



## C H A P T E R 4

---

# *Context menu and status strips*

- 4.1 Context menu strips 93
- 4.2 Drop-down events and event arguments 99
- 4.3 Status strips 108
- 4.4 Recap 115

We began chapter 3 with a quick tour through some of the Windows Forms classes and a discussion of menu objects in general, and saw how menus are part of the `ToolStrip` functionality provided by .NET. In this chapter we extend this discussion to cover context menus, as well as how to share a menu between a context menu and a menu bar.

The `StatusStrip` class is another kind of a tool strip, typically appearing at the bottom of a form to display various feedback to the user. This chapter looks at status bars in Windows Forms as well.

Figure 4.1 shows our application as it appears at the end of this discussion. In addition to our menus, you can see that the status bar contains two areas, called *panels* or *labels*. You can place any number of panels on a status bar, and display both textual and graphical information within each panel.

Continuing our tutorial approach, this chapter assumes you have the MyPhotos solution from section 3.4 available as the starting point for our discussion. You can also download this code from the book's website.



**Figure 4.1**  
Our status bar includes the optional sizing grip graphic at the lower right of the control, which allows a user to resize the form.

Before we get to status strips, we need to finish our menu discussion. We begin with the `ContextMenuStrip` class.

## 4.1 **CONTEXT MENU STRIPS**

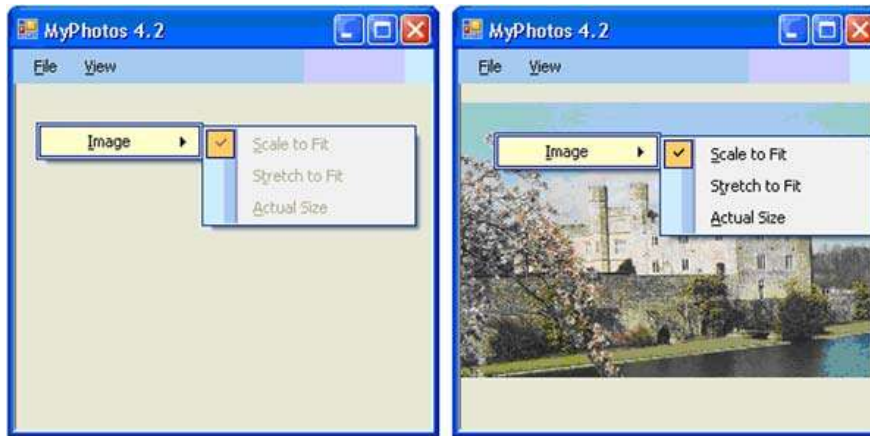
Although the creation of context menus requires some extra effort by a programmer, such menus improve the usability of an interface greatly and should be seriously considered in any application. The ability of a user to right-click a control and instantly see a list of commands is a powerful mechanism that experienced users especially appreciate. Context menus are typically associated with a specific graphical control, but can also be displayed programmatically. As a result, context menus provide quick access to commands immediately relevant to what the user is currently trying to accomplish or understand.

Most controls in the `System.Windows.Forms` namespace support the `ContextMenuStrip` property inherited from the `Control` class to specify a `ContextMenuStrip` object to associate with the control.<sup>1</sup> This setting can be changed dynamically to allow different context menus to display depending on the state of the control.

In this section we discuss context menus generally and add one to our `PictureBox` control. As shown in figure 4.2, this creates a shortcut for a user who wishes to alter how an image is displayed. We begin by adding a context menu to our application and populating its contents.

---


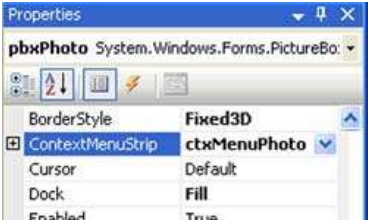
<sup>1</sup> All controls also support a `ContextMenu` property that links in the Win32-based `ContextMenu` class. These are fully supported, but do not appear in Visual Studio by default to discourage use of the Win32-based classes.



**Figure 4.2** Our application disables the Image submenu when no image is loaded, and marks the current display mode whenever the submenu is shown.

### 4.1.1 Creating a context menu

We begin by simply adding a new context menu to our application and associating it with the `pbxPhoto` control. The classes behind a context menu in .NET are discussed following these changes. The next section explains how to populate this menu with some menu items.

ADD A CONTEXT MENU		
	Action	Result
1	<p>Add a <code>ContextMenuStrip</code> object called <code>ctxMenuPhoto</code> to the form in the MainForm.cs [Design] window.</p> <p><b>How-to</b>            Drag a <code>ContextMenuStrip</code> item from the Menus &amp; Toolbars group in the Toolbox window onto the form, and set its (Name) to "ctxMenuPhoto."</p>	 <p>In the generated <code>InitializeComponent</code> method, the new context menu is defined and initialized.</p> <pre>private System.Windows.Forms.     ContextMenuStrip ctxMenuPhoto;</pre>
2	<p>Associate this new context menu with the <code>PictureBox</code> control.</p> <p><b>How-to</b></p> <ol style="list-style-type: none"> <li>Display the properties for the <code>pbxPhoto</code> control.</li> <li>Click to the right of the <code>ContextMenuStrip</code> entry.</li> <li>Click the down arrow.</li> <li>Select the <code>ctxMenuPhoto</code> item from the list.</li> </ol>	 <p>In the generated <code>InitializeComponent</code> method, the selected context menu is assigned to the picture box control.</p> <pre>private void InitializeComponent() {     . . .     this.pbxPhoto.ContextMenuStrip         = this.ctxMenuPhoto;</pre>

The `ContextMenuStrip` class is essentially a container for the `ToolStripItem` objects that appear within the menu. As we saw in figure 3.5, this class derives from the `ToolStripDropDownMenu` class, which in turn is based on the `ToolStripDropDown` class. While the core functionality for all of these controls comes from the `Control` and `ToolStrip` classes, of course, much of the drop-down functionality is defined by the `ToolStripDropDown` class. The key members of this class are shown in .NET Table 4.1.

**.NET Table 4.1** `ToolStripDropDown` class

**New in 2.0** The `ToolStripDropDown` class is a tool strip that displays a set of items from another tool strip item—for example, the drop-down list that appears when a `ToolStripDropDownButton` is clicked. This class is part of the `System.Windows.Forms` namespace, and inherits from the `ToolStrip` class. See .NET Table 16.1 for the members inherited from the `ToolStrip` class.


The `ToolStripDropDownMenu` class inherits from `ToolStripDropDown`, and is the base class for the `ContextMenuStrip` class.

<b>Public Properties</b>	<i>AutoClose</i>	Gets or sets whether the drop-down should automatically close when it loses focus.
	<i>CanOverflow</i> (overridden from <code>ToolStrip</code> )	Gets or sets whether items overflow. The default for drop-down strips is <code>false</code> .
	<i>OwnerItem</i>	Gets or sets the <code>ToolStripItem</code> that owns this drop-down strip.
<b>Public Methods</b>	<i>Close</i>	Closes the drop-down strip, optionally providing a reason as a <code>ToolStripDropDownCloseReason</code> value.
	<i>Show</i>	Displays the drop-down strip at the specified coordinates.
<b>Public Events</b>	<i>Closed</i>	Occurs after the drop-down strip has closed.
	<i>Closing</i>	Occurs just before the drop-down strip closes.
	<i>Opened</i>	Occurs after the drop-down strip is shown.
	<i>Opening</i>	Occurs before the drop-down strip is shown.

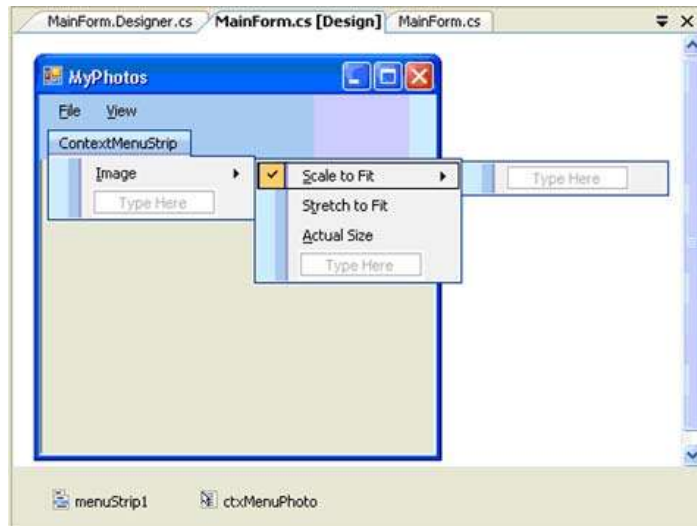
### 4.1.2 Adding items to a context menu

We are now ready to add items to our context menu, as illustrated in figure 4.3. Our context menu will define three different display settings for an image, corresponding to three different `PictureBoxSizeMode` enumeration values. The user can use this menu to alter how the image is displayed in the `PictureBox` control.

The creation of context menu items within Visual Studio is much the same as what we saw for menu strips in chapter 3: click the `ctxMenuView` object in the designer window to display a “Type Here” message, and enter the new menus. Since this is already familiar to us, the following table simply summarizes the changes required to add some menu items to our context menu strip.

POPULATE THE CONTEXT MENU STRIP														
	Action	Result												
1	<p>Add an Image menu item called “menuImage” to the context menu.</p>	<div></div> <p>A new ToolStripMenuItem object called menuImage is added to the MainForm.Designer.cs source code, and defined as a drop-down item for the ctxMenuPhoto menu.</p> <pre>. . . this.ctxMenuPhoto.Items.AddRange(     new System.Windows.Forms         .ToolStripItem[] {             this.menuImage         });</pre>												
2	<p>Add a “Scale to Fit” submenu item to the Image menu and assign its properties.</p> <p><b>How-to</b> Enter this item to the right of the Image menu.</p> <table><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>(Name)</td><td>menuImageScale</td></tr><tr><td>Checked</td><td>True</td></tr><tr><td>Text</td><td>&amp;Scale to Fit</td></tr></table>	Settings		Property	Value	(Name)	menuImageScale	Checked	True	Text	&Scale to Fit	<p>The new menu appears in Visual Studio as in figure 4.3. The menuImageScale menu item is created and initialized in the generated source code, and defined as a drop-down item for the menuImage menu.</p> <pre>// // menuImage //  this.menuImage.DropDownItems.AddRange(     new System.Windows.Forms.         ToolStripItem[] {             this.menuImageScale}); . . .</pre>		
Settings														
Property	Value													
(Name)	menuImageScale													
Checked	True													
Text	&Scale to Fit													
3	<p>Similarly, add the “Stretch to Fit” and “Actual Size” menus as submenus of the Image menu.</p> <table><tr><th colspan="3">Settings</th></tr><tr><th>Menu</th><th>Property</th><th>Value</th></tr><tr><td>Stretch to Fit</td><td>(Name) Text</td><td>menuImageStretch S&amp;tretch to Fit</td></tr><tr><td>Actual Size</td><td>(Name) Text</td><td>menuImageActual &amp;Actual Size</td></tr></table>	Settings			Menu	Property	Value	Stretch to Fit	(Name) Text	menuImageStretch S&tretch to Fit	Actual Size	(Name) Text	menuImageActual &Actual Size	<p>The new menus appear in Visual Studio, and are reflected in the generated source code. In particular, an array of ToolStripItem objects is created and defined as the drop-down items for the menuImage menu.</p> <pre>this.menuImage.DropDownItems.AddRange(     new System.Windows.Forms.         ToolStripItem[] {             this.menuImageScale,             this.menuImageStretch,             this.menuImageActual});</pre>
Settings														
Menu	Property	Value												
Stretch to Fit	(Name) Text	menuImageStretch S&tretch to Fit												
Actual Size	(Name) Text	menuImageActual &Actual Size												

The code generated in MainForm.Designer.cs for our context menu is similar to the code we examined in chapter 3, so we do not review it again here. Realize that the menus here are all the same kind of control—a ToolStripMenuItem object—regardless of where they happen to appear in the interface. The Image menu item, as well as the three submenu items, are all objects of type ToolStripMenuItem.



**Figure 4.3** The “Type Here” in the menu editor indicates where new items can be added, and the area in front of each item holds an optional image or check mark.

Before we actually process our context menu, it would be useful to have the context menu items accessible from the menu strip in our application as well. The next section shows how these items can be shared so they appear in both a new View menu and our context menu strip.

### 4.1.3 Sharing a context menu

While it is certainly possible to re-create the Image menu within our menu strip, it does not seem like the most elegant solution. Having this menu in two places would require that we update one menu when we make any change to the other. Ideally, we would prefer to have a single menu, and find a way to share this menu between the two strip objects.

The solution is to implement the menus in our context menu and then reuse these menus as the drop-down for a new View menu in the application. This is possible since both the context menu and menu item classes can both contain a submenu collection. The `ContextMenuStrip` class is based on the `ToolStripDropDown` class and defines the `Items` property to hold a collection of `ToolStripItem` instances; the `ToolStripMenuItem` object is based on the `ToolStripDropDownItem` class, which defines a `DropDown` property that holds a `ToolStripDropDown` instance.

As we will see, this allows us to assign a context menu as the drop-down for a menu item. The following code illustrates this capability:

```
// Create a menu strip with a single top-level item
MenuStrip ms = new MenuStrip();
ToolStripMenuItem menuTop = new ToolStripMenuItem("Top");
ms.Items.Add(menuTop);

// Create a context menu with three menu items
```

```

ContextMenuStrip ctxMenu= new ContextMenuStrip();
ctxMenu.Items.Add("Item 1");
ctxMenu.Items.Add("Item 2");
ctxMenu.Items.Add("Item 3");

// Assign context menu as dropdown for the top-level item
menuTop.DropDown = ctxMenu;

```

An overview of the `ToolStripDropDownItem` class is shown in .NET Table 4.2. As we saw in chapter 3 with our File menu, menu items use the `DropDownItems` property to define the individual items within the menu. The `DropDown` property used in the prior code snippet allows an existing `ToolStripDropDown` object to be assigned as the drop-down list.

**.NET Table 4.2** `ToolStripDropDownItem` class

**New in 2.0** The `ToolStripDropDownItem` class is a tool strip item that displays a set of drop-down items when clicked. This class is the basis for the drop-down button item, menu item, and other tool strip items with drop-down functionality.

The `ToolStripDropDownItem` class is part of the `System.Windows.Forms` namespace, and inherits from the `ToolStripItem` class. Members inherited from `ToolStripItem` are shown in .NET Table 3.4.

<b>Public Properties</b>	<i>DropDown</i>	Gets or sets the <code>ToolStripDropDown</code> object that is displayed when the item is clicked
	<i>DropDownItems</i>	Gets the collection of items in the <code>ToolStripDropDown</code> class associated with this drop-down item
	<i>DropDownDirection</i>	Gets or sets a <code>ToolStripDropDownDirection</code> enumeration value that indicates where the drop-down control is displayed relative to its parent control
	<i>HasDropDownItems</i>	Gets whether the drop-down list contains any items
<b>Public Methods</b>	<i>HideDropDown</i>	Hides the drop-down associated with the item
	<i>ShowDropDown</i>	Displays the <code>ToolStripDropDown</code> control associated with this item
<b>Public Events</b>	<i>DropDownClosed</i>	Occurs when the <code>ToolStripDropDown</code> control associated with this item has closed
	<i>DropDownItemClicked</i>	Occurs when an item within the associated <code>ToolStripDropDown</code> control has been clicked
	<i>DropDownOpened</i>	Occurs when the <code>ToolStripDropDown</code> control associated with this item has opened
	<i>DropDownOpening</i>	Occurs when the <code>ToolStripDropDown</code> control associated with this item is about to open

The `ToolStripDropDownItem` class also provides properties to indicate where the drop-down list is displayed and whether it contains any items. It also defines methods to show or hide the list, and events that fire as the list is opened and closed. In our application, we need to create the View menu to hold the contents of our context menu. As just discussed, we can assign the `DropDown` property of this menu to our context menu. The following steps illustrate this procedure.

Assign the Context Menu Strip as the View Menu Drop-Down										
	Action	Result								
1	<p>In the MainForm.cs [Design] window, add a top-level View menu to the right of our existing File menu.</p> <table><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>(Name)</td><td>menuView</td></tr><tr><td>Text</td><td>&amp;View</td></tr></table>	Settings		Property	Value	(Name)	menuView	Text	&View	<pre>private void InitializeComponent() {     . . .     this.menuStrip1.Items.AddRange(         new System...ToolStripItem[] {             this.menuFile,             this.menuView});     . . .     //     // menuView     //     this.menuView.Name = "menuView";     . . . }  . . . private System.Windows.     Forms.ToolStripMenuItem menuView;</pre>
Settings										
Property	Value									
(Name)	menuView									
Text	&View									
2	<p>In the MainForm constructor, add a line that assigns the DropDown property for the View menu to the ctxMenuPhoto context menu.</p>	<pre>public MainForm() {     InitializeComponent();     SetTitleBar();     <b>menuView.DropDown = ctxMenuPhoto;</b> }</pre>								

This change allows both a right-click on the picture box and a click on the new View menu to display the same menu. Compile and run this code to see this in action.

With our menus defined, we are ready for some event handlers for the Image menu to alter how the image displays based on the selected setting.

## 4.2 DROP-DOWN EVENTS AND EVENT ARGUMENTS

The Load and Exit menu items in our File menu are fairly straightforward as menus go. Each item raises a `Click` event when selected, and the associated event handler performs the appropriate action. In our Image menu, we need to alter the `PictureBox.SizeMode` setting whenever an Image submenu item is selected. One obvious solution is to handle the `Click` event associated with each submenu item, and explicitly alter the `SizeMode` property to the appropriate setting.

While this would work just fine, it is not our approach. Instead, we would prefer to process all three submenu items in a single event handler. There are a couple of reasons for this. First, of course, it provides an alternate example for this fine book



you are reading. More importantly, it encapsulates the logic for these menus in a single handler. If we later need to make a change or add a new submenu item, we only need to modify this one handler.

Another feature we'd like to add is the ability to check the submenu item associated with the current display setting. The `Click` event for the Image menu is still raised, so we could process a mouse click on the Image menu as the submenu is displayed, and update these menus accordingly.<sup>2</sup> Conceptually, the user is looking to display the submenu contents, not click the parent menu, so the `Click` event is not the right paradigm for this purpose.

Our Image menu is the parent menu for the Scale to Fit, Stretch to Fit, and Actual Size menu items. When the user clicks such a parent menu, they expect to see the submenu associated with the menu. The `ToolStripMenuItem` class supports various drop-down events before, during, and after such a submenu is displayed. This permits an event handler to modify the contents or appearance of the submenu as dictated by the application. An example of this concept can be found in the Windows operating system. Open the My Computer window and display the File menu. The contents of this menu change depending on the type of file currently selected.

For Windows Forms menu strips, the menu item drop-down events are inherited from the `ToolStripDropDownItem` class, shown in .NET Table 4.2. The drop-down events are activated when a user clicks the menu in order to display the collection of items in the associated drop-down object.

In this section we illustrate the `DropDownItemClicked` and `DropDownOpening` events to alter the appearance and behavior of the Image submenu items. The other drop-down events are handled in a similar manner.

#### 4.2.1 Handling a submenu item click

The submenu for the Image menu item pops up whenever the Image menu is clicked. When the user clicks an Image submenu item, we want the image display to change accordingly. To do this, we will assign the `SizeMode` property of our `PictureBox` control depending on which menu is selected. The `SizeMode` values are taken from the `PictureBoxSizeMode` enumeration, as shown in .NET Table 4.3.

As mentioned in the section intro, we use the drop-down events to manage the behavior of the Image submenu. The drop-down events work well in cases like this where the submenu contains a set of values or other related items that are applied in a similar fashion. These handlers take advantage of the fact that our View menu and context menu share the same menu items.

To facilitate this amazing behavior, we begin by employing the `Tag` property in our menus to record the desired `SizeMode` value for each menu.

---

<sup>2</sup> This is a change in behavior from the Win32-based menus encapsulated in the `Menu` and `MenuItem` classes. In these classes, the `Click` event is only raised if the menu does not contain a submenu.

Define the SizeMode settings for the Image submenu											
	Action	Result									
1	From the MainForm.cs [Design] window, assign the Tag property for each item in the Image submenu to contain the desired PictureBoxSizeMode enumeration value.	<div>The settings are assigned in the InitializeComponent method of the generated designer file.</div> <pre>this.menuImageScale.Tag = "Zoom"; ... this.menuImageStretch.Tag     = "StretchImage"; ... this.menuImageActual.Tag = "Normal";</pre>									
	<div>Settings</div> <table><tr><th>Menu Item</th><th>Tag Property</th></tr><tr><td>Scale to Fit</td><td>Zoom</td></tr><tr><td>Stretch to Fit</td><td>StretchImage</td></tr><tr><td>Actual Size</td><td>Normal</td></tr></table>		Menu Item	Tag Property	Scale to Fit	Zoom	Stretch to Fit	StretchImage	Actual Size	Normal	
	Menu Item		Tag Property								
Scale to Fit	Zoom										
Stretch to Fit	StretchImage										
Actual Size	Normal										

**.NET Table 4.3 PictureBoxSizeMode enumeration**

The `PictureBoxSizeMode` enumeration specifies the possible display modes for the `PictureBox` control, and is used with the `PictureBox.SizeMode` property. This enumeration is part of the `System.Windows.Forms` namespace.

<b>Enumeration Values</b>	AutoSize	The size of the <code>PictureBox</code> control adjusts automatically to the size of the contained image.
	CenterImage	The image is centered within the <code>PictureBox</code> control, clipping the outside edges of the image if necessary.
	Normal	The image is placed in the upper-left corner of the <code>PictureBox</code> control, clipping the right and bottom of the image as necessary.
	StretchImage	The image is stretched or shrunk to fit within the <code>PictureBox</code> control.
	Zoom	The image is scaled to fit within the <code>PictureBox</code> control, so that the aspect ratio of the image is preserved.

The `Tag` property is available in most Windows Forms components, including `ToolStripItem` objects, and allows pretty much anything to be associated with the component. The `Control` class defines this property for all Windows Forms controls. The property is of type `object`, so any class, structure, or other value can be assigned. In our code, we assign the string value of the enumeration setting we wish to associate with each item.

With these values set, we are ready to handle the events. The `DropDownItemClicked` event occurs when a child item is clicked. As mentioned in chapter 1, all .NET event handlers employ a common set of arguments. The first parameter is the object sending the event, while the second parameter is the event argument object.

For basic events like `Click`, as we have seen, the `EventArgs` class is a placeholder for this second parameter.

For more complex events, including the `DropDownItemClicked` event, more details are required to properly process the event. In such cases, a new class is derived from the base `EventArgs` class to hold this information. Let's see how this works by continuing our changes.

ADD DROP-DOWN EVENT HANDLERS FOR THE IMAGE SUBMENU		
	Action	Result
2	Add a <code>DropDownItemClicked</code> event handler to the Image menu.  <b>How-to</b> Display the events for the Image menu item and double-click the <code>DropDownItemClicked</code> event.	A new method is registered for the <code>menuImage</code> object in the generated designer file.  <pre> this.menuImage.DropDownItemClicked += new System.Windows.Forms     .ToolStripItemClickedEventHandler(         this.menuImage_DropDownItemClicked); </pre> The <code>MainForm.cs</code> code window is shown with the cursor at the beginning of this new method.  <pre> private void menuImage_DropDownItemClicked(     object sender,     ToolStripItemClickedEventArgs e) { </pre>
3	Within this handler, call a <code>ProcessImageClick</code> method with the given arguments.	<pre>     ProcessImageClick(e); } </pre>
4	Implement the <code>ProcessImageClick</code> method to modify the <code>SizeMode</code> property for the picture box control based on the selected item.	<pre> private void ProcessImageClick(     ToolStripItemClickedEventArgs e) {     ToolStripItem item = e.ClickedItem;     string enumVal = item.Tag as string;     if (enumVal != null)     {         pbxPhoto.SizeMode = (PictureBoxSizeMode)             Enum.Parse(typeof(PictureBoxSizeMode),                 enumVal);     } } </pre>

There is a lot going on here, so let's break this down to see exactly what is happening. First, note in step 2 how the event handler is assigned to our Image menu. For the `Click` event, the base `EventHandler` class is used to define this handler. Events that define an event argument other than the `EventArgs` class require an alternate event handler class. By convention, the event handler and event argument class for such events utilize the same name with different suffixes. For the `DropDownItemClicked` event, the event argument is a `ToolStripItemClickedEventArgs` object and the event handler is a `ToolStripItemClickedEventHandler` object.

#### Listing 4.1 ProcessItemClick method

```
private void ProcessItemClick(ToolStripItemClickedEventArgs e)
{
    ToolStripItem item = e.ClickedItem;
    string enumVal = item.Tag as string;
    if (enumVal != null)
    {
        pbxPhoto.SizeMode = (PictureBoxSizeMode)
            Enum.Parse(typeof(PictureBoxSizeMode), enumVal);
    }
}
```

1 Receives ItemClicked event argument

2 Converts Tag value to string

3 Converts string to SizeMode value

Listing 4.1 displays the code for the method that supports our new event handler. Let's look at the numbered points in this listing in more detail:

- 1 The `DropDownItemClicked` event handler receives the sender parameter, much like our `Click` event handlers earlier in the chapter. It also receives some event data as a `ToolStripItemClickedEventArgs` instance, derived from the `EventArgs` class as shown in figure 4.4. This event argument class provides a `ClickedItem` property that retrieves the associated `ToolStripItem` object clicked by the user. Our code utilizes this property to alter its behavior depending on which drop-down item was clicked.
- 2 Once we determine the item clicked by the user, we make use of the `Tag` property containing the desired `SizeMode` string value. Since the `Tag` property is an object, we must convert its value to a string to obtain the assigned value. The C# `as` keyword converts a variable to a given reference type, in this case a string object. If the variable cannot be converted to the given type, then `null` is returned.<sup>3</sup> In our example, we know that the `Tag` property always contains a string value. Still, it is good practice to check for `null` anyway, since you never know how the program may evolve over time.
- 3 The .NET Framework contains a special `Enum` structure that is the implicit base class for all enumerations. This class contains various static methods for manipulating enumeration values and their corresponding strings, and is summarized in .NET



**Figure 4.4** All event argument parameters inherit from the base `EventArgs` class.

<sup>3</sup> You can also cast the value directly to a string, using the code “`string enumVal = (string)item.Tag;`”. This code throws an `InvalidCastException` object if the cast cannot be performed, as opposed to the `as` keyword, which simply assigns the `null` value.

Table 4.4. The code here includes a few new concepts, so let's break this line down a bit further to see what is happening.

**.NET Table 4.4 Enum structure**

The `Enum` structure is the implicit base class for all enumerations. This structure is part of the `System` namespace, and implements the `IComparable`, `IFormattable`, and `IConvertible` interfaces.

<b>Public Methods</b>	<code>CompareTo</code>	Returns whether a given object is less than, equal to, or greater than the enumeration value
	<code>GetTypeCode</code>	Returns the <code>TypeCode</code> enumeration value representing the underlying <code>Type</code> of values for this instance
	<code>ToString</code> (overridden from <code>Object</code> )	Returns the string representation of the current value
<b>Public Static Methods</b>	<code>Format</code>	Converts a given value for a given enumeration type to a string, based on a provided format
	<code>GetNames</code>	Returns the array of string constants for a given enumeration type
	<code>GetValues</code>	Returns the array of values for the constants in a given enumeration type
	<code>IsDefined</code>	Returns whether a constant with a given value is defined in the given enumeration type
	<code>Parse</code>	Converts one or more string constants for a given enumeration type to the appropriate enumeration value

We use the `Enum.Parse` method to convert our `string` value to the appropriate `PictureBoxSizeMode` enumeration value. The `Parse` method used here takes an enumeration type and a `string`, and returns an object representing the enumeration value. It throws an `ArgumentException` object if an invalid `string` is provided.

The enumeration type is obtained using the `typeof` keyword. This keyword returns a `Type` object that represents the given type, in this case the `PictureBoxSizeMode` enumeration type.

Since the `Parse` method returns an object value, it must be converted to the appropriate enumeration value. This conversion is “down” the class hierarchy from the generic object type to the more specific `PictureBoxSizeMode` enumeration. In C++, such operations are dangerous since the language does not provide explicit support for such a downcast, as it is called. In C#, downcasting is explicitly permitted, and an illegal cast throws an exception of type `InvalidCastException`.

Using the `DropDownItemClicked` event ensures that our `PictureBox` is updated with the appropriate display behavior, regardless of which item was selected. Compile and run this code to see it in action.

**TRY IT!** The `PictureBoxSizeMode` enumeration contains more than just the three settings used here. Add a menu item to the Image menu called `menuImageCenter`, with the text “Center Image,” to handle the `CenterImage` value. Set the `Tag` property appropriately to see how our existing code automatically handles the new menu.

## 4.2.2 Altering a submenu before it appears

Users appreciate feedback from an application. Our current interface does not yet do this. The user has to understand the possible display modes in order to know what is currently selected, and then choose a different setting. A more intuitive interface would highlight the current selection in the `menuImage` submenu. This would immediately indicate what mode is currently displayed, and help our user make a more informed selection.

The `ToolStripMenuItem` class provides a `Checked` property that, when true, displays a check mark next to the menu. We could set this property explicitly whenever the selection is modified, so our user would see the appropriate feedback. Of course, as our program changes, there might be other commands or user interactions that alter the display mode of the image, so this approach could get complicated. An alternate solution might ensure that the display modes are checked or unchecked as they are displayed to the user. This approach is more robust in the face of future changes, creating an application that users, documenters, and testers will appreciate for years to come.

The `DropDownOpening` event is designed for just this purpose. This event occurs just before the drop-down list associated with an item is displayed, allowing its appearance or contents to be modified and then immediately displayed to the user. Let’s see how this works.

IMPLEMENT A <code>DROPDOWNOPENING</code> HANDLER FOR THE IMAGE MENU		
	Action	Result
1	In the <code>MainForm.cs</code> [Design] window, add a <code>DropDownOpening</code> event handler for the Image menu.	<pre>private void menuImage_DropDownOpening(     object sender, EventArgs e) {</pre>
2	Within this handler, call a <code>ProcessImageOpening</code> method, with the given drop-down item as an argument.	<pre>    ProcessImageOpening(         sender as ToolStripDropDownItem); }</pre>

IMPLEMENT A DROPDOWNOPENING HANDLER FOR THE IMAGE MENU (CONTINUED)		
	Action	Result
3	Implement the ProcessImageOpening method to assign the Enabled and Checked property for each menu item in the given drop-down list.	<pre> private void ProcessImageOpening(     ToolStripDropDownItem parent) {     if (parent != null)     {         string enumVal = pbxPhoto.SizeMode.ToString();         foreach (ToolStripMenuItem item             in parent.DropDownItems)         {             item.Enabled = (pbxPhoto.Image != null);             item.Checked = item.Tag.Equals(enumVal);         }     } } </pre>

The handler for the `DropDownOpening` event is similar to what we have seen for the `Click` event, with an object as the first argument and an `EventArgs` object as the second parameter. Our implementation invokes a `ProcessImageOpening` method to perform the desired actions. We could avoid this method and implement the logic directly in the handler. Using a supporting method as we do here is useful if you ever want to invoke the same functionality in another part of the program.

The `ProcessImageOpening` method ensures the given item is not null, and uses the `ToString` method to obtain the string representation of the current `SizeMode` enumeration value. It also uses the C# `foreach` keyword, which provides an easy way to enumerate the items in a collection. A `foreach` loop is much like a `for` loop, except that each iteration of the loop processes the next item in the collection rather than the next integer-based value. We discuss this concept in more detail in chapter 5.

For each submenu item in the Image menu, our code sets the `Enabled` property based on whether the `PictureBox` control currently contains an image, and the `Checked` property based on a comparison of the `SizeMode` string with the subitem's `Tag` value. The `ToString` and `Equals` methods used here are inherited from the base object class, which we also discuss in chapter 5.

Notice that there is nothing in the `ProcessImageOpening` method to indicate whether these menu items are part of a specific menu structure. The code works identically whether displayed from the View menu or as part of the context menu. Compile and run the application to verify that the menus



**Figure 4.5** Our Actual `SizeMode` display mode displays every pixel in the image within the window, starting with the upper-left corner.

work correctly. Figure 4.5 shows the application with an image displayed in Actual Size mode.

Unfortunately, this figure reveals a problem with our `PictureBox` control. In the figure, the image is larger than the display area, but there is no way to see the rest of the image without resizing the window. While this is possible when the image is small, a high-resolution image may contain more pixels than our screen. Ideally, the application would display scroll bars here. Since the `PictureBox` control does not support scroll bars, we are a bit stuck unless we display the picture in an alternate display mode.

You may be wondering about a book that teaches you how to build an application that doesn't quite work, and you should. We discuss how to solve this problem in chapter 13 using a scrollable panel or tab page to contain the picture box, and again in chapter 19 where we build a custom picture box that solves this problem directly.

**TRY IT!** Okay, I admit this has nothing to do with our application. Still, if you want to have fun with the `DropDownOpening` event, add a new menu, `menuCounter`, at the bottom of the `ctxMenuPhoto` context menu with the text "Counter" and insert a single menu with the text "DropDown" in its submenu. Define a `DropDownOpening` event for the `menuCounter` menu, which Visual Studio will name `menuCounter_DropDownOpening`. In this handler, dynamically create a new `ToolStripMenuItem` object and add it to the end of the `menuCounter` submenu. Set the `Text` property to your new menu to "Count #," where # is the number of drop-downs that have occurred on your new menu. Use a static integer `dropdownCount` in the `MainForm` class to track the number of drop-down occurrences. The lines to dynamically create the new menu in your `DropDownOpening` handler should look something like this:

```
ToolStripMenuItem mi = new ToolStripMenuItem();  
mi.Text = "Count " + dropdownCount.ToString();  
menuCounter.DropDownItems.Add(mi);
```

This example illustrates how easy it is to create menus on the fly with the .NET Framework, and how a parent menu can change the contents of its submenu as it is displayed. This might be used, for example, to display a list of files most recently opened by an application.

If all this makes no sense to you, download the code for this TRY IT! from the book's website. Have a look at the `menuCounter_DropDownOpening` handler to see the code required.

This concludes our discussion of menus for now. Before we leave this chapter, let's take a look at another type of tool strip: the status strip.



## 4.3 STATUS STRIPS

Most applications make a lot of information available to users. There is often a core subset that most users would appreciate having readily available. A status bar at the base of a window is a good place for this type of data, as it provides quick feedback related to the current task or cursor position. My word processor, for example, indicates the current page number, total number of pages, column and line position of the cursor, whether the Insert key has been pressed (which I seem to hit constantly while aiming for the Page Down key), and other information I may want to know at a glance. This helps me keep track of how this book is shaping up, when the Insert key has been pressed, and where these words you are reading appear on the page.

Status bars can also contain graphical information such as the status of a print request, whether the application is connected to the Internet, and pretty much anything else you can draw or animate.

In .NET, status bars are represented by the `StatusStrip` class, which is part of the `ToolStrip` class hierarchy we have been discussing. As a way to round out our discussion and examine another kind of tool strip, this section takes a quick look at this functionality. We discuss status strips and status panels, or labels, and add the status bar we saw in figure 4.1 to our MyPhotos application.

We should note that Win32 status bars are also supported by the `StatusBar` and `StatusBarPanel` classes. These classes are supported for existing and new applications, but are hidden in Visual Studio to encourage use of the new tool strip classes.

### 4.3.1 Creating a status strip

The `StatusStrip` control inherits directly from the base `ToolStrip` class. A summary of this control appears in .NET Table 4.5. We will not discuss the details of the

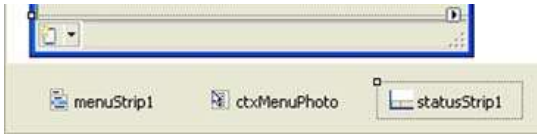
**.NET Table 4.5** `StatusStrip` class

**New in 2.0** The `StatusStrip` class is a tool strip used to show a status bar on a form. This class can display a collection of labels in the form of `ToolStripStatusLabel` objects. The `StatusStrip` class is part of the `System.Windows.Forms` namespace, and inherits from the `ToolStrip` class. See .NET Table 16.1 for a list of inherited members, and .NET Table 4.7 for information on the `ToolStripStatusLabel` class.

<b>Public Properties</b>	<code>Dock</code> (overridden from <code>Control</code> )	Gets or sets the docking style for the control. The default for status strips is <code>Bottom</code> .
	<code>GripStyle</code> (overridden from <code>ToolStrip</code> )	Gets or sets whether the move handle is hidden or visible. The default for status strips is <code>Hidden</code> .
	<code>ShowItemToolTips</code> (overridden from <code>ToolStrip</code> )	Gets or sets whether items display their tooltips. The default for status strips is <code>false</code> .
	<code>SizingGrip</code>	Gets or sets whether the sizing grip for resizing the form appears in the status bar.
	<code>Stretch</code> (overridden from <code>ToolStrip</code> )	Gets or sets whether the strip expands to fill the width of its parent's container. The default for status bars is <code>true</code> .

ToolStrip class until chapter 16, but one detail to take from our table here is that the StatusStrip class is basically a ToolStrip that overrides the base functionality to achieve the appearance of a Win32-style status bar. This includes docking the strip at the bottom of the form, hiding the move handle grip, not showing any tooltips, and stretching the strip across the entire width of the form.

A status strip can technically appear within any parent container and docked to any side of this container, so traditionally this control is placed only at the base of Form objects. The following step adds a status strip to our MyPhotos application.

ADD A STATUS STRIP TO OUR APPLICATION		
	Action	Result
1	In the MainForm.cs [Design] window, drag a StatusStrip control from the Toolbox window onto the form.	<p>The new status strip appears at the base of the form in the designer window, and in the component tray below the form.</p> 

Notice how the interface for populating a status strip is quite different than the one employed for menus. The small, mostly white icon on the left adds a ToolStripStatusLabel object, while the drop-down arrow next to this icon displays a context menu of various ToolStripItem objects that can be added to the strip.

The code generated in the MainForm.Designer.cs file is much like the code we have seen when adding other controls to our form. A statusStrip1 field is created at the top of the InitializeComponent method, its nondefault values assigned in the middle and added to the Form at the bottom of the method. The field is defined at the end of the generated file, as shown here:

```
private System.Windows.Forms.StatusStrip statusStrip1;
```

Most status strips contain one or more status panels, or labels. This is our next topic.

### 4.3.2 Adding status strip labels

In the Win32-based classes, status bars contain panels: the Windows Forms StatusBar class represents a status bar, and contains one or more StatusBarPanel components. In the tool strip world, the terminology is slightly different. The StatusStrip class represents a status bar, and contains one or more ToolStripStatusLabel objects. We will adopt the term *status labels* in this book to stay consistent with the new class name.

The ToolStripStatusLabel class for status labels is based on the ToolStripLabel class. A summary of this class appears in .NET Table 4.6. The ToolStripStatusLabel class adds some additional functionality specific to status bars, and is summarized in .NET Table 4.7.

**.NET Table 4.6 ToolStripLabel class**

**New in 2.0** The `ToolStripLabel` class is a tool strip item that displays nonselectable text and image as well as hyperlink items. The `ToolStripLabel` class is part of the `System.Windows.Forms` namespace, and inherits from the `ToolStripItem` class. See .NET Table 3.4 for a list of inherited members.

<b>Public Properties</b>	<i>CanSelect</i> (overridden from <code>ToolStripItem</code> )	Gets whether the contents of the item are selectable. Always <code>false</code> for label items.
	<i>IsLink</i>	Gets or sets whether the label is a hyperlink.
	<i>LinkBehavior</i>	Gets or sets the <code>LinkBehavior</code> enumeration value representing the appearance of a link.
	<i>LinkColor</i>	Gets or sets the color to use for a normal link. The <code>ActiveLinkColor</code> and <code>VisitedLinkColor</code> properties represent the color to use for an active and visited link, respectively.
	<i>LinkVisited</i>	Gets or sets whether the link should display as though it were visited.

As you can see in .NET Table 4.6, the `ToolStripLabel` class overrides the `CanSelect` property from the `ToolStripItem` class to enforce that such labels cannot be selected. The other properties relate to treating the label as a hyperlink. In particular, the `IsLink` property identifies whether the label appears as a hyperlink. A `Click` event handler can be used to process the link when it is clicked.

In .NET Table 4.7, the `ToolStripLabel` class is extended to work within a status strip. As you can see, three new properties are added: two to define the borders for the label, and a `Spring` property that allows the label to grow or contract as the status strip is resized.

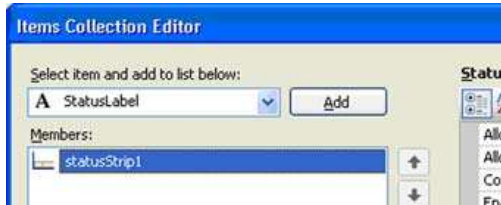
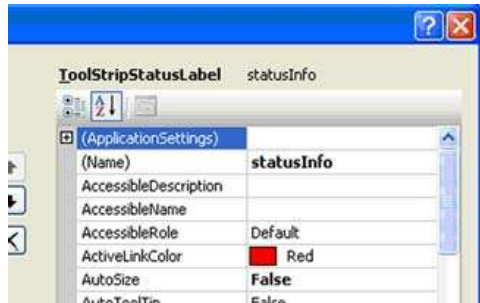
**.NET Table 4.7 ToolStripStatusLabel class**


**New in 2.0** The `ToolStripStatusLabel` class is a label item for use within a `StatusStrip` control. The `ToolStripStatusLabel` class is part of the `System.Windows.Forms` namespace, and inherits from the `ToolStripLabel` class.

<b>Public Properties</b>	<i>BorderSides</i>	Gets or sets the sides of the status label that should display borders
	<i>BorderStyle</i>	Gets or sets the <code>Border3DStyle</code> enumeration value for how the label's borders should appear
	<i>Spring</i>	Gets or sets whether the panel should expand or shrink to fit the available space when the status strip is resized

Returning to our MyPhotos application, we already have a `StatusStrip` object on our form, so the next task is to define a reasonable set of labels. We start with the three labels mentioned at the start of the chapter.

When we added a status strip to our form, we mentioned the interface provided to add labels and other items to the strip. While we could use this interface here, we instead use the Items Collection Editor window as an alternate method for this task. This editor is available for other tool strips as well, including menu strips.

ADD STATUS LABELS TO A STATUS STRIP																
	Action	Result														
1	<p>In the MainForm.cs [Design] window, display the items for the status strip.</p> <p><b>How-to</b> Right-click the status strip and select the Edit Items entry.</p> <p><b>Alternately</b> Click the smart tag associated with the status strip, and click Edit Items in the StatusStrip Tasks window.</p>	<p>The Items Collection Editor window appears. This editor allows a collection of items, in this case tool strip items on a status strip, to be configured for a parent control. A portion of this window is shown here.</p> 														
2	Click the Add button twice to add two status labels to the control.	The new ToolStripStatusLabel objects are shown in the editor.														
3	<p>Modify the properties of the first label as follows.</p> <table><thead><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr></thead><tbody><tr><td>(Name)</td><td>statusInfo</td></tr><tr><td>AutoSize</td><td>False</td></tr><tr><td>Spring</td><td>True</td></tr><tr><td>Text</td><td>Desc</td></tr><tr><td>TextAlign</td><td>MiddleLeft</td></tr></tbody></table>	Settings		Property	Value	(Name)	statusInfo	AutoSize	False	Spring	True	Text	Desc	TextAlign	MiddleLeft	
Settings																
Property	Value															
(Name)	statusInfo															
AutoSize	False															
Spring	True															
Text	Desc															
TextAlign	MiddleLeft															
4	Close the Item Collections Editor window by clicking the OK button.	The two status labels are shown in the designer window.														

ADD STATUS LABELS TO A STATUS STRIP (CONTINUED)													
	Action	Result											
5	Click the second status label and modify its properties as follows.	The properties are assigned in the generated MainForm.Designer.cs file.  <pre>// // statusImageSize // this.statusImageSize.BorderSides = ((System.Windows.Forms.     ToolStripStatusLabelBorderSides)     (((System.Windows.Forms.ToolStrip-         StatusLabelBorderSides.Left           System.Windows.Forms.ToolStrip-             StatusLabelBorderSides.Top)           System.Windows.Forms.ToolStrip-             StatusLabelBorderSides.Right)           System.Windows.Forms.ToolStrip-             StatusLabelBorderSides.Bottom)     )); this.statusImageSize.BorderStyle = System.Windows.Forms.     Border3DStyle.SunkenInner; this.statusImageSize.Name = "statusImageSize"; this.statusImageSize.Size = new System.Drawing.Size(40, 17); this.statusImageSize.Text = "W x H";</pre>											
	<table><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>(Name)</td><td>statusImageSize</td></tr><tr><td>BorderSides</td><td>All</td></tr><tr><td>BorderStyle</td><td>SunkenInner</td></tr><tr><td>Text</td><td>W x H</td></tr></table>		Settings		Property	Value	(Name)	statusImageSize	BorderSides	All	BorderStyle	SunkenInner	Text
Settings													
Property	Value												
(Name)	statusImageSize												
BorderSides	All												
BorderStyle	SunkenInner												
Text	W x H												
6	Add a third status label to the status strip.	The third label appears in the designer window, and is initialized in the generated code. The three status labels are added to the Items property of the StatusStrip object using the AddRange method.  <pre>this.statusStrip1.Items.AddRange(new     System.Windows.Forms.ToolStripItem[] {         this.statusInfo,         this.statusImageSize,         this.statusAlbumPos});</pre>											
	<p><b>How-to</b></p> <p>Click the small white icon at the right of the strip.</p>  <table><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>(Name)</td><td>statusAlbumPos</td></tr><tr><td>BorderSides</td><td>All</td></tr><tr><td>BorderStyle</td><td>SunkenInner</td></tr><tr><td>Text</td><td>1 / 1</td></tr></table>		Settings		Property	Value	(Name)	statusAlbumPos	BorderSides	All	BorderStyle	SunkenInner	Text
Settings													
Property	Value												
(Name)	statusAlbumPos												
BorderSides	All												
BorderStyle	SunkenInner												
Text	1 / 1												

These three labels will show a description of the current image, the size of the image, and the current position of the image within an album. Of course, the album position label is not needed in this chapter. We add it here to save time and space in chapter 6, when our application displays an album rather than a single image.

In our property settings, note how the `BorderSides` and `BorderStyle` properties work together to define how the label appears within the status strip. In our

example we set our size and position labels to have borders on all four sides using the `SunkenInner` value from the `Border3DStyle` enumeration.

As for the other two labels, the description and size, we should update them whenever we load an image to contain the filename as the description, and the width and height in pixels as the size. The `Text` property inherited from the `ToolStripItem` class defines the text to display on the label. You can also display an image by assigning the `Image` property, but we do not make use of this here.

One approach for updating these labels might be to assign each label's text in the `Click` event handler of our `Load` menu directly. This is another example where the most obvious approach may not be the best long-term solution. Since our status strip is likely to change, we encapsulate this logic in a private method. We continue our prior steps to make these changes.

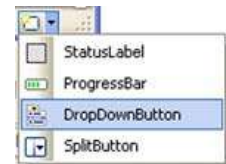
ASSIGN TEXT TO STATUS LABELS		
	Action	Result
7	In the <code>MainForm.cs</code> code window, define a new private method called <code>SetStatusStrip</code> that accepts a file path as the only argument.	<pre>private void SetStatusStrip(string path) {</pre>
8	If an image is assigned to the picture box, set the <code>statusInfo</code> label to the given path.	<pre>    if (pbxPhoto.Image != null)     {         statusInfo.Text = path;</pre>
9	Also set the <code>statusImageSize</code> label to the size of the image. <b>How-to</b> Use the <code>Width</code> and <code>Height</code> properties in the <code>Image</code> class.	<pre>        statusImageSize.Text             = String.Format("{0:#}x{1:#}",                 pbxPhoto.Image.Width,                 pbxPhoto.Image.Height);         // statusAlbumPos is set in ch. 6     }</pre>
10	If no image is assigned to the picture box control, set the status labels to blank. <b>How-to</b> Set each <code>Text</code> property to <code>null</code> .	<pre>    else     {         statusInfo.Text = null;         statusImageSize.Text = null;         statusAlbumPos.Text = null;     }</pre>
11	Update the status bar at the end of the <code>menuFileLoad_Click</code> event handler.	<pre>private void menuFileLoad_Click(     object sender, EventArgs e) {     . . .     if (dlg.ShowDialog() == DialogResult.OK)     {         . . .         SetStatusStrip(dlg.FileName);     }     dlg.Dispose(); }</pre>

ASSIGN TEXT TO STATUS LABELS (CONTINUED)		
	Action	Result
12	Also call <code>SetStatusStrip</code> in the form's constructor to initialize the status bar settings.	<pre> public MainForm() {     InitializeComponent();     SetTitleBar();     <b>SetStatusStrip(null);</b>      menuView.DropDown = ctxMenuPhoto; } </pre>

Compile and run the program to see the status strip update. Make sure you test loading a valid and an invalid file to see that the `SetStatusStrip` method handles both cases.

**TRY IT!** There are a number of changes you could make here to experiment with the various properties and settings supported by the `StatusStrip` and `ToolStripStatusLabel` classes. Here are a couple of suggestions:

- There is a slight problem with our first label, since if the file path is too long it cannot be read by the user. Assign the `ToolTipText` property for the `statusInfo` label so the user can see the full path in a tooltip. You will also need to modify the `ShowItemToolTips` property for the status strip, since this is set to `false` by default.
- For a more challenging task, create a new item in the status strip that modifies the `BorderStyle` property of the `statusImageSize` label. To do this, add a drop-down button to the strip by selecting the `DropDownButton` item from the drop-down menu, as shown in the graphic. Set the (Name) to “`statusBorder`” and Text to “Size Border.” Look up the `Border3DStyle` enumeration in the .NET documentation to find the ten possible values for this enumeration.



Use these values to replicate the logic from our Image menu handlers earlier in the chapter. The new item is a `ToolStripDropDownButton` object, and you can add drop-down items for each enumeration value to the button, set the `Tag` property on each, and handle the `DropDownItemClicked` event to modify the `statusImageSize.BorderStyle` property. Run the application to dynamically change the border style so you can view the effect of each option.

## 4.4 **RECAP**

This chapter discussed context menu strips and status strips. We began with the `ContextMenuStrip` class and its similarities to the `MenuStrip` control. An example in our `MyPhotos` application created an `Image` menu for altering how an image is displayed in our `PictureBox` control. A new `View` menu in our `MenuStrip` class cleverly linked to our context menu so both menus would exhibit the same contents and behavior. This logic for the `Image` menu required the use of the `Enum` structure to convert `PictureBoxSizeMode` enumeration values to and from `string` objects.

We concluded the chapter with a discussion of status strips. We showed how `StatusStrip` controls contain `ToolStripStatusLabel` items, and created a status strip with some labels within our application.

Some new `C#` keywords were examined along the way, namely the `foreach` and `as` keywords, and some common object methods such as `Equals` and `ToString` were employed. We looked at the `Properties` window in `Visual Studio` in more detail, and used this window to add various events to our program.

This chapter completes our early discussion of tool strips and tool strip items, done here in the context of menus and status strips. Future chapters rely on this knowledge to make additional menu or status strip changes as we progress through the book. We also discuss the `ToolStrip` and `ToolStripItem` classes in greater detail in chapter 16.

The next chapter takes us out of the `Windows Forms` namespace briefly in order to examine reusable libraries. This lays the foundation for our discussions on various `Windows Forms` controls in subsequent chapters.