

CHAPTER 3

Writing Windows C# Programs

The C# language has its roots in C++, Visual Basic, and Java. Both C# and VB.Net use the same libraries and compile to the same underlying code. Both are managed languages with garbage collection of unused variable space, and both can be used interchangeably. Both also use classes with method names that are very similar to those in Java, so if you are familiar with Java, you will have no trouble with C#.

Objects in C#

In C#, everything is treated as an object. Objects contain data and have methods that operate on them. For example, strings are now objects. They have methods such as these.

```
Substring  
ToLowerCase  
ToUpperCase  
IndexOf  
Insert
```

Integers, float, and double variables are also objects, and they have methods.

```
string s;  
float x;  
x = 12.3;  
s = x.ToString();
```

Note that conversion from numerical types is done using these methods rather than external functions. If you want to format a number as a particular kind of string, each numeric type has a `Format` method.

Managed Languages and Garbage Collection

C# and VB.Net are both *managed* languages. This has two major implications. First, both are compiled to an intermediate low-level language, and a common language runtime (CLR) is used to execute this compiled code, perhaps compiling it further first. So, not only do C# and VB.Net share the same runtime libraries, they are to a large degree two sides of the same coin and two aspects of the same language system. The differences are that VB7 is more Visual Basic-like and a bit easier for VB programmers to learn and use. C# on the other hand is more C++- and Java-like and may appeal more to programmers already experienced in those languages.

The other major implication is that managed languages are *garbage-collected*. Garbage-collected languages take care of releasing unused memory. (You never have to be concerned with this.) As soon as the garbage-collection system detects that there are no more active references to a variable, array, or object, the memory is released back to the system. Of course, it is still possible to write memory-eating code, but for the most part, you do not have to worry about memory allocation and release problems.

Classes and Namespaces in C#

All C# programs are composed entirely of classes. Visual windows forms are a type of class. You will see that all the program features we'll be writing are composed of classes. Since everything is a class, the number of names of class objects can be overwhelming. They have therefore been grouped into various functional libraries that you must specifically mention in order to use the functions in these libraries.

Under the covers these libraries are each individual DLLs. However, you need only refer to them by their base names, using the using statement, and the functions in that library are available to you.

```
using System;  
using System.Drawing;  
using System.Collections;
```

Logically, each of these libraries represents a different *namespace*. Each namespace is a separate group of class and method names, which the compiler will recognize after you declare that namespace. You can use namespaces that contain identically named classes or methods, but you will only be notified of a conflict if you try to use a class or method that is duplicated in more than one namespace.

The most common namespace is the System namespace, and it is imported by default without your needing to declare it. It contains many of the most fundamental classes and methods that C# uses for access to basic classes such as Application, Array, Console, Exceptions, Objects, and standard objects such as byte, bool, and string. In the simplest C# program we can simply write out a message to the console without ever bringing up a window or form.

```
class Hello {  
    static void Main(string[] args) {  
        Console.WriteLine ("Hello C# World");  
    }  
}
```

This program just writes the text “Hello C# World” to a command (DOS) window. The entry point of any program must be a Main method, and it must be declared as static.

Building a C# Application

Let’s start by creating a simple console application—that is, one without any windows that just runs from the command line. Start the Visual Studio.NET program, and select File | New Project. From the selection box, choose C# Console application, as shown in Figure 3-1.

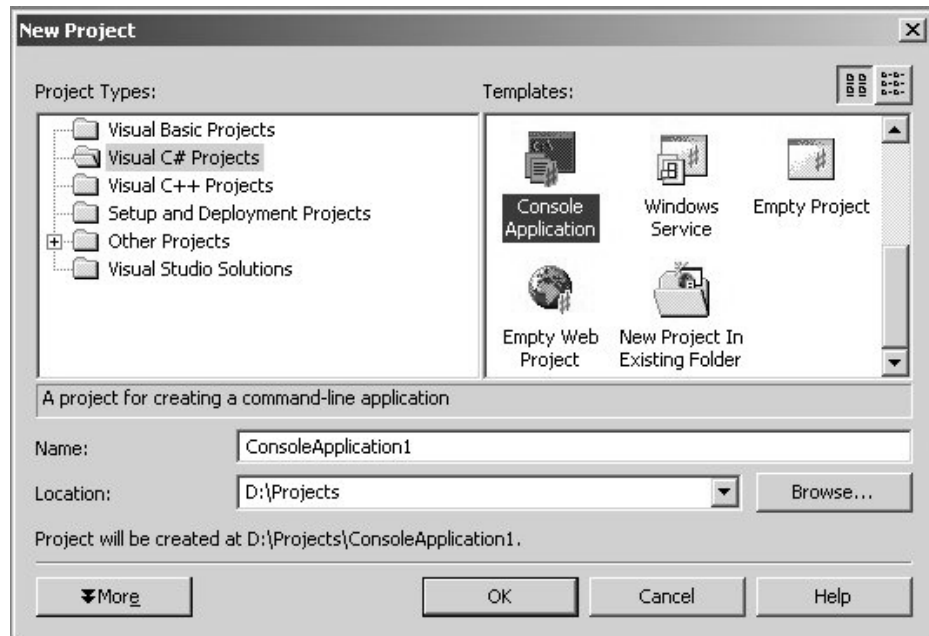


Figure 3-1 The New Project selection window: selecting a console application.

This will bring up a module with Main already filled in. You can type in the rest of the code as follows.

```
Console.WriteLine ("Hello C# World");
```

You can compile this and run it by pressing F5.

When you compile and run the program by pressing F5, a DOS window will appear and print out the message “Hello C# World” and then exit.

The Simplest Window Program in C#

C# makes it very easy to create Windows GUI programs. In fact, you can create most of it using the Windows Designer. To do this, start Visual Studio.NET, select File | New Project, and select C# Windows Application. The default name (and filename) is WindowsApplication1, but you can change this before you close the New dialog box. This brings up a single form project, initially called Form1.cs. You can then use the Toolbox to insert controls, just as you can in Visual Basic.

The Windows Designer for a simple form with one text field and one button is shown in Figure 3-2.

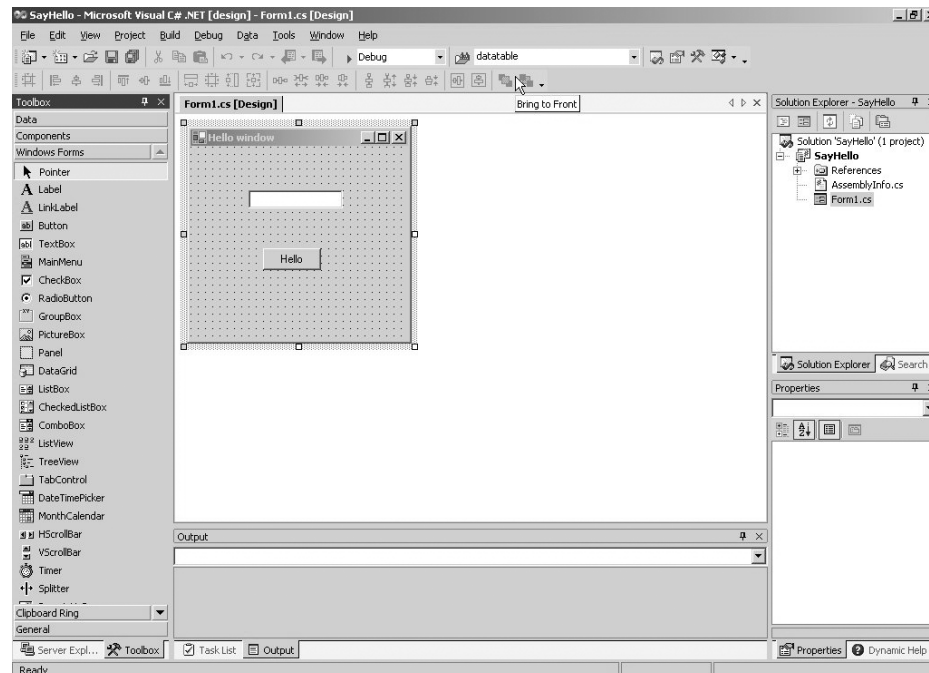


Figure 3-2 The Windows Designer in Visual Studio.NET

You can draw the controls on the form by selecting the `TextBox` from the Toolbox, dragging it onto the form, and then doing the same with the button. Then, to create program code, we need only double-click on the controls. In this simple form, we want to click on the “Hello” button, which copies the text from the text field to the textbox we called `txHi` and clears the text field. So in the designer, we double-click on that button, and this code is automatically generated.

```
private void btHello_Click(object sender, EventArgs e) {  
    txHi.Text = "Hello there";  
}
```

Note that the Click routine passes in a sender object and an event object that you can query for further information. Under the covers, it also connects the event to this method. The running program is shown in Figure 3-3.



Figure 3-3 The SimpleHello form after clicking the Say Hello button

While we only had to write one line of code inside the previous subroutine, it is instructive to see how different the rest of the code is for this program. We first see that several libraries of classes are imported so the program can use them.

```
using System;  
using System.Drawing;  
using System.Collections;  
using System.ComponentModel;  
using System.Windows.Forms;  
using System.Data;
```

Most significant is the `Windows.Forms` library, which is common to all the .NET languages.

The code the designer generates for the controls is illuminating—and it is right out there in the open for you to change if you want. Essentially, each control is declared as a variable and added to a container. Here are the control declarations. Note the event handler added to the `btHello.Click` event.

```
private System.Windows.Forms.TextBox txHi;
private System.Windows.Forms.Button btHello;

private void InitializeComponent()
{
    this.btHello = new System.Windows.Forms.Button();
    this.txHi = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // btHello
    //
    this.btHello.Location =
        new System.Drawing.Point(80, 112);
    this.btHello.Name = "btHello";
    this.btHello.Size = new System.Drawing.Size(64, 24);
    this.btHello.TabIndex = 1;
    this.btHello.Text = "Hello";
    this.btHello.Click +=
        new EventHandler(this.btHello_Click);
    //
    // txHi
    //
    this.txHi.Location =
        new System.Drawing.Point(64, 48);
    this.txHi.Name = "txHi";
    this.txHi.Size = new System.Drawing.Size(104, 20);
    this.txHi.TabIndex = 0;
    this.txHi.Text = "";
    //
    // Form1
    //
    this.AutoScaleBaseSize =
        new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(240, 213);
    this.Controls.AddRange(
        new System.Windows.Forms.Control[] {
            this.btHello,
            this.txHi } );
    this.Name = "Form1";
    this.Text = "Hello window";
    this.ResumeLayout(false);
}
}
```

If you change this code manually instead of using the property page, the window designer may not work anymore. We'll look more at the power of this system after we discuss objects and classes in the next chapter.

Windows Controls

All of the basic Windows controls work in much the same way as the `TextBox` and `Button` we have used so far. Many of the more common ones are shown in the Windows Controls program in Figure 3-4.



Figure 3-4 A selection of basic Windows controls

Each of these controls has properties such as `Name`, `Text`, `Font`, `ForeColor`, and `BorderStyle` that you can change most conveniently using the properties window shown at the right of Figure 3-2. You can also change these properties in your program code as well. The Windows Form class that the designer generates always creates a `Form1` constructor that calls an `InitializeComponent` method like the preceding one. Once that method has been called, the rest of the controls have been created, and you can change their properties in code. Generally, we will create a private `init()` method that is called right after the `InitializeComponent` method, in which we add any such additional initialization code.

Labels

A label is a field on the window form that simply displays text. Usually programmers use this to label the purpose of text boxes next to them. You can't click on a label or tab to it so it obtains the focus. However, if you want, you can change the major properties in Table 3-1 either in the designer or at runtime.

Table 3-1 Properties for the Label Control

Property	Value
<i>Name</i>	At design time only
BackColor	A Color object
BorderStyle	None, FixedSingle, or Fixed3D
Enabled	True or false. If false, grayed out.
Font	Set to a new Font object
ForeColor	A Color object
Image	An image to be displayed within the label
ImageAlign	Where in the label to place the image
Text	Text of the label
Visible	True or false

TextBox

The TextBox is a single line or multiline editable control. You can set or get the contents of that box using its Text property.

```
TextBox tbox = new TextBox();  
tbox.Text = "Hello there";
```

In addition to the properties in Table 3-1, the TextBox also supports the properties in Table 3-2.

CheckBox

A CheckBox can be either checked or not, depending on the value of the Checked property. You can set or interrogate this property in code as well as in the designer. You can create an event handler to catch the event when the box is checked or unchecked by double-clicking on the checkbox in the design mode.

CheckBoxes have an Appearance property that can be set to *Appearance.Normal* or *Appearance.Button*. When the appearance is set to the Button value,

Table 3-2 TextBox Properties

Property	Value
Lines	An array of strings, one per line
Locked	If true, you can't type into the text box
Multiline	True or false
ReadOnly	Same as locked. If true, you can still select the text and copy it, or set values from within code.
WordWrap	True or false

the control acts like a toggle button that stays depressed when you click on it and becomes raised when you click on it again. All the properties in Table 3-1 apply as well.

Buttons

A Button is usually used to send a command to a program. When you click on it, it causes an event that you usually catch with an event handler. Like the CheckBox, you create this event handler by double-clicking on the button in the designer. All of the properties in Table 3-1 can be used as well.

Buttons are also frequently shown with images on them. You can set the button image in the designer or at runtime. The images can be in bmp, gif, jpeg, or icon files.

Radio Buttons

Radio buttons or option buttons are round buttons that can be selected by clicking on them. Only one of a group of radio buttons can be selected at a time. If there is more than one group of radio buttons on a window form, you should put each set of buttons inside a Group box as we did in the program in Figure 3-4. As with checkboxes and buttons, you can attach events to clicking on these buttons by double-clicking on them in the designer. Radio buttons do not always have events associated with them. Instead, programmers check the Checked property of radio buttons when some other event, like an OK button click, occurs.

ListBoxes and ComboBoxes

Both ListBoxes and ComboBoxes contain an Items array of the elements in that list. A ComboBox is a single-line drop-down that programmers use to save space when selections are changed less frequently. ListBoxes allow you to set properties that allow multiple selections, but ComboBoxes do not. Some of their properties include those in Table 3-3.

Table 3-3 The ListBox and ComboBox Properties

Property	Value
Items	A collection of items in the list
MultiColumn	If true, the ColumnWidth property describes the width of each column. (Does not apply to ComboBox.)
SelectionMode	One, MultiSimple, or MultiExtended. If set to MultiSimple, you can select or deselect multiple items with a mouse click. If set to MultiExtended, you can select groups of adjacent items with a mouse. (Does not apply to ComboBox.)
SelectedIndex	Index of selected item
SelectedIndices	Returns collection of selections when ListBox selection mode is multiple.
SelectedItem	Returns the item selected

The Items Collection

You use the Items collection in the ListBox and ComboBox to add and remove elements in the displayed list. It is essentially an ArrayList, as we discuss in Chapter 7. The basic methods are shown in Table 3-4.

If you set a ListBox to a multiple selection mode, you can obtain a collection of the selected items or the selected indexes by

```
ListBox.SelectedIndexCollection it =
    new ListBox.SelectedIndexCollection (lsCommands);
ListBox.SelectedObjectCollection so =
    new ListBox.SelectedObjectCollection (lsCommands);
```

where *lsCommands* is the ListBox name.

Table 3-4 Methods for the Items Collection

Method	Value
Add	Add object to list
Count	Number in list
<i>Item</i> [i]	Element in collection
RemoveAt(i)	Remove element i

Menus

You add a menu to a window by adding a MainMenu control to the window form. Then you can select the menu control and edit its drop-down names and new main item entries, as shown in Figure 3-5.

**Figure 3-5** Adding a menu to a form

As with other clickable controls, double-clicking on one in the designer creates an event whose code you can fill in.

ToolTips

A Tooltip is a box that appears when your mouse pointer hovers over a control in a window. This feature is activated by adding an (invisible) Tooltip control to the form and then adding specific Tooltip control and text combinations to the control. In our example in Figure 3-4, we add ToolTips text to the button and ListBox using the *tips* control we have added to the window.

```
tips.SetToolTip (btPush, "Press to add text to list box");
tips.SetToolTip (lsCommands, "Click to copy to text box");
```

This is illustrated in Figure 3-6.

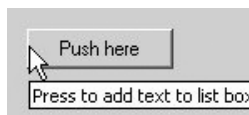


Figure 3-6 A Tooltip over a button

We discuss how to use the DataGrid and TreeList in Chapters 14 and 15, and Toolbar is discussed in Chapters 28 and 29.

The Windows Controls Program

The Windows Controls program, shown in Figure 3-4, controls changes in the text in the label.

- Font size is set from the combo box.
- Font color is set from the radio buttons.
- Boldface is set from the check box.

For the check box, we create a new font that is either lightface or boldface, depending on the state of the check box.

```
private void ckBold_CheckedChanged(object sender, EventArgs e) {
    if (ckBold.Checked) {
        lbText.Font = new Font ("Arial",
                               fontSize, FontStyle.Bold );
    }
    else {
        lbText.Font = new Font ("Arial", fontSize);
    }
}
```

When we create the form, we add the list of font sizes to the combo box.

```
private void init() {
    fontSize = 12;
    cbFont.Items.Add ("8");
    cbFont.Items.Add ("10");
    cbFont.Items.Add ("12");
    cbFont.Items.Add ("14");
    cbFont.Items.Add ("18");
    lbText.Text = "Greetings";
    tips.SetToolTip (btPush, "Press to add text to list box");
    tips.SetToolTip (lsCommands, "Click to copy to text box");
}
```

When someone clicks on a font size in the combo box, we convert that text to a number and create a font of that size. Note that we just call the check box changing code so we don't have to duplicate anything.

```
private void cbFont_SelectedIndexChanged(
    object sender, EventArgs e) {
    fontSize= Convert.ToInt16 (cbFont.SelectedItem );
    ckBold_CheckedChanged(null, null);
}
```

For each radio button, we click on it and insert color-changing code.

```
private void opGreen_CheckedChanged(object sender, EventArgs e) {
    lbText.ForeColor =Color.Green;
}

private void opRed_CheckedChanged(object sender, EventArgs e) {
    lbText.ForeColor =Color.Red ;
}

private void opBlack_CheckedChanged(object sender, EventArgs e) {
    lbText.ForeColor =Color.Black ;
}
```

When you click on the ListBox, it copies that text into the text box by getting the selected item as an object and converting it to a string.

```
private void lsCommands_SelectedIndexChanged(
    object sender, EventArgs e) {
    textBox.Text = lsCommands.SelectedItem.ToString () ;
}
```

Finally, when you click on the File | Exit menu item, it closes the form and, hence, the program.

```
private void menuItem2_Click(object sender, EventArgs e) {
    this.Close ();
}
```

Summary

Now that we've seen the basics of how to write programs in C#, we are ready to talk more about objects and OO programming in the chapters that follow.

Programs on the CD-ROM

Console Hello	\IntroCSharp\Hello
Windows Hello	\IntroCSharp\SayHello
Windows Controls	\IntroCSharp\WinControls

