WINDOWS FORMS
IN ACTION

Erik Brown

MANNING

Windows Forms in Action
by Erik Brown
Chapter 3

brief contents

C H A P T E R   3

Menus

Menu bars provide a good starting point for our discussion in part 2. Menus provide a convenient way to group similar or related commands in one place. Most users are familiar with the menu bar concept and expect standard menus such as File, Edit, and Help to appear in their applications. Even novice computer users quickly learn that clicking a menu on the menu bar displays a drop-down list of commands.

Menus became popular in Windows applications in the late 1980s, following their success on the Apple Macintosh. Prior to menus, users had to cope with a wide array of interfaces offered by desktop applications. The function keys still found at the top of computer keyboards were developed in part as a standard way to access common functions in an application, and some programs even provided a plastic template that sat on top of these function keys to help users remember the available commands.

Perhaps because of this history, many developers take the usefulness and popularity of menus for granted and do not spend sufficient time laying out a consistent, usable interface for their application. While graphical elements such as menus, toolbars, and other constructs make applications much friendlier, this is not an excuse to ignore good user design and rely on customers to become "experienced" to make effective use of the interface.

If that little lecture doesn't get your creative juices flowing, then nothing will. Back in .NET-land, Visual Studio provides a rather intuitive interface for the construction of menus that does away with some of the clunkiness found in earlier Windows development environments from Microsoft. No more dealing with menus in one place, the application in another place, and the processing code in a third place.

This chapter introduces some of the core classes used in Windows Forms in addition to discussing the menu interface in .NET. We cover the following aspects of Windows Forms and menu creation:

- The base classes for Windows Forms controls
- The different types of menus
- The classes required for Windows Forms menus
- How to create and modify menus and menu items

The examples in this chapter assume you have the code for MyPhotos version 2.4 available, as developed with Visual Studio in chapter 2. You can use this code with or without Visual Studio as a starting point for the tasks covered here. If you did not work through chapter 2, download the project from the book's website at http://www.manning.com/eebrown2. Follow the links and instructions on the site to retrieve version 2.4 of the application.

3.1 CONTROLS AND CONTAINERS

Before we plunge into menus, it is worth taking a look at the classes behind some of the .NET classes that support menus and other controls in Windows Forms, as shown in figure 3.1. This section walks through the class hierarchy behind Windows Forms controls and the `Form` class.

3.1.1 Control classes

All Windows Forms controls inherit from the `System.Windows.Forms.Control` class, or simply the `Control` class. The `System.Web.UI` namespace also contains a `Control` class for use in ASP.NET web pages, but since our focus is on Windows Forms, we use the terms control and `Control` class to mean the one in the `System.Windows.Forms` namespace.

So far we have seen the `Button` control and the `PictureBox` control classes. Figure 3.1 shows the class hierarchy for these classes. A *class hierarchy* is the set of classes from which a particular class is derived, and gives some indication of the purpose and capabilities behind the specific class. A brief discussion of the classes in figure 3.1 follows.

❶ All classes in C#, even internal types such as `int` and `char`, implicitly derive from the `object` class. In the .NET Framework, this class is equivalent to the `Object` class. We discuss this class in more detail in chapter 5.
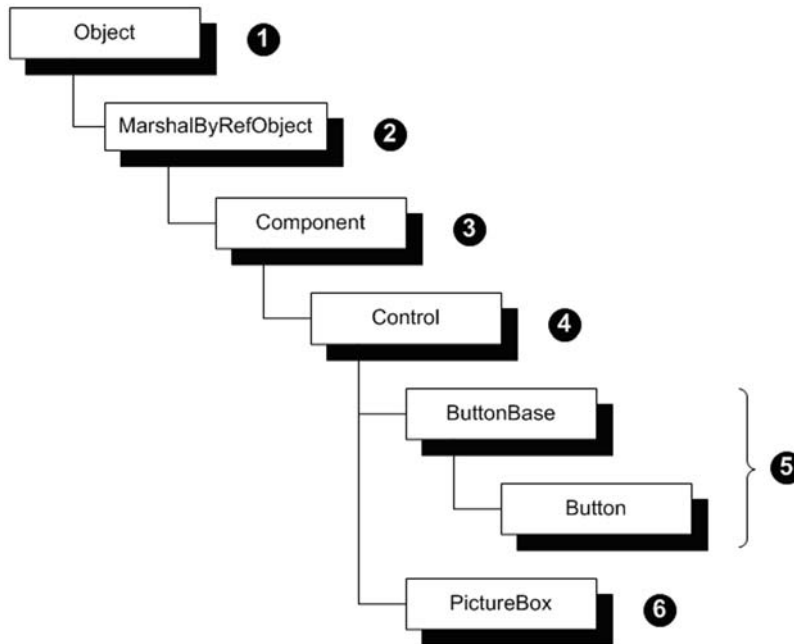
Figure 3.1   The class hierarchy for the Button and PictureBox controls is representative of the hierarchy for all Windows Forms controls.

❷ The `MarshalByRefObject` class is an object that must be marshaled by reference. *Marshaling* is a method of passing an item from one context so that it can be understood in another context. A typical use for marshaling is in remote procedure calls between two different machines, where each parameter of a function call must be converted into a common format (that is, marshaled) on the sending machine so that it may be interpreted on the receiving machine. In the .NET world, Windows controls are `MarshalByRefObject` objects since they are only valid in the process that creates them, and can be used outside this process only by reference.[1]

❸ The `Component` class is the base implementation of the `IComponent` interface for objects that marshal by reference. A *component* is an object that can exist within a container, and allows cleanup of system resources via the `Dispose` method. This class supports the `IDisposable` interface as well the `IComponent` interface. We cover interfaces in chapter 5, so don't get caught up in the terminology here. Since graphical controls exist within a `Form` window or other container control, all Windows Forms controls ultimately derive from this class.

---

[1] The details of marshaling are totally hidden for most Windows Forms applications, so you do not really need to know any of this. Hopefully, you find it somewhat interesting, if not useful.

❹ The Windows Forms `Control` class is a component with a visual representation on the Windows desktop. This class provides display functionality such as position and size, keyboard and mouse input, anchor and dock support, fonts, background images, and message routing. A summary of the members in this class is shown in .NET Table 3.1.

❺ The `ButtonBase` class is the base class for all buttons, including radio buttons and check box buttons in addition to the regular `Button` class we have already seen. Buttons are discussed in chapter 9.

❻ The `PictureBox` class is summarized in .NET Table 1 in the introduction.

3.1.2 Container classes

Controls that contain other controls are called *container controls*. The `Control` class itself provides support for containers, in members such as the `Controls` property or the `GetNextControl` method. Some container controls, such as the `GroupBox` control, inherit directly from the `Control` class. Group boxes are discussed in chapter 8. The `Form` class that we used in chapters 1 and 2 is also a container control. One of the unique features of this class is its ability to support scrolling for a contained set of controls. The `Form` class hierarchy supporting this and other functionality is shown in figure 3.2. Let's take a closer look at the numbered portions of this figure:

❶ You might think that all classes with scrolling inherit from the `ScrollableControl` class. In fact, this class is only for objects that support automated scrolling over a contained set of objects. Scrollable controls are discussed in chapter 13.

❷ The `ContainerControl` class is a control that provides focus management, providing a logical boundary for a contained set of controls. This class tracks the active control in a container even when the focus moves to an alternate container, and can manage the Tab key press for moving between the controls in the container.

❸ Almost all desktop windows in Windows Forms applications are represented by the `Form` class. This class is discussed throughout the book, of course, but especially in chapter 7.
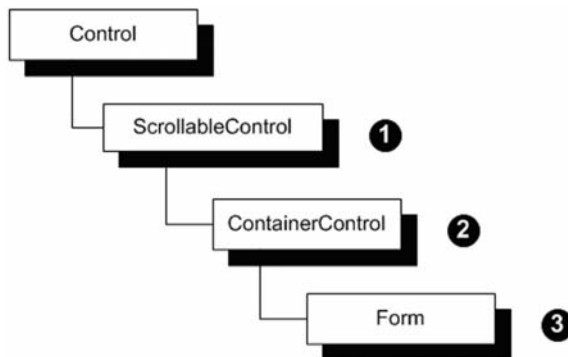

**Figure 3.2
The class hierarchy for the Form class is similar to the hierarchy for many Windows Forms containers.**

The `Control` class for Windows Forms is a component with a visual representation on the desktop. This class is part of the `System.Windows.Forms` namespace, and inherits from the `System.ComponentModel.Component` class. This class encapsulates the standard function-ality used by all Windows Forms controls.

Public Properties	AllowDrop	Gets or sets whether to allow drag-and-drop operations in this control. Drag-and-drop is discussed in chapter 23.
	Anchor	Gets or sets the anchor setting for the control. The `Dock` property gets or sets the dock setting.
	BackColor	Gets or sets the background color of the control.
	ContextMenuStrip	Gets or sets the context menu for the control.
	Controls	Gets or sets the controls contained by this control.
	ClientRectangle	Gets the client area of the control. The `DisplayRectangle` property gets the display area.
	Cursor	Gets or sets the `Cursor` to display when the mouse is over the control.
	Enabled	Gets or sets whether the control is enabled.
	Location	Gets or sets the control's location. The `Top`, `Bottom`, `Left`, and `Right` properties gets the control's edges.
	Parent	Gets or sets the parent of this control.
	TabIndex	Gets or sets the tab index of the control.
	TabStop	Gets or sets whether the user can use the Tab key to give the focus to the control.
	Text	Gets or sets the text associated with this control.
	Visible	Gets or sets whether control is visible. This also affects any controls contained by this control.
Public Methods	BringToFront	Brings the control to the front of the z-order. A similar `SendToBack` method also exists.
	GetNextControl	Returns the next or previous control in the tab order.
	Invalidate	Forces all or part of the control to be redrawn.
	PointToClient	Converts a screen location to client coordinates.
Public Events	Click	Occurs when the control is clicked.
	KeyPress	Occurs when a key is pressed while the control has focus.
	MouseUp	Occurs when a mouse button is released within the control.
	Paint	Occurs when all or part of the control should be redrawn.

3.2 THE NATURE OF MENUS

Let's turn our attention now to menu classes. Menus in .NET, as we see in this section, are container controls that contain menu items. We begin by presenting the different kinds of menus generally, most importantly menu bars and context menus, and then turn our attention to how menus are defined in the .NET Framework.

3.2.1 Menu terminology

The traditional *menu bar*, sometimes called the *main menu* or an *anchored menu*, is a set of menus shown horizontally across the top of an application. The menus in a typical menu bar display a drop-down list of commands when they are activated with the mouse or by a keyboard accelerator. Figure 3.3 shows an example of a menu bar with the File menu exposed, and a submenu of the Image menu item is displayed as well.

Figure 3.3 A traditional menu bar provides a set of menus across the top of an application.

Another type of menu is a *context menu*, also called a *pop-up menu* or *shortcut menu*. A context menu is a menu that appears in a particular situation, or context. Typically, a context menu contains a set of commands or menus related to a specific graphical element of the application. Such menus appear throughout the Windows environment at the right-click of the mouse. For example, right-click the Windows desktop, any program icon on your screen, or even the Windows Start menu, and a context menu appears with a set of commands related to the desktop display, the program, or the Start menu, respectively. Newer keyboards contain an accelerator key designed to simulate this behavior at the cursor's current location.

Figure 3.4 A context menu often provides quick access to items that also appear on the menu bar.

Context menus in .NET are typically associated with a specific control, the contents of which may change to reflect the condition of the control or type of item selected within the control. Figure 3.4 shows an example of a context menu associated with the main window of the application.

3.2.2 Menus in .NET

The menu classes provided by .NET received a complete rewrite for .NET 2.0. In .NET 1.*x*, the menu classes were based on the Win32 menu classes, and supported via the `Menu` class hierarchy. The `MainMenu`, `ContextMenu`, and `MenuItem` classes all derived from this `Menu` class, supporting Win32 menus and context menus within Windows Forms applications. These classes are still supported in .NET 2.0 for compatibility and use, but are no longer the preferred mechanism for menus in most applications.
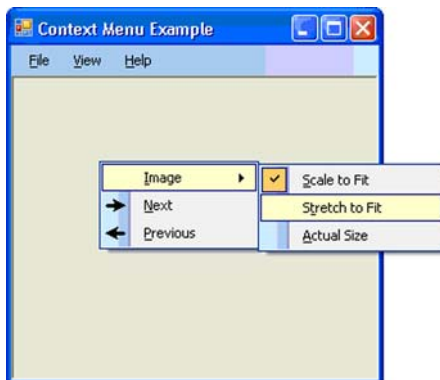
The new and improved classes for menus are based on the `ToolStrip` and `ToolStripItem` classes, which are the base classes for all manner of toolbar objects and the items within them. A `MenuStrip` class derives from `ToolStrip` and represents a menu, while a `ToolStripMenuItem` class derives from `ToolStripItem` to represent an item within a menu. The `ToolStrip` class and associated derived classes are shown in figure 3.5. We look at the `ToolStripItem` classes later in the chapter.

Between this and our prior class hierarchies, there is a lot to take in. As we did in part 1 of the book, we lay some groundwork here for future discussion, and revisit
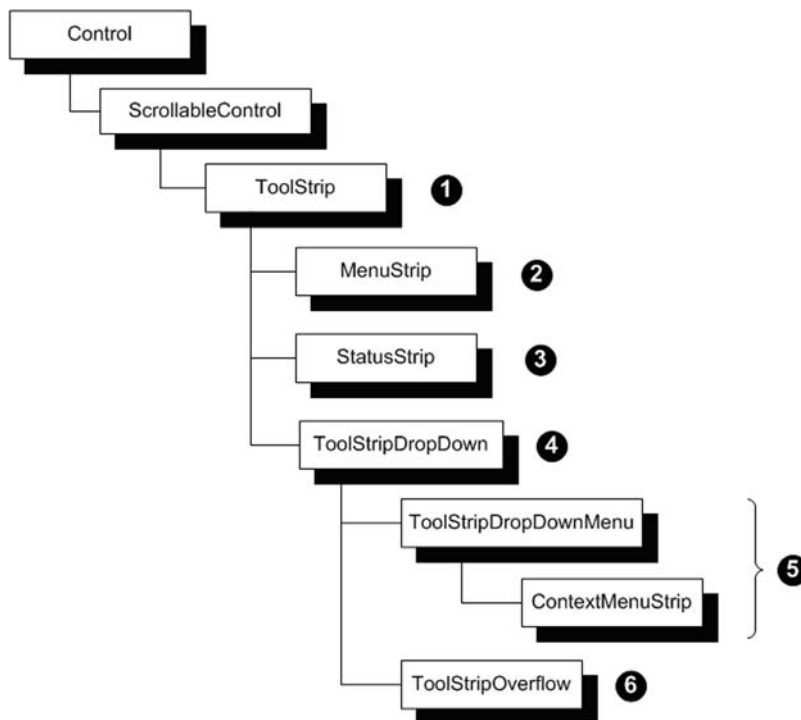

Figure 3.5   The ToolStrip classes replace the various *bar* classes available in prior versions of the .NET Framework, including the Menu, StatusBar, and ToolBar classes. These classes are still available to support Win32-style controls and for backward compatibility with existing applications.

the classes mentioned in passing here later in the book. Let's take a closer look at the classes in this hierarchy.

❶ The `ToolStrip` class is a scrollable control that contains a set of `ToolStripItem` objects. While tool strips do not scroll in the traditional sense, they allow controls to overflow into and out of the visible portion of the strip in a manner similar to scrolling. We discuss the details of tool strips in chapter 16.

❷ Menus are supported by the `MenuStrip` class. Menu strips behave like traditional menus and additionally support XP and Microsoft Office styles of appearance. The members of this class are shown in .NET Table 3.2.

❸ The `StatusStrip` control is a tool strip that acts as a traditional status bar, except that it additionally supports the functionality provided by tool strip objects. We discuss status strips in chapter 4.

❹ Tool strip objects that do not appear directly within a control are supported by the `ToolStripDropDown` class. This class is the generic base class for any drop-down strip, and is also discussed in chapter 4.

❺ The `ContextMenuStrip` class is specially designed to display menu item objects, or instances of the `ToolStripMenuItem` class, in a pop-up menu. This is the default drop-down created for drop-down items, instances of the `ToolStripDropDownItem` class. A context menu strip can be added to any Windows Forms control via the `Control.ContextMenuStrip` property. The `ToolStripDropDownMenu` class can be used to provide context-like functionality in a custom class, but it primarily serves as the base class for `ContextMenuStrip` objects.

❻ The final control in the figure, the `ToolStripOverflow` class, supports the overflow behavior of tool strip objects. This behavior is discussed in chapter 16.

.NET Table 3.2 MenuStrip class		
New in 2.0 The `MenuStrip` class is a tool strip control that represents a menu bar on a form. Menu strips contain one or more menu items, as `ToolStripMenuItem` objects, that represent clickable menus within the menu bar. This class is part of the `System.Windows.Forms` namespace, and inherits from the `ToolStrip` class.		
Public Properties	*MdiWindowListItem*	Gets or sets the menu item contained by this menu strip that displays a list of MDI child forms for the associated form object
Public Events	*MenuActivate*	Occurs whenever the menu is accessed via the keyboard or mouse

As you can see in .NET Table 3.2, most of the functionality for menu strips is contained in the `ToolStrip` class. In order to behave more like a traditional menu, the `MenuStrip` class also turns off various features of its parent class. By default, tool strips support item overflow functionality, a positioning grip, tooltips, and the ability

to fit multiple strips in a single row on a form. Menu strips define these properties more appropriately for menus, so the `CanOverflow` property defaults to `false`, rather than `true` as in the `ToolStrip` class. Similarly, the `GripStyle` property defaults to `Hidden`; `ShowToolTips` to `false`; and `Stretch` to `true`. These and other properties in the `ToolStrip` class are discussed in chapter 16.

3.3 MENU BARS

So, let's do it. Looking at our MyPhotos application, it would be nice to replace the Load button with a menu option. This allows more space in our window for the displayed image, and permits additional commands to be added in the future related to loading images. As an added benefit, it provides a nice example for this book, which is, of course, our ultimate goal.

Our new application is shown in figure 3.6. Load and Exit menu items have been added to a File menu on the main menu bar. The Load menu item replaces our Load button from the previous chapter. The line separating these items is called a *menu separator*. A View menu is also shown, which is discussed later in this section.

Figure 3.6 Notice in this File menu how the Load item displays Ctrl+L as its keyboard shortcut.

As you may expect, the menu bar appears in our code as a `MenuStrip` object. Menus such as the File menu are represented as `ToolStripMenuItem` objects contained within the menu strip. The Load and Exit menu items underneath the File menu are also `ToolStripMenuItem` objects. The menu separator is a special `ToolStripSeparator` object.

3.3.1 Adding a menu strip

In this section, we show the steps for adding our main menu. As already mentioned, this book uses Visual Studio for all example programs. If you are writing the code by hand and using the C# compiler on the command line, read through the steps and use the code inside or follow the task description as a model for your own program. Note that the downloadable code from the book's website alters the version number for the program at the beginning of each section. This tracks our progress throughout the book. If you recall, the version number is modified in the AssemblyInfo.cs file of the project.

Before we add our menu, we need to remove the existing Load button from the form.

REMOVE THE LOAD BUTTON		
	Action	**Result**
1	Remove the Load button from the form. **How-to** a. Right-click the Load button in the MainForm.cs [Design] window. b. Select the Delete option. **Alternately** Simply select the button and press the Delete key.	Visual Studio automatically removes all generated code related to the button from the `InitializeComponent` method of the MainForm.cs file. **Note:** When a control is deleted, any assignments of event handlers to the control are removed as well. The actual event handling code, in this case our `btnLoad_Click` method, is still in the source file and must be removed manually. We leave this code in the MainForm.cs file for now, and deal with it later in the chapter.
2	Set the `Dock` property for the `PictureBox` control to `Fill`. **How-to** a. Click the control. b. Click the small arrow at the top right of the control. c. Click the "Dock in parent container" link. 	The `PictureBox` control is docked within the form in the designer. **Note:** You can, of course, assign the `Dock` property as described at the end of chapter 2. The designer provides the small arrow, called a smart tag, for most controls to give quick access to common settings and tasks. We do not discuss the smart tags for every control, but you can look for them as we progress through the book.

With the Load button gone, our way is clear to move the Load functionality into a menu. To do this, we need to add a `MenuStrip` to our form, to act as a container for the menu items to display. The following table continues the above steps.

CREATE THE MENU BAR		
3	Drag a `MenuStrip` object from the Toolbox onto your form. This object appears in the Menus and Toolbars group within the toolbox.	A `MenuStrip` object called `menuStrip1` is added to your form. This object is shown within the form and in the *component tray*, located below the form as in the graphic. The component tray displays objects that may not have a physical presence in the window, such as timers, database connections, menu strips, and context menu strips.

Let's take a look at the source code generated by these actions in the Main-Form.Designer.cs window. As you can see in listing 3.1, the Windows Forms Designer has replaced the button control with our menu strip. The overall structure of this code follows what we saw in chapter 2: controls are created, layout is suspended, controls are initialized, controls are added to the form, layout is resumed, and finally the control definitions appear at the end. The annotated points highlight the menu strip portion of this code.

Listing 3.1   The designer region after adding a MenuStrip

```
#region Windows Form Designer generated code
. . .
private void InitializeComponent()
{
  this.pbxPhoto = new System.Windows.Forms.PictureBox();
  this.menuStrip1 = new System.Windows.Forms.MenuStrip();
  ((System.ComponentModel.ISupportInitialize)
      (this.pbxPhoto)).BeginInit();
  this.SuspendLayout();
  //
  // pbxPhoto
  //
  this.pbxPhoto.BorderStyle
      = System.Windows.Forms.BorderStyle.Fixed3D;
  this.pbxPhoto.Dock = System.Windows.Forms.DockStyle.Fill;
  this.pbxPhoto.Location = new System.Drawing.Point(0, 24);
  this.pbxPhoto.Name = "pbxPhoto";
  this.pbxPhoto.Size = new System.Drawing.Size(292, 242);
  this.pbxPhoto.SizeMode
      = System.Windows.Forms.PictureBoxSizeMode.Zoom;
  this.pbxPhoto.TabIndex = 1;
  this.pbxPhoto.TabStop = false;
  //                        ❶ Initializes menu strip
  // menuStrip1
  //
  this.menuStrip1.Location = new System.Drawing.Point(0, 0);
  this.menuStrip1.Name = "menuStrip1";
  this.menuStrip1.Size = new System.Drawing.Size(292, 24);
  this.menuStrip1.TabIndex = 2;
  this.menuStrip1.Text = "menuStrip1";
  //
  // MainForm
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size(292, 266);
  this.Controls.Add(this.menuStrip1);      ←
  this.Controls.Add(this.pbxPhoto);
  this.MainMenuStrip = this.menuStrip1;     ❷ Adds menu
  this.Name = "MainForm";                      strip to form
```

```
      this.Text = "MyPhotos";
      ((System.ComponentModel.ISupportInitialize)
          (this.pbxPhoto)).EndInit();
      this.ResumeLayout(false);
      this.PerformLayout();
  }
  #endregion

  private System.Windows.Forms.PictureBox pbxPhoto;
  private System.Windows.Forms.MenuStrip menuStrip1;
```
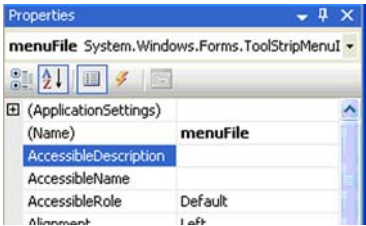
❶ The `MenuStrip` control is initialized much like the `Button` and `PictureBox` controls in chapter 2. The `Dock` property for a menu strip control is set to `DockStyle.Top` by default, which is why this setting does not appear here.

❷ The strip is added to the control just like any other control, even though it really does not have much visual presence except through its contained menu items. As we see shortly, menu items are added to the menu strip control, rather than to the form itself.

3.3.2   Adding a menu item

With a `MenuStrip` on our form, we can now add menu items. You may have noticed that the menu strip represents the container rather than the actual menu items. Each menu item is created using the `ToolStripMenuItem` class. In this section we create a top-level File menu. In the next section we create the drop-down menu that appears when the user clicks on this menu.

	CREATE THE FILE MENU	
	Action	**Result**
1	Edit the menu strip in the MainMenu.cs [Design] window. **How-to** Click on the menuStrip1 item that appears below the form.	An empty menu bar appears at the top of the form. The space for the first top-level menu contains the words "Type Here."
2	Type in a top-level File menu as "&File."	A File menu appears on the form. **Note:** The ampersand (&) specifies the character, in this case F, to use as the mnemonic for this menu. Such mnemonics are used with the Alt key. In our application the File menu is displayed whenever the users clicks on it or when they enter Alt+F via the keyboard.

CREATE THE FILE MENU *(CONTINUED)*		
	Action	**Result**
3	Modify the (Name) property for this menu to be "menuFile." **How-to** Use the Properties window for the new File menu item, and modify the (Name) entry.	The variable name for the control is renamed to menuFile in the source code files.

		Properties ▼ 廿 ✕
		menuFile System.Windows.Forms.ToolStripMenuI ▼
		▣ (ApplicationSettings)
		(Name) **menuFile**
		AccessibleDescription
		AccessibleName
		AccessibleRole Default
		Alignment Left

Note: The string "&File" we entered for the menu appears in the Text property for the item.

Your application now contains a File menu on the menu bar. The designer file has been updated to define and initialize the menu item. The relevant portions of this code are shown in listing 3.2. The InitializeComponent method now contains additional lines to initialize this menu item and add it to our MenuStrip object.

Listing 3.2   Designer code after portion of File menu is created

```
#region Windows Form Designer generated code
. . .
private void InitializeComponent()
{
  this.pbxPhoto = new System.Windows.Forms.PictureBox();
  this.menuStrip1 = new System.Windows.Forms.MenuStrip();
  this.menuFile = new System.Windows.Forms.ToolStripMenuItem();
  . . .
  this.menuStrip1.SuspendLayout();
  . . .
  //
  // menuStrip1
  //
  this.menuStrip1.Items.AddRange(new
      System.Windows.Forms.ToolStripItem[] {
          this.menuFile});
  this.menuStrip1.Location = new System.Drawing.Point(0, 0);
  this.menuStrip1.Name = "menuStrip1";
  this.menuStrip1.Size = new System.Drawing.Size(292, 24);
  this.menuStrip1.TabIndex = 2;
  this.menuStrip1.Text = "menuStrip1";
  //
  // menuFile
  //
  this.menuFile.Name = "menuFile";
  this.menuFile.Size = new System.Drawing.Size(35, 20);
```

```
    this.menuFile.Text = "&File";
    . . .
}
#endregion

private System.Windows.Forms.PictureBox pbxPhoto;
private System.Windows.Forms.MenuStrip menuStrip1;
private System.Windows.Forms.ToolStripMenuItem menuFile;
```

<table>
<tr><td colspan="3">.NET Table 3.3   ToolStripMenuItem class</td></tr>
<tr><td colspan="3">New in 2.0 The <code>ToolStripMenuItem</code> class represents a menu within a <code>MenuStrip</code> or <code>ContextMenuStrip</code> object, or a submenu of another <code>ToolStripMenuItem</code> object. <code>ToolStripMenuItem</code> objects are displayed to the user, while <code>MenuStrip</code> and <code>ContextMenuStrip</code> objects simply establish a container in which such menu items can appear. The <code>ToolStripMenuItem</code> class is part of the <code>System.Windows.Forms</code> namespace, and inherits from the <code>ToolStripDropDownItem</code> class. See .NET Table 4.2 for a list of members inherited from this base class.</td></tr>
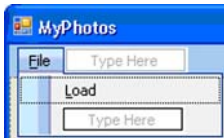<tr>
<td rowspan="7">Public Properties</td>
<td>Checked</td>
<td>Gets or sets whether a checkmark appears next to the text of the menu item.</td>
</tr>
<tr>
<td>CheckState</td>
<td>Gets or sets a three-state value for the menu item, based on the <code>CheckState</code> enumeration. This is similar to <code>Checked</code>, but allows an indeterminate setting when the checked or unchecked state cannot be determined.</td>
</tr>
<tr>
<td>Enabled (overridden from ToolStripItem)</td>
<td>Gets or sets whether the menu item is enabled. A disabled menu is displayed in a grayed-out color, cannot be selected, and does not display any child menu items.</td>
</tr>
<tr>
<td>Overflow (overridden from ToolStripItem)</td>
<td>Gets or sets how the menu item interacts with an overflow button, based on the <code>ToolStripItemOverflow</code> enumeration.</td>
</tr>
<tr>
<td>ShortcutKeyDisplayString</td>
<td>Gets or sets the string to display as the shortcut for the menu. If this is blank, the actual shortcut key setting is shown.</td>
</tr>
<tr>
<td>ShortcutKeys</td>
<td>Gets or sets the shortcut keys for this menu item, using the <code>Keys</code> enumeration values.</td>
</tr>
<tr>
<td>ShowShortcutKeys</td>
<td>Gets or sets whether to display the <code>ShortcutKeys</code> setting when displaying the menu.</td>
</tr>
<tr>
<td rowspan="2">Public Events</td>
<td>CheckedChanged</td>
<td>Occurs when the <code>Checked</code> property value changes.</td>
</tr>
<tr>
<td>CheckStateChanged</td>
<td>Occurs when the <code>CheckState</code> property value changes.</td>
</tr>
</table>

CHAPTER 3   MENUS

This code follows the now familiar pattern of creating the control, initializing the control in its own section, and adding the control to a parent container. One difference here is that the File menu item is contained within the menu strip, rather than the `Form` itself. Note how the menu item is added to the `menuStrip1` control by creating an array of `ToolStripItem` objects with `menuFile` as the only entry. Arrays of objects in C# are created just like any other class, with the addition of square brackets, `[]`, to indicate that an array of objects should be created rather than a single object.

The File menu is listing 3.2 is defined as a `ToolStripMenuItem` object. An overview of this class appears in .NET Table 3.3. We discuss the class hierarchy for this class in a moment.

3.3.3 Adding drop-down menu items

So far, we have created a main menu with a single File menu item. Our next step is to create the drop-down menu, or submenu, that appears when this menu is clicked.

	CREATE THE FILE DROP-DOWN MENU	
	Action	**Result**
1	Create a Load menu item within the File menu. Use the text "&Load." **How-to** a. In the designer window, click the File menu. b. Press the down arrow key to highlight the "Type Here" entry below the File menu. c. Enter the text "&Load." d. Press the Enter key.	The item appears as the first item in the drop-down list for the File menu.
2	Display the Properties window for the Load menu item and set the following property values. **Settings**	Property	Value		(Name)	menuFileLoad		ShortcutKeys	Ctrl+L		Text	&Load		The modified properties are displayed in the Properties window. **Note:** The `ShortcutKeys` property defines a keyboard shortcut, in this case Ctrl+L, that immediately invokes the menu as if it were clicked, without actually displaying the menu. In the Properties window, the `ShortcutKeys` property sports a special interface, as shown here.

	Action	Result
3	Add a menu separator after the Load menu. **How-to** Enter a dash (-) character as the text for the menu. **Alternately** Select Separator from the drop-down menu associated with the item, as shown here.	A menu separator is added to the menu. This item is implemented as a `ToolStripSeparator` object. We retain the default (Name) and other settings for this item.
4	Finally, add the Exit menu item, assigning the properties as follows. **Settings** | Property	Value
---	---
(Name)	menuFileExit
Text	E&xit		This completes the File menu, at least for now. **Note:** Of course, the Windows keyboard shortcut Alt-F4 can be used to close the application. There is no need to add this keystroke to our menu as it is imposed by the operating system.

As you might expect, the code generated for the MainForm.cs file uses `ToolStrip-MenuItem` objects to construct the drop-down list for the File menu, with the objects initialized in the `InitializeComponent` method. The relevant code from the designer generated region is extracted in listing 3.3.

Listing 3.3 Designer code after complete File menu is created

```
private void InitializeComponent()
{
  this.pbxPhoto = new System.Windows.Forms.PictureBox();
  this.menuStrip1 = new System.Windows.Forms.MenuStrip();
  this.menuFile = new System.Windows.Forms.ToolStripMenuItem();
  this.menuFileLoad
        = new System.Windows.Forms.ToolStripMenuItem();
  this.toolStripMenuItem1
        = new System.Windows.Forms.ToolStripSeparator();
```

```
      this.menuFileExit = new . . .;
      . . .
      this.menuStrip1.Items.AddRange(
        new System.Windows.Forms.ToolStripItem[] {
        this.menuFile});
      . . .
      //
      // menuFile
      //
      this.menuFile.DropDownItems.AddRange(
        new System.Windows.Forms.ToolStripItem[] {            ❶ Creates File
        this.menuFileLoad,|#1                                    drop-down menu
        this. toolStripMenuItem1,|#1
        this.menuFileExit});|#1
      . . .
      //
      // menuFileLoad
      //
      this.menuFileLoad.Name = "menuFileLoad";
      this.menuFileLoad.ShortcutKeys = ((System.Windows.Forms.Keys)
          ((System.Windows.Forms.Keys.Control             Defines keyboard
              | System.Windows.Forms.Keys.L)));                 shortcut ❷
      this.menuFileLoad.Size = new System.Drawing.Size(152, 22);
      this.menuFileLoad.Text = "&Load";
      //
      // toolStripMenuItem1
      //
      this.toolStripMenuItem1.Name = "toolStripSeparator1";
      this.toolStripMenuItem1.Size = new System.Drawing.Size(149, 6);
      //
      // menuFileExit
      //
      this.menuFileExit.Name = "menuFileExit";
      this.menuFileExit.Size = new System.Drawing.Size(152, 22);
      this.menuFileExit.Text = "E&xit";
      . . .
    }
    . . .
    private System.Windows.Forms.MenuStrip menuStrip1;
    private System.Windows.Forms.ToolStripMenuItem menuFile;
    private System.Windows.Forms.ToolStripMenuItem menuFileLoad;
    private System.Windows.Forms.ToolStripSeparator
            toolStripMenuItem1;
    private System.Windows.Forms.ToolStripMenuItem menuFileExit;
```

While much of this code is similar to what we have seen for other controls, a couple aspects are worth highlighting:

❶ The items to appear under the File menu are added by constructing an array of the desired objects and assigning them to the menuFile.DropDownItems property. The ToolStripMenuItem class derives from the ToolStripDropDownItem class.

This base class defines the `DropDownItems` property used here, which contains the collection of items to associate with the menu. The `AddRange` method on this collection adds a set of items to the menu. The order of objects in the array establishes the order in which these items appear within the File menu.

❷ The Ctrl+L shortcut for the Load menu item is defined through the use of the `System.Windows.Forms.Keys` enumeration. Note how the `Keys.Control` and `Keys.L` values are or'd together using the vertical bar (`|`) operator.

If you wish to see the application so far, compile and run the code to view the File menu. You may notice that our menus still do not actually do anything. To fix this, we need to handle the `Click` event on our menus, which is the subject of our final section.

The `ToolStripMenuItem` class used to define the Load and Exit menu items, as well as the File menu, is part of the `ToolStripItem` class hierarchy. This class is a component, and serves as the base class for the various tool strip items available in Windows Forms. Since this class is not a control, it defines a number of members similar to the `Control` class discussed earlier in the chapter to maintain consistency between control and tool strip item objects. We discuss the `StatusStrip` and `ContextMenuStrip` classes in the next chapter; and `ToolStrip` classes in general in chapter 16. Here, let's take a quick look at the rather large `ToolStripItem` class hierarchy, as shown in figure 3.7. Let's discuss the annotated areas:

❶ The `ToolStripItem` class is the basis for the hierarchy in the figure. Details on this class appear in .NET Table 3.4.

❷ A common use of tool strips is to display a button that responds to a click of the mouse, much like a normal button on a form. The `ToolStripButton` class encapsulates this functionality.

❸ One of the more interesting features of tool strips is their ability to host almost any Windows Forms control as an item within the strip. The `ToolStripControlHost` class encapsulates this functionality, with predefined classes for the `ComboBox`, `ProgressBar`, and `TextBox` controls. The base class can be used to host other controls as well, as we see in chapter 16.

❹ Some tool strip items support the ability to display additional items in an associated drop-down list, such as a menu item that displays a submenu. The `ToolStripDropDownItem` class is the base class for such items, and uses a `ToolStripDropDown` instance as a container for the set of drop-down items. We have already used the `ToolStripMenuItem` class for our menu items. The `ToolStripDropDownButton` item is a button that displays a drop-down list when clicked. The `ToolStripSplitButton` item is a normal button next to a drop-down button, such as a graphic with an associated drop-down arrow used in many applications. We discuss these and other tool strip items, including the overflow functionality supported by the `ToolStripOverflowButton` class, in chapter 16.
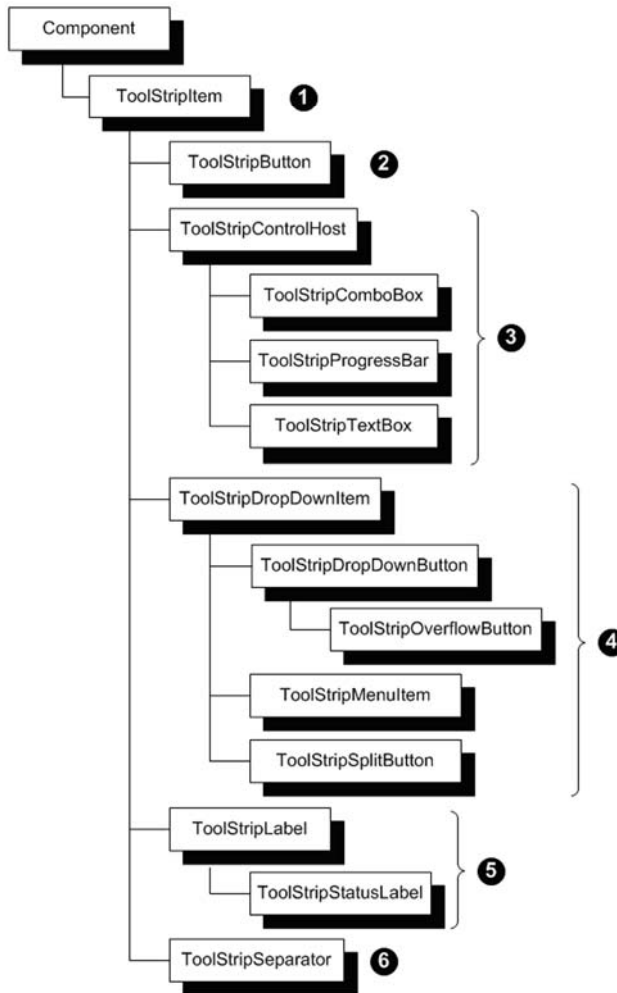
Figure 3.7 The classes derived from ToolStripItem are all components, and cannot exist on a form outside of a ToolStrip object.

❺ The `ToolStripLabel` class displays nonselectable text and graphics, or can link to other information by acting as a hyperlink. A special type of label is the text that appears in a status bar, represented by the `ToolStripStatusLabel` class discussed in the next chapter.

❻ When a large number of items appear in a single strip, whether buttons in a toolbar or menu items in a menu, there is a need to partition the strip into logical areas to make it easier for users to locate and understand the desired functionality. The `ToolStripSeparator` class encapsulates this functionality, as we saw for the separator between our Load and Exit menu items.

New in 2.0 The `ToolStripItem` class is a component that represents an item on a `Tool-Strip` object, and encapsulates the standard functionality used by all tool strip items. This class is part of the `System.Windows.Forms` namespace, and inherits from the `System.ComponentModel.Component` class.

Public Properties	AllowDrop	Gets or sets whether item reordering and drag-and-drop operations use the default (false) or custom behavior (true).
	Alignment	Gets or sets whether the item aligns toward the beginning or end of the containing tool strip.
	Anchor	Gets or sets how the item attaches to the edges of its container. A `Dock` property also exists.
	BackColor	Gets or sets the background color of the item.
	ClientRectangle	Gets the area where content can be drawn within the item without overwriting background borders.
	DisplayStyle	Gets or sets the `ToolStripItemDisplayStyle` enumeration value that defines whether text and images are displayed for the item.
	Enabled	Gets or sets whether this item can respond to user interaction.
	Image	Gets or sets the image displayed on the item.
	MergeAction	Gets or sets how the item merges into a target tool strip.
	Parent	Gets or sets the parent of this item.
	Text	Gets or sets the text associated with this item.
	ToolTipText	Gets or sets the tooltip text. If not set and `AutoToolTip` is `true`, the `Text` property is used as the tooltip.
	Visible	Gets or sets whether item and any subitems are visible.
Public Methods	Invalidate	Indicates that all or part of the item should be redrawn.
	PerformClick	Invokes the `Click` event behavior for this item.
Public Events	Click	Occurs when the item is clicked.
	DragDrop	Occurs when a drag-and-drop operation on the item is completed.
	MouseUp	Occurs when a mouse button is released within the bounds for the item.
	Paint	Occurs when all or part of the item should be repainted.

3.4   MENU HANDLING

A menu, of course, is not very useful if you can't make it do something. In this section we define some event handlers for our File menu items, and examine how event handlers work in more detail than we covered in prior chapters.

In part 1 we saw how an event was defined using the += syntax in C#, and how Visual Studio generates this code whenever an event is defined for a Windows Forms control. Events can be added from the Windows Forms Designer window directly, or via the Properties window. We discuss and demonstrate each method separately in the context of our File menu items.

3.4.1   Adding handlers via the designer window

As you may expect, Visual Studio adds a Click event handler whenever you double-click a menu item control in the Windows Forms Designer window. We saw this behavior for buttons in chapter 2, so let's use this feature to add a handler to the Load menu item here.

Since this code matches the handler we discussed in chapter 2 for the Load button, we will not discuss it again.

Compile the application to verify that the Load menu item works just like the Load button in chapter 2. You should be able to load a new image using the menu bar via the mouse, using the access keys Alt+F and then Alt+L to invoke the menu item from the keyboard, or using the keyboard shortcut Ctrl+L.

ADD CLICK HANDLER FOR THE LOAD MENU	
Action	**Result**
1 In the MainForm.cs [Design] window, add a Click handler for the Load menu item. **How-to** Double-click the Load menu item.	A new event handler for the item is added and the cursor is placed in the code window within the newly added handler. ``` private void menuFileLoad_Click(object sender, EventArgs e) { } ``` The new handler is also registered as a Click handler for the item in the InitializeComponent method of the MainForm.Designer.cs file. ``` menuFileLoad.Click += new System.EventHandler (this.menuFileLoad_Click); ```

	Action	Result

2	Copy the code from the now defunct `btnLoad_Click` handler into our new method and delete the old method.
	
	Note: Unless you removed it, the code for `btnLoad_Click` should still be present in your MainForm.cs file.
		This code is identical to the code used with our Load button in chapter 2; it is just invoked via a menu item rather than a button.
		
		<pre>private void menuFileLoad_Click
		    (object sender, System.EventArgs e)
		{
		  OpenFileDialog dlg = new OpenFileDialog();
		
		  dlg.Title = "Load Photo";
		  dlg.Filter = "jpg files (*.jpg)"
		      + "|*.jpg|All files (*.*)|*.*";
		
		  if (dlg.ShowDialog() == DialogResult.OK)
		  {
		    try
		    {
		      pbxPhoto.Image = new Bitmap(dlg.OpenFile());
		    }
		    catch (ArgumentException ex)
		    {
		      MessageBox.Show("Unable to load file: "
		                      + ex.Message);
		    }
		  }
		  dlg.Dispose();
		}</pre>

3.4.2 Adding handlers via the properties window

Most controls in Windows Forms define a default event. Visual Studio adds an event handler for this event whenever a control is double-clicked in the designer window. As we have seen, the default event for button and menu controls is the `Click` event. We discuss default events for other controls throughout the book.

The .NET classes provide a rich set of events for everything from key presses and mouse clicks to redrawing or resizing a control. To support these and other events, Visual Studio provides a generic way to add event handlers using the Properties window.

We have seen how the Properties window provides the list of properties associated with a specific control. It also provides the list of events and allows new event handlers to be added, as illustrated in figure 3.8. Note the small toolbar buttons between the object drop-down and the list of object members. Clicking the Properties button displays a list of properties for the
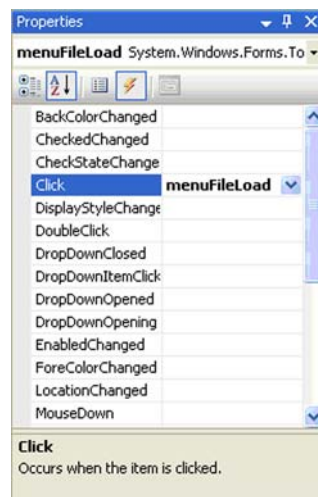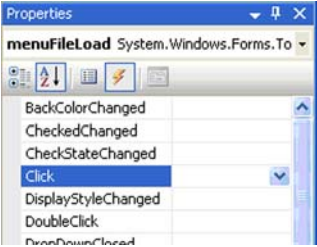

Figure 3.8 This Properties window shows the events for the Load menu item in our application, including the menuFileLoad _Click event handler.

current object. If you click the Events button, the lightning bolt icon, this window displays a list of events. The events for our `menuFileLoad` menu item are shown in the figure.

We can use this window to add an event handler for the Exit menu item. The following steps add a `Click` event handler for this menu that closes the application.

	Action	**Result**
1	Display the Events for the Exit menu item in the Properties toolbar. **How-to** Display the Properties window for the item and click the Events button.	
2	Double-click the Click item listed in the window.	A `Click` event handler is added to the `menuFileExit` object. ```\nprivate void menuFileExit_Click(\n object sender, EventArgs e)\n{\n```
3	Call the `Form.Close` method within this handler.	```\nClose();\n}\n``` **Note:** The code for this event handler is split across steps 2 and 3. We do this throughout the book as a convenient way to discuss different portions of code for a single member of a class.

The `Form.Close` method is used to exit the application. This method closes the associated `Form`, or the entire application if the form was the startup window for the application.

As you may have noticed in chapter 1, the `Application` class provides an `Exit` method that we could use instead here. This call forces all message loops started by `Application.Run` methods to exit, and closes any forms associated with them as well.

In our existing code, either method would close the application. As we discuss in chapter 7, however, the `Close` method ensures that all resources associated with a form are disposed, and invokes various closing events to permit additional processing as required. As a result, use of the `Close` method is normally preferred to exit a `Form` rather than the `Application.Exit` method.

TRY IT!   Compile and run the code to verify that the Load and Exit menu items now work. If you feel like experimenting, here are a couple areas worth exploring:

- Set the `ShowShortcutKeys` property for the Load menu item to `false` in order to prevent the Ctrl+L shortcut from appearing on the menu. Note that the keyboard shortcut still works, even though it is not displayed.
- Modify the `Enabled` and `Visible` properties for the Exit menu item to see how they change the behavior of this menu when the application runs.
- Create a new Clear item between the Load item and the subsequent separator that clears the picture box control by assigning `null` to the `Image` property.

Our handling of the File menu is now complete, and we have seen the two main ways to add event handlers in Visual Studio.

Sit back for a moment and think about what we have done here. If you have used Visual C++ with MFC, realize that the secret macros and magic interface files required by this environment are gone. In their place are well-designed objects that can quickly and easily be used to create arbitrarily complex menu structures. Also realize that we created these menus with very little explicit code. The designer interface handled much of the work required to define and arrange these menus within the application.

3.5   RECAP

In this chapter we modified our application to use a Load menu item, rather than a Load button to open and display an image in the `PictureBox` control. We looked at various kinds of menus, and examined the classes required to build and manipulate menus in Windows applications with the .NET Framework.

Along the way we took a quick tour through many of the foundational classes in the Windows Forms namespace. We discussed how the `Component` class is the basis for objects that can exist within a container, and the `Control` class is the basis for all Windows Forms controls. We also saw the class hierarchy for the `Form` class, including the `ScrollableControl` and `ContainerControl` classes.

We discussed the different types of menus, and the `MenuStrip` and `ToolStrip-MenuItem` classes used to create menus in Windows Forms. The `MenuStrip` class is part of the `ToolStrip` classes that are used to define menu bars, status bars, and all manner of tool bars in .NET. These classes display the `ToolStripItem` classes within their borders to represent various types of items. The `ToolStripMenuItem` class is one such item, and represents a menu item within a menu. We also saw the `ToolStripSeparator` class, used to create a separator line within a menu.

The Visual Studio interface for creating menus was also discussed, and we created some top-level menus for our sample MyPhotos application. We looked at the code generated for these menus, and how collections of menus are defined as arrays within the designer file.

In chapter 4 we discuss additional aspects of tool strips in Windows Forms by examining the classes for creating context menus and status bars.

.NET PROGRAMMING

WINDOWS FORMS IN ACTION

Erik Brown

With the .NET Framework you can create rich graphical interfaces and compelling user experiences. Whether you design business forms, build interactive graphical interfaces, or code smart client applications, you need to master Windows Forms.

Windows Forms in Action, a revised edition of the popular *Windows Forms Programming with C#*, gets you going quickly by immersing you in a practical running example. Easy to follow instructions, numerous graphics, and unique "information maps" guide you through the entire Windows Forms namespace—all in the powerful C# language. If you're new to Windows Forms, you'll explore fundamentals like labels, menus, buttons, and panels. Seasoned developers can sink their teeth into advanced concepts such as reusable control libraries, merging tool strips, persistent application settings, explorer-style interfaces, and two-way data binding. The appendices include a handy visual index of the Windows Forms classes.

What's Inside

- Owner-drawn lists and tabs
- List, tree, and data grid views
- Data streaming and encryption
- Masked text boxes and date controls
- Standard and ClickOnce deployment
- How to spice up your applications with: custom controls, drag and drop, background processing, automatic text completion, embedded web browsing

With over 17 years of experience, **Erik Brown** is a developer, architect, consultant, and manager. He is a veteran of three successful startups and currently works as a program manager at Unisys Corporation. Erik lives in northern Virginia with his wife and two daughters.

"Each chapter is worth the price of the whole book!"

—Berndt Hamboeck
Senior Architect and Team
Leader, United Nations

"Excellent for developers new to Windows Forms."

—Dave Corun
Director of Microsoft
Technologies, Catalyst IT
Services

"Precisely what I needed to get up to speed quickly. Engaging."

—Joe Litton
Enterprise Application Developer
MCP, SCJP, PCLP

"Right on target."

—Jack Herrington
Author and Software Engineer

AUTHOR ONLINE
Ask the Author

Ebook edition

www.manning.com/eebrown2