



C# for Visual Basic 6 Developers

This appendix presents a brief introduction to the C# language, specifically aimed at those developers whose experience up until now has been mostly or entirely with Visual Basic 6.

Note that throughout this appendix, references to Visual Basic indicate Visual Basic 6. On those few occasions when we mention Visual Basic 2005, we will explicitly name it as such.

C# and Visual Basic are very different languages, both in their syntactical style and in the fundamental concepts that they are based on. This means that Visual Basic developers will find they have quite a steep learning curve to climb in order to become familiar with C#, even at a basic level. The aim of this appendix is to make that learning curve easier by providing a tutorial to C#. This tutorial presumes knowledge of Visual Basic, and focuses on the main conceptual differences between the two languages. The approach in this appendix will be to compare Visual Basic solutions with C# solutions programmatically.

This does mean that coverage of the C# language will be restricted to a basic level. This appendix does not cover the more advanced features of the language (which is covered in Part I of this book). The emphasis is on showing you the different methodologies involved in writing code using the C# language.

Differences Between C# and Visual Basic

Beyond the obvious syntactical differences between these two languages, there are really two key concepts with which you have to familiarize yourself to progress from Visual Basic to C#:

- 1. The concept of the complete flow of execution of a program from start to finish:** Visual Basic hides this aspect of programs from you, so that the only elements of a Visual Basic program you code are the event handlers and any methods in class modules. C# makes the complete program available to you as source code. The reason for this has to do with the fact that C# can be seen, philosophically, as next-generation C++. The roots of C++ go back to the 1960s and predate windowed user interfaces and sophisticated operating systems. C++ evolved as a low-level, close-to-the-machine, all-purpose language. To write GUI applications with C++ meant that you had to invoke the system calls to create and interact with the windowed forms. C# has been designed to build on this tradition while simplifying and modernizing C++, to combine the low-level performance benefits of C++ with the ease of coding in Visual Basic. Visual Basic, on the other hand, is designed specifically for rapid application development of Windows GUI applications. For this reason, in Visual Basic all the GUI boilerplate code is hidden, and all the Visual Basic programmer implements are the event handlers. In C# on the other hand, this boilerplate code is exposed as part of your source code.
- 2. Classes and inheritance:** C# is a genuine object-oriented language, unlike Visual Basic, requiring all code to be a part of a class. It also includes extensive support for implementation inheritance. Indeed, most well-designed C# programs will be very much designed around this form of inheritance, which is completely absent in Visual Basic.

The bulk of this appendix is devoted to developing two sample applications. The first example is a simple form, written in both Visual Basic and C#, that asks the user for a number and displays the square root and sign of the number. By comparing the Visual Basic and C# versions of the sample in some detail, you will learn basic C# syntax and also understand the concepts behind the flow of execution of a program.

Next, you are presented with a Visual Basic class module that stores information about employees and its C# equivalent. This example demonstrates the real power of C# by showing you the shortcomings of Visual Basic.

The appendix finishes with a short tour of some of the remaining differences between Visual Basic and C#.

Before you start, however, a couple of concepts require some clarification: classes, compilation, and the .NET base classes.

Classes

Throughout this appendix, you use C# classes quite extensively. C# classes represent precisely defined objects (see Chapter 3, “Objects and Types,” and Appendix A, “Principles of Object-Oriented Programming”). However, for the purposes here, you are better off thinking of them as the C# equivalent to Visual Basic class modules, because they are quite similar entities: Like a Visual Basic class module, a C# class implements properties and methods, and contains member variables. Like a Visual Basic class module, you can create objects of a given C# class (class instances) using the operator `new`. Beyond these similarities, however, there are many differences. For example, a Visual Basic class module is really a COM class. C# classes, by contrast, are always integrated into the .NET Framework. C# classes are also more lightweight than their Visual Basic or COM counterparts, in the sense that they are designed for performance and give a smaller performance hit when instantiated. However, for the most part these differences will not affect the discussion of the C# language here.

Compilation

As you know, computers never directly execute code in any high-level language, whether it is Visual Basic, C++, C, or any other language. Instead, all source code is first translated into native executable code, a process usually known as *compilation*. When you are debugging, Visual Basic offers the option of just running the code (meaning that each line of Visual Basic code is *interpreted* as the computer executes that line), or of doing a full compile (meaning that the entire program is first translated into executable code, and then execution starts). Performing a full compile first means that any syntax errors are discovered by the compiler before the program starts running. It also yields much higher performance when running and is therefore the only option permitted in C#.

In C#, compilation is done in two stages. First, code is compiled into the Microsoft intermediate language (IL), a process commonly referred to as compilation. Then the code is converted into native executable code at runtime. This is not the same as interpreting. Entire portions of code are converted from IL to assembly language at a time and the resultant native executable is then stored so it doesn't need to be recompiled the next time that portion of code is executed. Combined with various optimizations, Microsoft believes that this will ultimately lead to code that is actually faster to execute than with the previous system of compiling direct from source code to native executable. Although the existence of IL is something that you need to bear in mind, it won't affect the discussion in this appendix, because it doesn't really affect C# language syntax.

The .NET Base Classes

Visual Basic has a large number of associated functions, such as the conversion functions `CInt`, `CStr`, and so on, the file system functions, date-time functions, and many more. Visual Basic also relies on the presence of ActiveX controls to provide the standard controls that you put on your form, such as list boxes, buttons, text boxes, and so on.

C# also relies on extensive support for these sorts of areas. However, in the case of C#, this support comes from a very large set of classes known as the .NET base classes. These classes provide support for almost every aspect of Windows development. There are classes that represent all the standard controls, classes that perform conversions, classes that perform date-time and file system access, classes that access the Internet, and many more. This appendix won't go into the .NET base class library in detail, but it will frequently refer to it. Indeed, C# is so well integrated with the .NET base classes that many C# keywords just provide wrappers around particular base classes. In particular, all the basic C# data types that are used to represent integers, floating-point numbers, strings, and so on are actually base classes.

In this respect, there is a marked distinction between Visual Basic and C#; the Visual Basic system functions are specific to Visual Basic, whereas the respective functionality of C# is provided by the .NET base classes, which are accessible to any .NET-aware language.

Conventions

This appendix frequently compares code in C# and Visual Basic. To make it easier to identify code in these two languages, C# code is presented in this format:

```
// C# code that we have already seen
// C# code that we want to draw attention to or which is new
```

However, all Visual Basic code is presented in this format:

```
' Visual Basic code is presented with a white background
```

Example: The Square Root Form

In this section, you examine a simple application called `SquareRoot`, which has been developed in both Visual Basic and C#. The application is a simple dialog box that asks the user to type in a number, and then, when the user clicks a button, displays the sign and square root of that number. If the number is negative, the square root needs to be displayed as a complex number, which means taking the square root of the number and adding “i” after it. Figure B-1 shows the C# version of this example. The Visual Basic version is pretty much identical in appearance except that it has a standard Visual Basic icon in place of the .NET Windows Forms icon in the top-left corner.

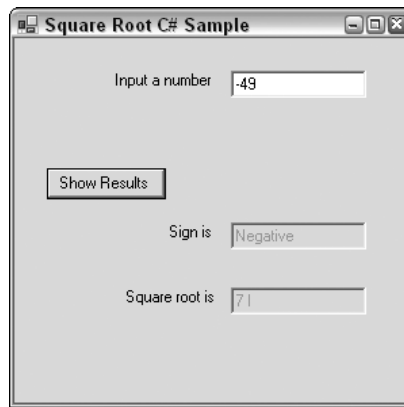


Figure B-1

SquareRoot Visual Basic Version

To get this application working in Visual Basic, you simply need to add an event handler for the event of clicking the button. The button has been given the name `cmdShowResults`, and the `TextBox` controls have the intuitive names of `txtNumber`, `txtSign`, and `txtResult`. With these names, the event handler for the button looks like this:

```
Option Explicit
Private Sub cmdShowResults_Click()
    Dim sngNumberInput As Single
    sngNumberInput = CSng(Me.txtNumber.Text)
    If (sngNumberInput < 0) Then
        Me.txtSign.Text = "Negative"
        Me.txtResult.Text = CStr(Sqr(-sngNumberInput)) & " i"
    ElseIf (sngNumberInput = 0) Then
        txtSign.Text = "Zero"
        txtResult.Text = "0"
    Else
        Me.txtSign.Text = "Positive"
```

```

        Me.txtResult.Text = CStr(Sqr(sngNumberInput))
    End If
End Sub

```

That is all the Visual Basic code that you need to write.

SquareRoot C# Version

In C# you also need to write an event handler for the event of the button being clicked. You keep the same names for the button and the text boxes, but in C# the code looks like this:

```

// Event handler for user clicking Show Results button.
// Displays square root and sign of number
private void OnClickShowResults(object sender, System.EventArgs e)
{
    float NumberInput = float.Parse(this.txtNumber.Text);
    if (NumberInput < 0)
    {
        this.txtSign.Text = "Negative";
        this.txtResult.Text = Math.Sqrt(-NumberInput).ToString() + " i";
    }
    else if (NumberInput == 0)
    {
        txtSign.Text = "Zero";
        txtResult.Text = "0";
    }
    else
    {
        this.txtSign.Text = "Positive";
        this.txtResult.Text = Math.Sqrt(NumberInput).ToString ();
    }
}

```

Comparing these two code samples, you can see the similarity in the code structure, and even without any knowledge of C#, you can probably get some idea of what is going on. It is also evident that a huge number of differences exist in the syntax between the two languages. Over the next couple of pages, you compare these samples to see what you can learn about C# syntax in the process. In the process, you will also uncover some of the differences between the basic methodologies of C# and Visual Basic.

Basic Syntax

This section examines the two `SquareRoot` programs to see what they teach you about C# syntax.

C# requires all variables to be declared

If you start with the first line of Visual Basic code, you encounter the `Option Explicit` declaration. This statement has no equivalent in C#. The reason is that in C# variables must always be declared before they are used. It's as if C# always runs with `Option Explicit` turned on and doesn't allow you to switch it off. Hence there's no need to declare `Option Explicit`. The point of this restriction is that C# has been very carefully designed to make it difficult for you to accidentally introduce bugs into your

Appendix B

code. Standard advice in Visual Basic is always to use `Option Explicit` because it prevents hard-to-find bugs caused by misspelled variable names. Generally, you will find that C# doesn't allow you to do things that have a high risk of causing bugs.

Comments

Because commenting code is always important, the next thing you do in both samples (or the first thing in the C# sample!) is add a comment:

```
// Event handler for user clicking Show Results button.
// Displays square root and sign of number
private void OnClickShowResults(object sender, System.EventArgs e)
{
```

In Visual Basic you use an apostrophe to denote the start of a comment, and the comment lasts until the end of the line. The C# comments in the code work the same way, except they start with two forward slashes: `//`. Just as for Visual Basic comments, you can use an entire line for a comment or append a comment to the end of a line:

```
// This code works out the results

int Result = 10*Input; // get result
```

However, C# is more flexible in its comments because it allows two other ways of indicating comments, and each has a slightly different effect. A comment may also be delimited by the sequences `/*` and `*/`. In other words, if the compiler sees an opening `/*` sequence, it assumes all the following text is a comment until it sees a closing `*/` sequence. This allows you to have long, multiple-line comments:

```
/* this text is a really long
long
long
long
comment */
```

Short comments within a line are very useful if you just want to temporarily swap something in a line while you are debugging:

```
x = /*20*/ 15;
```

The third option is very similar to the first option. However, now you use three forward slashes:

```
/// <summary>
/// Event handler for user clicking Show Results button.
/// Displays square root and sign of number
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void OnClickShowResults(object sender, System.EventArgs e)
```

If you use three forward slashes instead of two, the comment still lasts until the end of that line. However, this comment now has an additional effect: The C# compiler is actually capable of using comments that start with three slashes to automatically generate documentation for your source code as a separate XML

file. That's why the comment text in the previous example appears to have a fairly formal structure—it is ready to be placed into an XML file. We won't go into the details of this process here (it's covered in Chapter 2, "C# Basics"). We will, however, say that this means that commenting each method of your code allows you to have complete documentation automatically generated and updated when you modify your code. The compiler even checks that the documentation matches the method signatures.

Statement separation and grouping

The most visible difference between the preceding C# and Visual Basic code is the presence of semicolons and curly braces in the C# code. Although this can make C# code look daunting, the principle is actually very simple. Visual Basic uses carriage returns to indicate the ends of statements, whereas C# uses semicolons for the same purpose. In fact, the C# compiler completely ignores all excess whitespace, including carriage returns. These features of C# syntax can combine to give you a lot of freedom in laying out your code. For example, the following snippet (reformatted from part of the preceding sample) is also perfectly valid C# code:

```
        this.txtSign.Text =  
  
        "Negative"; this.txtResult.Text = Math.Sqrt  
        (-NumberInput) + " i";
```

Obviously, if you want other people to be able to read your code, you will opt for the first coding style, and Visual Studio 2005 will automatically lay out your code in that style.

The braces are used to group statements together into what are known as *block statements* (or *compound statements*). This is a concept that doesn't exist in Visual Basic. In C#, you can group any statements by placing braces around them. The group is now regarded as one single block statement and can be used anywhere in C# where a single statement is expected.

Block statements are used a lot in C#. For example, in the previous C# code there is no explicit indication of the end of the `OnClickShowResults()` method (C# has methods, written in text with the `()` appended, whereas Visual Basic has functions and subs). Visual Basic needs an `End Sub` statement at the end of any sub because a sub can contain as many statements as you want, so a specific marker is the only way that Visual Basic knows where you intend the sub to end. C# works differently. In C# a method is formed from exactly one compound statement. Because of this, the method ends with the closing curly brace matching the opening one at the start of the method.

You find this a lot in C#: where Visual Basic uses some keyword to mark the end of a block of code, C# simply organizes the block into one compound statement. The `if` statement in the previous samples illustrates the same point. In Visual Basic, you need an `End If` statement to mark where the `If` block ends, if the `If` statement is more than one line. In C#, the rule is simply that an `if` clause always contains exactly one statement, and the `else` clause also contains one statement. If you want to put more than one statement into either clause, as is the case in the previous example, you use a compound statement.

Capitalization

One other point you may notice about the syntax is that all the keywords—`if`, `else`, `int`, and so on—in the C# code are in lowercase.

Remember that unlike Visual Basic, C# is case-sensitive.

In C#, if you write `IF` instead of `if`, the compiler won't understand your code. One advantage of being case-sensitive, however, is that you can have two variables whose names differ only in case, such as `Name` and `name`. You'll encounter this in the second sample application later in this appendix.

In general, you'll find that all C# keywords are entirely lowercase.

Methods

Compare the syntax that Visual Basic and C# use to declare the part of the code that handles the event:

```
Private Sub cmdShowResults_Click()
```

and:

```
private void OnClickShowResults(object sender, System.EventArgs e)
```

The Visual Basic version declares a sub, whereas the C# version declares a method. In Visual Basic, code is traditionally grouped into subs and functions, with the concept of a procedure being either. Additionally, Visual Basic class objects have what are known as methods, which for all practical purposes means the same thing as procedures except that they are part of a class module.

C#, by contrast, only has methods (that's because everything in C# is part of a class). C# does not support the concept of functions and subroutines; these terms don't even exist in the C# language specification. In Visual Basic, the only real difference between a sub and a function is that a sub never returns a value. In C#, if a method does not need to return a value, it is declared as returning `void` (as the `OnClickShowResults()` method illustrated earlier).

The syntax for declaring a method is similar in the two languages, at least to the extent that the parameters follow the method name in brackets. Note, however, that whereas in Visual Basic declaring a sub is indicated with the word `Sub`, there is no corresponding word in the C# version. In C#, the return type (`void` in this case), followed by the method name, followed by the opening bracket, is sufficient to tell the compiler that you are declaring a method, because no other construct in C# has this syntax (arrays in C# are marked with square rather than round brackets so there is no risk of confusion with arrays).

Like the Visual Basic `Sub`, the C# method declaration in the preceding code is preceded by the keyword `private`. This has roughly the same meaning as in Visual Basic — it prevents outside code from being able to see the method. (You'll examine the notion of outside code shortly.)

There are two other differences to remark on about the method declaration: the C# version takes two parameters, and it has a different name from the Visual Basic event handler.

The name of the event handler in Visual Basic is supplied for you by the Visual Basic IDE. The reason that Visual Basic knows that the `Sub` is the event handler for a button click is because of the name, `cmdShowResults_Click`. If you renamed the sub, it wouldn't get called when you click the button. However, C# doesn't use the name in this way. In C#, there is some other code that tells the compiler

which method is the event handler for this event. That means you can give the handler whatever name you want. However, something starting with `On` for an event handler is traditional, and in C#, common practice is to name methods (and for that matter most other items) using *Pascal* casing, which means that words are joined together with their first letters capitalized. Using underscores in names in C# is not recommended, and the example uses a name in accordance with these guidelines:

```
OnClickShowResults()
```

Now for the parameters. You don't need to worry about the details of these parameters in this appendix. All you need to know for now is that all event handlers in C# are required to take two parameters similar to these, and these parameters can provide some useful extra information about the event in question (for example, for a mouse move event the parameters might indicate the location of the mouse pointer).

Variables

The `SquareRoot` sample can tell you quite a lot about the differences between the variable declarations in C# and Visual Basic. In the Visual Basic version, you declare a floating-point number and set up its value as follows:

```
Dim sngNumberInput As Single sngNumberInput = CSng(Me.txtNumber.Text)
```

The C# version looks like this:

```
float NumberInput = float.Parse(this.txtNumber.Text);
```

As you'd expect, the data types in C# aren't exactly the same as in Visual Basic. `float` is the C# equivalent to `Single`. It's probably easier for you to understand what's going on if you split up the C# version into two lines. The following C# code has exactly the same effect as the preceding line:

```
float NumberInput; NumberInput = float.Parse(this.txtNumber.Text);
```

Now you can compare the declaration and initialization of the variable separately.

Declarations

The obvious syntactical difference between C# and VB6, as far as variable declarations are concerned, is that in C#, the data type precedes rather than follows the name of the variable, with no other keywords. This gives C# declarations a more compact format than their Visual Basic counterparts.

You'll notice that this idea of a declaration consisting only of a type followed by a name is used elsewhere too. Look again at the method declaration in C#:

```
private void OnClickShowResults(object sender, System.EventArgs e)
```

The type (`void`) precedes the name of the method, with no other keywords to indicate what you are declaring—that's obvious from the context. The same is also true for the parameters. The types of the parameters are `object` and `System.EventArgs`. The `object` type in C# plays a similar role to `Object` in Visual Basic—it indicates something for which you are choosing not to specify its type. However, the C# `object` type is much more powerful than the Visual Basic `Object`. In C#, `object` also replaces VB6's `Variant` data type. You'll look at `object` later on. `System.EventArgs` is not covered in any detail in this appendix. It's a .NET base class, and it has no equivalent in Visual Basic.

Appendix B

In the case of variables, the declaration syntax used in C# allows you to combine the declaration with the setting of an initial value for the variable. In the code sample `sngNumberInput` is initialized to what looks like a complicated expression, which is explained shortly. To take two simpler examples:

```
int X = 10; // int is similar to Long in Visual Basic
string Message = "Hello World"; // string is similar to String in Visual Basic
```

While we are on the subject, you should know a couple of other points about variables.

No suffixes in C#

Visual Basic allows you to attach suffixes to variables to indicate their data types, with `$` for `String`, `%` for `Int`, and `&` for `Long`:

```
Dim Message$ ' will be a string
```

This syntax is not supported in C#. Variable names may contain only letters, numbers, and the underscore character, and you must always indicate the data type.

No default values for local variables

In the Visual Basic code sample, the variable `sngNumberInput` is assigned the default value of 0 when it is declared. This is actually a waste of processor time because you immediately assign it a new value in the next statement. C# is a little more performance-conscious and does not bother putting any default values in local variables when they are declared. Instead, it requires that you always initialize such variables yourself before you use them. The C# compiler will raise a compilation error if you attempt to read the value in any local variable before you have set it.

Assigning values to variables

Assigning values to variables in C# is done with the same syntax as in VB6. You simply put an `=` sign after the variable name, followed by the value you are assigning to it. However, one point to watch out for is that this is the only syntax used in C#. In some cases in Visual Basic, you use `Let`, whereas for objects, Visual Basic always uses the `Set` keyword:

```
Set MyListBox = new ListBox
```

C# does not use a separate syntax for assigning to object references. The C# equivalent of the preceding is:

```
MyListBox = new ListBox();
```

Remember that in C#, variables are always assigned using the syntax

```
<VariableName>=<Expression>;
```

Classes

Here you look at what's going on in the expression used to initialize the variable `sngNumberInput` in the `SquareRoot` sample. The C# and Visual Basic examples are both doing exactly the same thing: They grab the text from the `txtNumber` `TextBox` control; but the syntax is different:

```
sngNumberInput = CSng (Me.txtNumber.Text)
```

and:

```
float NumberInput = float.Parse(this.txtNumber.Text);
```

Getting the value out of the `TextBox` controls is quite similar in both cases. The only difference here is the syntax: Visual Basic uses the keyword `Me` whereas C# uses the keyword `this`, which has exactly the same meaning (in fact, in C# you can omit `this` if you want, just as you can omit `Me` in Visual Basic). In C# you could equally well have written:

```
float NumberInput = float.Parse(txtNumber.Text);
```

The more interesting part is how the string retrieved from the `TextBox` control is converted to a `float` (or `single`), because this illustrates a fundamental point of the C# language, which was hinted at earlier: *everything in C# is part of a class.*

In Visual Basic, the conversion is carried out by a function, `CSng`. However, C# does not have functions of the Visual Basic variety. C# is totally object-oriented and will only allow you to declare methods that are part of a class.

In C#, the conversion from `string` to `float` is carried out by the `Parse()` method. However, because `Parse()` is part of a class, it has to be preceded by the name of the class. The class against which you need to call the `Parse()` method is `float`. Yes, I did say that right. Up until now `float` has been treated as simply being the C# equivalent to the Visual Basic `Single` type. However, it is actually a class as well. In C#, all data types are classes as well, which means even things like `int`, `float`, and `string` have methods and properties that you can call (although you should know that `int` and `float` are special types of class known in C# as `structs`. The difference is not important for the code here but it is explained later).

If you are looking really carefully at the code, you might notice a slight apparent problem with the analogy with Visual Basic class modules. In Visual Basic, you call methods by specifying the name of an object, not the name of the class module, but you've called `Parse` by specifying the name of the class, `float`, instead of the name of an object. `Parse()` is a special type of method known as a static method. It has no equivalent in Visual Basic, and a static method can be called without creating an instance of a class. Hence you specify the class name, `float`, rather than a variable name. By the way, `static` does not have the same meaning in C# as it does in Visual Basic. There is no equivalent in C# to the Visual Basic static variables—there is no need for these in the C# object-oriented programming methodology, because you use C# fields in their stead.

Also, to be strictly accurate, it should be pointed out that the name of the class is actually `System.Single`, not `float`. `System.Single` is one of the .NET base classes, and C# uses the keyword `float` to indicate this class.

If Statements

Next you come to the main part of the event handler—the “If” statement. Recall that the Visual Basic version looks like this:

```
If (sngNumberInput < 0) Then
    Me.txtSign.Text = "Negative"
    Me.txtResult.Text = CStr(Sqr(-sngNumberInput)) & " i"
```

Appendix B

```
ElseIf (sngNumberInput = 0) Then
    txtSign.Text = "Zero"
    txtResult.Text = "0"
Else Me.txtSign.Text = "Positive"
    Me.txtResult.Text = CStr(Sqr(sngNumberInput))
End If
```

In the C# version, it looks like this:

```
if (NumberInput < 0)
{
    this.txtSign.Text = "Negative";
    this.txtResult.Text = Math.Sqrt(-NumberInput).ToString() + " i";
}
else if (NumberInput == 0)
{
    txtSign.Text = "Zero";
    txtResult.Text = "0";
}
else
{
    this.txtSign.Text = "Positive";
    this.txtResult.Text = Math.Sqrt(NumberInput).ToString();
}
```

You have already learned the biggest syntactical difference here: that each part of the `if` statement in C# must be a single statement; hence, if you have to conditionally execute more than one statement, you must combine them into a single block statement. In C#, if there is only one statement to be conditionally executed, you don't need to form a block statement. For example, if you want to skip setting the text in the `txtSign` `TextBox` control in the preceding code, you can write:

```
if (NumberInput < 0)
    this.txtResult.Text = Math.Sqrt(-NumberInput) + " i";
else if (NumberInput == 0)
    txtSign.Text = "Zero";
else
    this.txtResult.Text = Math.Sqrt(NumberInput).ToString();
```

Other syntax differences worth noting include the following: In C#, the parentheses around the condition to be tested in an `if` statement are compulsory. In Visual Basic you could have written:

```
If sngNumberInput < 0 Then
```

Trying the same trick in C# would result in a compilation error. In general, C# is much pickier about the syntax it expects than Visual Basic. Also, notice that when you test whether `NumberInput` is zero, you use two equals signs in succession for the comparison:

```
else if (NumberInput == 0)
```

In Visual Basic, the symbol `=` has two purposes: it is used for assigning values to variables and for comparing values. C# formally recognizes these as two very different types of operation, and so uses different symbols: `=` for assignment and `==` for comparison.

There is one other important difference that you should be aware of, because this one can easily catch you by surprise when making the transition from Visual Basic to C#: `else if` is two words in C# whereas it is one word in Visual Basic: `ElseIf`.

Calculating square roots: Another class method

Given the earlier comments about everything in C# being a member of the class, you won't be surprised to learn that C#'s equivalent of the Visual Basic `Sqr` function, which calculates square roots, is also a method that is a member of a class. In this case, it is the `Sqrt()` method, which is a static member of another .NET base class, `System.Math`, which you can abbreviate to just `Math` in your code.

Note also that when dealing with the condition of the number input being exactly zero, you don't specify the `this` keyword in the C# code:

```
txtSign.Text = "Zero";  
txtResult.Text = "0";
```

In the corresponding Visual Basic code, you don't specify `Me` explicitly either. In C#, just as in Visual Basic, you don't have to explicitly specify `this` (`Me`) unless the context is unclear.

Strings

To display the square root of a negative number, you use string processing:

```
this.txtResult.Text = Math.Sqrt(-NumberInput).ToString() + " i";
```

Note that in C# concatenation of strings is done using the symbol `+` rather than `&`. Note also that you convert from a `float` to a `String` by calling a method on the `float` object. The method is called `ToString()`, and this method is not static, so it is called using the same syntax as in Visual Basic when you call methods on objects: by prefixing the name of the method with the name of the variable that represents the object, followed by a dot. One useful thing to remember about C# is that every object (and hence every variable) inherits the `ToString()` functionality and can provide its own custom method.

Extra Code in C#

The comparison of the event handler routines in C# and Visual Basic is now complete. In the process, you've learned a lot about the syntactical differences between the languages. In fact, you have learned most of the basic syntax that C# uses to piece statements together. You have also had your first brush with the fact that everything in C# is a class. However, if you have downloaded the sample code for these samples from the Wrox Press Web site (www.wrox.com) and looked at the code, you will have almost certainly noticed that we have carefully avoided any discussion of the most obvious difference between the samples: there is a lot more code in the C# sample than simply an event handler. For the Visual Basic version of the `SquareRoot` sample, the code for the event handler presented here represents the total of all the source code in the project that is visible to you. However, in the C# version of the project, this event handler is just one method in a large source code file that contains a lot more code.

The reason why there is so much additional code in the C# project has to do with the fact that the Visual Basic IDE hides a lot of what's going on in your program from you. In Visual Basic, all you needed to write was the event handler, but in fact the sample is doing a lot more. It needs to start up, display the form on the screen, send information to Windows regarding what it wants to do with events, and shut it

down when you have finished. In Visual Basic, you don't have access to any of the code that does this. By contrast, C# takes a completely different approach and leaves all this code in the open. That might make your source code look more complicated, but it does have the advantage that if the code is available, then you can edit it, which means you gain much more flexibility in deciding how your application should behave.

What Happens When You Run a Program

Any program involves a precise sequence of execution. When an application is launched the computer comes across an instruction that identifies the start of the program. It will then carry on executing the next instruction, and the next, and the next, and so on. Some of these commands will tell the computer to jump to a different instruction, depending on the values contained in certain variables, for example. Very often the computer will jump back and execute the same instructions again. However, there is always this continuous sequence of executing the next instruction until the computer comes across a command that tells it to terminate the execution of the code. This linear sequence is true of any program. Some programs may be multithreaded, in which case there are several sequences of execution (threads). However, each thread still follows this sequence from an initial instruction through to termination of the program.

Of course, this sequence is not what you see when you write a Visual Basic executable program. In Visual Basic, what you write is essentially a set of event handlers or subs, each of which is called when the user does something. There's typically no single start to the program, although the `Form_Load` event handler comes close to that in concept. Even so, `Form_Load` is really only another event handler. It just happens to be the handler for the event that gets raised when the form is loaded, which means it'll be the first event that runs. Similarly, if, instead of an executable, you are writing a control or a class object, you don't have a start point. You simply write a class and add lots of methods and properties to it. Each method or property will execute when the client code calls it.

Note also that in Visual Basic, `Sub Main` does exist, and acts as the entry point to a program, but unlike the `Main()` method of C#, `Sub Main` is optional.

To see how you can relate the two programming ideas, take a look at what actually happens when any Visual Basic application — or for that matter any Windows GUI application, no matter what language it is written in — executes. This is a bit more restrictive than the applications mentioned before, because now you are focusing on Windows GUI applications (in other words, not consoles, services, and so on).

As usual, execution starts at some well-defined point. The commands executed probably involve the creation of some Windows and controls, and displaying those controls on the screen. At that point, the program then does something that is known as *entering a message loop*. What effectively happens is that the program puts itself to sleep and tells Windows to wake it up when something interesting happens that it needs to know about. These "interesting" things are the events that you have written handlers for, and also a good few events that you haven't written your own event handlers for, because even if you don't write a handler for a particular event, the Visual Basic IDE may quietly supply one for you. A good example of this is the handlers that deal with resizing a form. You never see the source code for this in Visual Basic, but a Visual Basic application is still able to respond correctly when the user attempts to resize the form because the Visual Basic IDE has invisibly added event handlers to your project to correctly handle this situation.

Whenever an event occurs, Windows wakes up the application and calls the relevant event handler — that's when the code that you wrote might start executing. When the event handler subroutine exits, the application puts itself to sleep again, once again telling Windows to wake it up when another interesting event happens. Finally, assuming nothing goes wrong, at some point Windows will wake up the application and inform it that it needs to shut down. At that point, the application takes any appropriate action — for example, displaying a message box asking users if they want to save a file — and then terminates itself. Again, most of the code to do this has been quietly added to your project behind the scenes by the Visual Basic IDE, and you never get to see it.

Figure B-2 shows the thread of execution in a typical Windows GUI application.

In Figure B-2, the box with a dashed border indicates the part of execution that the Visual Basic IDE lets you get access to, and for which you can write source code: some of the event handlers. The rest of the code is inaccessible to you, though you can specify it to some extent through your choice of application type when you first ask Visual Basic to create a project. Recall that when you create a new project in Visual Basic, you get a dialog box asking you what type of application you want to create: Standard EXE, ActiveX EXE, ActiveX DLL, and so on. After you make your selection, the Visual Basic IDE uses your choice to generate all the appropriate code for the part of the program that is outside the dashed box in Figure B-2. The diagram shows the situation when you choose to create a Standard EXE project and will differ for other types of project (for example, an ActiveX DLL doesn't have a message loop at all but relies on clients to call the methods instead), but it should give you a rough idea of what's going on.

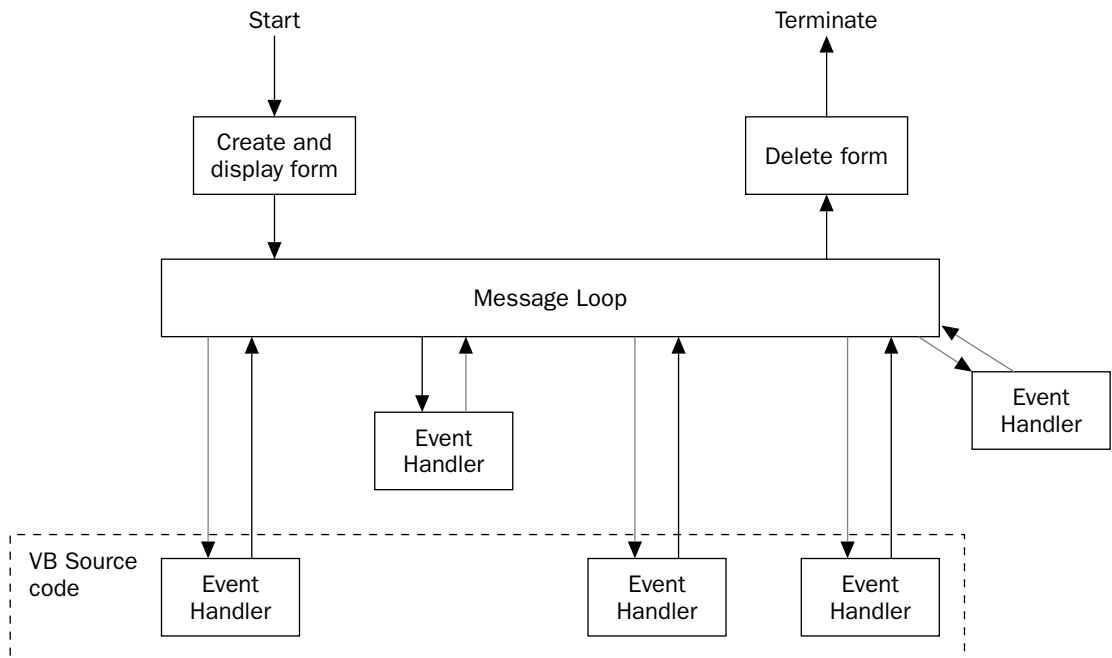


Figure B-2

In C# you can view (if not modify) the code that does everything. All the nitty-gritty details of things, such as what's going on inside the message loop, are hidden inside various DLLs that Microsoft has written, but you do get to see the high-level methods that call up the various bits of processing. So, for example, you have access to the code that starts up the program, the call to a library method that makes your program enter the message loop and puts it to sleep, and so on. You also get access to the source code that instantiates all the various controls you place on your form, makes them visible, and sorts out their initial positions and sizes, and all the rest. You don't need to write any of this code yourself. When you use Visual Studio 2005 to create a C# project, you will still get a dialog box asking you which type of project you want to create, and Visual Studio will still write all the background code for you. The difference is that Visual Studio writes this background code as source C# code, which then becomes code that you can edit directly.

Doing things this way does, as mentioned, bloat your source code. However, the huge advantage is that you have much more flexibility in what your program does and how it behaves. It also means that you can write many more types of projects in C#. Whereas in Visual Basic, the only things you can write are different kinds of form, and COM components, in C# you can write any of the different types of programs that run on Windows.

The C# Code for the Rest of the Program

This section discusses the rest of the code for the `SquareRoot` sample. In the process you will learn a bit more about classes in C#.

The C# `SquareRoot` sample was created in Visual Studio 2005, and the Visual Basic one was created in the Visual Basic 6 IDE. However, the code presented here isn't quite what Visual Studio generated for us. Apart from adding the event handler, a couple of other tweaks have been made to the code in order to better illustrate the principles of C# programming. However, it will still give you a good idea of the sort of work that Visual Studio does when it creates a project for you.

The full text of the source code is quite long. It will not be presented here; you can find it in the accompanying document, `VBToCSharp_CSharpSource.pdf`.

Namespaces

The main part of the C# `SquareRoot` source code begins with a couple of namespace declarations and a class declaration:

```
namespace Wrox.ProCSharp.VbToCSharp.SquareRootSample
{
    public class SquareRootForm : System.Windows.Forms.Form
    {
```

The `SquareRootForm` class holds almost all the code with the exception a small amount of code that is contained in the class `MainEntryClass`. Remember that it's easiest here to think of a C# class resembling a Visual Basic class object, with the exception that you can see the source code that begins the declaration of the class. The Visual Basic IDE only gives you a separate window with the contents of the class in it.

A namespace is something that doesn't really have an analogy in Visual Basic, and the easiest way to think of it is as a way of organizing the names of your classes in much the same way as a file system organizes the names of your files. For example, you almost certainly have a number of files on your hard

drive all called `ReadMe.Txt`. If that name, `ReadMe.Txt`, were the only information you had about each file, you'd have no way of distinguishing between them all. However, you can distinguish between them using their full pathnames; for example, on my computer one of them is actually `C:\Program Files\ReadMe.txt` and another is `G:\Program Files\HTML Help Workshop\ReadMe.txt`.

Namespaces work in the same way, but without all the overhead of having an actual file system — they are basically no more than labels. You don't have to do anything to create a namespace, other than declare it in your code in the way you've done in the preceding sample. The preceding code means that the full name of the class you have defined is not `SquareRootForm`, but `Wrox.ProCSharp.VbToCSharp.SquareRootSample.SquareRootForm`. It is extremely unlikely that anyone else will write a class with that full name. On the other hand, if you didn't have the namespace, there would be more risk of confusion because someone else might conceivably write a class called `SquareRootForm`.

Avoiding clashes in this way is important in C#, because the .NET environment uses only these names to identify classes, whereas the ActiveX controls created by Visual Basic used a complex mechanism involving GUIDs to avoid name clashes. Microsoft has opted for the simpler concept of namespaces because of concerns that some of the complexities of COM, such as GUIDs, made it unnecessarily difficult for developers to write good Windows applications.

In C#, although namespaces are not strictly required, it is strongly advised that you place all your C# for classes in a namespace in order to prevent any possible name clashes with other software. In fact, it is quite rare to see C# code that does not start with a namespace declaration, and namespaces are an excellent method to organize related classes into logical order.

Namespaces can be nested. For example, this namespace code

```
namespace Wrox.ProCSharp.VbToCSharp.SquareRootSample
{
    public class SquareRootForm : System.Windows.Forms.Form
    {
        // and so on
    }
}
```

could have been written like this:

```
namespace Wrox
{
    namespace ProCSharp
    {
        namespace VbToCSharp
        {
            namespace SquareRootSample
            {
                public class SquareRootForm : System.Windows.Forms.Form
                {
                    // and so on
                }
            }
        }
    }
}
```

Appendix B

In this code, the closing curly braces are added just to emphasize that they always have to match up. Curly braces are used to mark the boundaries of namespaces and classes just as they are used to mark the boundaries of methods and compound statements.

The using directive

The final part of the code that begins the `SquareRoot` project consists of using directives:

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

namespace Wrox.ProCSharp.VbToCSharp.SquareRootSample
{
```

These using directives are here to simplify the code. Full names of classes, including the namespace names, are long. For example, later in this code you will be defining a couple of text boxes. A `TextBox` control is represented by the class `System.Windows.Forms.TextBox`. If you had to write that in your code every time you wanted to refer to `TextBox`, your code would look very messy. Instead, the statement `using System.Windows.Forms;` instructs the compiler to look in this namespace for any classes that are not in the current namespace, and for which you have not specified a namespace. Now you can simply write `TextBox` whenever you want to refer to that class. It is common to start any C# program with a number of using directives that bring in all the namespaces you are going to use into the set of namespaces searched by the compiler. The namespaces specified in the previous code are all namespaces that cover various parts of the .NET base-class library, and so allow you to conveniently use various .NET base classes.

The class definition: Inheritance

Now you come to the definition of the `SquareRootForm` class. The definition itself is fairly simple:

```
public class SquareRootForm : System.Windows.Forms.Form
{
```

The keyword `class` tells the compiler that you are about to define a class. The interesting part is the colon after the name of the class, which is followed by another name, `Form`. This is the point at which you need to bring in that other important C# concept mentioned earlier: *inheritance*.

What the previous code does is tell the compiler that the `SquareRootForm` class inherits from the class `Form` (actually `System.Windows.Forms.Form`). What this means is that the class has not only any methods, properties, and so on that you define; it also inherits everything that was in `Form`. `Form` is an extremely powerful .NET base class, which gives you all the features of a basic form. It contains methods that get the form to display itself, and a large number of properties including `Height`, `Width`, `DesktopLocation`, and `BackColor` (the background color of the form), which control the appearance of the form on the screen. By inheriting from this class, your own class gets all these features as well, and is therefore already a fully fledged form. The class you inherit from is known as the base class, and the new class is known as the *derived class*.

If you have worked with interfaces before, the concept of inheritance will not be new to you. What you have here, however, is much more powerful than interface inheritance. When a COM interface inherits from another interface, it only gets what the interface contains — the names and signatures of the methods and properties. However, a class contains all the code that implements these methods and so on as well, just as in Visual Basic a class object does. This means that `SquareRootForm` gets all the implementations of just about everything in `Form`, as well as the method names. This kind of inheritance is known as *implementation inheritance* and is not new to C#: It has been a fundamental concept of classic object-oriented programming (OOP) for decades. C++ and Java programs, in particular, use this concept extensively, but it was not supported in Visual Basic. (Implementation inheritance does have similarities to subclassing.) As you get used to writing C# programs, you will find that the entire architecture of a typical C# program is almost invariably based around implementation inheritance.

But implementation inheritance is even more powerful than that. As you will see later on, when a class inherits from another class, it doesn't have to take *all* the implementations of everything in the base class. If you want, you can modify the implementations of particular methods and properties using a technique called *overriding*. This means that you can create a class that is very similar to an existing class, but has some differences in how it works or what it does. That makes it very easy for you to reuse code that other people have written, thereby saving yourself a lot of development time. It is also important to understand that you don't need access to the source code of the base class in order to derive from it. For obvious commercial reasons, Microsoft is keeping the source code of `Form` to itself. The fact that the compiled library is available in the form of an assembly is sufficient for you to be able to inherit from the class, taking those methods you want and overriding those that you don't want.

Program Entry Point

You now jump to near the end of the sample code, to examine the main program entry point. That is the `Main()` function, reproduced here:

```
class MainEntryClass
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>

    [STAThread] static void Main()
    {
        SquareRootForm TheMainForm = new SquareRootForm();
        Application.Run(TheMainForm);
    }
}
```

This doesn't look at first sight like a very obvious program entry point, but it is. The rule in C# is that program execution starts at a method called `Main()`. This method must be defined as a static method in some class. There can normally be only one `Main()` method throughout all the classes in the source code — otherwise the compiler won't know which one to choose, without further compilation switches. `Main()` here is defined as not taking any parameters and returning `void` (in other words, not returning anything). This isn't the only possible signature for the method, but it is the usual one for a Windows application (command-line applications may take parameters; these are any command-line arguments you specify).

Because `Main()` has to be in a class, you've put it in one: a class called `MainEntryClass`. Although this class doesn't contain anything else, it's legitimate for a class that contains the main entry point to contain other methods as well. The fact that `Main()` is a static method is important. Recall that static methods can be run without actually creating an object of the class first. Because the very first thing that happens when the program is run is that `Main()` is called, there aren't yet any instances of any classes, hence the entry point has to be static.

Apart from the static keyword, the definition of `Main()` looks much like the earlier method definition you examined. However, it is prefixed by the word `[STAThread]` in square brackets. `STAThread` is an example of an *attribute*—another concept that has no equivalent in Visual Basic source code.

An attribute is something that provides extra information to the compiler about some item in the code, and always takes the form of a word (possibly with some parameters as well, though not in this case) in square brackets, immediately before the item to which it applies. This particular attribute tells the compiler about the threading model that the code needs to run in. Without going into details, note that writing `[STAThread]` in the C# source code has a similar effect to selecting the threading model under Project Properties in the Visual Basic IDE, although in Visual Basic you can only do this for ActiveX DLL and ActiveX Control projects.

This comparison shows once again the different philosophy of C# compared to Visual Basic. In Visual Basic, the threading model is there and needs to be specified, but it is all but hidden by the Visual Basic IDE, so you can't get to it in the Visual Basic source code—in Visual Basic you have to access it through the project settings.

Instantiating Classes

Now let's examine the code inside the `Main()` method. The first thing you need to do is create the form, that is, instantiate a `SquareRootForm` object. This is dealt with by the first line of code:

```
SquareRootForm TheMainForm = new SquareRootForm();
```

You obviously can't compare this with the corresponding Visual Basic code, because the corresponding Visual Basic commands aren't available as source code, but you can do a comparison—if you imagine that in some Visual Basic code you are going to create a dialog box. In VB6, the way you would do that would look something like this:

```
Dim SomeDialog As MyDialogClass  
Set SomeDialog = New MyDialogClass
```

In this Visual Basic code, you first declare a variable that is an object reference: `SomeDialog` refers to a `MyDialogClass` instance. Then you actually instantiate an object using the Visual Basic `New` keyword and set your variable to refer to it.

That's exactly what is going on in the C# code too: you declare a variable called `TheMainForm`, which is a reference to a `SquareRootForm` object; then you use the C# `new` keyword to create an instance of `SquareRootForm`, and set your variable to refer to it. The main syntactical difference is that C# allows you to combine both operations into one statement, in the same way that you were previously able to declare and initialize the `NumberInput` variable in one go. Note also the parentheses after the `new` expression. That is a requirement of C#. When creating objects, you always have to write these brackets

in. The reason is that C# treats creating an object a bit like a method call, to the extent that you can even pass parameters into the call to `new`, to indicate how you want the new object to be initialized. In this case, you don't pass in any parameters, but you still need the parentheses.

C# classes

So far you've seen that C# classes are similar to class modules in Visual Basic. You've already seen one difference in that C# classes allow static methods. The code for the `Main()` method now highlights another difference. If you were doing something like this in Visual Basic, you would also need to set the object created to `Nothing` when you have finished with it. However, nothing like that appears in the C# code, because in C# it is not necessary to do this. That's because C# classes are more efficient and lightweight than their Visual Basic counterparts. Visual Basic class objects are really COM objects, which means they include some sophisticated code that checks how many references to the object are being held, so that each object can destroy itself when it detects it is no longer needed. In Visual Basic, if you don't set your object reference to `Nothing` when you have finished with the object, this is considered bad practice because it means the object does not know that it is no longer needed, so it can stay in memory, possibly until the whole process ends.

However, for performance reasons, C# objects don't perform this kind of checking. Instead, C# makes use of the .NET *garbage collection* mechanism. What happens is that, instead of each object checking whether it should still be alive, every so often the .NET runtime hands control to the garbage collector. The garbage collector examines the state of memory, uses a very efficient algorithm to identify those objects that are no longer referenced by your code, and removes them. Because of this mechanism, it is not considered important that you reset references when you have finished with them — it is normally sufficient to simply wait until the variable goes out of scope.

If, however, you do want to set reference variables not to refer to anything, then the relevant C# keyword is `null`, which is identical to `Nothing` in Visual Basic. Hence, where in Visual Basic you would write

```
Set SomeDialog = Nothing
```

in C# you would write something like:

```
TheMainForm = null;
```

Note, however, that this by itself doesn't achieve much in C# unless the variable `TheMainForm` still has a substantial lifetime left, because the object won't be destroyed until the garbage collector is called up.

Entering the message loop

Now consider the final statement in the `Main()` method:

```
Application.Run(TheMainForm);
```

This statement is the one that enters the message loop. What you are actually doing is calling a static method of the class `System.Windows.Forms.Application`. The method in question is the `Run()` method. This method handles the message loop. It puts the application (or strictly speaking, the thread) to sleep and requests Windows to wake it up whenever an interesting event occurs. The `Run()` method can take one parameter, which is a reference to the form that handles all events. `Run()` exits when an event instructing the form to terminate has occurred and been handled.

After the `Run()` method has exited, there is nothing else to be done, so the `Main()` method returns. Because this method was the entry point to the program, when it returns, execution of the entire process stops.

One piece of syntax in the preceding statements that you might find surprising is that you use parentheses when calling the `Run()` method, even though you are not using any return value from this method, and hence you are doing the equivalent of calling a Visual Basic sub. In this situation, Visual Basic does not require parentheses, but the rule is that in C# you always use parentheses when calling any method.

Always use parentheses in C# when calling any method, whether or not you are going to use any return value.

The *SquareRootForm* Class

You have now seen how C# enters a message loop, but you have not yet seen the process of displaying and creating the form itself, and the text has also been rather vague about the calling of the event handlers. You have seen that Windows calls event handlers, such as the `OnClickButtonResults()` method. But how does Windows know that that is the method to be called? You can find the answers to those questions in the `SquareRootForm` class definition, and in its base class, `Form`.

First, note that the `SquareRootForm` class has quite a number of member fields. (Member field is C# parlance for a variable that is defined as a member of a class. You can think of it as being like a Visual Basic variable that has form scope, or alternatively as being like a Visual Basic variable that is defined as a member of a class module. Each such variable is associated with a particular instance of a class—a particular object—and stays in scope for as long as its containing object remains alive.)

```
public class SquareRootForm : System.Windows.Forms.Form
{
    private System.Windows.Forms.TextBox txtNumber;
    private System.Windows.Forms.TextBox txtSign;
    private System.Windows.Forms.TextBox txtResult;
    private System.Windows.Forms.Button cmdShowResults;
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Label label2;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.Label label4;
}
```

These fields each correspond to one of the controls. You can see clearly the three `TextBox` and the `Button` controls, as well as the four `Label` controls, corresponding to the areas of text on the form. You won't be doing anything with the labels so there's no need to give them more user-friendly names.

However, each of these variables is just a reference to an object, so the fact that these variables exist doesn't imply any instances of these objects exist—the objects have to be instantiated separately. The process of instantiating these controls is done in a *constructor*. A constructor in C# is analogous to Visual Basic subs such as `Form_Load`, `Form_Initialize`, and `Class_Initialize`. It is a special method that is automatically called whenever an instance of the class is created, and it contains whatever code is needed to initialize the instance.

You can spot the constructor in the class because a constructor always has the same name and casing as the class itself. In this case you just look for a method called `SquareRootForm`:

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

Note that because this is a constructor, not a method, that you can call, it doesn't have any return type specified. It does, however, have parentheses after its name just like a method. You can use these parentheses to specify parameters to be passed to the constructor (you can pass parameters in the parentheses after the new clause when creating a variable). The definition of the constructor indicates if any parameters are needed to create an instance of the object. However, you don't have any parameters in this example; they are included in the `Employee` code sample later in the appendix.

In this case the constructor just calls a method, `InitializeComponent()`. This is because of Visual Studio 2005. Visual Studio has the same set of features as the Visual Basic IDE for manipulating controls graphically — clicking to place controls on the form and so on. However, because now with C# the definitions of all the controls are set out in the source code, Visual Studio 2005 has to be able to read the source code to find out what controls are around on your form. It does this by looking for an `InitializeComponent()` method, and seeing what controls are instantiated there.

`InitializeComponent()` is a huge method, so you won't look at it all, but it starts off like this:

```
private void InitializeComponent()  
{  
    this.txtNumber = new System.Windows.Forms.TextBox();  
    this.txtSign = new System.Windows.Forms.TextBox();  
    this.cmdShowResults = new System.Windows.Forms.Button();  
    this.label3 = new System.Windows.Forms.Label();  
    this.label4 = new System.Windows.Forms.Label();  
    this.label1 = new System.Windows.Forms.Label();  
    this.label2 = new System.Windows.Forms.Label();  
    this.txtResult = new System.Windows.Forms.TextBox();
```

The previous code is a set of calls to actually instantiate all the controls on the form. This snippet doesn't really contain any new pieces of C# syntax. The next part of the code starts setting properties on the controls:

```
//  
// txtNumber  
//  
this.txtNumber.Location = new System.Drawing.Point(160, 24);  
this.txtNumber.Name = "txtNumber";  
this.txtNumber.TabIndex = 0;  
this.txtNumber.Text = "";  
  
//  
// txtSign  
//
```

```
this.txtSign.Enabled = false;
this.txtSign.Location = new System.Drawing.Point(160, 136);
this.txtSign.Name = "txtSign";
this.txtSign.TabIndex = 1;
this.txtSign.Text = "";
```

This code sets up the start positions and initial text of two of the controls, the input `TextBox` control and the `TextBox` control that displays the sign of the number input. One new bit of code is that the location relative to the top-left corner of the screen is specified using a `Point`. `Point` is a .NET base class (in fact, a struct) that stores *x* and *y* coordinates. The syntax for the two lines that set the `Location` is instructive. The `TextBox.Location` property is just a reference to a `Point`, so in order to set it to a value you need to create and initialize a `Point` object that holds the correct coordinates. This is the first time that you've seen a constructor that takes parameters—in this case, the horizontal and vertical coordinates of the `Point` and hence of the control. If you'd wanted to translate one of these lines into Visual Basic, assuming you'd defined some Visual Basic class module called `Point`, and you had a class that had such a property, the best you would be able to do would look something like this:

```
Dim Location As Point
Set Location = New Point
Location.X = 160
Location.Y = 24
SomeObject.Location = Location
```

Compare this to the C# code:

```
someObject.Location = new System.Drawing.Point(160, 24);
```

The relative compactness and readability of the equivalent C# statement should be obvious! Now look at the same commands for the button. In this case, you see the same kinds of properties being set up, but here there is one other thing that needs to be done: you need to tell Windows to call your event handler when the button is clicked. The line that does this is shown in bold:

```
this.cmdShowResults.Name = "cmdShowResults";
this.cmdShowResults.Size = new System.Drawing.Size(88, 23);
this.cmdShowResults.TabIndex = 3;
this.cmdShowResults.Text = "Show Results";
this.cmdShowResults.Click += new System.EventHandler(this.OnClickShowResults);
```

What's going on here is this: The button, which is referred to by the `cmdShowResults` button object, contains an event, `Click`, that will be raised when the user clicks the button. You need to add your event handler to this event. Now C# doesn't allow you to pass names of methods around directly; instead you have to wrap them up into something called a delegate. This is done to ensure type safety (see Chapter 6, "Delegates and Events"), and is the reason for the new `System.EventHandler()` text in the code. Once you've wrapped the name of the event handler up, you add it to the event using an operator `+=`, which is discussed next.

Arithmetic assignment operators

The `+=` symbol represents what is known as the addition-assignment operator in C#. It provides a convenient shorthand for cases where you want to add some quantity to another quantity. How it works is this. Say, in Visual Basic you had declared two `Integers`, *A* and *B*, and you were going to write

```
B = B + A
```

In C# the equivalent type is `int`, and you can write something very similar:

```
B = B + A;
```

However, in C#, there is an alternative shorthand for this:

```
B += A;
```

`+=` really means “add the expression on the right to the variable on the left,” and it works for all the numeric data types, not just `int`. Not only that but there are other similar operators, `*=`, `/=`, and `-=` which respectively multiply, divide, and subtract the quantity on the left by the one on the right. So for example, to divide a number by 2, and assign the result back to `B`, you’d write:

```
B /= 2;
```

C# has other operators that represent bitwise operations, as well as `%` that takes the remainder on division—and almost all of these have corresponding operation assignment operators (see Chapter 2, “C# Basics”).

In the `SquareRootForm` sample, you have simply applied the addition assignment operator to an event; the line

```
this.cmdShowResults.Click += new System.EventHandler(this.OnClickShowResults);
```

simply means “add this handler to the event.”

Note that operators like `+`, `-`, `*`, and so on in Visual Basic only have meaning when applied to numeric data. In C#, however, they can be applied to any type of object.

The previous statement needs to be qualified a bit. In order to be able to apply these operators to other types of objects, you have to first tell the compiler what these operators mean for other types of objects—a process known as operator overloading. Suppose you want to write a class that represented a mathematical vector. In Visual Basic you write a class module, and then add:

```
Dim V1 As Vector
Set V1 = New Vector
```

In mathematics, it’s possible to add vectors, which is where operator overloading comes in. But Visual Basic doesn’t support operator overloading, so instead in Visual Basic you’d probably define a method, `Add`, on the `Vector`, so you could do this:

```
' V1, V2, and V3 are Vectors
Set V3 = V1.Add(V2)
```

In Visual Basic, that’s the best you can do. However, in C#, if you define a `Vector` class, you can add an operator overload for `+` to it. The operator overload is basically a method that has the name operator `+`, and which the compiler will call up if it sees `+` applied to a `Vector`. That means that in C# you are able to write:

```
// V1, V2 and V3 are Vectors  
V3 = V1 + V2;
```

Chapter 5, “Operators and Casts,” details the code for overloading such a `Vector` class, and discusses operator overloading in more detail.

Obviously you wouldn’t want to define operator overloads for all classes. For most classes that you write, it wouldn’t make sense to do things like add or multiply objects together. However, for the classes for which it does make sense to do this, operator overloads can go a long way toward making your code easier to read. That’s what has happened with events. Because it makes sense to talk about adding a handler to an event, an operator overload has been supplied to let you do this using the intuitive syntax using the `+` (and `+=`) operators. You can also use `-` or `-=` to remove a handler from an event.

You’ve really gone as far as you can go with the `SquareRootForm` code samples. There is a lot more C# code that you haven’t examined in the C# version of this application, but this extra code has largely to do with setting up the various other controls on the form, and doesn’t introduce any new principles.

Up to now, you’ve had a flavor of the syntax of C#. You’ve seen how it lets you write statements in a way that is often much shorter than the corresponding Visual Basic code. You have also seen the way that C# places all the code in the source file, unlike Visual Basic, where much of the background code is hidden from you—something that makes your code simpler at the cost of reducing your flexibility in the kinds of applications you can write. You’ve also had your first hints at the concepts behind inheritance.

However, what you have not yet seen is a real example of some code that you can write in C#, where it would be extremely hard to write Visual Basic code to achieve the same result. You are going to see an example of this in the next code sample, in which you write a couple of classes that illustrate the kinds of things you can do with inheritance.

Example: Employees and Managers

For this example, assume that you are writing an application that does some sort of processing on data that pertains to company employees. You are not really going to worry about what sort of processing this involves—you are more interested in the fact that this means it will be quite useful to write a C# class (or a Visual Basic class module) that represents employees. You are assuming that this will form part of a software package that you can sell to companies to help them with their salary payments and so on.

The Visual Basic Employee Class Module

The following code represents an attempt to code an `Employee` class module in Visual Basic. The class module exposes two public properties, `EmployeeName` and `Salary`, as well as a public method, `GetMonthlyPayment`, that returns the amount the company needs to pay the employee each month. This isn’t the same as the salary, partly because the salary is assumed to be the salary per year, and partly because later on you want to allow for the possibility of adding more money to what the company pays its employees (such as performance-related bonuses):

```

'local variable(s) to hold property value(s)

Private mStrEmployeeName As String 'local copy
Private mCurSalary As Currency 'local copy

Public Property Let Salary(ByVal curData As Currency)
    mCurSalary = curData
End Property

Public Property Get Salary() As Currency
    Salary = mCurSalary
End Property

Public Property Get EmployeeName() As String
    EmployeeName = mStrEmployeeName
End Property

Public Sub Create(sEmployeeName As String, curSalary As Currency)
    mStrEmployeeName = sEmployeeName
    mCurSalary = curSalary
End Sub

Public Function GetMonthlyPayment() As Currency
    GetMonthlyPayment = mCurSalary/12
End Function

```

In real life you'd probably be writing something more complex than this, but this class suffices for the purpose here. In fact, you already have a problem with this Visual Basic class module. Most people's names do not change very often, which is why the `EmployeeName` property is read-only. That still requires you to set up the name in the first place. This is done using a `Create` method, which sets the name and the salary. That means that the process of creating an employee object looks like this:

```

Dim Britney As Employee
Set Britney = New Employee
Britney.Create "Britney Spears", 20000

```

This is workable but messy. The problem is that you have to write a separate initialization method, `Create`, instead, and hope that everyone writing client code will always remember to call it. This solution is awkward, because it doesn't make any sense to have an `Employee` object lying around that doesn't have a name and a salary set. However, that is exactly what you have in this code for the brief instant between instantiating `Britney` and initializing the object in the code. As long as you always remember to call `Create`, you won't run into any problems, but there is a potential source of bugs here.

In C# the situation is completely different. In C# you are able to supply parameters to constructors. All you need to do is make sure that when you define your C# `Employee` class, the constructor takes the name and salary as parameters. Then you can write:

```

Employee Britney = new Employee("Britney Spears", 20000.00M);

```

This is a lot neater and less prone to bugs. Of course, you could overload the constructor to only supply a name, for example. By the way, note the "M" appended to the salary. This is because the C# equivalent

Appendix B

to the Visual Basic Currency type is called `decimal`, and “M” appended to a number in C# indicates you want the number interpreted as a `decimal`. You don’t have to supply it, but it makes for a useful extra compile-time check.

The C# Employee class

Here is the first definition of the C# version of `Employee` (note that this example only shows the class definition, not the containing namespace definition):

```
class Employee
{
    private readonly string name;
    private decimal salary;

    public Employee(string name, decimal salary)
    {
        this.name = name;
        this.salary = salary;
    }

    public string Name
    {
        get
        {
            return name;
        }
    }

    public virtual decimal Salary
    {
        get
        {
            return salary;
        }
        set
        {
            salary = value;
        }
    }

    public decimal GetMonthlyPayment()
    {
        return salary/12;
    }

    public override string ToString()
    {
        return "Name: " + name + ", Salary: $" + salary.ToString();
    }
}
```

Working through this code, you first see a couple of private variables, the so-called member fields corresponding to the member variables in the Visual Basic class module. The field `name` is marked `readonly`. Roughly speaking, this ensures that this field must be set when an `Employee` object is created and cannot subsequently be modified. In C# it isn't usual to use Hungarian notation for the names of variables, so they are called simply `name` and `salary`, rather than `mStrEmployeeName` and `mCurSalary`. (Hungarian notation means that you prefix the names of variables with some letters that indicate their type [`mStr`, `mCur`, and so on]). This type of notation is not considered important nowadays, because editors are more sophisticated and can supply automatic information about data types. Hence, the recommendation is not to use Hungarian notation in C# programs.)

The `Employee` class also contains a constructor, a couple of properties (`Name` and `Salary`), and two methods (`GetMonthlyPayment()` and `ToString()`). All of these are discussed next.

Note that the names of the properties `Name` and `Salary` differ only in case from the names of their corresponding fields. This isn't a problem, because C# is case-sensitive. The way the properties and fields are named here corresponds to the usual convention in C# and shows how you can actually take advantage of case sensitivity.

The Employee constructor

Following the field declarations in the previous code, you have a "method" that has the same name as the class, `Employee`. This tells you that it is a constructor. However, this constructor takes parameters and does the same thing as the `Create` method in the Visual Basic version. It uses the parameters to initialize the member fields:

```
public Employee(string name, decimal salary)
{
    this.name = name;
    this.salary = salary;
}
```

There's a potential syntax problem, because the obvious names for the parameters are the same as the names of the fields: `name` and `salary`. But you've resolved this problem using the `this` reference to mark the fields. You could have given the parameters different names instead, but the way you've done it is still clear enough and means that the parameters keep the simple names that correspond to their meanings. It's also the conventional way of dealing with this situation in C#.

The precise meaning of the `readonly` qualifier on the `name` field can now be explained:

```
private readonly string name;
```

If a field is marked as `readonly`, the only place in which it may be assigned to is in the constructor to the class. The compiler will raise an error if it finds any code in which you attempt to modify the value of a `readonly` variable anywhere except in a constructor. This provides a very good way of guaranteeing that a variable cannot be modified after it has been set. It wouldn't be possible to do anything like this in Visual Basic because Visual Basic doesn't have constructors that take parameters, so class-level variables in Visual Basic have to be initialized via methods or properties that are called after the object has been instantiated.

Appendix B

Incidentally, this constructor doesn't just allow you to supply parameters to initialize an `Employee` object: it actually forces you to do so. If you tried to write the following code, it would not compile:

```
Employee Britney = new Employee(); // will not compile now
```

The compiler would raise an error because, in C#, a constructor must always be called when a new object is created. However, you have not supplied any parameters, and the only constructor available requires two parameters. Therefore, it is simply not possible to create an `Employee` object without supplying any parameters. This provides a good guarantee against bugs caused by uninitialized `Employee` objects!

It is possible to supply more than one constructor to a class so that you get a choice of what sets of parameters you want to pass in when you create a new object of that class. You'll see how to do this later in the chapter. However, for this particular class, your one constructor is quite adequate.

Properties of Employee

You next come to the properties `Name` and `Salary`. The C# syntax for declaring a property is very different from the corresponding Visual Basic syntax, but the basic principles are unchanged. You need to define two accessors to respectively get and set the values of the property. In Visual Basic, these are syntactically treated like methods, but in C# you declare the property as a whole and then define the accessors within the definition of the property:

```
public decimal Salary
{
    get
    {
        return salary;
    }
    set
    {
        salary = value;
    }
}
```

In Visual Basic, the compiler knows that you are defining a property, because you use the keyword `Property`. In C# this information is conveyed by the fact that the name of the property is followed immediately by an opening brace. If you were defining a method, this would be an opening parenthesis signaling the start of the parameter list; in the case of a field, this would be a semicolon, marking the end of the definition.

Note also that the definitions of the `get` and `set` accessors do not contain any parameter lists. That's because you know that `Salary` is a decimal and that the `get` accessor will return a decimal and take no parameters, while the `set` accessor will take one decimal parameter and return `void`. For the `set` accessor, this parameter is not explicitly declared, but the compiler always interprets the word `value` as referring to it.

Once again, the syntax for defining properties shows how C# syntax is more compact, and can save you a fair bit of typing!

If you want to make a property read-only, you simply omit the set accessor, as you have done for the Name property:

```
public string Name
{
    get
    {
        return name;
    }
}
```

Methods of Employee

The example also includes two methods: `GetMonthlySalary()` and `ToString()`.

`GetMonthlySalary()` requires little explanation, because you have seen most of the relevant C# syntax already. It simply takes the salary and divides it by 12 to convert it from an annual to a monthly salary, and returns the result:

```
public decimal GetMonthlyPayment()
{
    return salary/12;
}
```

The only new piece of syntax here is the return statement. In Visual Basic, you specify a return value from a method by setting a dummy variable that has the same name as the function to the required value:

```
GetMonthlyPayment = mCurSalary/12
```

In C#, you achieve the same result by appending a parameter to a return statement (without parentheses). `return` in C# also specifies that you are exiting from the function, so the C# statement

```
return salary/12;
```

is actually equivalent to the following Visual Basic code:

```
GetMonthlyPayment = mCurSalary/12
Exit Function
```

`ToString()` is slightly more interesting. In most cases, when you write a C# class, it is a good idea to write a `ToString()` method that can be used to get a quick view of the contents of an object. As mentioned before, `ToString()` is already available because all classes inherit it from `System.Object`. However, the version in `System.Object` simply displays the name of the class, not any data in the class instance. Microsoft has already overridden this method for all the numeric data types (`int`, `float`, and so on) to display the actual value of the variable, and it's quite useful for you to do the same in your classes. If nothing else, it can be a useful way of seeing the contents of an object when you are debugging:

```
public override string ToString()
{
    return "Name: " + name + ", Salary: $" + salary.ToString();
}
```

The override here simply displays the name and the salary of the employee. One new piece of syntax is that you have declared the method as `override`. C# requires that you mark method overrides in this way; it will raise a compilation error if you don't. This eliminates the risk of any potential bugs that might lead you, for example, to accidentally override an existing method by that name without realizing it.

You have now completed writing the `Employee` class in both Visual Basic and C#, and so far, although there is a bit of awkwardness about constructing and initializing an `Employee` instance in the Visual Basic version, both languages have coped reasonably well with your requirements. However, one of the aims of this appendix is to show you why C# can be so much more powerful than Visual Basic in some situations. So it's about time you started seeing some useful C# code where it would be very difficult if not impossible to achieve the same result using Visual Basic. Let's start with a `static` field and property.

Static Members

It's been mentioned a few times that in C# classes can have special methods referred to as static methods, which can be called without instantiating any objects. These methods do not have any counterpart in VB6. In fact, not only methods but also fields, properties, or any other class member can be static.

The term "static" has a very different meaning in C# from its meaning in Visual Basic.

To illustrate how static members work and why you would use them, imagine that you would like your `Employee` class to support retrieving the name of the company that each employee works for. Now there is an important difference here between the company name and the employee name, in that each employee object represents a different employee, and therefore needs to store a different employee's name. This is the usual behavior for variables in class modules in Visual Basic, and the default behavior for fields in C#. However, if your organization has just purchased the software that contains the `Employee` class, obviously all of the employees will have the same company name. This means that it would be wasteful to store the company name separately for each employee. You'd just be duplicating the string unnecessarily. Instead, what you want is just to store the company name once, and then have every employee object access the same data. This is how a static field works. Declare a static field, `companyName`:

```
class Employee
{
    private string name;
    private decimal salary;
    private static readonly string companyName;
```

In this code, you have simply declared another field, but by marking it as `static` you have instructed the compiler to store this variable only once, no matter how many `Employee` objects are created. In a real sense, this static field is associated with the class as a whole, rather than with any one object.

You have also declared this field as read-only. This makes sense because, like the employee's name, the company name should not be changed when the program is running.

Of course, merely declaring this field isn't enough. You also need to make sure it is initialized with the correct data. Where should you do that? Not in your constructor; the constructor is called every time you create an `Employee` object, whereas you only want to initialize `companyName` once. The answer is

that C# provides another construct for this purpose, known as the *static constructor*. The static constructor acts like any other constructor, but it works for the class as a whole, not for any particular object. If you define a static constructor for a class, it will be executed just once. As a rule, it will execute before any client code attempts to access the class, typically when the program first starts up. Add a static constructor to the `Employee` class:

```
static Employee()  
{  
    companyName = "Wrox Press Pop Stars";  
}
```

As usual, you identify the constructor because it has the same name as the class. This one is also identified as `static`; hence, it is the static constructor. It is marked neither as `public` nor as `private` because it is called by the .NET runtime, not by any other C# code. So, just for the static constructor, you don't need any access modifier.

In the example, you have implemented the static constructor by hard-coding in a company name. More realistically, you might read a registry entry or a file, or connect to a database to find out the company name. Incidentally, because the `companyName` field has been declared as both static and read-only, the static constructor is the only place in which you can legally assign a value to it. You have one last thing to do, which is to define a public property that lets you access the company name.

```
public static string CompanyName  
{  
    get  
    {  
        return companyName;  
    }  
}
```

The `CompanyName` property has also been declared as static, and you can now see the real significance of a static method or property: a method or property can be declared as static if it accesses only static fields and does not access any data that is associated with a particular object.

As you have already seen, the syntax for calling static members of the class from outside the class is slightly different from that used for other members. Because a static member is associated with the class rather than with any object, you use the class name rather than the name of a variable to call it:

```
string Company = Employee.CompanyName;
```

The concept of static members is very powerful and provides a very useful means for a class to implement any functionality that is the same for every object of that class. The only way that you can achieve anything like this in Visual Basic is by defining global variables. The disadvantage of global variables is that they are not associated with any particular class, and this can lead to name conflict issues.

Here are two more situations in which you might use static class members:

- ❑ You might choose to implement a `MaxLength` property for your `Employee` class, or for that matter for any other class that contains a name, where you might need to specify the maximum length of the name.

- ❑ In C#, most of the numeric data types have static properties that indicate their maximum possible values. For example, to find out the biggest values that can be stored in an `int` and a `float`, you could write:

```
int MaxIntValue = int.MaxValue;
float MaxFloatValue = float.MaxValue;
```

Inheritance

In this section you take a closer look at how implementation inheritance works. Suppose that a year after you have shipped your software package it's time for the next version. One point that your customers have commented on is that some of their employees are actually managers, and managers usually get profit-related bonuses as well as regular salaries. This means that your `GetMonthlyPayment()` method doesn't give the complete information for managers. The practical upshot of this is that you have to have some way of dealing with managers, too.

For the purposes of this example, assume that the bonus is some constant figure that can be specified when you create a manager. You don't want to get bogged down in doing profit-related calculations here.

If you were coding in Visual Basic, how would you set about upgrading your software? There are two possible approaches; both of them have severe disadvantages:

- ❑ You could write a new class, `Manager`.
- ❑ You could modify the `Employee` class.

Writing a new class is probably the approach that would result in the least amount of work for you, because you'd probably start by simply copying and pasting all the code for the `Employee` class module and then modifying your copy of the code. The trouble is that `Employee` and `Manager` have an awful lot of code in common, such as the code surrounding the `Name`, `CompanyName`, and `Salary` properties. Having the same code duplicated is dangerous. What happens if, at some point in the future, you need to modify the code? Some poor developer is going to have to remember to make exactly the same changes to both classes. That is just asking for bugs to creep in. Another problem is that there are now two unrelated classes that client code will have to deal with, which is likely to make it harder for the people writing the code that uses `Employee` and `Manager`. (Although you could get around this by wrapping the common properties into an interface and having both `Employee` and `Manager` implement this interface.)

A slightly different alternative is to write a `Manager` class and put an `Employee` object inside it as a class-scoped variable. This solves the problem of duplicating code, but still leaves you with two separate objects, as well as an awkward, indirect, syntax for calling employee methods and properties (for example, `objManager.objEmployee.Name`).

If you opt for modifying the `Employee` class module, then you could, for example, add an extra field, a `Boolean`, that indicates whether or not this `Employee` is a manager. Then, at relevant parts of the code, you would test this `Boolean` in an `If` statement, to check what to do. This solves the problem of having two unrelated classes. However, it introduces a new difficulty: As mentioned earlier, you decide *a year or so later* to add manager support. This means that the `Employee` class module has presumably been shipped, tested, fully debugged, and is known to be working correctly. Do you really want to have to dive in and start pulling working code to bits, with all the associated risk of introducing new bugs?

In short, you have reached a point at which Visual Basic cannot offer any satisfactory solutions. Enter C#, which does offer a way out of this quandary, through inheritance.

As mentioned earlier, inheritance involves adding or replacing features of classes. In the previous example, the `SquareRootForm` class added stuff to the .NET class, `System.Windows.Forms.Form`. It defined the controls to go on the `SquareRootForm` as member fields, and also added an event handler. The `Employee` example will demonstrate both adding and replacing features of a base class. You will define a `Manager` class, which is derived from `Employee`. You will add a field and property that represent the bonus, and replace the `GetMonthlyPayment()` method (for completeness, you'll also replace `ToString()` so that it displays the bonus as well as the name and salary). This all means that you will have a separate class. But you do not need to duplicate any code, nor do you need to make any big changes to the `Employee` class either. You might think that you still have a problem of two different classes—which makes it more difficult to write client code. However, C# provides a solution for this problem as well.

Inheriting from the Employee Class

Before you define the `Manager` class you need to make one small change to `Employee`—you have to declare the `GetMonthlyPayment()` method as `virtual`:

```
public virtual decimal GetMonthlyPayment()
{
    return salary/12;
}
```

Roughly speaking, this is the C# way of saying that this is a method that in principle can be overridden.

You might think that this means you are changing the base class, which invalidates the argument about not needing to change the base class. However, adding a `virtual` keyword isn't really the sort of major change that carries a risk of new bugs—with the Visual Basic approach you were going to have to actually rewrite the implementations of several methods. Besides, usually when you write classes in C#, you plan in advance for the methods that are suitable candidates for overriding. If this was a real-life example, `GetMonthlyPayment()` would almost certainly have been declared `virtual` in the first place, so then you really would have been able to add the `Manager` class without making any changes to the `Employee` class.

The Manager Class

You can now define the `Manager` class:

```
class Manager : Employee
{
    private decimal bonus;

    public Manager(string name, decimal salary, decimal bonus): base(name, salary)
    {
        this.bonus = bonus;
    }

    public Manager(string name, decimal salary): this(name, salary, 100000M)
    {

```

```
    }

    public decimal Bonus
    {
        get
        {
            return bonus;
        }
    }

    public override string ToString()
    {
        return base.ToString() + ", bonus: " + bonus;
    }

    public override decimal GetMonthlyPayment()
    {
        return base.GetMonthlyPayment() + bonus/12;
    }
}
```

Besides the near-complete implementation of the `Employee` class that you have inherited, `Manager` contains the following members:

- A field, `bonus`, which will be used to store the manager's bonus, and a corresponding property, `Bonus`
- The overloaded `GetMonthlyPayment()` method, as well as a new overload of `ToString()`
- Two constructors

The `bonus` field and corresponding `Bonus` property shouldn't need any further discussion. However, the next section looks in detail at the overridden methods and the new constructors, because they illustrate important C# language features.

Method Overrides

The override of `GetMonthlyPayment()` is reasonably simple. Notice that you have marked it with the keyword `override` to tell the compiler that you are overriding a base class method, as you did with `Employee.ToString()`:

```
public override decimal GetMonthlyPayment()
{
    return base.GetMonthlyPayment() + bonus/12;
}
```

Your override also contains a call to the base-class version of this method. This method uses a new keyword, `base`. `base` works in the same way as your override, except that it indicates that you want to grab a method, or property, from the definition in the base class. Alternatively, you could have implemented your override of `GetMonthlyPayment()` like this:

```
public override decimal GetMonthlyPayment()
{
    return (Salary + bonus)/12;
}
```

However, you cannot use this code:

```
public override decimal GetMonthlyPayment()
{
    return (salary + bonus)/12; // wrong
}
```

This code looks almost exactly like the previous version, except that you are hitting the `salary` field directly instead of going through the `Salary` property. You might think that this looks like a more efficient solution, because you are saving what is effectively a method call. The trouble is that the compiler will raise an error because the `salary` field has been declared as `private`. That means that nothing outside the `Employee` class is allowed to see this field. Even derived classes are not aware of private fields in base classes.

If you do want derived classes to be able to see a field, but not unrelated classes, C# provides an alternative level of protection, `protected`:

```
protected decimal salary; // we could have done this
```

If a member of a class is declared as `protected`, it is visible only in that class and in derived classes. However, in general, you are strongly advised to keep all fields `private` for exactly the same reason that you are advised to keep variables `private` in Visual Basic class modules: by hiding the implementation of a class (or class module) you are making it easier to carry out future maintenance of that class. Usually, you will use the `protected` modifier for properties and methods that are intended purely to allow derived classes access to certain features of the base class definition.

The Manager Constructors

You need to add at least one constructor to the `Manager` class for two reasons:

- ❑ There is now an extra piece of information, the manager's bonus, which you need to specify when you create a `Manager` instance.
- ❑ Unlike methods, properties, and fields, constructors are not inherited by derived classes.

In fact, you have added two constructors. This is because you have decided to assume that the manager's bonus normally defaults to \$100,000 if it is not explicitly specified. In Visual Basic you can specify default parameters to methods, but C# does not allow you to do this directly. Instead, C# offers a more powerful technique that can achieve the same effect, *method overloads*. Defining two constructors here will illustrate this technique.

The first `Manager` constructor takes three parameters:

```
public Manager(string name, decimal salary, decimal bonus) : base(name, salary)
{
    this.bonus = bonus;
}
```

The first thing you notice about this constructor is a call to the base class constructor using a slightly strange syntax. The syntax is known as a *constructor initializer*. What happens is that any constructor is allowed to call one other constructor before it executes. This call is made in a constructor initializer with the syntax shown in the preceding code. It is permitted for a constructor to call either another constructor in the same class, or a constructor in the base class. This might sound restrictive, but it is done for good reasons in terms of imposing a well-designed architecture on the constructors. These issues are discussed in Chapter 3, “Objects and Types.” The syntax for the constructor initializer requires a colon, followed by one of the keywords `base` or `this` to specify the class from which you are calling the second constructor, followed by the parameters you are passing on to the second constructor.

The constructor shown in the preceding code takes three parameters. However, two of these parameters, `name` and `salary`, are really there in order to initialize base class fields in `Employee`. These parameters are the responsibility of the `Employee` class rather than the `Manager` class, so what you do is simply pass them on to the `Employee` constructor for it to deal with—that’s what the call to `base(name, salary)` achieves. And as you saw earlier, the `Employee` constructor will simply use these parameters to initialize the `name` and `salary` fields. Finally, you take the `bonus` parameter, which is the responsibility of the `Manager` class, and use it to initialize the `bonus` field. The second `Manager` constructor that you’ve supplied also uses a constructor initialization list:

```
public Manager(string name, decimal salary) : this(name, salary, 100000M)
{
}
```

In this case, what is happening is that you set up the value of the default parameter, and then pass everything on to the three-parameter constructor. The three-parameter constructor, in turn, calls the base class constructor to deal with the `name` and `salary` parameters. You might wonder why you haven’t used the following alternative way of implementing the two-parameter constructor:

```
public Manager(string name, decimal salary,) : base(name, salary) // not so good
{
    this.bonus = 100000M;
}
```

The reason is that this involves some potential duplication of code. The two constructors each separately initialize the `bonus` field, and this might cause problems in the future in terms of both constructors needing separately to be modified if, for example, in some future version of `Manager` you change how you store the `bonus`. In general, in C# just as in any programming language, you should avoid duplicating code if you can. For this reason, the previous implementation of the two-parameter constructor is preferred.

Method Overloading

The fact that you have supplied two constructors for the `Manager` class illustrates the principle of method overloading in C#. Method overloading occurs when a class has more than one method of the same name, but different numbers of parameters. In the case of method overloading, the same principles apply as in constructor overloading.

Don’t confuse the terms method overloading and method overriding. Despite the similar names, they are different, and completely unrelated, concepts!

When the compiler encounters a call to a method that has been overloaded, it examines the parameters you are attempting to pass in, in order to figure out which method is the one to call. In the case of

creating a `Manager` object, because one constructor takes three parameters and the other only takes two, the compiler examines the number of parameters first. Hence, if you write

```
Manager SomeManager = new Manager ("Name", 300000.00M);
```

the compiler will arrange for a `Manager` object to be instantiated, with the two-parameter constructor being used, which means that the bonus will be given its default value of `100000M`. If, on the other hand, you write this:

```
Manager SomeManager = new Manager ("Name", 300000.00M, 50000.00M);
```

the compiler will arrange for the three-parameter constructor to be called, so `bonus` will now be given the specified value of `50000.00M`. If several overloads are available, but the compiler is unable to find one that is suitable, it will raise a compilation error. For example, if you wrote

```
Manager SomeManager = new Manager (100, 300000.00M, 50000.00M); // wrong
```

you would get a compilation error because both of the available `Manager` constructors require a string, and not a numeric type, as the first parameter. The C# compiler can arrange for some type conversions between the different numeric types to be done automatically, but it will not convert automatically from a numeric value to a string.

Note that C# does not allow methods to take default parameters in the way Visual Basic does. However, it is very easy to achieve the same effect using method overloads, as done in this example. The usual way is simply to have the overloads with fewer parameters supply default values for the remaining parameters and then call the other overloads.

Using the *Employee* and *Manager* Classes

Now that you have completed defining the `Employee` and `Manager` classes, you can write some code that uses them. In fact, if you download the source code for this project from the Wrox Press Web site (www.wrox.com), you will find that these two classes are defined as part of a standard Windows Forms project, quite similar to the `SquareRoot` sample. In this case, however, the main form has only one control, a list box. You use the constructor of the main form class (`MainForm`) to instantiate a couple of instances of `Employee` and `Manager` objects and then display data for these objects in the list box. Figure B-3 shows the results of this operation.

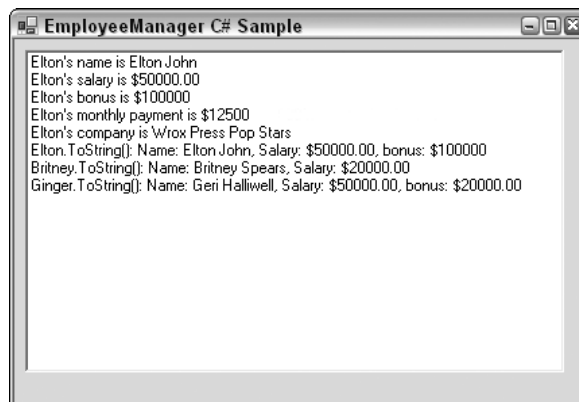


Figure B-3

The code used to generate these results is this:

```
public MainForm()
{
    InitializeComponent();
    Employee Britney = new Employee("Britney Spears", 20000.00M);
    Employee Elton = new Manager("Elton John", 50000.00M);
    Manager Ginger = new Manager("Geri Halliwell", 50000.00M, 20000.00M);
    this.listBox1.Items.Add("Elton's name is $" + Elton.Name);
    this.listBox1.Items.Add("Elton's salary is $" + Elton.Salary);
    this.listBox1.Items.Add("Elton's bonus is " + ((Manager)Elton).Bonus);
    this.listBox1.Items.Add("Elton's monthly payment is $" +
        Elton.GetMonthlyPayment());
    this.listBox1.Items.Add("Elton's company is " + Employee.CompanyName);
    this.listBox1.Items.Add("Elton.ToString(): " + Elton.ToString());
    this.listBox1.Items.Add("Britney.ToString(): " + Britney.ToString());
    this.listBox1.Items.Add("Ginger.ToString(): " + Ginger.ToString());
}
```

This code should be self-explanatory, based on the C# that you have learned up to now, apart from one little oddity — one of the `Manager` objects, `Elton`, is being referred to by an `Employee` reference instead of a `Manager` reference. How does this work? Keep reading.

References to Derived Classes

Take a closer look at the `Manager` class that is referenced by a variable declared as a reference to `Employee`:

```
Employee Elton = new Manager("Elton John", 50000.00M);
```

This is perfectly legal C# syntax. The rule is quite simple: if you declare a reference to a type `B`, then that reference is permitted to refer to instances of `B` or to instances of any class derived from `B`. This works because any class derived from `B` must also implement any methods or properties and so on that `B` implements. So in the previous example, you call `Elton.Name`, `Elton.Salary`, and `Elton.GetMonthlyPayment()`. The fact that `Employee` implements all these members guarantees that any class derived from `Employee` will do the same. So it doesn't matter if a reference points to a derived class — you can still use the reference to call up any member of the class the reference is defined as and be confident that that method exists in the derived class.

On the other hand, notice the syntax that is used when you call the `Bonus` property against `Elton`: `((Manager) Elton).Bonus`. In this case, you need to convert `Elton` to a `Manager` reference, because `Bonus` is not implemented by `Employee`. The compiler knows this and would raise a compilation error if you tried to call `Bonus` through an `Employee` reference. That line of code is shorthand for writing:

```
Manager ManagerElton = (Manager) Elton;
this.listBox1.Items.Add("Elton's bonus is " + ManagerElton.Bonus);
```

As in Visual Basic, conversion between data types in C# is known as *casting*. You can see from the previous code that the syntax for casting involves placing the name of the destination data type in parentheses before the name of the variable you are attempting to cast. Of course, the object being referred to must be of the correct type in the first place. If you wrote

```
Manager ManagerBritney = (Manager) Britney;
```

the code would compile correctly, but when you ran it, you would get an error, because the .NET runtime would see that `Britney` is just an `Employee` instance, not a `Manager` instance. References are permitted to refer to instances of derived classes, but not to instances of base classes of their native type. It's not permitted for a `Manager` reference to refer to an `Employee` object. (You can't permit it because if you did, what would happen if you attempted to call the `Bonus` property through such a reference?)

Because Visual Basic doesn't support implementation inheritance, there is no direct parallel in Visual Basic for C#'s support for references referring to objects of derived classes. However, there is some similarity with the fact that in Visual Basic you can declare an interface reference, and then it does not matter what type of object that interface refers to, as long as the object in question implements that interface. If you were coding the `Employee` and `Manager` classes in Visual Basic, you might as well have done so by defining an `IEmployee` interface that both class modules implement, and then access the `Employer` features through this interface.

Arrays of Objects

One important benefit of being able to have references refer to derived class instances is that you can form arrays of object references, where the different objects in the array might be of different types. This is analogous to the situation in Visual Basic where you could form arrays of interface references and not care about the fact that these interface references might be implemented by completely different classes of objects.

In order to see how C# deals with arrays, rewrite the test harness code for the `Employee` and `Manager` classes so that it forms an array of object references. You can download the revised code, called `EmployeeManagerWithArrays`, from the Wrox Press Web site (www.wrox.com). The new code looks like this:

```
public MainForm()
{
    InitializeComponent();

    Employee Britney = new Employee("Britney Spears", 20000.00M);
    Employee Elton = new Manager("Elton John", 50000.00M);
    Manager Ginger = new Manager("Geri Halliwell", 50000.00M, 20000.00M);

    Employee [] Employees = new Employee[3];
    Employees[0] = Britney;
    Employees[1] = Elton;
    Employees[2] = Ginger;

    for (int I=0 ; I<3 ; I++)
    {
        this.listBox1.Items.Add(Employees[I].Name);
        this.listBox1.Items.Add(Employees[I].ToString());
        this.listBox1.Items.Add("");
    }
}
```

Appendix B

You simply call up the `Name` property and the `ToString()` method of each element of the array. Figure B-4 shows the results of running this code.

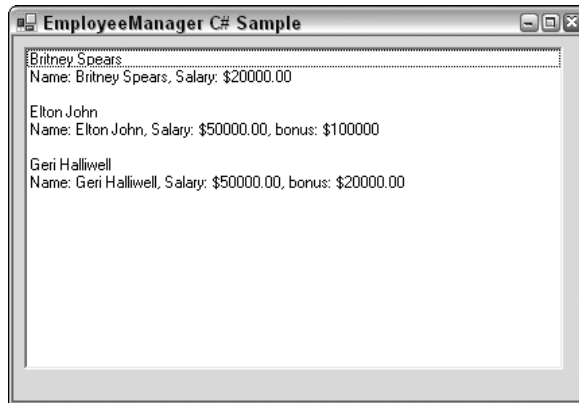


Figure B-4

From Figure B-4 you can see that C# uses square brackets for dealing with arrays. This means that, unlike in Visual Basic, there is no danger of any confusion about whether you're talking about an array or a method or function call. The syntax for declaring an array looks like this:

```
Employee [] Employees = new Employee[3];
```

As you can see, you declare an array of variables of a certain type by putting square brackets after the name of the type. An array in C# always counts as a reference object (even if its elements are simple types like `int` or `double`) so there are actually two stages: declaring the reference and instantiating the array. To make this clearer you could have split up the previous line of code like this:

```
Employee [] Employees;  
Employees = new Employee[3];
```

There is no difference between what you're doing here and how you instantiate objects, except that you are using square brackets to indicate that this is an array. Also note that the size of the array is established when you instantiate the object—the reference itself doesn't contain details of the size of the array—only its dimension. The dimension is specified by commas in the array declaration. For example, if you want to declare a two-dimensional, 3x4 array of doubles, you write this:

```
double [,] DoubleArray = new double[3,4];
```

When you have the array, you simply assign values to its elements in the usual way.

Note that one difference between C# and Visual Basic is that arrays in C# always start at element 0.

In Visual Basic you have the option to change this behavior to element 1 using the `Option Base` statement. You can also specify lower boundaries for any array. But this feature doesn't really add any benefits, and it can impact performance, because it means that whenever you access an element in an array in Visual Basic, the code has to do some extra checking to find out what of the lower bound of that array is for this collection. C# does not support changing the base of an array in this way.

In the previous code, once you have initialized the elements of the array, you just loop through them. If the "strange-looking" syntax of the `for` loop worries you, hang in there—you'll come back to it shortly.

Note that because the array has been declared as an array of `Employee`, you can access only those members of each object defined for the `Employee` class. If you wanted to access the `Bonus` property of any object in the array, you first would have to cast the corresponding reference to a `Manager` reference, which would mean checking whether the object is a `Manager` object. That is not difficult to do but is beyond the scope of this appendix.

Although you are using `Employee` references, you do always pick up the correct version of `ToString()`. If the object you're referring to is a `Manager` object, then, when you call `ToString()`, the version of `ToString()` defined in the `Manager` class is the one that is executed for that object. That is the beauty of overriding methods in C#. You can replace some method in the derived class, and know that no matter through which reference type this object is accessed, you will always run the correct method for that object.

The for Loop

This section discusses the `for` loop, introduced in the previous code snippet. What you have here is the C# equivalent of this Visual Basic code:

```
Dim I As Integer
For I = 1 To 3
    listBox1.Items.Add "Details of the Employee"
Next
```

The idea of the `For` loop in Visual Basic is that you start off by initializing a variable, called the *loop control variable*, and each time you go round the loop, you add something to the loop control variable until it exceeds a final value. This is quite useful, but gives you almost no flexibility in how the loop works. Although you can change the value of the increment, or even make the increment negative, by using the `Step` facility, the loop always works by counting, and the test of whether the loop exits is always whether the variable has reached a preset minimum or maximum value.

In C# the `for` loop generalizes this concept. The basic idea of the `for` loop in C# is this: At the beginning of the loop you do something, at each step of the loop you do something else in order to move to the next iteration, and in order to determine when to exit from the loop, you perform some test. The following table provides a comparison between the Visual Basic and C# versions of this loop.

Loop	Visual Basic	C#
At start of loop...	Initialize the loop control variable.	Do something.
To test whether to exit loop...	Check whether the loop control variable has reached a certain value.	Test some condition.
At end of each iteration...	Increment the loop control variable.	Do something.

This might look a bit vague, but it does give you a lot of flexibility! For example, in C#, instead of adding a quantity to the loop control variable at each iteration, you might multiply its value by some number. Or instead of adding on a fixed amount you might add some number that you've read in from a file and which changes with each iteration. The test doesn't have to be a test of the value of the loop control variable. It could be a test of whether you have reached the end of the file. What this adds up to is that, by a suitable choice of the start action, test, and action at the end of each iteration, the `for` loop can effectively perform the same task as any of the other loops in Visual Basic (`For`, `For Each`, `Do`, and `While`). Alternatively the loop can work in some manner for which there is no simple equivalent in Visual Basic. The C# `for` loop really gives you complete freedom to control the loop in whatever manner is appropriate for the task at hand.

It should be noted, however, that C# also does support `foreach`, `do`, and `while` loops.

Now let's look at the syntax. The C# version of the previous `for` loop looks like this:

```
for (int I=0 ; I<3 ; I++)
{
    this.listBox1.Items.Add(Employees[I].Name);
    this.listBox1.Items.Add(Employees[I].ToString());
    this.listBox1.Items.Add(" ");
}
```

As you can see, the `for` statement itself takes three different items inside its parentheses. These items are separated by semicolons:

- ❑ The first item is the action that is performed right at the start of the loop in order to initialize the loop. In this case you declare and initialize the loop control variable.
- ❑ The next item is the condition that will be evaluated to determine whether the loop should exit. In this case your condition is that `I` must be less than 3. The loop continues as long as this condition is `true` and exits as soon as the condition evaluates to `false`. The condition will be evaluated at the beginning of each iteration, so that if it turns out to be `false` right at the start, the statement inside the loop does not get executed at all.
- ❑ In the third item is the statement that is executed at the end of each iteration of the loop. Visual Basic loops always work by incrementing some number.

Even though the syntax looks unfamiliar, once you've familiarized yourself with it, you can use the `for` loop in very powerful ways. For example, if you want to display all the integer powers of 2 that are less than 4000 in a list box, you can write this:

```
for (int I = 2 ; I<4000 ; I*=2)
    listBox1.Items.Add(I.ToString());
```

You can achieve the same result in Visual Basic, but it wouldn't be as easy; for this particular example, you might want to opt for a `while` loop in Visual Basic.

Other C# Features

You have now completed examining the code samples. The remainder of this appendix briefly examines a couple of features of C# that you need to be aware of when making the transition from Visual Basic to C#, and which haven't yet been discussed; in particular some of the C# concepts relating to data types and operators.

Data Types

As indicated, the data types available in C# do differ in detail from those available in Visual Basic. Furthermore, all C# data types have features that you would normally associate with an object. For example, every type, even simple types such as `int` and `float`, supports the calling of methods. (Incidentally, this feature does not cause any loss of performance.)

Although the types available in C# are slightly different from Visual Basic types, most of the types that you are familiar with in Visual Basic do have equivalents in C#. For example, the Visual Basic `Double` type translates to `double` in C#. The C# equivalent of the `Date` type is the .NET base class, `DateTime`, which implements a huge number of methods and properties to allow you to extract or set the date using different formats.

One exception, however, is `Variant`, for which there is no equivalent in C#. The `Variant` type is a very generic type, which to some extent exists only in order to support scripting languages that are not aware of any other data types. The philosophy of C#, however, is that the language is strongly typed. The idea is that if, at each point in the program, you have to indicate the data type you are referring to, at least one major source of runtime bugs is eliminated. Because of this, a `Variant` type isn't really appropriate to C#. However, there are still some situations in which you do need to refer to a variable without indicating what type that variable is, and for those cases C# does have the `object` type. C#'s `object` is analogous to `Object` in Visual Basic. However, `Object` specifically refers to a COM object, and therefore can only be used to refer to objects, which in Visual Basic terms means to reference data types. For example, you cannot use an `object` reference to refer to an `Integer` or to a `Single` type. In C#, by contrast, an `object0` method can be used to refer to any .NET data type, and because all data types are .NET data types, this means that you can legitimately convert anything to an `object`, including `int`, `float`, and all the predefined data types. To this extent, `object` in C# does perform a similar role to `Variant` in Visual Basic.

Value and Reference Types

In Visual Basic there is a sharp distinction between value types and reference types. Value types include most of the predefined data types: `Integer`, `Single`, `Double`, and even `Variant` (though strictly speaking `Variant` can also contain a reference). Reference types are any object, including class modules that you define and ActiveX objects. As you will have noticed through the samples in this appendix, C# also makes the distinction between value and reference types. However, C# is more flexible to the extent that it permits you, when defining a class, to specify that that class should be a value type. You do this by declaring the class as something called a *struct*. As far as C# is concerned, a *struct* is basically a special

type of class that is represented as a value rather than a reference. The overhead involved in instantiating structs and destroying them when you are finished with them is less than that involved when instantiating and destroying classes. However, C# does restrict the features supported by structs. In particular, you cannot derive classes or other structs from structs. The reasoning here is that structs are intended to be used for really lightweight, simple objects, for which inheritance isn't really appropriate. In fact, all the predefined classes in C#, such as `int`, `long`, `float`, and `double` are actually .NET structs, which is why you can call methods such as `ToString()` against them. The data type `string`, however, is a reference type and so is really just a class.

Operators

You need to know a couple things about operators in C#, because they do differ somewhat from Visual Basic operators, and this can catch you off guard if you are used to the Visual Basic way of working. In Visual Basic there are really two types of operators:

- ❑ The assignment operator, `=`, which assigns values to variables
- ❑ All the other operators, such as `+`, `-`, `*`, and `/`, which each return some value

There is an important distinction here in that none of the operators, apart from `=`, has any effect in terms of modifying any value. On the other hand, `=` assigns a value but does not return anything. There are no operators that do both.

In C#, this categorization simply does not exist. The rule in C# is that *all* operators return a value, and some operators also assign some value to a variable. You have already seen an example of this when you examined the addition assignment operator, `+=`:

```
int A=5, B=15;
A+=B; // performs an arithmetic operation AND assigns result (20) to A
```

`+=` returns a value as well as assigning the value. It returns the new value that has been assigned. Because of this you could actually write:

```
int A=5, B=15;
int C = (A+=B);
```

This will have the results that both `A` and `C` will be assigned the value 20. The assignment operator, `=`, also returns a value. It returns the value that has been assigned to the variable on the left side of the expression. This means that you can write code like this:

```
C = (A = B);
```

This code sets `A` equal to whatever value is in `B`, and then sets `C` to this same value too. You can also write this statement more simply as:

```
C = A = B;
```

A common use of this type of syntax is to evaluate some condition inside an `if` statement and simultaneously set a variable of type `bool` (the C# equivalent of `Boolean` in Visual Basic) to the result of this condition, so you can reuse this value later:

```
// assume X and Y are some other variables that have been initialized

bool B;
if ( B = (X==Y) )
    DoSomething();
```

This code looks daunting at first sight, but it is quite logical. Let's break it down. The first thing the computer will do is check the condition `X==Y`. Depending on whether `X` and `Y` contain the same data, this will either return `true` or `false` and this value will be assigned to the variable `B`. However, because the assignment operator also returns the value that has just been assigned to it, the complete expression `B = (X==Y)` will also return this same value (`true` or `false`). This return value will then be used by the `if` clause to determine whether to execute the conditional `DoSomething()` statement. The result of this code is that the condition `X==Y` is tested to determine whether the conditional statements should be executed, and at the same time you have stored the results of this test in the variable `B`.

The ternary operator

There is not space in this appendix to go over all the various operators that are available in C#. They are detailed in Chapter 2, "C# Basics," of this book. However, this section will mention the *ternary operator* (also known as the conditional operator) because it has a very unusual syntax. The ternary operator is formed from the two symbols `?` and `:`. It takes three parameters and is actually equivalent to an `IIf` statement in Visual Basic. It is used syntactically like this:

```
// B, X and Y are some variables or expressions. B is a Boolean.

B ? X : Y
```

The way it works is that the first expression (the one before the `?` symbol) is evaluated. If it evaluates to `true`, then the result of the second expression is returned, but if it evaluates to `false` then the result of the third expression is returned instead. This provides an extremely compact syntax for conditionally setting the value of variable. For example, you can write

```
string animal = (legs==8) ? "octopus" : "dog";
```

which yields the same result as:

```
string animal;
if (legs==8)
    animal="octopus";
else
    animal="dog";
```

With the Visual Basic `IIf` function, this can be achieved with:

```
strAnimal = IIf(intLegs = 8, "octopus", "dog")
```

Summary

This appendix has presented a brief introduction to C# through the eyes of a Visual Basic 6.0 programmer. You have seen quite a few differences in syntax. In general, C# syntax allows most statements to be expressed in a more compact way. You have also seen many similarities between the languages; for example in their use of classes (or class modules in VB6), value and reference types, and many of the syntactical structures. However, you have also seen how C# supports many powerful features, particularly those related to inheritance and classic object-oriented programming that are not available in Visual Basic 6.0.

Appendix A of this book contains an introduction to object-oriented programming, which is key to any serious C# development effort.

Making the transfer from Visual Basic 6.0 to C# does require a fair bit of learning, but is well worth it, because the methodology of C# allows you to code not only any application that you could have developed in Visual Basic 6.0, but also a wide range of other applications that would be too difficult, if not impossible, to design in a good, well-structured, and maintainable manner in Visual Basic 6.0. With C# you also get the added bonus of the .NET runtime and all its associated benefits.