



Creating Forms and Using Inheritance



ESTIMATED
TIME
30 min.

In this additional lesson, you will learn how to:

- ✓ *Create Windows Forms applications that have common properties, methods, and events by using inheritance.*
- ✓ *Make global changes to multiple forms in an application by using subclasses.*
- ✓ *Execute a method periodically by using a Timer control.*

Creating Forms

In Chapter 23, you saw how to take advantage of inheritance when building your own customized version of a *ToggleButton* control. A major benefit of creating your own subclass is that if you need to make a global change to all of the applications that use the control, you actually need to make the change only once—in the class definition. After the control has been updated, all applications that use it can pick up the amended version of the control automatically through the dynamic assembly linking mechanism implemented by the .NET Framework. You can perform the same task with Windows forms, which is what you will do in this lesson. As a by-product of all of this, you will also see how to use a *Timer* control.

Forms as Templates

You can think of a form as a template. When you create a Microsoft Windows application, you are presented with a form that has a particular, predefined style. You then set properties, write methods, and customize the form to your own requirements. If you need to perform the same customization on different forms, you might end up repeating your code and actions. This is the same problem that can be avoided by using user controls or by subclassing. The good news is that you can use the same technique with forms that you used with controls—

you can define your own specialized form class by using inheritance. You can then use this new class as the basis for your forms. If you need to make a global change to all the forms in your application, you simply update the class—once. Applications that use forms based on this class can be configured to pick up the changes automatically.

Subclassing a Form

In the following exercises, you will subclass the *System.Windows.Forms.Form* class and add standard features that you want all of your applications to use. In this example, the features in question are the default font used by the form, the background image containing the company logo for the Honest John Heavy Industries Corporation, and the way in which the form appears and disappears as it opens and closes.

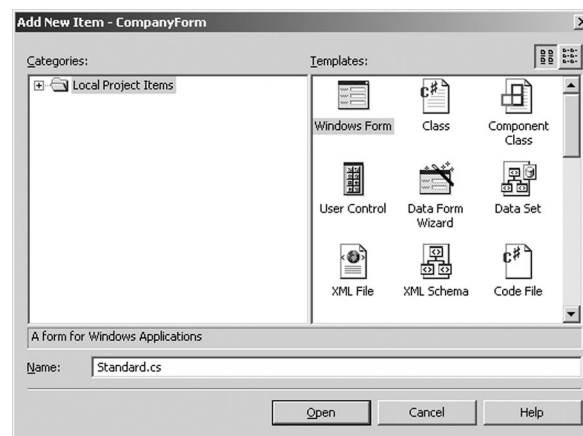
Automatic Updates Using Assemblies

One of the many joys of the .NET Framework is the assembly linkage feature that allows you to update a .NET component and have the updated version used automatically by programs that depend on it. To use this feature, you must place the component in the Global Assembly Cache (GAC)—use the `gacutil` command. When you create a reference to the component in your project, set the *Copy Local* property of the reference to *false* in the Properties window (by default it is *true*, which creates a local copy of the component DLL in your project's `bin\Debug` or `bin\Release` folder). The .NET run time will attempt to locate the component in the GAC at run time, and if successful, it will link to it. If the component is not there, you will get a run-time error instead. If you don't want to place a component in the GAC, you must rebuild all applications that reference the component if you modify the component.

There are many other issues to think about if you are planning to use the GAC. For example, if you place a component in the GAC and then update the component, you can get two versions of the same component stored in the GAC. This is primarily because the .NET Framework assumes that existing programs linked to the earlier version of the component might not work using the new version (remember the joys of DLL—when you updated a driver or component in the good old days, half of your applications would not work because they were no longer compatible). You can also replace all previous versions of a component in the GAC with a new version and configure your applications (or components) to use the latest version. You achieve this feat by using `.config` files. It is beyond the scope of this lesson to cover assembly linkage and application configuration in detail. Just be aware that you can do it, it works (most of the time), and life is much easier because of it.

Create the *Standard* form class

- 1 In Microsoft Visual Studio .NET, create a new project by using the Class Library template. Call the project CompanyForm, and create it in the \Additional Lessons\Creating Forms and Using Inheritance folder.
- 2 In the Solution Explorer, right-click Class1.cs, and then click Delete on the pop-up menu—you do not need this class. A dialog box appears stating that Class1.cs will be deleted permanently. Click OK.
- 3 On the Project menu, click Add Windows Form. When the Add New Item dialog box appears, verify that the Windows Form template is selected and type **Standard.cs** in the Name text box. Click Open to create the new form.



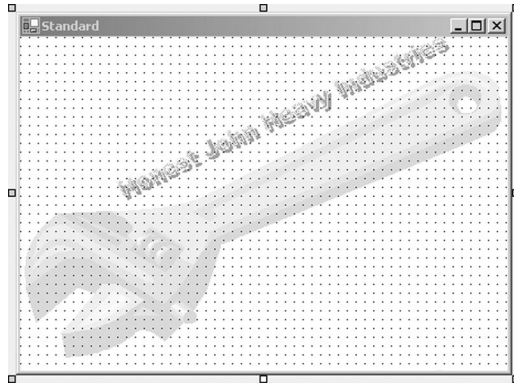
- 4 On the View menu, click Code to display the Code pane. Verify that the *Standard* class inherits from *System.Windows.Forms.Form*:

```
:  
public class Standard : System.Windows.Forms.Form  
{  
:  
}
```

Set the properties of the *Standard* form class

- 1 Switch back to Design View, and change the *Size* property of the *Standard* form to 450, 330.
- 2 Select the *BackgroundImage* property, and click the Ellipses button. In the Open dialog box, navigate to the \Additional Lessons\Creating Forms and Using Inheritance folder, select Background.bmp, and then click Open.

The company logo (Honest John Heavy Industries) appears as the background on the form.

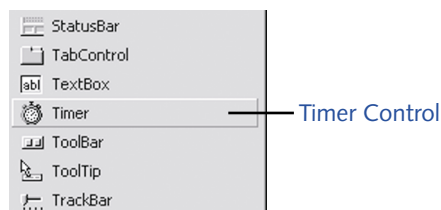


- 3 In the Properties window, expand the *Font* property. Set the font name to *Times New Roman* and the font size to 12.

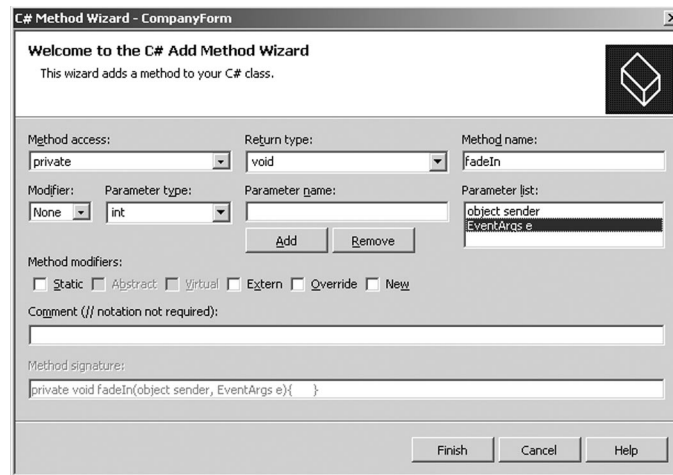
Extend the **Load** and **Closing** events

When the form opens and closes, it will fade in and out. You can achieve this effect by using a *Timer* control and the *Opacity* property of the form. You will extend the *Load* event to set the *Opacity* property to 0 and start a timer running. When the timer expires (the interval will be very short), a *Tick* event method will increase the *Opacity* by 0.1 and start the timer running. When the *Opacity* reaches 1, the *Timer* will halt. The opposite will occur when the form closes, using the *Closing* event.

- 1 In the Toolbox, drag a *Timer* control anywhere onto the *Standard* form. The *Timer* control appears under the form. In the Properties window, change the (Name) property of the *Timer* control to *formTimer* and change the *Interval* property to 50 (for 50 milliseconds).



- 2 In the Class View, navigate to the *Standard* class in the *CompanyForm* namespace. Right-click the *Standard* class. On the pop-up menu, point to Add, and then click Add Method. When the C# Method Wizard appears, set the method access to private, set the return type to void, and set the method name to *fadeIn*. Add two parameters; the first should be called *sender* and have its parameter type set to *object*, and the second parameter should be called *args* and be of type *EventArgs* (you have to type this in because it does not appear in the drop-down list). Click Finish.



- 1 In the body of the *fadeIn* method, add the following statements:

```
if (this.Opacity < 1.0)
{
    this.Opacity += 0.1;
    formTimer.Enabled = true;
}
else
{
    // More to be added in a moment
}
```

The *fadeIn* method is called each time the timer expires—its signature matches that of the *Tick* event method (you will use this in a moment)—when the form is loading. The *if* statement examines the current *Opacity* property value of the form. If it is less than 1, the *Opacity* property is incremented by 0.1 (the *Opacity* can range from 0 to 1, 0 meaning transparent and 1 meaning opaque). Setting the *Enabled* property of the timer starts it running again. You will fill in the *else* part in step 7.

- 4** Move to the top of the *Standard* class and add a *private EventHandler* variable called *loading* just before the *formTimer* variable by using the following statement:

```
private EventHandler loading;
```

- 5** Create a new *protected* method called *OnLoad* by adding the following statements:

```
protected override void OnLoad(EventArgs e)
{
}
```

This method is called whenever the *Load* event of the form is raised—you used this same procedure in Chapter 23 to override the *Click* event of the *ToggleButton* control. Recall that each event defined for a form or control has a corresponding *On* method that raises the event. Overriding the *On* method allows you to insert your own custom processing while permitting developers using your control or form to subscribe to the event and do their own processing as well.

- 6** In the *OnLoad* method, add the following statements:

```
base.OnLoad(e);
this.Opacity = 0;
loading = new EventHandler(this.fadeIn);
formTimer.Tick += loading;
formTimer.Enabled = true;
```

The first statement calls the *OnLoad* method of the *Form* class (you want to extend the existing behavior of the form). The second statement makes the form totally transparent. The next statement creates a delegate referring to the *fadeIn* method. This delegate is used to subscribe to the *Tick* event of the *formTimer* control—when the *Tick* event occurs, the *fadeIn* method is called. The final statement enables the timer and starts it running.

- 7** Return to the *fadeIn* method. In the *else* statement (that you left empty previously), insert the following statement:

```
formTimer.Tick -= loading;
```

This statement unsubscribes from the *Tick* event of the *formTimer* control—the form should now be fully visible.

- 8** Create another method called *fadeOut* that has the same signature as *fadeIn* by adding the following code:

```
private void fadeOut(object sender, EventArgs e)
{
}
```



In the body of the *fadeOut* method, type the following statements:

```
if (this.Opacity > 0)
{
    this.Opacity -= 0.1;
    formTimer.Enabled = true;
}
else
{
    // More to be added in a moment
}
```

This method is the opposite of *fadeIn*. It is also called when the *Tick* event of the timer occurs—this time when the form is closing. If the *Opacity* of the form is greater than 0, it is decremented by 0.1 and the timer is restarted.

- 9** Move to the top of the *Standard* class, and create another *private EventHandler* variable called *closing* after the *loading* variable by adding the following statement:

```
private EventHandler closing;
```

- 10** Create a *protected* method called *OnClosing* by adding the following code:

```
protected override void OnClosing(CancelEventArgs e)
{
}
```

This method is invoked whenever the form raises the *Closing* event.

Type the following statements in the body of this method:

```
base.OnClosing(e);
if (this.Opacity > 0)
{
    closing = new EventHandler(this.fadeOut);
    formTimer.Tick += closing;
    formTimer.Enabled = true;
    e.Cancel = true;
}
```

A little explanation is required here. Remember that the *Closing* event occurs when the form is being closed but before it actually closes. You can use the *Closing* event to determine whether you want the form to really close, and you can abort the close operation by setting the *Cancel* property of the *CancelEventArgs* parameter passed in to *true*. The thing to consider in this case is what happens when *e.Cancel* is not set to *true*. Under those circumstances, the form closes and disappears before the timer has had a chance to fade it out gracefully by using the *fadeOut* method. For this reason, the code shown above allows the form to close only after the *Opacity* has

dropped to 0. There is still an issue though—the *Closing* event will be called when the form is totally opaque (when *Opacity* is 1.0). So how does this work? The answer is that you need to call the *Closing* event again in the *fadeOut* method after the *Opacity* has dropped to 0.

- 11 Return to the *fadeOut* method. In the *else* statement, add the following code:

```
formTimer.Tick -= closing;
this.Close();
```

The first line of code unsubscribes from the *Tick* event because the form should now be totally transparent. The second line of code closes the form. It raises the *Closing* event. The *Opacity* of the form is 0, and the form then closes.

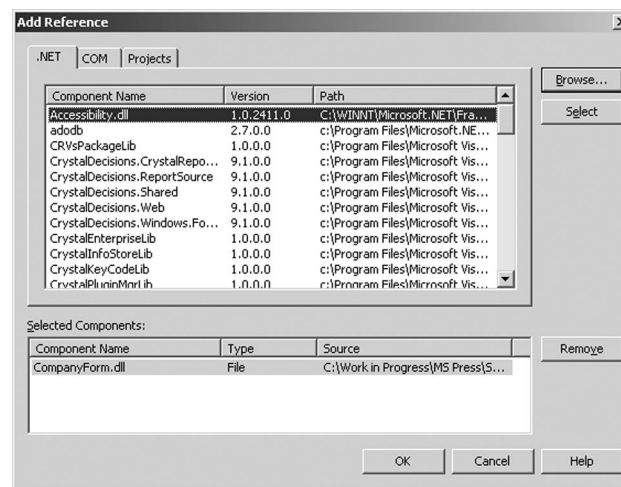
- 2 On the Build menu, click Build Solution to compile the class.

Using the *Standard Form Class*

In the following set of exercises, you will build a simple application to test the *Standard* form class.

Create the test application

- 1 Create a new Windows Application project called *StandardFormTest* in the \Additional Lessons\Creating Forms and Using Inheritance folder.
- 2 On the Project menu, click Add Reference. When the Add Reference dialog box opens, verify that the .NET tab is selected and click Browse. Navigate to the \Additional Lessons\Creating Forms and Using Inheritance\CompanyForm\bin\Debug folder, select *CompanyForm.dll*, and then click Open. In the Add Reference dialog box, verify that *CompanyForm.dll* appears as a selected component, and then click OK.



- 3 Display Form1.cs in the Code pane. Add the following *using* statement to the list at the top:

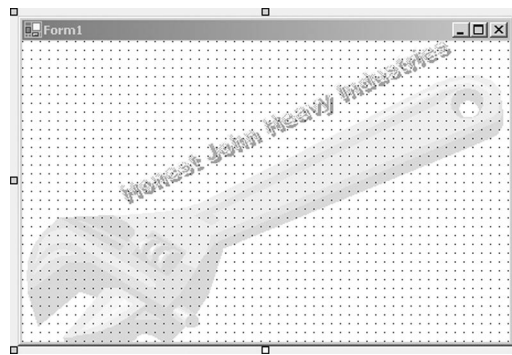
```
using CompanyForm;
```

This is the namespace holding the *Standard* form class.

- 4 Change the definition of the *Form1* class so that it inherits from *Standard* rather than *System.Windows.Forms.Form*:

```
public class Form1 : Standard
{
    :
}
```

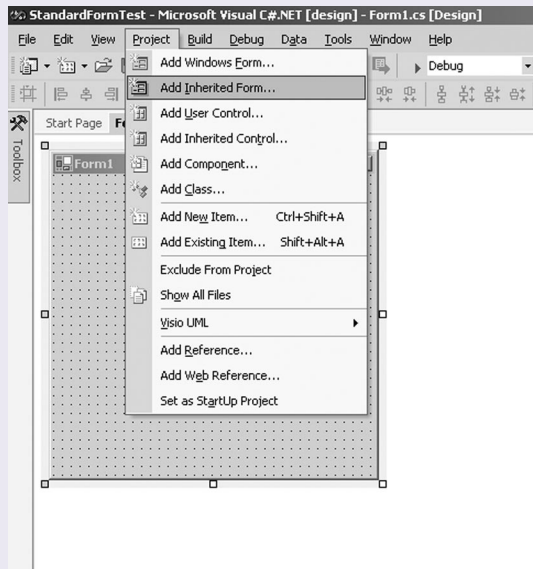
- 5 Switch back to Design View for Form1. You should now see the Honest John Heavy Industries logo appear as the background image of the form. Change the *Size* property of *Form1* to 450, 300 so that you can see the entire logo.



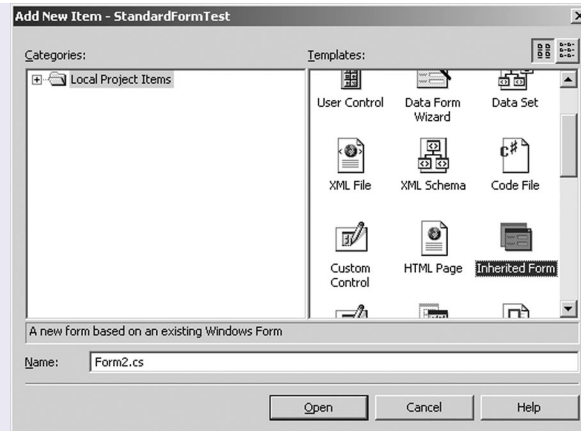
Forms Inheritance, Properties, and the Visual Studio IDE

You may be surprised to find that *Form1* is not automatically sized to be 450, 300 because this was the size you specified earlier in the *Standard* class. After all, the background image changed automatically, and if you look at the *Font* property, it is now set to Times New Roman, 12 point. You can override the inherited value of any property, but property values normally default to the inherited value. So why did the *Size* property of *Form1* not change automatically? Well, the Microsoft Visual Studio integrated development environment (IDE) itself actually overrides some properties of forms, with *Size* being one of them and *Text* being another (the text in the Title bar of the form is *Form1* and not *Standard*, which is the value defined in the *Standard* class).

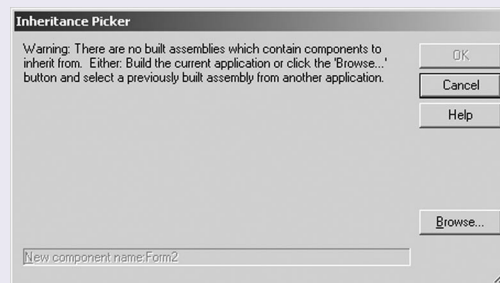
However, Visual Studio .NET does give you another way to create a form by using inheritance. If you want to create a new form (rather than use the existing one given to you when you create a project by using the Windows Application template), click Add Inherited Form on the Project menu.



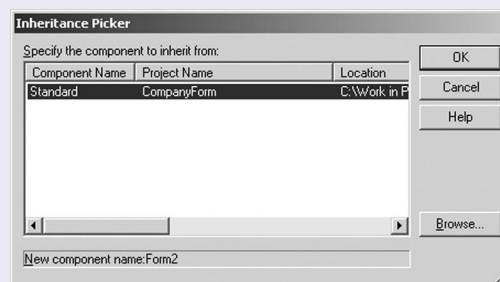
The Add New Item dialog box appears, and the Inherited Form icon is selected.



After typing the name of your new form and clicking Open, you may be prompted by the Inheritance Picker dialog box to specify the DLL containing the form from which you want to inherit.



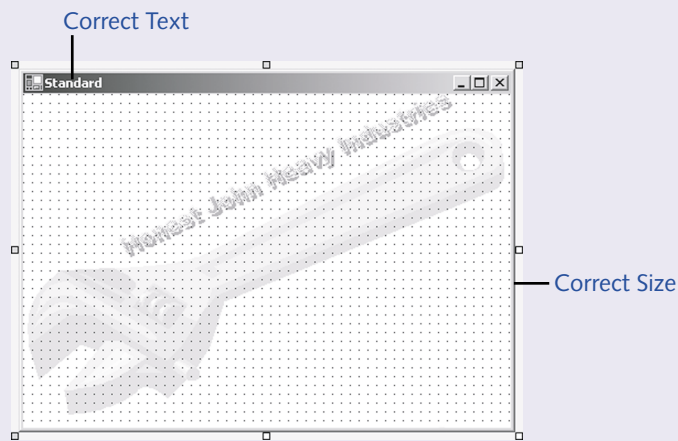
Click Browse, and locate the DLL you want to use. When you have selected the DLL and the form to inherit from (there may be more than one in the DLL), click OK in the Inheritance Picker dialog box.



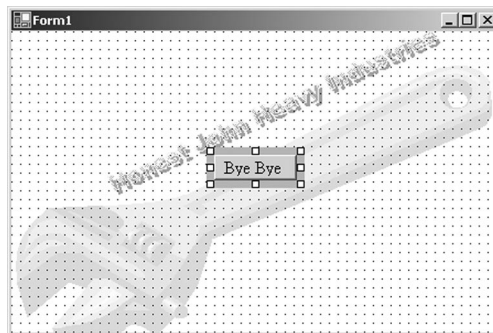
(continued)

(continued)

Your new form is created and uses the inherited values for the *Size* and *Text* properties.



- 6 In the Toolbox, drag a *Button* control onto the middle of the form. Notice that the text (button1) appears in the Times New Roman font. Change the *Text* property of *button1* to Bye Bye.



- 7 While *button1* is still selected, in the Properties window, click the Events button, and then select the *Click* event. Type **byeByeClick** and press Enter. The *byeByeClick* method appears in the Code pane. In the method, add the following statement:

```
this.Close();
```


This code closes the form (after raising the *Closing* event).



Execute the test application

- 1 Build and run the form. When the form appears, it fades in, taking about a second or so to appear. Click Bye Bye and the form fades away in the same way.
- 2 Close the form, and return to Visual Studio .NET.

Quick Reference

To	Do this	Button
Create your own customized <i>Form</i> class using subclassing	Create a new Class Library project. Delete the <i>Class1</i> class that is added automatically. Add a Windows Form to the project. Set the properties of the form, create methods, and override <i>On</i> methods for any events you want to extend.	
Use a customized <i>Form</i> class	Create a new project using the Windows Application template. Add a reference to the class library containing the subclassed form, and add an appropriate <i>using</i> statement. Change the definition of the form in the application so that it inherits from the subclass and not from <i>System.Windows.Forms.Form</i> . For example, change: <pre>public class myForm : System.Windows.Forms.Form { : } to: public class myForm : Standard { : }</pre>	
Execute a method at periodic intervals	Use a <i>Timer</i> control. Create an event method that handles the <i>Tick</i> event. Set the <i>Interval</i> property of the timer to your requirements, and then set the <i>Enabled</i> property to <i>true</i> to start the timer running. In the <i>Tick</i> event method, set the <i>Enabled</i> property to <i>true</i> again to raise another <i>Tick</i> event, or set it to <i>false</i> to disable the timer and stop raising <i>Tick</i> events.	 Timer