CHAPTER 3

Using GUI Objects and the Visual Studio IDE

In this chapter you will:

- Create a `Form` in the Visual Studio IDE
- Use the `Toolbox` to add a `Button` to a `Form`
- Add `Labels` and `TextBoxes` to a `Form`
- Name `Forms` and controls
- Correct errors
- Decide which interface to use

You have learned to write simple C# programs that accept input from a user at the console and produce output at the command line. The environment the user sees is a program's **interface**; unfortunately, the interfaces in the applications you have written so far look dull. Most modern applications use visually pleasing graphic objects to interact with users. These graphical user interface (GUI) objects include the labels, buttons, and text boxes you manipulate or control with a mouse, touch screen, or keyboard when interacting with Windows-type programs.

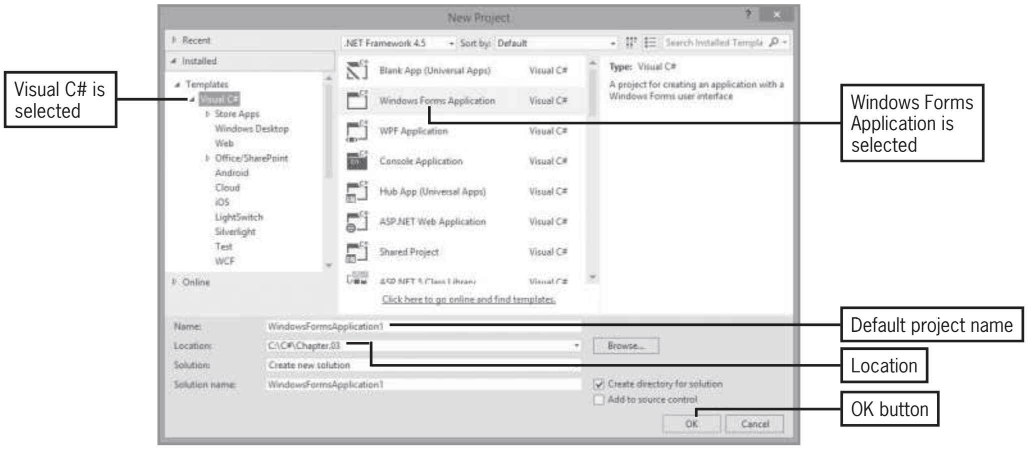
In Chapter 1, you learned that when you write a console application, you can use a simple text editor such as Notepad or you can use the Visual Studio integrated development environment (IDE). Technically, you have the same options when you write a GUI program. However, so much code is needed to create even the simplest of GUI programs that it is far more practical to develop the user interface visually in the IDE. This approach allows the IDE to automatically generate much of the code you need to develop appealing GUI programs that are easy to use.

In Visual Studio versions before .NET, C#, C++, and Visual Basic each had its own IDE, so you had to learn about a new environment with each programming language you studied. Now, you can use one IDE to create projects in all the supported languages.

Creating a Form in the IDE

Forms are rectangular GUI objects that provide an interface for collecting, displaying, and delivering information. Although they are not required, forms almost always include **controls**, which are devices such as labels, text boxes, and buttons that users can manipulate to interact with a program. The C# class that creates a form is `Form`.

To create a `Form` visually using the IDE, you start Visual Studio, select **New Project**, and then choose **Windows Forms Application**, as shown in Figure 3-1. By default, Visual Studio names your first `Forms` application *WindowsFormsApplication1*. You can change the name if you want, and you can browse to choose a location to save the application. You should almost always provide a more meaningful name for applications than the default name suggested. Most of the examples in this chapter retain the default names to minimize the number of changes required if you want to replicate the steps on your computer.


New Project
Recent
Installed
Templates
Visual C#
Store Apps
Windows Desktop
Web
Office/SharePoint
Android
Cloud
iOS
LightSwitch
Silverlight
Test
WCF
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates
Blank App (Universal Apps) Visual C#
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Hub App (Universal Apps) Visual C#
ASP.NET Web Application Visual C#
Shared Project Visual C#
ASP.NET 5 Class Library Visual C#
Click here to go online and find templates.
Type: Visual C#
A project for creating an application with a Windows Forms user interface
Browse...
OK
Cancel
Visual C# is selected
Windows Forms Application is selected
Default project name
Location
OK button

Name: WindowsFormsApplication1
Location: C:\C#\Chapter.03
Solution: Create new solution
Solution name: WindowsFormsApplication1
[✓] Create directory for solution
[ ] Add to source control

Figure 3-1 Choosing Windows Forms Application in the New Project window

After you click **OK** in the New Project window, you see the IDE main window, as shown in Figure 3-2. The main window contains several smaller windows, each of which can be resized, relocated, or closed. If a window is not currently visible, you can select it from the View menu in the menu bar. Some key features in the IDE follow:

- The *name of the project* shown in three places: on the title bar, and in two locations in the Solution Explorer. In Figure 3-2, the application has the default name WindowsFormsApplication1.
- The *menu bar*, which lies horizontally across the top of the window and includes a File menu from which you open, close, and save projects. It also contains submenus for editing, debugging, and help tasks, among others.

As you work through any project, you should choose **Save All** frequently. You can select this action from the File menu or click the **Save All** button, which has an icon that looks like two overlapping disks.

- The *Toolbox tab*, which, when opened, provides lists of controls you can drag onto a `Form` so that you can develop programs visually, using a mouse.
- The *Form Designer*, which appears in the center of the screen. This is where you design applications visually.
- The *Solution Explorer*, for viewing and managing project files and settings.
- The *Properties window*, for configuring properties and events on controls in your user interface. For example, you can use this window to set the `Size` property of a `Button` or the `Text` property of a `Form`.
- The *Error list*, which displays any warnings and errors that occur when you build or execute a program.

If some of these features are not visible after you start a project in the IDE, you can select them from the View menu.

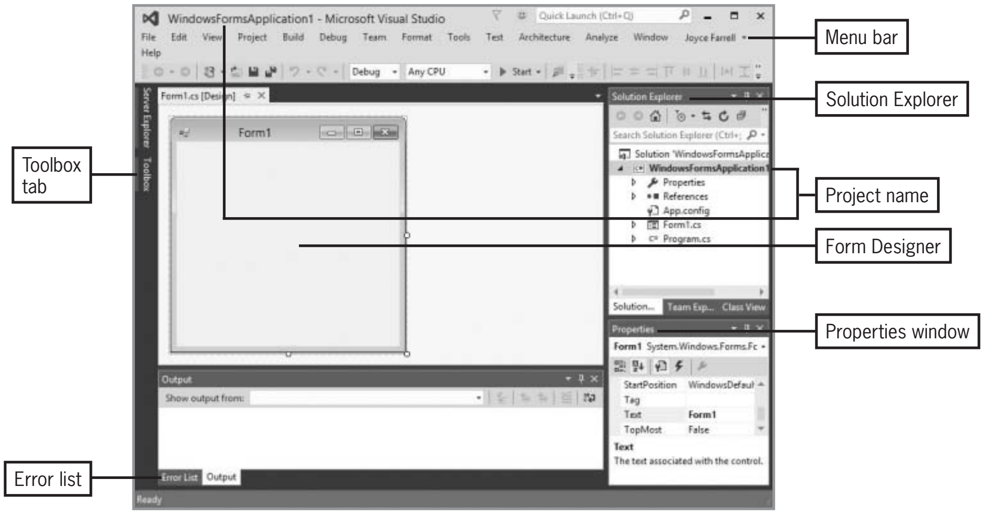


Figure 3-2 The IDE main window

The Solution Explorer file list shows the files that are part of the current project:

- The *Program.cs* file contains the automatically generated `Main()` method of the application.
- The *Form1.cs* file contains other automatically generated code and is where you write code describing the tasks you will assign to controls in your application.
- To the left of the Form1.cs file, you see a small triangle node. A **node** is an icon that can be used to expand or condense a list or a section of code. In Visual Studio, some nodes appear as triangles, and others appear as small plus or minus signs. When a triangle-shaped node points straight to the right, you see a condensed view. Clicking the triangle reveals hidden items or code—the triangle will point diagonally down and to the right, and you see all the items in the list (or all the hidden code). Clicking the triangle again collapses the list so you can work with a condensed view.

If you expand the Form1.cs node by clicking it, you see the following:

- A file named *Form1.Designer.cs*. The Windows Form Designer automatically writes code in the Designer.cs file; the code created there implements all the actions that are performed when you drag and drop controls from the Toolbox. You should avoid making manual changes to this file because you could easily corrupt a program.
- A file named *Form1* that can hold resources such as images and audio clips; you will add such resources to your programs as you learn more about C#.

When you create a Windows Forms project, Visual C# adds a form to the project and calls it `Form1`. After you click the Form in the Form Designer area, you can see the name of the form in the following locations as shown in Figure 3-3:

- On the folder tab at the top of the Form Designer area (followed by *[Design]*)
- In the title bar of the form in the Form Designer area
- In the Solution Explorer file list (with a .cs extension)
- At the top of the Properties window indicating that the properties listed are for Form1
- As the contents of the `Text` property listed within the Properties window

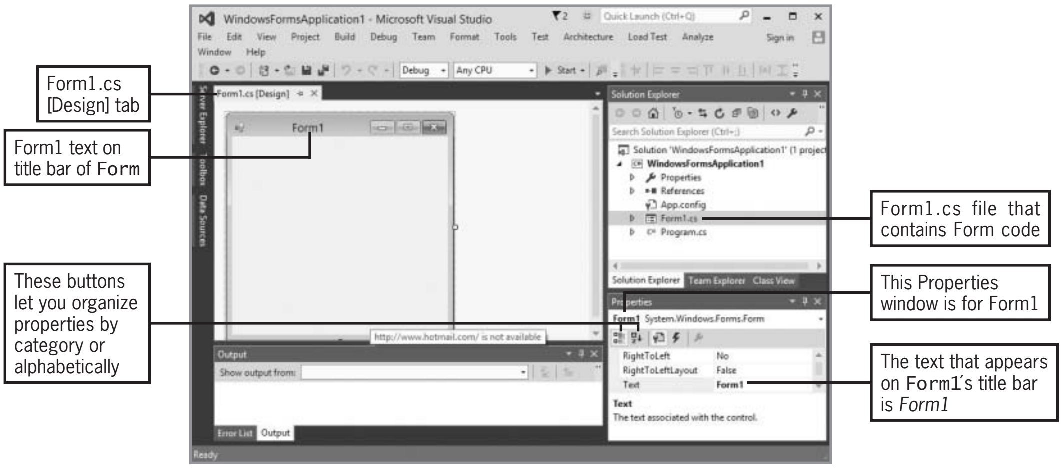

Form1.cs [Design] tab
Form1 text on title bar of Form
These buttons let you organize properties by category or alphabetically
Form1.cs file that contains Form code
This Properties window is for Form1
The text that appears on Form1's title bar is *Form1*
WindowsFormsApplication1 - Microsoft Visual Studio
Form1.cs [Design]
Form1
Solution Explorer
Solution 'WindowsFormsApplication1' (1 project)
WindowsFormsApplication1
Properties
References
App.config
Form1.cs
Program.cs
Properties
Form1 System.Windows.Forms.Form
RightToLeft No
RightToLeftLayout False
Text Form1
Text
The text associated with the control.
Output
Ready

Figure 3-3 Displaying the properties of `Form1`

You can scroll through a list of properties in the Properties window. For example, `Size`, `Text`, and `Font` are listed. If you click a property, a brief description of it appears at the bottom of the Properties window. For example, Figure 3-3 displays a description of the `Text` property of `Form1`.

You can change the appearance, size, color, and window-management features of a `Form` by setting its properties. The `Form` class contains approximately 100 properties; Table 3-1 lists a few of the `Form` features you can change. For example, setting the `Text` property allows you to specify the caption of the `Form` in the title bar, and setting the `Size`, `BackColor`, and `ForeColor` allows you to further customize the `Form`. The two left buttons at the top of the Properties list allow you to organize properties by category or alphabetically. Not every property that can be used with a `Form` appears in the Properties window in the Visual Studio IDE—only the most frequently used properties are listed.

Property Name	Description
`AcceptButton`	Gets or sets the button on the form that is clicked when the user presses the Enter key
`BackColor`	Gets or sets the background color of the `Form`
`CancelButton`	Gets or sets the button control that is clicked when the user presses the Esc key
`ForeColor`	Gets or sets the foreground color of the `Form`
`Name`	Gets or sets the name of the `Form`
`Size`	Gets or sets the size of the `Form`
`Text`	Gets or sets the text associated with the control
`Visible`	Gets or sets a value indicating whether the control is visible

Table 3-1 Selected properties of `Forms`

Some of the properties listed in the Properties window have a small plus sign that appears to the left of the property name. This is a node that you can click to expand or condense a property. For example, if you click the `Size` node, you will see an expanded list that displays the two parts of `Size` (`Width` and `Height`) separately. The node changes to a minus sign, which you can then click to condense the expanded list.

When you sort Properties in alphabetical order in the Properties window, the `Name` entry is not in alphabetical order—it appears near the top of the list. `Name` appears in parentheses because the opening parenthesis has a lower value than an *A*, so when the Properties list is sorted in alphabetical order, `Name` is near the top and easy to find. (An opening parenthesis has a lower Unicode value than any alphabetic letter; Appendix B contains information on Unicode.) For most professional applications, you will want to provide a `Form` with a reasonable identifier that is more descriptive than `Form1`.

Do not confuse a `Form`'s `Name` with its `Text` property. If you change a `Form`'s `Name`, you don't notice any difference in the visual design, but all references to the form are changed in the code. Therefore, a `Form`'s `Name` property is an identifier, and it must follow the rules for variable names. This means, for example, that a `Form`'s `Name` cannot contain spaces and cannot start with a digit. However, a `Form's Text` is what appears in its title bar, and it can be any string. For example, in an accounting application, you might name a form `BudgetForm` but assign *2016 Budget Calculator* as its `Text`.

In Chapter 2, you learned to enclose a string's contents in quotation marks. Although a `Form`'s `Text` is a string, you do not type containing quotes when you enter a value for the `Text` property in the Properties list. However, when you view C# code, you will see quotation marks around the `Text` value.

Watch the video *The Visual Studio IDE*.

TWO TRUTHS & A LIE

Creating a `Form` in the IDE

1. The Visual Studio IDE allows you to use a visual environment for designing `Forms`.
2. Some key features in Visual C# include the Toolbox, Form Designer, and Solution Explorer.
3. When you create a first Windows Forms project in any folder, Visual C# names the project MyFirstForm by default.

The false statement is #3. When you create a first Windows Forms project in any folder, Visual C# names the project WindowsFormsApplication1 by default.

Using the Toolbox to Add a `Button` to a `Form`

When you open the IDE, the left border displays a Toolbox tab. (If you don't see the tab, you can select **View** and then **Toolbox** from the IDE's menu bar.) When you open the Toolbox, a list of tool groups is displayed. The list automatically closes when you move your mouse off the list and click elsewhere, or you can pin the Toolbox to the screen to keep it open by clicking the pushpin icon at the top of the list. When you pin the Toolbox, the `Form` moves to the right to remain as visible as possible, depending on how you have sized the windows in the IDE. Selecting **All Windows Forms** at the top of the Toolbox displays a complete list of available tools; selecting **Common Controls** displays a smaller list that is a subset of the original one. As shown in Figure 3-4, the Common Controls list contains many controls: the GUI objects a user can click or manipulate. The list includes controls you probably have seen when using Windows applications—for example, `Button` and `CheckBox`. You can drag these controls onto the `Form`. This chapter features only three controls from the Toolbox—`Label`, `Button`, and `TextBox`. You will learn about many of the other controls in the chapter called "Using Controls."

All windows in Visual C# can be made dockable or floating, hidden or visible, or can be moved to new locations. To change the behavior of a window, click the down arrow or pushpin icon on the title bar, and select from the available options. You can customize many aspects of the IDE by clicking the **Tools** menu, then clicking **Options**. To reset the windows to their original states, click **Window** on the main Visual Studio menu bar, and then click **Reset Window Layout**.

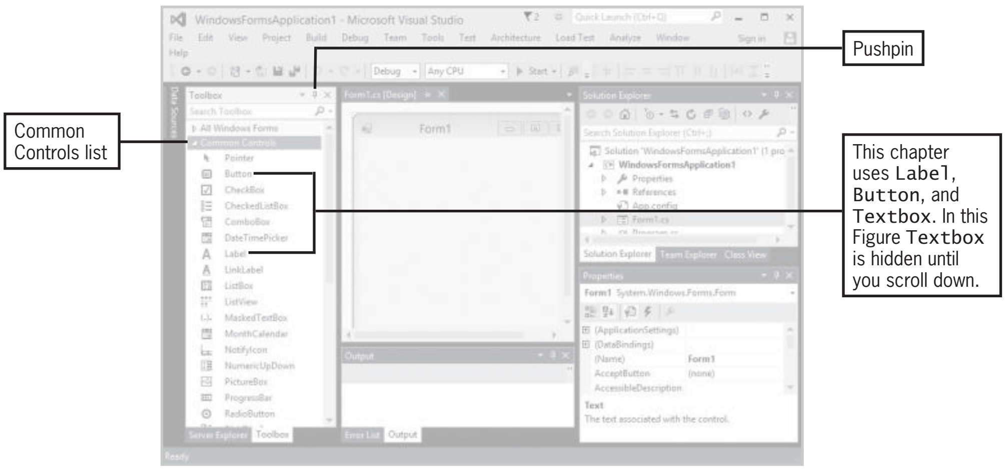

Pushpin
Common Controls list
This chapter uses Label, Button, and Textbox. In this Figure Textbox is hidden until you scroll down.

Figure 3-4 The open Toolbox in the IDE

You can drag controls onto a `Form` where you think they will be most useful or contribute to a pleasing design. After a control is on a `Form`, you can relocate it by dragging it, or delete the control by selecting it and pressing the **Del** key on your keyboard. You also can delete a control by right-clicking it and selecting the **Delete** option from the menu that appears.

In Figure 3-5, the programmer has dragged a button onto the `Form`. By default, its `Name` is `button1`. (You might guess that if you drag a second `Button` onto a `Form`, its `Name` automatically becomes `button2`.) In a professional application, you would probably want to change the `Name` property of the `Button` to a unique and memorable identifier.

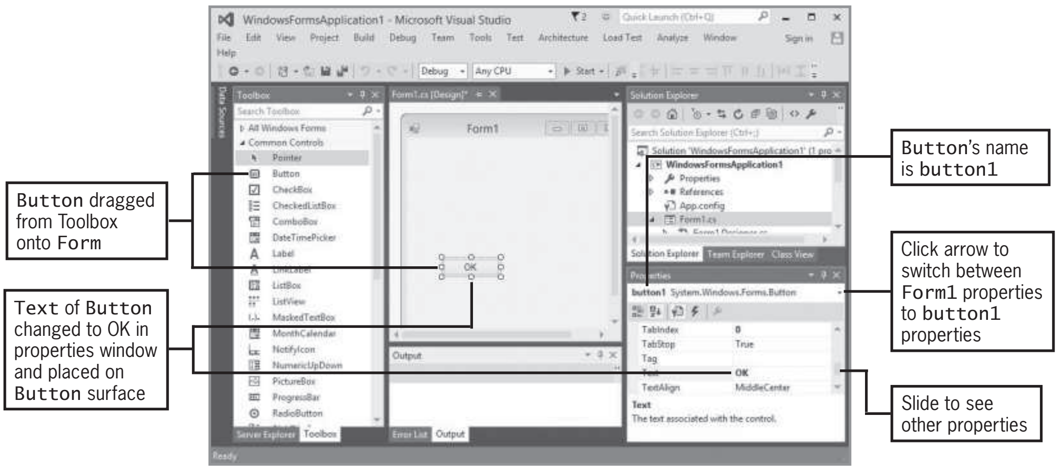

Button's name is button1
Button dragged from Toolbox onto Form
Click arrow to switch between Form1 properties to button1 properties
Text of Button changed to OK in properties window and placed on Button surface
Slide to see other properties

Figure 3-5 A `Button` dragged onto a `Form`

In Figure 3-5, the programmer has clicked button1, so the Properties window shows the properties for button1. (If you click the Form, the Properties window changes to show Form properties instead of Button properties.) The Text property of button1 has been changed to *OK*, and the button on the Form has handles that can be used to resize it. You can scroll through the Properties list and examine other properties that can be modified for the Button. For example, you can change its BackColor, ForeColor, Font, and Size. Table 3-2 describes a few of the properties available for Button objects. This chapter discusses only a few properties of each control. You can learn about other properties in the chapter "Using Controls."

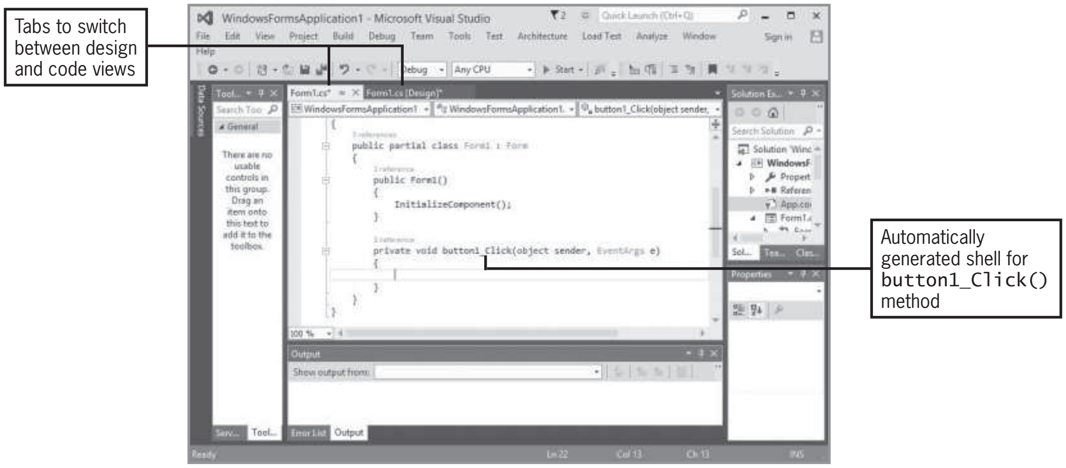
Property Name	Description
BackColor	Gets or sets the background color for the control
Enabled	Gets or sets whether the control is enabled
Font	Gets or sets the current font for the control
ForeColor	Gets or sets the foreground color of the control
Name	Gets or sets the name of the control
Size	Gets or sets the size of the control
Text	Gets or sets the text associated with the control
Visible	Gets or sets a value indicating whether the control is visible

Table 3-2 Selected properties of Buttons

You can click a control on a Form to display its Properties list in the Properties window. Alternatively, you can click the arrow next to the current control name shown at the top of the Properties list and select the desired control to view from a drop-down list. See Figure 3-5.

Adding Functionality to a Button on a Form

Adding functionality to a Button is easy when you use the IDE. After you have dragged a Button onto a Form, you can double-click it to create a method that executes when the user clicks the Button. For example, if you have dragged a Button onto a Form and have not changed its default name from button1, code is automatically generated when you double-click the button, as shown in Figure 3-6. (The viewing windows in the figure have been resized by dragging their borders so that you can see more of the code.) You can view the Form Designer again by selecting **View** and then **Designer** from the menu bar. Then, you can revert to the code window by selecting **View** and **Code**. As you create GUI applications, you frequently will switch back and forth between the visual environment and the code. Alternatively, you can use the tabs or press one of the shortcut keys displayed on the drop-down menu to switch the view.


Tabs to switch between design and code views
Automatically generated shell for button1_Click() method
WindowsFormsApplication1 - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Architecture Load Test Analyze Window Help
Form1.cs*
Form1.cs [Design]*
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.
Output
Show output from:
Ready

```
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {

        }
    }
}
```

Figure 3-6 The automatically generated code after double-clicking `button1`

When you double-click a button to generate a click method, you generate code for the button's default event. Clicking is a button's default event because users typically expect to click a button. As you learn more about C#, you will find that you can write other events for a button. For example, perhaps you want to take a specific action if a user double-clicks or right-clicks a button.

If you scroll through the exposed code, you see many generated statements, some of which are confusing. However, to make a `Button` perform an action, you can ignore most of the statements. You only need to write code between the curly braces of the method with the following header:

```
private void button1_Click(object sender, EventArgs e)
```

Instead of automatically generating the method header that executes when a user clicks a `Button`, you could write it yourself. However, you would have to know more about both methods and event handlers because other code is also automatically written elsewhere in the project that links the `Form`'s button to the method. You will learn about these topics in later chapters.

You have seen the following method header used in console applications:

```
static void Main()
```

The GUI application also contains a `Main()` method that is created automatically; you can see it by opening the Program.cs file that is part of the application.

The button1_Click() method and the Main() method headers both use the void return type, but the methods differ in the following ways:

- The button1_Click() method has a private access specifier. An **access specifier** is a keyword that dictates which types of outside classes can use a method. By default, Visual Studio makes the button1_Click() method private, which limits the ability of other classes to use the method, but you could remove the word private and notice no difference in how the button works. You will learn more about access specifiers later in this book.
- The Main() method is static, whereas the button1_Click() method is not. A static method executes without an object. The button1_Click() method cannot be static because it needs an object, which is the Form in which it is running. The chapter called "Using Classes and Objects" provides more detail about the keyword static.
- The Main() method header shown above contains no parameters between its parentheses, but the button1_Click() method has two parameters, separated with a comma. These represent values about the event (the user clicking the button) that causes execution of the method. When a user interacts with a GUI object, an **event** is generated that causes the program to perform a task. When a user clicks a Button, the action fires a **click event**, which causes the Button's Click() method to execute. This chapter discusses only click events. You can learn about other events in the "Event Handling" chapter.

If you change the Name property of the button1 object in the IDE, the name of its subsequently created Click() method will also change automatically. For example, if you rename button1 to okButton in the Properties list and then double-click it, the name of the generated method is okButton_Click().

If you change a Button's Name from button1 and then place a second button on the Form, the new Button is named button1. If you leave the first Button's default name as button1, then the second Button you place on the Form is named button2, and its default Click() method is named button2_Click(). C# automatically creates a method name for each Button based on its associated object's name. Later in this chapter you will learn what to do if you create the event method first and change the control's name later.

You are not required to create a Click() method for a Button. If, for some reason, you did not want to take any action when a user clicked a Button, you simply would not include the method in your program. Alternatively, you could create an empty method that contains no statements between its curly braces and thus does nothing. However, these choices would be unusual and a poor programming practice—you usually place a Button on a Form because you expect it to be clicked at some point. You will frustrate users if you do not allow your controls to act in expected ways.

You can write any statements you want between the curly braces of the `button1_Click()` method. For example, you can declare variables, perform arithmetic statements, and produce output. You also can include block or line comments. The next sections of this chapter include several statements added to a `Click()` method.

Watch the video *Creating a Functional Button.*

TWO TRUTHS & A LIE

Using the Toolbox to Add a `Button` to a `Form`

1. When a user clicks a `Button`, the action fires a click event that causes the `Button`'s `Click()` method to execute.
2. If a `Button`'s identifier is `reportButton`, then the name of its `Click()` method is `reportButton.Click()`.
3. You can write any statements you want between the curly braces of a `Button`'s `Click()` method.

The false statement is #2. If a `Button`'s identifier is `reportButton`, then the name of its `Click()` method is `reportButton_Click()`.

Adding `Labels` and `TextBoxes` to a `Form`

Suppose that you want to create an interactive GUI program that accepts two numbers from a user and outputs their sum when the user clicks a `Button`. To provide prompts for the user and to display output, you can add `Labels` to a `Form`, and to get input from a user, you can provide `TextBoxes`.

A **label** is a control that you use to display text to communicate with an application's user; the C# class that creates a label is `Label`. Just like a `Button`, you can drag a `Label` onto a `Form`, as shown in Figure 3-7. By default, the first `Label` you drag onto a `Form` is named `label1`, and the text that appears on the `Label` also is `label1`. You can change its `Text` property to display any string of text; depending on the amount of text you enter, you might need to change the size of the `Label` by dragging its resizing handles or altering its `Size` property. In Figure 3-7, *Enter a number* has been assigned to `label1`'s `Text` property. If you want to create multiple lines of text on a `Label`, click the small, downward-pointing arrow to the right of the `Text` property value to open an editor, and type the needed text in separate lines. You might want a program to start with no text appearing on a label. If so, just delete the entry from the Text property in the Properties list. If you execute a program that starts with a blank label and don't make any changes to its color, the label will be invisible to the user. Note that a label's Text property can be blank, but a Label's Name property cannot.

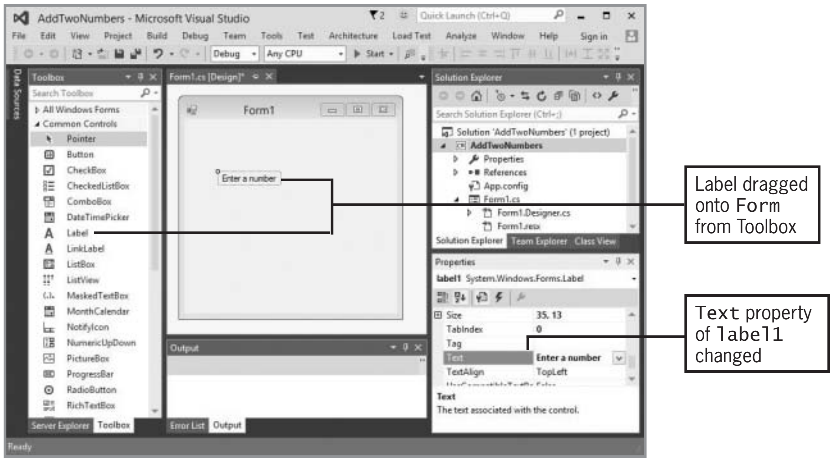

Label dragged onto Form from Toolbox
Text property of label1 changed

Figure 3-7 A Label on a Form

A **text box** is a GUI control through which a user can enter input data; the C# class is TextBox. Figure 3-8 shows a TextBox on a Form. If the intention is for a user to enter data in a TextBox, you might want to start with its Text property empty, but you are allowed to add text to its Text property if desired.

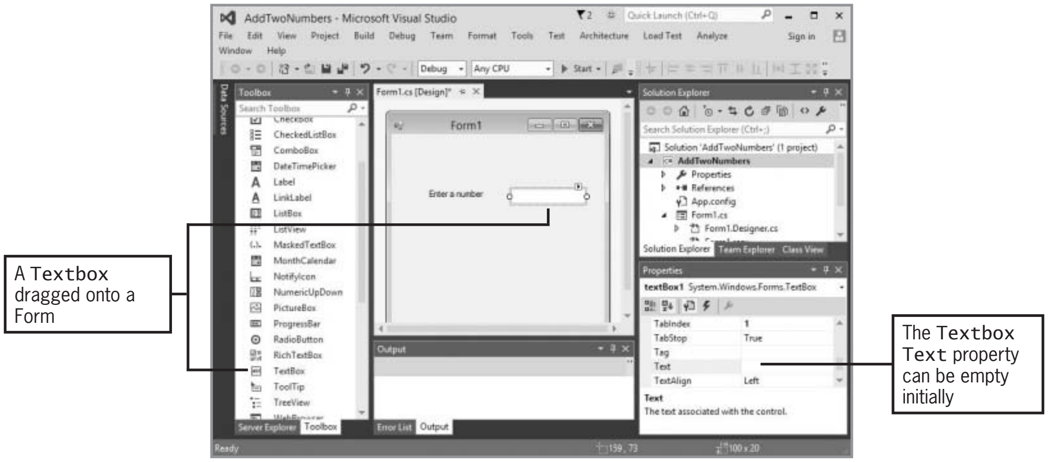

A Textbox dragged onto a Form
The Textbox Text property can be empty initially

Figure 3-8 A Label and TextBox on a form

Figure 3-9 shows a `Form` with three `Label`s, two `TextBox`es, and a `Button`. The third `Label` has not yet been provided with new text, so it still contains *label3*. You can delete the `Text` value in the Properties window if you want the `Label` to be empty at first.

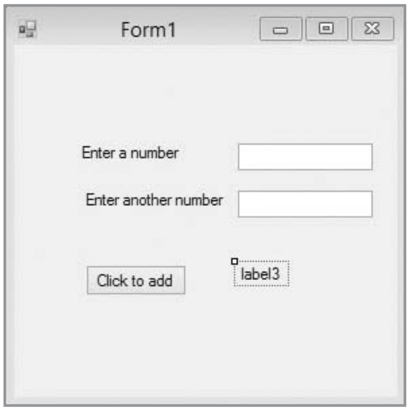


Figure 3-9 A `Form` with several controls

When a user runs the program that displays the `Form` in Figure 3-9, the intention is for the user to enter a number in each `TextBox`. When the user clicks the `Button`, the sum of the two numbers will be displayed in `label3`. To sum the numbers, you must write code for the `Button`. In the Form Designer, you can double-click the `Button` to expose the prewritten shell of the `button1_Click()` method.

Figure 3-10 shows the code you can write within the `button1_Click()` method. The method contains declarations for three integers. When a user types a value into a `TextBox` in an executing program, it becomes the value for the `Text` property of the `TextBox`. Because a user might type any value, the value in a `TextBox` is a `string` by default. When you write an application in which the user is supposed to enter a number, you must convert the entered `string` into a number. You are familiar with this process because it's exactly the same action required when a user enters data in response to a `ReadLine()` statement in a console application.

The last two statements in the method perform the necessary addition operation and assign a `string` with the concatenated `sum` to the `Text` property of the `Label` that displays the output.

```
private void button1_Click(object sender, EventArgs e)
{
   int num1;
   int num2;
   int sum;
   num1 = Convert.ToInt32(textBox1.Text);
   num2 = Convert.ToInt32(textBox2.Text);
   sum = num1 + num2;
   label3.Text = "Sum is " + sum;
}
```

Figure 3-10 The `button1_Click()` method that calculates the sum of the entries in two `TextBoxes`

You can execute a program from the IDE by selecting **Debug** from the menu bar and then **Start Without Debugging**. (As an alternative, you can hold down the **Ctrl** key and press **F5**.) Figure 3-11 shows how the `Form` appears when the program executes and the result after the user has entered numbers and clicked the `Button`. The period during which you design a program is called **design time**. When you execute a program, the stage is called **runtime**.

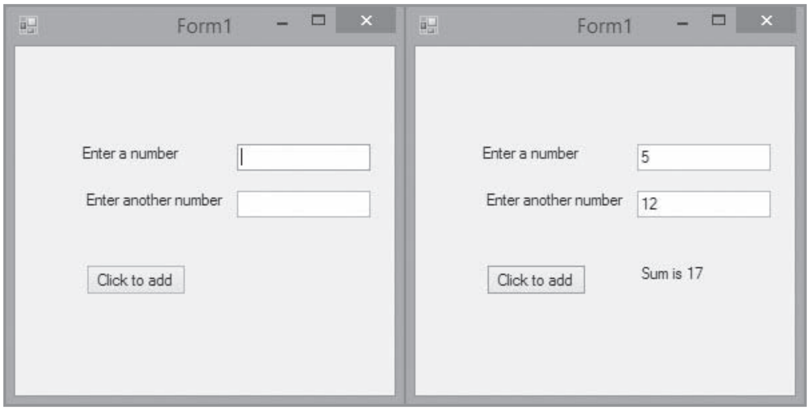


Figure 3-11 The `Form` when it first appears and after a user has entered two integers and clicked the `Button`

You can change the values in either or both of the `TextBoxes` in the `Form` and click the `Button` to see a new result. When you finish using the application, you can close it by clicking the **Close** button in the upper-right corner of the `Form`.

Understanding Focus and Tab Control

When a GUI program displays a `Form`, one of the components on the `Form` has focus. The control with **focus** is the one that can receive keyboard input. When a `Button` has focus, a thin, bright line appears around it to draw your attention, and the `Button`'s associated event executes when you click the button or press the **Enter** key on the keyboard. When a `TextBox` has focus, a blinking cursor appears inside it.

When an application is running, the user can give focus to a component by clicking it or can switch focus from one component to another using the Tab key. The order in which controls receive focus from successive Tab key presses is their **tab order**. By default, controls are assigned a tab order based on the order in which you place them on a `Form` when designing a program. The first control you place on a `Form` has `TabIndex 0`, the next control has `TabIndex 1`, and so on. When you start a program, the control with `TabIndex 0` automatically has focus. You can check a control's tab order by viewing its `TabIndex` property in the Properties list of the IDE. If you do not want a control to be part of the tabbing sequence for a `Form`, you can change its `TabStop` property from `true` to `false` in the Properties list.

You can change a control's `TabIndex` value by changing the entry next to `TabIndex` in the Properties list of the IDE. Alternatively, you can select **View** from the menu bar and then select **Tab Order**. You see a representation of the `Form` with a small blue number next to each control to represent the tab values. To change the focus order for the controls, simply click each control in the desired order. Select **View** and then **Tab Order** again to remove the blue numbers.

Watch the video *Visual Studio's Automatically Generated Code*.

Formatting Data in GUI Applications

In Chapter 2, you learned about format strings that can be used to make numeric data align appropriately, display the desired number of decimal places, and display dollar signs. For example, the following displays *Total is $4.55* because the `double` variable `myMoney` is converted to currency format and placed in position `0` in a string:

```
double myMoney = 4.558;
string mySentence = String.Format("Total is {0}",
   myMoney.ToString("C"));
WriteLine(mySentence);
```

You can use the same `String.Format()` method to create formatted strings in your GUI applications. For example, suppose you have retrieved a double from a `TextBox` with a statement similar to the following:

```
double money = Convert.ToDouble(textBox1.Text);
```

You can display the value on a label with explanatory text and as currency with two decimal places with a statement similar to the following:

```
label1.Text = String.Format("The money value is {0}",
   money.ToString("C2"));
```

If necessary, you also can use escape characters in strings that are assigned to controls such as labels. For example, a two-line label might be created using the following code:

```
label1.Text = "Hello\nthere";
```

Changing a Label's Font

The Font property for controls was described briefly in Table 3-2 earlier in this chapter. You will learn more about this property in Chapter 12, but you might want to experiment with it now to align numbers.

When you click any control on a Form and examine its Properties list, you discover that the default font for controls is Microsoft Sans Serif. This font is a **proportional font**, which means that different characters have different pitches or widths. For example, a *w* is wider than an *i*. The opposite of a proportional font is a **fixed-pitch font**, or a **monospaced font**, in which each character has the same width. You might want to use a fixed-pitch font to help align text values on controls. For example, when you stack $11.11 and $88.88 on a Label, you typically want the characters to be an equal width so that the values align.

To change the font for a Label, select the small box with an ellipsis (three dots) to the right of the existing Font name in the Properties list, and select a new Font from the dialog box that opens. For example, Courier New and Consolas are popular fixed-pitch fonts. (You can also change the font style and size if desired.)

TWO TRUTHS & A LIE

Adding Labels and TextBoxes to a Form

1. A Label and a TextBox both have a Text property.
2. When a user types a value on a Label in an executing program, it becomes the value for the Text property of the Label.
3. You can display the value on a Label using a format string.

The false statement is #2. A user does not type on a Label. When a user types a value into a TextBox in an executing program, it becomes the value for the Text property of the TextBox.

Naming `Forms` and `Controls`

Usually, you want to provide reasonable `Name` property values for all the controls you place on a `Form`. Although any identifier that starts with a letter is legal, note the following conventions:

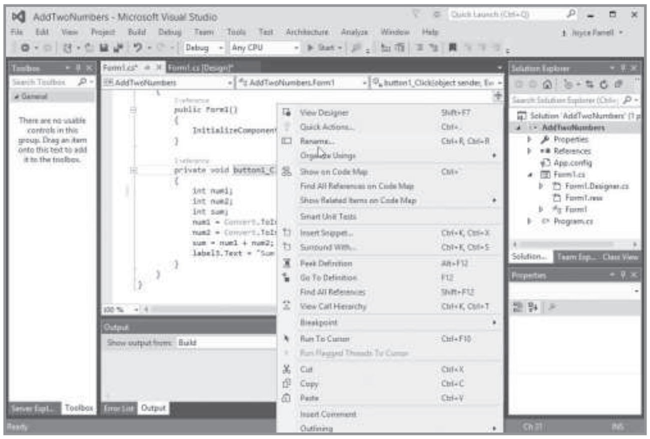
- Start control names with a lowercase letter, and use camel casing as appropriate.
- Start `Form` names with an uppercase letter, and use camel casing as appropriate.
- Use the type of object in the name. For example, use names such as `okButton`, `firstValueTextBox`, or `ArithmeticForm`.

Professional programmers usually do not retain the default names for `Forms` and controls. An exception is sometimes made for `Labels` that never change. For example, if three labels provide directions or explanations to the user and are never altered during a program's execution, many programmers would approve of retaining their names as `label1`, `label2`, and `label3`.

Most often, you will want to name controls as soon as you add them to a `Form`. If you rename a control after you have created an event for it, the program still works, but the name of the control no longer corresponds to its associated method, making the code harder to understand. If you simply change the method name to match the control name, the event code will generate errors. For example, assume that you have created a button named `button1` and that you have double-clicked it, creating a method named `button1_Click()`.

Then suppose that you decide to change the `Name` property of the `Button` to `sumButton`. Afterward, when you look at the program code, the `Click()` method is still named `button1_Click()`. To fix this, you must refactor the code. **Code refactoring** is the process of changing a program's internal structure without changing the way the program works. To refactor the code after changing the button's name, do the following:

- After you change the name of the control using the Designer, switch to view the code.
- Right-click the name of the method `button1_Click()`.
- From the menu that appears, click **Rename**, as shown in Figure 3-12.

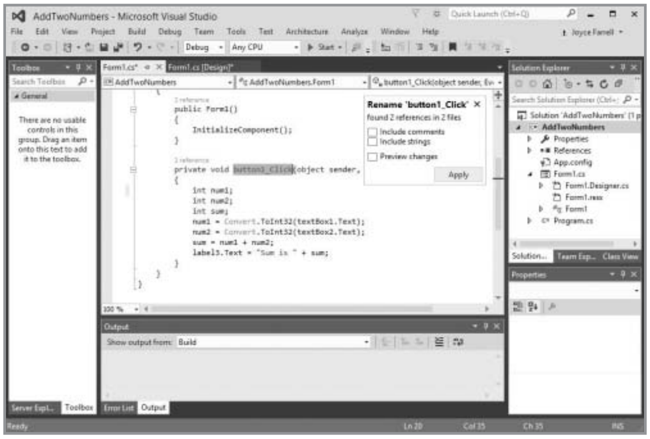


```
public Form1()
{
    InitializeComponent
}

private void button1_C
{
    int num1;
    int num2;
    int sum;
    num1 = Convert.ToI
    num2 = Convert.ToI
    sum = num1 + num2;
    label3.Text = "Sum
}
```

Figure 3-12 Choosing the Rename option for a method

- When you click **Rename**, the `button1_Click` method name is highlighted and a box appears with the heading *Rename button1_Click*, as shown in Figure 3-13.
- Type the new method name (`sumButton_Click`) to replace `button1_Click` in the code.
- When you finish, you can check the Preview changes checkbox if you want to view all the changes before they are applied, although at this point, the changes will not have much meaning for you. Click **Apply** or press **Enter**. A Preview Changes dialog box will highlight the change. You can confirm the change by clicking **Apply**.



```
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender,
        {
            int num1;
            int num2;
            int sum;
            num1 = Convert.ToInt32(textBox1.Text);
            num2 = Convert.ToInt32(textBox2.Text);
            sum = num1 + num2;
            label3.Text = "Sum is " + sum;
        }
    }
}
```

Figure 3-13 Renaming the `button1_Click()` method

TWO TRUTHS & A LIE

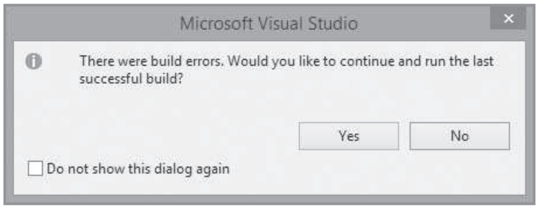
Naming `Forms` and `Controls`

1. `Label` and `TextBox` names usually start with an uppercase letter, but `Form` names usually start with a lowercase letter.
2. Professional programmers usually do not retain the default names for `Forms` and controls.
3. If you rename a control after you have created an event for it, you must refactor the code.

The false statement is #1. `Label` and `TextBox` names usually start with a lowercase letter, but `Form` names usually starts with an uppercase letter.

Correcting Errors

Just like in console-based programs, you will often generate syntax errors when you use the visual developer to create GUI applications. If you build or run a program that contains a syntax error, you see *Build failed* in the lower-left corner of the IDE. You also see an error dialog box like the one shown in Figure 3-14. When you are developing a program, you should always click **No** in response to “Would you like to continue and run the last successful build?” Clicking **Yes** will run the previous version of the program you created before inserting the mistake that caused the dialog box to appear. Instead, click **No**, examine the error messages and code, and correct the mistake.



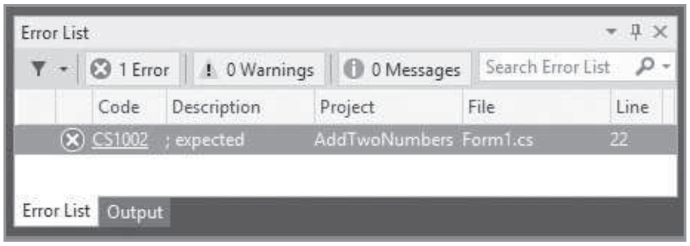
Microsoft Visual Studio

There were build errors. Would you like to continue and run the last successful build?

Figure 3-14 Dialog box that appears when an error occurs during compilation

When you select **Build** from the menu bar, two options are Build Solution and Rebuild Solution. The Build Solution option uses only files that have changed since the most recent build, whereas Rebuild Solution works with all the files in a project, regardless of whether they have changed. The Build Solution option is usually sufficient and faster, but if you have made many changes to several parts of a project, you might choose Rebuild Solution to make sure that all files are currently coordinated.

When errors occur, they are shown in the error list at the bottom of the screen. If you do not see error messages, you can click **View** from the IDE’s menu bar and then click **Error List**. If the list is partially or completely hidden, you can drag up the divider between the development window and the error list and then click the **Error List** tab. Figure 3-15 shows a single error message, *; expected*, which means *semicolon expected*. The error list also shows the file and line number in which the error occurred. If you double-click the error message, the cursor is placed at the location in the code where the error was found. A wiggly red underline will help you locate the error in the code window. In this case, a semicolon is missing at the end of a statement. You can fix the error and attempt another compilation.



Error List

Code	Description	Project	File	Line
CS1002	; expected	AddTwoNumbers	Form1.cs	22

Figure 3-15 The error list

You can add line numbers to your code for reference by selecting **Tools** from Visual Studio's menu bar, then selecting **Options**. Use the triangle-shaped node to expand the Text Editor, and then click the node next to C# from the list that appears. Click **General**, select the **Line numbers** check box, and then click **OK**.

As with console-based programs, GUI applications might compile successfully but contain logical errors. For example, you might type a minus sign instead of a plus sign in the statement that is intended to calculate the sum of two input numbers. You should be prepared to execute a program multiple times and carefully examine the output to discover any logical errors.

Deleting an Unwanted Event-Handling Method

When you are working in the Form Designer in the IDE, it is easy to inadvertently double-click a control and automatically create an event-handling method that you do not want. For example, you might double-click a `Label` named `label1` by mistake and generate a method named `label1_Click()`. You can leave the automatically created method empty so that no actions result, but in a professional program you typically would not include empty methods. Such empty methods that never execute are known as **orphaned methods**. You should not just delete an orphaned method because, behind the scenes, other code might have been created that refers to the method; if so, the program will not compile. Instead of deleting the method code, you should click the control with the unwanted method so it appears in the Properties window, then click the Events button, which looks like a lightning bolt. Select the event you want to eliminate, and delete the name of the method (see Figure 3-16). This eliminates all the references to the method, and your program can again run successfully.

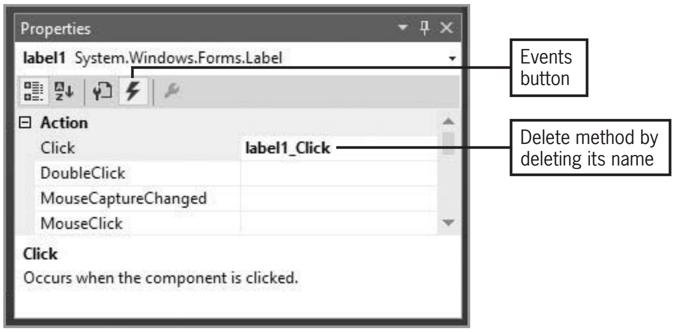

Properties
label1 System.Windows.Forms.Label
Action
Click
label1_Click
DoubleClick
MouseCaptureChanged
MouseClick
Click
Occurs when the component is clicked.
Events button
Delete method by deleting its name

Figure 3-16 Deleting an event from the Properties window

Failing to Close a `Form` Before Attempting to Reexecute a Program

Often you will execute a GUI application and notice something you want to correct in the program. For example, you might want to reword the `Text` on a `Label`, or reposition a `Button` on the `Form`. In your haste to make the improvement, you might forget to close the application that is still running. If you make a change to the program and then try to rerun it, you get an error message that indicates that changes are not allowed while the program is running. The solution is simply to close the previous execution of the application and then try again.

Using Visual Studio Help

When working with a class that is new to you, such as `Button` or `Form`, no book can answer all of your questions. The ultimate authority on C# classes is the Visual Studio Help documentation. You should use this tool often as you continue to learn about C# in particular and the Visual Studio products in general. The Help documentation for Visual Studio is in the MSDN Library, which you can install locally on your own computer. It is also available at http://msdn.microsoft.com. From within Visual Studio, you can click **Help** on the menu bar, and then click **View Help**. You will be taken to the Visual Studio Web site where you can type a topic in the Search box.

TWO TRUTHS & A LIE

Correcting Errors

1. When a program in Visual Studio has a syntax error, you see an error dialog box that asks if you want to continue with the last successful build. You should always respond No.
2. Program errors are harder to locate in the Visual Studio IDE than they are in applications written using a plain text editor.
3. If you inadvertently create a `Click()` method in Visual Studio, you should not delete its code. You should select the event in the Properties window and delete it there.

The false statement is #2. In the Visual Studio IDE, any displayed syntax errors include the filename, line, position in the line, and project in which the error occurred.

Deciding Which Interface to Use

You have learned to create console applications in which most of the action occurs in a `Main()` method and the `WriteLine()` and `ReadLine()` methods are used for input and output. You also have learned to create GUI applications in which most of the action occurs within an event-handling method such as a `Click()` method and `Label`s and `TextBox`es are used for input and output. Both types of applications can use declared variables and constants, and, as you will learn in the next few chapters, both types of programs can contain the basic building blocks of applications—decisions, loops, arrays, and calls to other methods. When you want to write a program that displays a greeting or sums some integers, you can do so using either type of interface and employing the same logic to get the same results. So, should you develop programs using a GUI interface or a console interface?

- GUI applications look "snazzier." It is easy to add colors and fonts to them, and they contain controls that a user can manipulate with a mouse or touch screen. Also, users are accustomed to GUI applications from their experiences on the Web. However, GUI programs usually take longer to develop than their console counterparts because you can spend a lot of time setting up the controls, placing them on a `Form`, and adjusting their sizes and relative positions before you write the code that does the actual work of the program. GUI applications created in the IDE also require much more disk space to store.

Designing aesthetically pleasing, functional, and user-friendly `Forms` is an art; entire books are devoted to the topic.

- Console applications look dull in comparison to GUIs, but they can often be developed more quickly because you do not spend much time setting up objects for a user to manipulate. When you are learning programming concepts like decision making and looping, you might prefer to keep the interface simple so that you can better concentrate on the new logical constructs being presented.

In short, it doesn't matter which interface you use to develop programs while learning the intricacies of the C# programming language. In the following "You Do It" section, you develop an application that is similar to the one you developed at the console in Chapter 1. In the programming exercises at the end of this chapter, you will develop programs that are identical in purpose to programs written using the console in the chapter called "Using Data." Throughout the next several chapters on decisions, looping, and arrays, many concepts will be illustrated in console applications, because the programs are shorter and the development is simpler. However, you also will occasionally see the same concept illustrated in a program that uses a GUI interface, to remind you that the program logic is the same no matter which interface is used. After you complete the chapters "Using Classes and Objects," "Using Controls," and "Handling Events," you will be able to write even more sophisticated GUI applications.

When writing your own programs, you will use the interface you prefer or one that your instructor or boss requires. If time permits, you might even want to develop programs both ways in future chapter exercises.

TWO TRUTHS & A LIE

Deciding Which Interface to Use

1. Console applications are used for declaring variables and constants and performing arithmetic, but GUI applications are not.
2. GUI programs usually take longer to develop than their console counterparts because you can spend a lot of time setting up and arranging the controls.
3. Console applications look dull compared to GUI applications, but they can often be developed more quickly because you do not spend much time setting up objects for a user to manipulate.

The false statement is #1. Both console and GUI applications can use declared variables and constants and perform arithmetic.

You Do It

Working with the Visual Studio IDE

In the next steps, you use the IDE to create a `Form` with a `Button` and a `Label`. Your first console application in Chapter 1's "You Do It" section displayed "Hello, world!" at the command prompt. This application displays "Hello, Visual World!" on a `Form`.

1. Open Microsoft Visual Studio. You might have a desktop shortcut you can double-click, or in Windows 8.1 you can swipe from the right, click **Search**, type the first few letters of **Visual Studio** in the search box, and then select it from the list of choices. Your steps might differ slightly from the ones listed here if you are using a different version of Visual Studio or a different operating system. If you are using a school network, you might be able to select **Visual Studio** from the school's computing menu.
2. Select **New Project**. Then, in the New Project window, select **Visual C#** and **Windows Forms Application**. (Refer to Figure 3-1.) Instead of using the default project name, use **HelloVisualWorld**, and select a location to store the project. Then click **OK**.

(continues)

(continued)

3. The HelloVisualWorld development environment opens, as shown in Figure 3-17. The text in the title bar of the blank `Form` contains the default text `Form1`. If you click the `Form`, its Properties window appears in the lower-right portion of the screen, and you can see that the `Text` property for the `Form` is set to `Form1`. Take a moment to scroll through the list in the Properties window, examining the values of other properties of the `Form`. For example, the value of the `Size` property is 300, 300 by default.

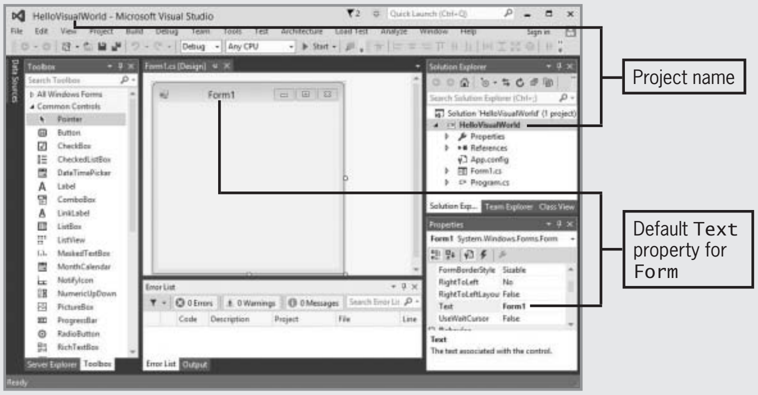


Figure 3-17 The HelloVisualWorld project development environment

If you do not see the Properties window in the lower-right corner of your screen, click the title bar on the `Form`. Alternatively, click **View** in the menu bar and click **Properties Window**.

If you do not see the Toolbox shown in Figure 3-17, click the **Toolbox tab** at the left side of the screen, and then click the pushpin near the top to pin the Toolbox to the screen. Alternatively, click **View** from the menu bar and then click **Toolbox**.

If you do not see the error list at the bottom of the screen, as in Figure 3-17, you can select **View** from the menu bar and then select **Error List**. You might have to drag up the divider that separates the error list from the window above it.

4. In the Properties window, change the `Name` of the `Form` to **HelloForm**. (The `Name` property is under Design if you choose to have the list categorized; if you choose to display it alphabetically, then `Name` is near the top.) Then change the `Text` of the `Form` to **Hello Visual World**. Press **Enter**; the title of the `Form` in the center of the screen changes to *Hello Visual World*, as shown in Figure 3-18.

(continues)

(continued)

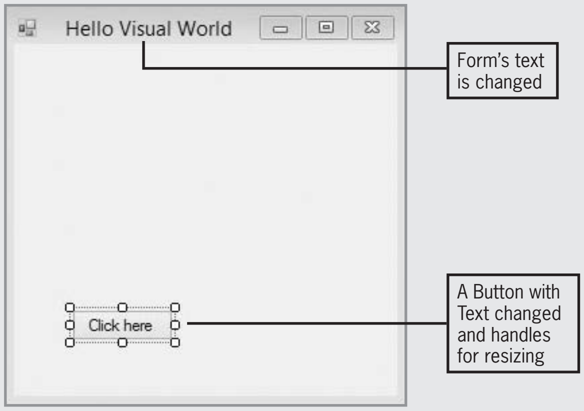


Figure 3-18 A `Form` with changed `Text` and a `Button`

5. Examine the Toolbox on the left side of the screen. Select **Common Controls** if it is not already selected. In the Toolbox, click and hold **Button**. As you move your mouse off the Toolbox and onto the `Form`, the mouse pointer changes so that it appears to carry a `Button`. Position your mouse anywhere on the form, then release the mouse button. The `Button` appears on the `Form` and contains the text *button1*. When you click the `Button`, handles appear that you can drag to resize it. When you click off the `Button` on the `Form`, the handles disappear. Click the `Button` to display the properties for `button1`. Change its `Name` to **`displayOutputButton`**, and change its `Text` property to **Click here**. When you press **Enter** on the keyboard or click anywhere on the `Form`, the text of the `Button` on the `Form` changes to *Click here*. See Figure 3-18.
6. Scroll through the other `displayOutputButton` properties. For example, examine the `Location` property. The first value listed for `Location` indicates horizontal position, and the second value indicates vertical position. Drag the `Button` across the `Form` to a new position. Each time you release your mouse button, the value of the `Form` `Button`'s `Location` property is updated to reflect the new location. Try to drag the `Button` to `Location` 100, 150. Alternatively, delete the contents of the `Location` property field and type **100, 150**. The `Button` moves to the requested location on the `Form`.
7. Save your form by clicking **File** on the menu bar, then clicking **Save All**. Alternatively, you can click the **Save All** button on the toolbar; its icon is two overlapping disks.

(continues)

(continued)

8. Although your Form doesn't do much yet, you can execute the program anyway. Click **Debug** on the menu bar, and then click **Start Without Debugging**. The Form appears. You can drag, minimize, and restore it, and you can click its Button. The Button has not yet been programmed to do anything, but it appears to be pressed when you click it. Click the Form's **Close** button to dismiss the Form.
9. You can close a project at any time and come back to it later. Exit Visual Studio now by clicking the **Close** button in the upper-right corner of the screen, or by clicking **File** on the menu bar and then clicking **Exit**. If you have made more changes since the last time you saved, you will be prompted to save again. When you choose **Yes**, the program closes.

Providing Functionality for a Button

In the next steps, you make the Button on the HelloVisualWorld Form functional; it will display a message when the user clicks it.

1. Start Visual Studio. Select **File** from the menu, click **Open**, and browse for the **HelloVisualWorld** project. Alternatively, select **File** from the menu, click **Recent Projects and Solutions**, and select the project from the pop-up list.
2. When the Form appears, drag a Label from the Toolbox to the Form. Change its Name to **helloLabel** and its Text property to **Hello, Visual World!**. If you want your Form to match Figure 3-19 exactly, change the Label's Location property to **90, 75**.

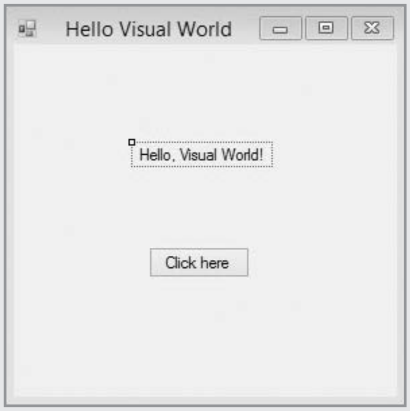


Figure 3-19 The helloLabel on the Form

(continues)

(continued)

3. Scroll through the Properties list for the `Label`, and change its `Visible` property from `True` to `False`. You can still see the `Label` in the Designer, but the `Label` will be invisible when you execute the program.
4. Double-click the `Button` on the `Form`. A new window that contains program code is displayed, revealing a newly created `displayOutputButton_Click()` method with no statements. Between the curly braces, type the following statement that causes the `Label` to appear when the user clicks the `Button` on the `Form`:

   ```
   helloLabel.Visible = true;
   ```
5. Save the file, then run the program by clicking **Debug** on the menu bar and clicking **Start Without Debugging**, or by pressing **Ctrl+F5**. When the `Form` appears, click the **Click here** button to reveal the greeting. If you want, experiment with values for some of the `Label` properties, such as `Font`, `BackColor`, and `ForeColor`.

Adding a Second `Button` *to a* `Form`

In the next steps, you add a second `Button` to the `Form` in the `HelloVisualWorld` project.

1. In the IDE, switch to Design View. Drag a second button onto the `Form` below the first `Button`. Change the new `Button`'s `Name` property to **`changeOutputButton`** and change its `Text` to **Click me last**.
2. Double-click the `changeOutputButton` to expose the `changeOutputButton_Click()` method.
3. Between the curly braces of the method, add the following code:

   ```
   helloLabel.Text = "Goodbye";
   ```
4. Save the project, then execute the program. Click the `displayOutputButton` to reveal *Hello, V isual World!*. Then click the `changeOutputButton` to reveal *Goodbye*.
5. Execute the program again. This time, click the second `Button` first. No message appears because the `Label` with the message has not been made visible. When you click the `displayOutputButton`, the `Label` becomes visible, but it displays *Goodbye* because the `changeOutputButton`'s `Click()` method has already changed the `Label`'s `Text`.

(continues)

(continued)

6. Return to the Form1.cs [Design] tab. Click the `changeOutputButton`, and change its `Enabled` property to `False`. Now, when the program executes, the user will not be immediately able to click the second `Button`.
7. Double-click the `displayOutputButton`. Add the following statement to the method so that the `changeOutputButton` becomes enabled when the `displayOutputButton` is clicked:

```
changeOutputButton.Enabled = true;
```

8. Save the project, then execute the program. When the `Form` appears, the `changeOutputButton` is dimmed and not clickable because it is not enabled. Click the enabled `displayOutputButton` to reveal the *Hello, Visual World!* message and to enable the second `Button`. Then click the `changeOutputButton` to expose the *Goodbye* message.
9. Dismiss the `Form`, and close Visual Studio.

Chapter Summary

- `Forms` are GUI objects that provide an interface for collecting, displaying, and delivering information. They almost always include controls such as labels, text boxes, and buttons that users can manipulate to interact with a program. Every `Form` and control on a `Form` has multiple properties you can set using the IDE.
- The Toolbox displays a list of available controls you can add to a `Form`. The list includes `Button`, `CheckBox`, and `Label`. From the Toolbox, you can drag controls onto a `Form` where they will be most useful. After you have dragged a `Button` onto a `Form`, you can double-click it to create the method that executes when a user clicks the `Button`, and you can write any statements you want between the curly braces of the `Click()` method.
- `Label`s are controls that you use to display text to communicate with an application's user. `TextBox`es are controls through which a user can enter input data in a GUI application. Both have a `Text` property; frequently an application starts with the `Text` empty for `TextBox`es. Because a user might type any value, the value in a `TextBox` is a `string` by default.

- Usually, you want to provide reasonable `Name` property values for all the controls you place on a `Form`. Although any identifier that starts with a letter is legal, by convention you should start control names with a lowercase letter and use camel casing as appropriate, start `Form` names with an uppercase letter and use camel casing as appropriate, and use the type of object in the name.
- If you build or run a program that contains a syntax error, you see *Build failed* in the lower-left corner of the IDE and an error dialog box. An error list shows the filename, line, position in the line, and project for each error. If you double-click an error message, the cursor is placed at the location in the code where the error was found. If you inadvertently create an event-handling method that you do not want, you should eliminate the event using the Properties window in the IDE. If you rename a control after you have created an event for it, you must refactor the code. The ultimate authority on C# classes is the Visual Studio Help documentation.
- Both console and GUI applications can contain variables and constants, decisions, loops, arrays, and calls to other methods. GUI applications look "snazzier," and they contain controls that a user can manipulate with a mouse. However, GUI programs usually take longer to develop than their console counterparts. When writing your own programs, you will use the interface you prefer or one that your instructor or boss requires.

Key Terms

The **interface** is the environment a user sees when a program executes.

Forms are GUI objects that provide an interface for collecting, displaying, and delivering information.

Controls are devices such as labels, text boxes, and buttons that users can manipulate to communicate with a GUI program.

A **node** is an icon that appears beside a list or a section of code and that can be expanded or condensed.

An **access specifier** is a keyword that dictates what outside classes can use a method.

An **event** causes a program to perform a task.

A **click event** is the event generated when a user clicks a control in a GUI program.

A **label** is a control that you use to display text to communicate with an application's user.

A **text box** is a control through which a user can enter input data in a GUI application.

Design time is the period of time during which a programmer designs the interface and writes the code.

Runtime is the period of time during which a program executes.

Focus describes the attribute of a `Form` control that stands out visually from the others and that reacts to keyboard input.

Tab order describes the sequence of controls selected when the user presses the Tab key.

A **proportional font** is one in which different characters have different pitches or widths.

A **fixed-pitch font** is one in which each character occupies the same width.

A **monospaced font** is one in which each character occupies the same width.

Code refactoring is the process of changing a program's internal structure without changing the way the program works.

An **orphaned method** is one that never executes in an application and thus serves no purpose.

Review Questions

1. Which of the following is a GUI object that provides an interface for collecting, displaying, and delivering information and that contains other controls?

   a. `Form`
   b. `Button`
   c. `TextBox`
   d. `Label`

2. In the Visual Studio IDE main window, where does the menu bar lie?

   a. vertically along the left border
   b. vertically along the right border
   c. horizontally across the top of the window
   d. horizontally across the bottom of the window

3. In the IDE, the area where you visually construct a `Form` is the ____________________ .

   a. Toolbox
   b. Palette
   c. Easel
   d. Form Designer

4. When you create a new Windows Forms project, by default the first `Form` you see is named ____________________ .

   a. `Form`
   b. `Form1`
   c. `FormA`
   d. `FormAlpha`

5. The `Form` class has ____________________ properties.

   a. three
   b. ten
   c. about 100
   d. about 1000

6. Which of the following is not a `Form` property?

   a. `BackColor`
   b. `ProjectName`
   c. `Size`
   d. `Text`

7. Which of the following is a legal `Form Name` property?
   a. `Payroll Form`
   b. `PayrollForm`
   c. either of the above
   d. none of the above

8. Which of the following is a legal `Form Text` property?
   a. `Payroll Form`
   b. `PayrollForm`
   c. either of the above
   d. none of the above

9. Which of the following does not appear in the IDE's Toolbox list?
   a. `Text`
   b. `Button`
   c. `Label`
   d. `TextBox`

10. After you have dragged a `Button` onto a `Form` in the IDE, you can double-click it to ____________________.
    a. delete it
    b. view its properties
    c. create a method that executes when a user clicks the `Button`
    d. execute a method when a user clicks the `Button`

11. The `button1_Click()` method that is generated by the IDE ____________________.
    a. has a `private` access specifier
    b. is nonstatic
    c. contains parameters between its parentheses
    d. all of the above

12. A(n) ____________________ is generated when a user interacts with a GUI object.
    a. event
    b. occasion
    c. method
    d. error

13. If you create a `Button` named `yesButton`, the name of the method that responds to clicks on it is ____________________.
    a. `button1_Click()`
    b. `yesButton_Method()`
    c. `click_YesButton()`
    d. `yesButton_Click()`

14. Statements allowed in a `Click()` method include ____________________.
    a. variable declarations
    b. arithmetic statements
    c. both of the above
    d. none of the above

15. ______________________ are controls through which a user can enter input data in a GUI application.

    a. `Label`s
    b. `Tag`s
    c. `Ticket`s
    d. `TextBox`es

16. The value in a `TextBox` is ______________________.

    a. an `int`
    b. a `double`
    c. a `string`
    d. It might be any of the above.

17. Which of the following is a legal and conventional name for a `TextBox`?

    a. `Salary TextBox`
    b. `salaryTextBox`
    c. both of the above
    d. none of the above

18. The process of changing a program's internal structure without changing the way the program works is ______________________.

    a. compiling
    b. debugging
    c. code refactoring
    d. systems analysis

19. If you inadvertently create a `Click()` method for a control that should not generate a click event, you can successfully eliminate the method by ______________________.

    a. deleting the method code from the Form1.cs file
    b. eliminating the method from the Events list in the Properties window
    c. adding the method to the Discard window
    d. making the method a comment by placing two forward slashes at the start of each line

20. Of the following, the most significant difference between many console applications and GUI applications is ______________________.

    a. their appearance
    b. their ability to accept input
    c. their ability to perform calculations
    d. their ability to be created using C#

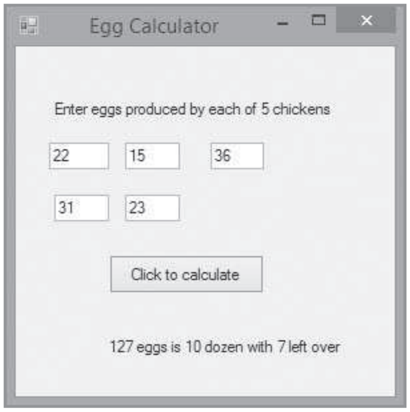
Exercises

Programming Exercises

1. Write a GUI program named **MilesToKilometersGUI** that allows the user to input a distance in miles and output the value in kilometers. There are 1.6 kilometers in a mile.

The exercises in this section should look familiar to you. Each is similar to an exercise in Chapter 2, where you created solutions using console input and output.

2. Write a GUI program named **ProjectedRaisesGUI** that allows a user to enter an employee's salary. Then display, with explanatory text, next year's salary, which reflects a 4 percent increase.
3. Write a program named **CarRentalInteractiveGUI** that prompts a user for days and miles for a car rental and displays the total rental fee computed as $20 per day plus 25 cents per mile.
4. Write a GUI program named **EggsInteractiveGUI** that allows a user to input the number of eggs produced in a month by each of five chickens. Sum the eggs, then display the total in dozens and eggs. For example, a total of 127 eggs is 10 dozen and 7 eggs. Figure 3-20 shows a typical execution.


Egg Calculator

Enter eggs produced by each of 5 chickens

22 15 36

31 23

127 eggs is 10 dozen with 7 left over

Figure 3-20 An `EggsInteractiveGUI` application

5. Write a GUI program named **TestsInteractiveGUI** that allows a user to enter scores for five tests he has taken. Display the average of the test scores to two decimal places.

6. Write a GUI program named **PayrollGUI** that prompts the user for a name, Social Security number, hourly pay rate, and number of hours worked. In an attractive format, display all the input data as well as the following:
   - Gross pay, defined as hourly pay rate times hours worked
   - Federal withholding tax, defined as 15 percent of the gross pay
   - State withholding tax, defined as 5 percent of the gross pay
   - Net pay, defined as gross pay minus taxes
7. Create an enumeration named `Month` that holds values for the months of the year, starting with `JANUARY` equal to `1`. (Recall that an enumeration must be placed within a class but outside of any method.) Write a GUI program named **MonthNamesGUI** that prompts the user for a month integer. Convert the user's entry to a `Month` value, and display it.
8. Pig Latin is a nonsense language. To create a word in pig Latin, you remove the first letter and then add the first letter and *ay* at the end of the word. For example, *dog* becomes *ogday*, and *cat* becomes *atcay*. Write a GUI program named **PigLatinGUI** that allows the user to enter a word and displays the pig Latin version.

Debugging Exercises

1. Each of the following projects in the Chapter.03 folder of your downloadable student files has syntax and/or logical errors. In each case, immediately save a copy of the project folder with a new name that begins with *Fixed* before you open the project in Visual Studio. For example, the project folder for DebugThree1 will become FixedDebugThree1. All the files within the folders have already been named with the *Fixed* prefix, so you do not need to provide new filenames for any of the files in the top-level folder. After naming the new folder, open the project, determine the problems, and fix them.
   a. DebugThree1
   b. DebugThree2
   c. DebugThree3
   d. DebugThree4

Case Problems

1. In Chapter 2, you created a program for the Greenville Idol competition that prompts a user for the number of contestants entered in last year's competition and in this year's competition. The program displays the revenue expected for this year's competition if each contestant pays a $25 entrance fee. The application also displays a statement that indicates whether this year's competition has more contestants than last year's. Now, create an interactive GUI program named **GreenvilleRevenueGUI** that performs all the same tasks.

2. In Chapter 2, you created a program for Marshall's Murals that prompts a user for the number of interior and exterior murals scheduled to be painted during the next month. The program computes the expected revenue for each type of mural when interior murals cost $500 each and exterior murals cost $750 each. The application should display, by mural type, the number of murals ordered, the cost for each type, and a subtotal. The application also displays the total expected revenue and a statement that indicates whether more interior murals are scheduled than exterior ones. Now create an interactive GUI program named **MarshallsRevenueGUI** that performs all the same tasks.