStream.Close();
        Invalidate();
      }
    }
  }
}
```

After Chapter 20, the overall serialization logic here should look familiar. It is worth pointing out that the `SaveFileDialog` and `OpenFileDialog` types both support a `Filter` property that is assigned a rather cryptic string value. This filter controls a number of settings for the save/open dialog boxes, such as the file extension (`*.shapes`). You use the `FileName` property to control what the default name of the file you want to create—`MyShapes`, in this example.

At this point, your painting application is complete. You should now be able to save and load your current graphical data to any number of `*.shapes` files. If you want to enhance this Windows Forms program, you might wish to account for additional shapes, or even to allow the user to control the size of the shape to draw or perhaps select the format used to save the data (e.g., binary, XML, or SOAP; see Chapter 20).

Summary

This chapter examined the process of building traditional desktop applications using the Windows Forms and GDI+ APIs, which have been part of the .NET Framework since version 1.0. At minimum, a Windows Forms application consists of a type-extending `Form` and a `Main()` method that interacts with the `Application` type.

When you want to populate your forms with UI elements (e.g., menu systems and GUI input controls), you do so by inserting new objects into the inherited `Controls` collection. This chapter also showed you how to capture mouse, keyboard, and rendering events. Along the way, you learned about the `Graphics` type and many ways to generate graphical data at runtime.

As mentioned in this chapter's overview, the Windows Forms API has been (in some ways) superseded by the WPF API introduced with the release of .NET 3.0 (which you learned about in some detail in Part 6 of this book). While it is true that WPF is the choice for supercharged UI front ends, the Windows Forms API remains the simplest (and in many cases, most direct) way to author standard business applications, in-house applications, and simple configuration utilities. For these reasons, Windows Forms will be part of the .NET base class libraries for years to come.