Chapter 20

Introducing Windows Forms

After completing this chapter, you will be able to:

- Create Windows Forms applications.
- Use common Windows Forms controls such as labels, text boxes, and buttons.
- Change the properties of Windows Forms and controls at design time and programmatically at run time.
- Subscribe to and process events exposed by Windows Forms and controls.

Now that you have completed the exercises and examined the examples in the first three parts of this book, you should be well-versed in the Microsoft Visual C# language. You have learned how to write programs and create components by using C#, and you should understand many of the finer points of the language, such as the differences between value types and reference types. Because you now have the essential language skills, Part IV will show you how to expand upon them and use C# to take advantage of the graphical user interface (GUI) libraries provided as part of the Microsoft .NET Framework. In particular, you will see how to use the objects in the System.Windows.Forms namespace to create Windows Forms applications.

In this chapter, you will learn how to build a basic Windows Forms application using the common components that are a feature of most GUI applications. You will see how to set the properties of Windows Forms and components by using the Visual Designer and the Properties windows. You’ll also learn how to change or examine the values of these properties dynamically by using C# code. Finally, you will learn how to intercept and handle some of the common events that Windows Forms and components expose.

Creating Your Application

As an example, you are going to create an application that allows a user to input and display details for members of the Middleshire Bell Ringers Association, an esteemed collection of the finest campanologists. Initially you will keep the application very simple, concentrating on laying out the form and making sure that it all works. In later chapters, you will provide menus and learn how to implement validation to ensure that the data that is entered makes sense. The following graphic shows what the application will look like after you have completed it. (You can see the completed version by running BellRingers.exe, located in the \Microsoft Press\Visual CSharp Step by Step\Chapter 20\BellRingers Complete\BellRingers\bin\Debug folder in your My Documents folder.)
Creating a Windows Forms Application

In this exercise, you’ll start building the Middleshire Bell Ringers Association application by creating a new project, laying out the form, and adding Windows Forms controls to the form. Because you have been using existing Windows Forms applications in Microsoft Visual Studio 2005 in previous chapters, much of the first couple of exercises will be a review for you.

Create the Middleshire Bell Ringers Association project

2. On the File menu, point to New, and then click Project.
3. In the Project Types pane, select Visual C#.
4. In the Templates pane, select Windows Application.
5. In the Name text box, type BellRingers.
6. In the Location list box, navigate to the Microsoft Press\Visual CSharp Step by Step\Chapter 20 folder in your My Documents folder.
7. Click OK.

The new project is created and contains a blank form called Form1.
Set the properties of the form

1. Select the form in the Designer View window. In the Properties window, click the (Name) property, and then type **MemberForm** in the (Name) text box to change the name of the form. (If the Properties window is not displayed, click Properties Window on the View menu, or press F4.)

2. In the Properties window, click the **Text** property, and then type **Middleshire Bell Ringers Association – Members**, to change the text in the title bar of the form.

3. In the Properties window, click the **BackgroundImage** property, and then click the Ellipses button in the adjacent text box.

   The Select Resource dialog box opens.

4. In the Select Resource dialog box, click Local resource and then click Import.

   The Open dialog box opens.

5. In the Open dialog box, navigate to the \Microsoft Press\Visual CSharp Step by Step\Chapter 20 folder in your My Documents folder, select the Bell.gif file, and then click Open.

   Part of the image will be displayed in the Select Resource dialog box.

6. In the Select Resource dialog box, click OK.

   The **BackgroundImage** property is now set to the bell image.

7. In the Properties window, click the **BackColor** property, and then click the down-arrow button in the adjacent text box.

   A dialog box opens.
8. On the System tab of the dialog box, click Window. This value sets the background color of all the controls that you drop onto the form to the same color as the window itself.

9. Select the Font property. This is a composite property that has many attributes. In the Properties window, click the plus sign (+) to expand the Font property and display the attributes. Type 12 for the Size attribute of the font, and set the Bold attribute to True.

Tip: You can also change some composite properties, such as Font, by clicking the ellipses button that appears when you select the property. When you click the ellipses button in the Font property, the standard Font dialog box opens and allows you to select the font and effects that you want.

10. Change the form’s Size property, which is also a composite property. In the Properties window, click the plus sign (+) to expand the Size property and display the attributes. Set the Width attribute to 600 and the Height attribute to 470.

The form should look like the image in the following graphic.
11. On the Build menu, click Build Solution.
   The form should build successfully.

   The application will start running and will display the main form containing the image.
   The form does not do anything useful yet, so close it and return to Visual Studio.

---

### How a Windows Forms Application Runs

A Windows Forms application can comprise any number of forms—you can add additional forms to an application by using the Add Windows Form command on the Project menu in Visual Studio 2005. How does an application know which form to display when an application starts?

If you look in the Solution Explorer, you will see another file called Program.cs. You can right-click this file and select View Code to display its contents in the Code And Text Editor window. This file contains the `Main` method, defining the entry point for the application. The key statement in this method is:

```
Application.Run(new MemberForm());
```

This statement creates a new instance of `MemberForm` and displays it. When the form closes, the `Application.Run` statement terminates, and as this is the final statement in `Main`, the program exits.

**Tip** If you have previously developed applications using Visual Basic 6, you will know that in that application, you can designate a form as the default form; this form is displayed automatically when the application starts. There is no such option in Visual Studio 2005. If you want to change the form that a .NET Framework application runs when an application starts, edit the `Application.Run` statement in the `Main` method.

You should only use the `Application.Run` statement for displaying the initial form for an application. If you have defined additional forms, you can display them from your own code, typically in an event handler, using the `Show` method inherited by all Windows Forms objects. For example, if you have added another form called `AnotherForm` to your application, you can display it like this:

```
AnotherForm aForm = new AnotherForm();
aForm.Show();
```
What Are the Common Windows Forms Properties?

If you look closely at the Properties window when a form is selected, you can see that there are over fifty properties available. Some of them are fairly obvious; for example, the Text property that corresponds to the text displayed in the form’s title bar. Some properties are useful under certain circumstances; for example, you can remove the Minimize, Maximize, and Close buttons, or remove the System menu from the title bar of a form by setting the `ControlBox` property to `False`—useful if you want to ensure users cannot close the form unless they execute some code that closes it explicitly. Other properties apply to very specific circumstances; for example, the `Opacity` property can be used to control the level of transparency of the form.

The following table describes some of the common form properties that you can change at design time. You should also be aware that there are additional properties not listed in the Properties window that you can use only programmatically at run time. For example, the `ActiveControl` property shows which control in the form currently has the focus.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Name)</td>
<td>The name of the form. Two forms in the same project cannot have the same name.</td>
</tr>
<tr>
<td>BackColor</td>
<td>The default background color of any text and graphics in the form.</td>
</tr>
<tr>
<td>BackgroundImage</td>
<td>A bitmap, icon, or other graphic file to be used as a backdrop to the form. If the image is smaller than the form, it can be tiled to fill the form, stretched, centered, or zoomed by using the <code>BackgroundImageLayout</code> property.</td>
</tr>
<tr>
<td>Font</td>
<td>The default font used by the controls embedded on the form that display text. This is a compound property—you can set many attributes of the font including the font name, size, and whether the font appears italic, bold, or underlined.</td>
</tr>
<tr>
<td>ForeColor</td>
<td>The default foreground color of any text and graphics in the form.</td>
</tr>
<tr>
<td>FormBorderStyle</td>
<td>This controls the appearance and type of border of the form. The default setting is Sizable. Other options specify borders that are not resizable or do not have the various System menu buttons.</td>
</tr>
<tr>
<td>Icon</td>
<td>This specifies the icon that appears in the form’s System menu and on the Microsoft Windows taskbar. You can create your own icons by using Visual Studio 2005.</td>
</tr>
<tr>
<td>Location</td>
<td>This is another compound property that specifies the coordinates of the top left corner of the form with respect to its container, which might be another form or the screen.</td>
</tr>
<tr>
<td>MaximizeBox</td>
<td>This property specifies whether the Maximize command on the System menu and caption bar is enabled or disabled. By default, it is enabled.</td>
</tr>
<tr>
<td>MaximumSize</td>
<td>This specifies the maximum size of the form. The default value (0, 0) indicates that there is no maximum size and the user can resize the form to any size.</td>
</tr>
<tr>
<td>MinimizeBox</td>
<td>This property is similar to the <code>MaximizeBox</code> property. It specifies whether the Minimize command on the System menu and title bar is enabled or disabled. By default, it is enabled.</td>
</tr>
<tr>
<td>MinimumSize</td>
<td>This property specifies the minimum size of the form.</td>
</tr>
</tbody>
</table>
Tip You can view a summary of a property by selecting the property in the Properties window, right-clicking it, and then clicking Description. A pane displaying a description of any selected property appears at the bottom of the Properties window. Clicking Description again hides the description of the property.

Changing Properties Programmatically

In addition to setting properties statically at design time, you can write code that changes properties dynamically at run time. For example, you can change the Size property of a form in your program to make it shrink or grow without the user dragging the border to resize it. In actual fact, if you look at the code behind the form, you will see that Visual Studio 2005 generates code to change the properties of a form at run time according to the values you specify at design time. If you click the + sign adjacent to Form1.cs in the Solution Explorer you will see the file Form1.Designer.cs (you will also see Form1.resx which contains information about resources, such as bitmaps, used by your application). Right-click the file Form1.Designer.cs and click View Code to display the generated code. You already saw this code in Chapter 1, “Welcome to C#,” but now you can start to appreciate what it actually does.

In this code, you will notice that the form is simply a class that contains a private System.ComponentModel.IContainer variable, called components, and a Dispose method. IContainer is an interface that includes a collection for holding references to the components belonging to the form. The Dispose method implements the disposal pattern described in Chapter 13, “Using Garbage Collection and Resource Management,” to quickly release any unmanaged resources used by the form when it is closed.

Expanding the Windows Forms Designer generated code region reveals another method called InitializeComponent. If you expand this method, you can see how the property values you specified in the Properties window are translated into code. Later, when you add additional controls to the form, code will be inserted into this method to create them and set their properties as well. If you change the values in the Properties window, Visual Studio 2005 will keep the code in this method synchronized with your changes.

Important You should not modify the code in the InitializeComponent method, or anywhere else in the Windows Forms Designer-generated code region. If you do, the changes you make will likely be lost the next time any property values are amended in Design View.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>This is the default size of the form when it is first displayed.</td>
</tr>
<tr>
<td>Text</td>
<td>This property contains the text that appears on the title bar of the form.</td>
</tr>
<tr>
<td>WindowState</td>
<td>This property determines the initial state of the form when it is first displayed. The default state (Normal) positions the form according to the Location and Size properties. The other options are Minimized and Maximized.</td>
</tr>
</tbody>
</table>
You should notice that the code in Form1.Designer.cs is actually a partial class, used to separate the statements and methods generated by Visual Studio from your own code. In the Solution Explorer, right-click Form1.cs and then click View Code to display the file that you add your own methods and fields to. You will see that this file already contains a default constructor that simply calls the `InitializeComponent` method to create and layout the form at runtime.

**Adding Controls to the Form**

So far you have created a form, set some of its properties, and examined the code that Visual Studio 2005 generates. To make the form useful, you need to add controls and write some code of your own. The Windows Forms library contains a varied collection of controls. The purposes of some are fairly obvious—for example, `TextBox`, `ListBox`, `CheckBox`, and `ComboBox`—whereas other, more powerful controls (such as the `DateTimePicker`) might not be so familiar.

**Using Windows Forms Controls**

In the next exercise, you will add controls to the form that allow a user to input member details. You will use a variety of different controls, each suited to a particular type of data entry.

You will use `TextBox` controls for entering the first name and last name of the member. Each member belongs to a “tower” (where bells hang). The Middleshire district has several towers, but the list is static—new towers are not built very often and hopefully, old towers do not fall down with any great frequency. The ideal control for handling this type of data is a `ComboBox`. The form also records whether the member is the tower “captain” (the person in charge of the tower who conducts the other ringers). A `CheckBox` is the best sort of control for this; it can either be selected (`True`) or cleared (`False`).

---

**Tip**  
`CheckBox` controls can actually have three states if the `ThreeState` property is set to `True`. The states are `True`, `False`, and `Indeterminate`. These states are useful if you are displaying information that has been retrieved from a relational database. Some columns in a table in a database allow `null` values, indicating that the value held is not defined or is unknown. If you want to display this data in a `CheckBox`, you can use the `Indeterminate` state to handle `null` values.

The application also gathers statistical information about when members joined the association and how much bell ringing experience they have (up to one year, between one and four years, between five and nine years, and ten or more years). A `DateTimePicker` control is very suitable for selecting and displaying dates, and a group of options, or radio buttons, is useful for indicating the member’s experience—radio buttons provide a mutually exclusive set of values.

Finally, the application records the tunes the member can ring—rather confusingly, these tunes are referred to as “methods” by the bell-ringing fraternity. Although a bell ringer only rings one bell at a time, a group of bell ringers under the direction of the tower captain can ring their bells in different sequences and generally play simple music. There are a variety of
bell ringing methods, and they have names like Canterbury Minimus, Plain Bob Doubles, and Old Oxford Delight Minor. New methods are being written with alarming regularity, so the list of methods can vary over time. In a real-world application, you would store this list in a database. In this application, you will use a small selection of methods that you will hard-wire into the form. (You will see how to use databases in the next part of the book.) A good control for displaying this information and indicating whether a member can ring a method is the CheckedListBox.

When the user has entered the member’s details, the Add button will validate and store the data. The user can click Clear to reset the controls on the form and cancel any data entered.

Add Windows Forms controls

1. Ensure that Form1 is displayed in the Designer View window. Using the Toolbox, verify that the Common Controls category is expanded, and then drag a Label control onto MemberForm. (If the Toolbox is not displayed, click Toolbox from the View menu, or click the Toolbox tab in the left-hand border of Visual Studio.)

2. In the Properties window, click the Location property, and then type 10,40 to set the Location property of the label.

3. From the Toolbox, drag a TextBox control onto MemberForm, to the right of the label. Do not worry about aligning the TextBox exactly because you will set the Location property for this and the following controls later.

4. Add a second Label to the form. Place it to the right of the TextBox.

5. Add another TextBox to MemberForm and position it to the right of the second Label.

6. From the Toolbox, drag a third Label onto the form. Place it directly under the first Label.

7. From the Toolbox, drag a ComboBox control onto the form. Place it on MemberForm under the first TextBox and to the right of the third Label.

8. From the Toolbox, drag a CheckBox control onto the form and place it under the second TextBox.

9. Add a fourth Label to MemberForm and place it under the third Label.

10. From the Toolbox, drag a DateTimePicker control and place it under the ComboBox.

11. In the Toolbox, expand the Containers category. Drag a GroupBox control from the Toolbox and place it under the fourth Label control.

12. From the Common Controls category in the Toolbox, drag the RadioButton control and place it inside the GroupBox control you just added.
13. Add three more RadioButton controls, vertically aligned with each other, to the GroupBox. You might need to make the GroupBox bigger to accommodate them.

14. From the Toolbox, drag a CheckedListBox control and place it under the second Label and to the right of the GroupBox control.

15. From the Toolbox, drag a Button control and place it near the bottom on the lower-left side of MemberForm.

16. Add another Button control to the bottom of the form, just to the right of the first.

Setting Control Properties

You now need to set the properties of the controls you just added to the form. To change the value of a control’s property, click the control on the form to select it, and then enter the correct value in the Properties window. You will start with the basic properties. The following table lists the properties and values you need to assign to each of the controls.

<table>
<thead>
<tr>
<th>Control</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>label1</td>
<td>Text</td>
<td>First Name</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>10, 40</td>
</tr>
<tr>
<td>textBox1</td>
<td>(Name)</td>
<td>firstName</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>120, 40</td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>170, 26</td>
</tr>
<tr>
<td>label2</td>
<td>Text</td>
<td>Last Name</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>300, 40</td>
</tr>
<tr>
<td>textBox2</td>
<td>(Name)</td>
<td>lastName</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>400, 40</td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>170, 26</td>
</tr>
<tr>
<td>label3</td>
<td>Text</td>
<td>Tower</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>10, 92</td>
</tr>
<tr>
<td>comboBox1</td>
<td>(Name)</td>
<td>towerNames</td>
</tr>
<tr>
<td></td>
<td>DropDownList</td>
<td>DropDownList (This setting forces users to pick one of the items in the list; users cannot type in a value of their own.)</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>120, 92</td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>260, 28</td>
</tr>
<tr>
<td>checkBox1</td>
<td>(Name)</td>
<td>isCaptain</td>
</tr>
</tbody>
</table>
This property specifies the location of the checkbox relative to the text in the control. When you click the drop-down arrow for this property, an interesting graphic containing a grid appears. Click the left square in the middle row.)

<table>
<thead>
<tr>
<th>Control</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>label4</td>
<td>Text</td>
<td>Member</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>10, 148</td>
</tr>
<tr>
<td>DateTimePicker</td>
<td>(Name)</td>
<td>memberSince</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>120, 148</td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>290, 26</td>
</tr>
<tr>
<td>groupBox1</td>
<td>(Name)</td>
<td>yearsExperience</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>10, 204</td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>260, 160</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>Experience</td>
</tr>
<tr>
<td>radioButton1</td>
<td>(Name)</td>
<td>novice</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>16, 32</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>Up to 1 year</td>
</tr>
<tr>
<td>radioButton2</td>
<td>(Name)</td>
<td>intermediate</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>16, 64</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>1 to 4 years</td>
</tr>
<tr>
<td>radioButton3</td>
<td>(Name)</td>
<td>experienced</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>16, 96</td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>5 to 9 years</td>
</tr>
<tr>
<td>radioButton4</td>
<td>(Name)</td>
<td>accomplished</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>16, 128</td>
</tr>
</tbody>
</table>
Part IV: Working with Windows Applications

It is a good idea to save your project at this point.

<table>
<thead>
<tr>
<th>Control</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td></td>
<td>10 or more years</td>
</tr>
<tr>
<td>checkedListBox1</td>
<td>(Name)</td>
<td>methods</td>
</tr>
<tr>
<td>Location</td>
<td></td>
<td>300, 212</td>
</tr>
<tr>
<td>Size</td>
<td></td>
<td>270, 165</td>
</tr>
<tr>
<td>Sorted</td>
<td></td>
<td>True</td>
</tr>
<tr>
<td>button1</td>
<td>(Name)</td>
<td>Add</td>
</tr>
<tr>
<td>Location</td>
<td></td>
<td>190, 388</td>
</tr>
<tr>
<td>Size</td>
<td></td>
<td>75, 40</td>
</tr>
<tr>
<td>Text</td>
<td></td>
<td>Add</td>
</tr>
<tr>
<td>button2</td>
<td>(Name)</td>
<td>Clear</td>
</tr>
<tr>
<td>Location</td>
<td></td>
<td>335, 388</td>
</tr>
<tr>
<td>Size</td>
<td></td>
<td>75, 40</td>
</tr>
<tr>
<td>Text</td>
<td></td>
<td>Clear</td>
</tr>
</tbody>
</table>

Control Properties

As you have just seen, like forms, controls have many properties that you can set. Each different type of control has different properties. Also, like forms, you can set and query control properties dynamically in your own programs, and there are a number of properties that are available only at run time. If you want to learn more about the different properties available for each type of control, you can find a list of them in the MSDN Library for Visual Studio 2005 supplied with Visual Studio 2005.

Changing Properties Dynamically

You have been using the Design View to set properties statically. When the form runs, it would be useful to reset the value of each control to an initial default value. To do this, you will need to write some code (at last). In the following exercises, you will create a private method called Reset. Later, you will invoke the Reset method when the form first starts, and when the user clicks the Clear button.
Rather than coding the method from scratch, you will use the Class Diagram editor to generate the method. The Class Diagram editor provides a schematic way to view and amend classes.

Create the Reset method

1. In the Solution Explorer, right-click Form1.cs. A menu appears.
2. On the menu, click View Class Diagram.
   
   A new class diagram appears displaying the MemberForm class.
3. Right-click the MemberForm class in the diagram, point to Add, and then click Method. The MemberForm class expands to display a list of all defined methods (Dispose, InitializeComponent, and the MemberForm constructor). A new method, simply called Method, is added. Change the name of this method to Reset by overtyping the name and pressing the Enter key.
4. In the Class Details pane that appears underneath the class diagram, verify that the Type of the Reset method is void, and that the Modifier is public. If they are wrong, you can click these fields in the Class Details pane and modify them.

The following graphic shows the class diagram with the new method added:
5. In the Class Details pane that appears underneath In the Class Diagram, right-click the Reset method and then click View Code.

You are placed in the Code and Text Editor window displaying the MemberForm class. The Reset method has been added with a default implementation that throws a NotImplementedException:

```csharp
public void Reset()
{
    throw new System.NotImplementedException();
}
```

6. In the Code And Text Editor window, replace the throw statement in the Reset method with the following lines of code:

```csharp
firstName.Text = "";
l.lastName.Text = "";
```

These two statements ensure that the firstName and lastName text boxes are blank by assigning an empty string to their Text property.

**Programming the User Interface**

You now need to configure the properties of the remaining controls on the form. You will do this programmatically.

**Populating the ComboBox** If you recall, the towerName ComboBox will contain a list of all the bell towers in the Middleshire district. This information would usually be held in a database and you would write code to retrieve the list of towers and populate the ComboBox. Because you have not been shown how to access a database yet, the application will use a hard-coded collection.

A ComboBox has a property called Items that contains a list of the data to be displayed. In the Reset method, after the code you have already written, add the following statements to clear this list (this is important because otherwise you would end up with many duplicate values in the list) and create four items in the ComboBox:

```csharp
towerNames.Items.Clear();
towerNames.Items.Add("Great Shevington");
towerNames.Items.Add("Little Mudford");
towerNames.Items.Add("Upper Gumtree");
towerNames.Items.Add("Downley Hatch");
```

**Set the current date** The next step is to initialize the memberSince DateTimePicker control to the current date. The date can be set by using the Value property. You can obtain the current date by using the static property Today of the DateTime class. Add the following statement to the Reset method:

```csharp
memberSince.Value = DateTime.Today;
```
Chapter 20: Introducing Windows Forms

Initialize the CheckBox  The isCaptain CheckBox should default to False. To do this, you need to set the Checked property. Add the following statement to the Reset method:

    isCaptain.checked = false;

Initialize the radio button group  The form contains four radio buttons that indicate the number of years of bell ringing experience the member has. A radio button is similar to a CheckBox in that it can contain a True or False value. However, the power of radio buttons increases when you put them together in a GroupBox. In this case, the radio buttons form a mutually exclusive collection—at most, only one radio button in a group can be selected (set to true), and all the others will automatically be cleared (set to false). By default, none of the buttons will be selected. You should rectify this by setting the Checked property of the novice radio button. Add the following statement to the Reset method:

    novice.checked = true;

Fill the ListBox  Like the Tower ComboBox, the CheckedListBox containing the list of bell ringing methods has a property called Items that contains a collection of values to be displayed. Also, like the ComboBox, it could be populated from a database. However, as before, you will supply some hard-coded values for now. Complete the Reset method by adding the following code:

    methods.Items.Clear();
    methods.Items.Add("Canterbury Minimus");
    methods.Items.Add("Reverse St Nicholas");
    methods.Items.Add("Plain Bob Doubles");
    methods.Items.Add("Grandsire Doubles");
    methods.Items.Add("Cambridge Minor");
    methods.Items.Add("Old Oxford Delight Minor");
    methods.Items.Add("Kent Treble Bob Major");

Call the Reset method  You need to arrange for the Reset method to be called when the form is first displayed. A good place to do this is in the MemberForm constructor. In the Code And Text Editor window, scroll to the beginning of the MemberForm class in the file Form1.cs, and find the constructor (it is called MemberForm, just like the class). Insert a call to the Reset method after the statement that calls the InitializeComponent method:

    this.Reset();

Compile and test the application

1.  It is a good practice to name the file containing a form after the form itself. In the Solution Explorer, right-click Form1.cs, click Rename, and then type MemberForm.cs.

2.  On the Debug menu, click Start Without Debugging to verify that the project compiles and runs.

3.  When the form runs, click the Tower ComboBox.
    You will see the list of bell towers, and you can select one of them.
4. Click the drop-down arrow on the right side of the Member Since date/time picker. You will be presented with a calendar of dates. The default value will be the current date. You can click a date, and use the arrows to select a month. You can also click the month name to display the months as a drop-down list, and click the year to allow you to select a year using a numeric up-down control.

5. Click each of the radio buttons in the Experience group. Notice that you cannot select more than one at a time.

6. In the Methods list box, click some of the methods and select the corresponding check box. You will have to click once to select a method and a second time to select or clear the checkbox.


Publishing Events in Windows Forms

If you are familiar with Microsoft Visual Basic, Microsoft Foundation Classes (MFC), or any of the other tools available for building GUI applications for Windows, you are aware that Windows uses an event-driven model to determine when to execute code. In Chapter 16, “Delegates and Events,” you saw how to publish your own events and subscribe to them. Windows Forms and controls have their own predefined events that you can subscribe to, and these events should be sufficient to handle most eventualities.

Processing Events in Windows Forms

Your task as a developer is to capture the events that you feel are relevant to your application and write the code that responds to these events. A familiar example is the Button control, which raises a “Somebody clicked the button” event when a user clicks it with the mouse or presses Enter when the button has the focus. If you want the button to do something, you write code that responds to this event. This is what you will do in the final exercise in this chapter.

Handle the Click event for the Clear button

1. In Design View (on the View menu, click Designer), select the Clear button on Member-Form.

When the user clicks the Clear button, you want the form to be reset to its default values.

2. In the Properties window, click the Events button.
Chapter 20: Introducing Windows Forms

365

The list of properties is replaced with a list of events that you can intercept.

3. Select the **Click** event.

4. Type **clearClick** in the text box and press Enter.

   A new event method called **clearClick** is created and displayed in the Code And Text Editor window. Notice that the event method conforms to the convention in that it takes two parameters: the sender (**object**) and additional arguments (**EventArgs**). The Windows Forms runtime will populate these parameters with information about the source of the event and any additional information that might be useful when handling the event. You will not use these parameters in this exercise.

5. In the body of the **clearClick** method, call the **Reset** method.

   The body of the method now should look exactly like this:

   ```
   private void clearClick(object sender, System.EventArgs e)
   {
       this.Reset();
   }
   ```

**Handle the Click event for the Add button**

The users will click the Add button when they have filled in all the data for a member and want to store the information. The **Click** event should validate the information entered to ensure it makes sense (for example, should you allow a tower captain to have less than one year of experience?) and, if it is okay, arrange for the data to be sent to a database or other persistent store. You will learn more about validation and storing data in later chapters. For now, the code for the **Click** event of the Add button will display a message box echoing the data input.

1. Return to Design View and select the Add button.
2. In the Properties window, ensure that you are displaying events rather than properties, type `addClick` in the `Click` event, and then press Enter.

Another event method called `addClick` is created.

3. Add the following code to the `addClick` method:

```csharp
string details;
details = "Member name " + firstName.Text + " 
   + lastName.Text + " from the tower at " + towerNames.Text;
MessageBox.Show(details, "Member Information");
```

This block of code creates a string variable called `details` that it fills with the name of the member and the tower that the member belongs to. Notice how the code accesses the `Text` property of the `TextBox` and `ComboBox` to read the current values of those controls. The `MessageBox` class provides static methods for displaying dialog boxes on the screen. The `Show` method used here will display the contents of the `details` string in the body of the message box and will put the text “Member Information” in the title bar. `Show` is an overloaded method, and there are other variants that allow you to specify icons and buttons to display in the message box.

**Handle the Closing event for the form**

As an example of an event that can take a different set of parameters, you will also trap the `FormClosing` event for a form. The `FormClosing` event is raised when the user attempts to close the form but before the form actually closes. You can use this event to prompt the user to save any unsaved data or even ask the user whether she really wants to close the form—you can cancel the event in the event handler and prevent the form from closing.

1. Return to Design View and select the form (click anywhere on the background of the form rather than selecting a control).

2. In the Properties window, ensure that you are displaying events, type `memberFormClosing` in the `FormClosing` event, and then press Enter.

   An event method called `memberFormClosing` is created.

   You should observe that the second parameter for this method has the type `FormClosingEventArgs`. The `FormClosingEventArgs` class has a Boolean property called `Cancel`. If you set `Cancel` to `true` in the event handler, the form will not close. If you set `Cancel` to `false` (the default value), the form will close when the event handler finishes.

3. Add the following statements to the `memberFormClosing` method:

```csharp
DialogResult key = MessageBox.Show(  
    "Are you sure you want to quit",  
    "Confirm",  
    MessageBoxButtons.YesNo,  
    MessageBoxIcon.Question);  
e.Cancel = (key == DialogResult.No);
```

These statements display a message box asking the user to confirm whether to quit the application. The message box will contain Yes and No buttons and a question mark.
When the user clicks either the Yes or No button, the message box will close and the button clicked will be returned as the value of the method (as a DialogResult— an enumeration identifying which button was clicked). If the user clicks No, the second statement will set the Cancel property of the CancelEventArgs parameter (e) to true, preventing the form from closing.

#### Delegates for Windows Forms Events

When you use the Properties window to define an event method (see Chapter 16), Visual Studio 2005 generates code that creates a delegate that references the method and subscribes to the event. If you look at the block of code that defines the Clear button in the InitializeComponent method in the MemberForm.Designer.cs file, you will see the following statement:

```csharp
// clear
...
this.clear.Click += new System.EventHandler(this.clearClick);
```

The statement creates an EventHandler delegate pointing to the clearClick method. It then adds the delegate to the Click event for the Clear button. As you create additional event methods, Visual Studio 2005 will generate the required delegates and subscribe to the events for you.

### Run the Application

1. On the Debug menu, click Start Without Debugging to run the application.
2. Type in a first name and a last name, and then select a tower from the list. Click Add. In the message box that appears displaying the member data you entered, click OK.
3. Try and close the form. In the message box that appears, click No.
   The form should continue running.
4. Try and close the form again. In the message box, click Yes.
   This time the form closes and the application finishes.

- **If you want to continue to the next chapter**

- **If you want to exit Visual Studio 2005 for now**
  On the File menu, click Exit. If you see a Save dialog box, click Yes.
## Chapter 20 Quick Reference

<table>
<thead>
<tr>
<th>To</th>
<th>Do this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a Windows Forms project.</td>
<td>Use the Windows Application template.</td>
</tr>
<tr>
<td>Change the properties of a form.</td>
<td>Click the form in Design View. In the Properties window, select the property you want to change and enter the new value.</td>
</tr>
<tr>
<td>View the code behind a form.</td>
<td>On the View menu, click Code.</td>
</tr>
<tr>
<td>Add controls to a form.</td>
<td>Drag the control from the Toolbox onto the form.</td>
</tr>
<tr>
<td>Change the properties of a control.</td>
<td>While in Design View, click the control. In the Properties window, select the property you want to change and enter the new value.</td>
</tr>
</tbody>
</table>
| Dynamically populate a ComboBox or a ListBox. | Use the Add method of the Items property. For example: 
```
towerNames.Items.Add("Upper Gumtree");
```
You might need to clear the Items property first, depending on whether you want to retain the existing contents of the list. For example: 
```
towerNames.Items.Clear();
```
| Initialize a CheckBox or radio button. | Set the Checked property to true or false. For example: 
```
novice.Checked = true;
```
| Handle an event for a control or form. | Select the control or form in Design View. In the Properties window, click the Events button. Find the event you want to handle and type the name of an event method. Write the code that handles the event in the event method. |