C# for VB 6 Developers

In this appendix we will present 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 VB indicate VB 6. On those few occasions when we mention Visual Basic .NET, 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 introduction to C#, which specifically presumes knowledge of VB, and focuses on the main conceptual differences between the two languages. Our approach through much of the appendix will be to compare how you would code up the solution to a problem in VB and in C#, presenting C# and corresponding VB code together.

This does mean that our coverage of the C# language will be restricted to a basic level. We are not going to cover the more advanced features of the language – you'll need to look at Chapters 2 to 4 of Professional C# 2nd Edition for that. The emphasis will be on showing you the different methodologies involved in writing code using the C# language.

> **Note that any chapter references in this appendix refer to Professional C# 2nd Edition.**

Differences Between C# and VB

Beyond the obvious syntactical differences between the languages, there are really two main concepts that you need to become familiar with in order to progress from VB 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 part of a VB program you code up is the event handlers and any methods and so on in class modules. C# makes the complete program available to you as sourcecode. The reason for this is to do with the fact that C# can be seen, philosophically, as next-generation C++, and the roots of C++ go back to the 1960s. C++ predates windowed user interfaces and sophisticated operating systems. C++ evolved as a low-level, close-to-the-machine, all-purpose language. To write GUI apps with C++ meant that you needed to explicitly invoke the system calls to create and interact with the windowed forms. C# builds upon this heritage while simplifying and modernizing C++, with the aim being that the low-level performance benefits of C++ can

be roughly achieved with coding not being much harder than it is in VB. VB, on the other hand, is designed specifically for rapid application development of Windows GUI applications. For this reason, in VB all the GUI boilerplate code is hidden, and all the VB programmer implements are the event handlers. In C# on the other hand, this boilerplate code is exposed as part of your sourcecode.

2. Classes and inheritance: C# is a genuine object-oriented language, unlike VB, 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 VB.

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

Next we will present a VB class module that stores information about employees, and its C# equivalent. However, in this case we will start to see the real power of C#, since as we add features to the samples, we will rapidly discover that VB simply doesn't support the concepts we need to develop the class module according to the requirements we will set ourselves, and we will have to proceed in C# alone.

We will then finish off the appendix with a short tour of some of the remaining differences between VB and C# not illustrated by the examples.

Before we start, however, we need to clarify a couple of concepts: classes, compilation, and the .NET base classes. Let's take a look at each of these in turn.

Classes

Throughout the appendix we will be using C# classes quite extensively. C# classes represent precisely defined objects, which we explain in detail in Chapter 3 and Appendix A. However, for our purposes, we are better off thinking of them as the C# equivalent to VB class modules, because they are quite similar entities: Like a VB class module, a C# class implements properties and methods, and contains member variables. Like a VB class module, you can create objects of a given C# class (class instances) using the operator `new`. Behind these similarities, however, there are many differences. For example, a VB class module is really a COM class. C# classes, by contrast, are not usually COM classes, but they are always integrated into the .NET Framework. C# classes are also more lightweight than their VB/COM counterparts, in the sense that they are more defined for performance and give a smaller performance hit when instantiated. However, these differences will largely not affect our discussion of the C# language here.

Compilation

You are almost certainly aware that the computer never directly executes code in any high level-language, whether it is VB, C++, C, or any other language. Instead, all sourcecode is first translated into native executable code, a process usually known as **compilation**. When you are debugging, VB offers the option of just running the code straight away (meaning that each line of VB code is **interpreted** as the computer comes to execute 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 leads to much higher performance when running, and is therefore the only option permitted in C#.

In C#, compilation is done in two stages, with the first stage being into the so-called Intermediate Language (IL). This is the stage that we will informally refer to as compilation. The second stage, converting into native executable code, may be done at run time, but is a much simpler stage so it doesn't lead to such significant performance problems. It's also 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 sourcecode to native executable. Although the existence of IL is something that you need to bear in mind, it won't affect any of our discussion in this appendix, as it doesn't really affect C# language syntax.

The .NET Base Classes

VB 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. VB also relies on the presence of ActiveX controls to provide the standard controls that you put on your form – listboxes, buttons, textboxes, and so on.

C# also relies on extensive support for these sorts of areas – but in the C# case, the 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 normal controls, classes that perform conversions, classes that perform date-time and file system access, classes that access the Internet, and many more. We won't go into the .NET base-class library in detail here, but we will frequently refer to it. Indeed, C# is so well integrated with the .NET base classes that we will find 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 VB and C#; VB's system functions are specific to VB, whereas the respective functionality of C# is provided by the .NET base classes, accessible to any .NET-aware language.

Conventions

In this appendix we will frequently be comparing code in C# and Visual Basic. In order to make it easier to identify code in the two languages, we will present C# code in this format:

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

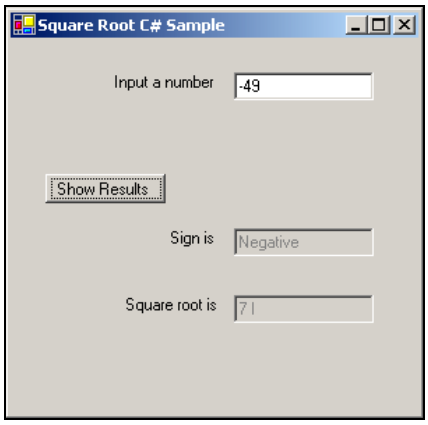
However, all VB code will be presented in this format:

```
' VB code is presented with a white background
```

Example: The Square Root Form

In this section, we are going to examine a simple application called `SquareRoot`, which we have developed in both VB and C#. The application is a simple dialog box, which invites 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 simply means taking the square root of minus the number and adding 'i' after it. The C# version of the example looks like this. The VB version is pretty much identical in appearance except that it has a standard VB icon in place of the .NET windows forms icon in the top left corner:



SquareRoot VB Version

To get this application working in Visual Basic, we simply need to add an event handler for the event of clicking the button. We have given the button the name `cmdShowResults`, and the textboxes 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 (NumberInput < 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 the only bit of VB code that we need to write.

SquareRoot C# Version

In C# we also need to write an event handler for the event of the button being clicked. We have kept the same names for the button and the textboxes, 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 almost certainly 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 there are a huge number of differences in the syntax between the two languages. Over the next couple of pages, we are going to compare these samples to see what we can learn about C# syntax in the process. In the process, we will also uncover some of the differences between the basic methodologies of C# and VB.

Basic Syntax

In this section we will examine the two `SquareRoot` programs to see what they teach us about C# syntax.

C# Requires All Variables to be Declared

If we start with the first line of VB code, we encounter the `Option Explicit` declaration. This statement has no counterpart 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` explicitly.

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 code. Standard advice in VB 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

Since commenting code is always important, the next thing we 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 VB we use an apostrophe to denote the start of a comment, and the comment lasts until the end of

the line. Our C# comments in the code work the same way, except they start with two forward slashes: `//`. Just as for VB comments, we 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, which each have a slightly different effect.

A comment may be also be delimited by the sequences `/*` and `*/`. In other words, if the compiler sees a `/*` sequence, it assumes all the following text is a comment until it sees a `*/` sequence. This allows you to have long comments that stretch over several lines:

```
/* 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 way is very similar to the first way. However, now we 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 sourcecode as a separate XML file. That's why the example above appears to have a fairly formal structure to the actual comment text: The structure 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). 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 will even check that the documentation matches the method signatures and so on.

Statement Separation and Grouping

The most visible difference between the above C# and VB codes will almost certainly be the presence of all the semicolons and the curly braces in the C# code. Although this can make C# code look daunting, the principle is actually very simple. VB 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 (reformatted from part of the above sample) is also perfectly valid C# code:

```
        this.txtSign.Text =

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

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

The braces are used to group statements together into what are known as **block statements** (or sometimes as **compound statements**). This is a concept that doesn't really exist in VB. In C#, you can group together 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 C# code above there is no explicit indication of the end of the `OnClickShowResults()` method (C# has methods, written in text with the `()` appended, whereas VB has functions and subs). VB 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 VB 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 VB 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 above samples illustrates the same point. In VB, we need an `EndIf` 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 we want to put more than one statement into either clause, as is the case in the above example, we 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.

> **Unlike VB, 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`. We'll encounter this in the second example later in the appendix.

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

Methods

Let's compare the syntax that VB and C# use to declare the part of the code that will handle the event: In VB:

```
Private Sub cmdShowResults_Click()
```

and in C#:

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

The first point we should make is that the VB version declares a sub, whereas the C# version declares a method. In VB, code is traditionally grouped into subs and functions, with the concept of a procedure being either. Additionally, VB 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 (this is connected with the fact that, as we will see later, everything in C# is part of a class). In C# there is no separate concept of functions and subroutines – those terms don't even exist in the C# language specification. In VB, 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 here).

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 VB we indicated that we were declaring a sub 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 we are declaring a method, since 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 VB `Sub`, the C# method declaration above is preceded by the keyword `private`. This has roughly the same meaning as in VB – it prevents outside code from being able to see the method. We'll examine just what exactly we mean by 'outside code' later.

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

We'll tackle the name first. The name of the event handler in VB is supplied for you by the VB IDE. The reason that VB knows that the `Sub` is the event handler for when the button gets clicked is because of the name, `cmdShowResults_Click`. If you renamed the sub, then it wouldn't get called when you clicked the button. However, C# doesn't use the name in this way. In C#, as we'll see soon, there is some other code that tells the compiler which method is the event handler for this event. That means we can give the handler whatever name we 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) with something called **Pascal** casing, which means that words are joined together with their first letters capitalized. Using underscores in names in C# is not recommended, and we've chosen a name in accordance with these guidelines: `OnClickShowResults()`.

Now for the parameters. We won't worry about the details of these parameters in this appendix, but we'll just say 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 concerned (for example, for a mouse move event the parameters might indicate the location of the mouse pointer).

Variables

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

1234

```
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 VB. `float` is C#'s equivalent to `Single`. It's probably easier for us to understand what's going on if we split up the C# version into two lines. The following C# code has exactly the same effect as the line above:

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

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

Declarations

The obvious syntactical difference between C# and VB, 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 VB 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 we 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#, incidentally, plays a similar role to `Object` in VB – it indicates something for which we are choosing not to specify its type. However, C#'s `object` is much more powerful than VB's `Object`. In C#, `object` also replaces VB's `Variant` data type. We'll look at `object` later on. We won't really cover `System.EventArgs` in any detail in this appendix. It's a .NET base class, and it has no equivalent in VB.

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 quite a complicated-looking expression, which we will examine soon. But to take two simpler examples:

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

While we are on the subject, we ought to mention a couple of other points about variables.

No Suffixes in C#

VB 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 explicitly indicate the data type.

No Default Values for Local Variables

In the VB code sample, the variable `sngNumberInput` will be assigned the default value of 0 when it is declared. This is actually a waste of processor time since we 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 VB. 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 VB we use `Let`, while for objects VB 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 above is:

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

> **Remember that in C#, variables are always assigned using the syntax <VariableName>=<Expression>;**

Classes

Now we come to look at what's going on in the expression used to initialize the variable `sngNumberInput` in the `SquareRoot` sample. The C# and VB examples are both actually doing exactly the same thing: grabbing the text from the `txtNumber` textbox; but the syntax to do this looked rather different in the two languages:

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

and:

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

Getting the value out of the textboxes is quite similar in both cases. The only difference for that part of the process is the purely syntactical one that VB uses the keyword `Me` while 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 VB). In C# we could equally well have written:

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

The more interesting part is how the string retrieved from the TextBox is converted to a `float` (or `single`), because this illustrates a fundamental point of the C# language, which we've hinted briefly at earlier:

Everything in C# is part of a class.

In VB, the conversion is carried out by a function, `CSng`. However, C# does not have functions in the way that VB has. 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 we need to call the `Parse()` method is `float`. Yes, I did say that right. Up until now we have treated `float` as simply being C#'s equivalent to VB's `Single`. 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 we should point out that `int` and `float` are special types of class known in C# as **structs**. The difference is not important for our code here, but we will explain it later).

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

Also, to be strictly accurate, we should point 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 we come to the main part of the event handler. The "If" statement. Recall that the VB version looks like this:

```
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
```

while this is the C# version:

```
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();
      }
```

In fact, we have already explained the biggest syntactical difference here: that each part of the `if` statement in C# must be a single statement, hence, if we need to conditionally execute more than one statement we must combine them into a single block statement. In C#, if there is only one statement to be conditionally executed, we don't need to form a block statement. For example, if we skipped setting the text in the `txtSign` textbox in the above code we could have written:

```
      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();
```

There are some other differences in syntax that we should remark on. In C#, the parentheses around the condition to be tested in an `if` statement are compulsory. In VB we could have written:

```
If NumberInput < 0 Then
```

Trying the same trick in C# would result in a compilation error immediately. In general, C# is much more precise about the syntax expected than VB. Also, notice that when we test whether `NumberInput` is zero, we use two equal signs in succession for the comparison:

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

In VB, the symbol = doubles up for two purposes: it is used for assigning values to variables, and it is used 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 out when making the transition from VB to C#:

> **`else if` is two words in C# whereas it is one word in VB: `ElseIf`.**

Calculating Square Roots: Another Class Method

Given our 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 VB's `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 we can abbreviate to just `Math` in our code.

You'll have also noticed that, in the sample code, when dealing with the condition of the number input being exactly zero, we don't specify the `this` keyword in the C# code:

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

and in the corresponding VB code we don't specify `Me` explicitly either. In C#, just as in VB, you don't have to explicitly specify `this` (`Me`) unless, for any reason, the context is unclear. We've chosen not to do so here just to illustrate the point.

Strings

When we display the square root of a negative number, we have our first bit of string processing:

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

You will notice from this code that in C# concatenation of strings is done using the symbol + rather than `&`. You will also notice that we 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 VB 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 their own custom method.

Extra Code in C#

We have now completed the comparison of the event handler routines in C# and VB. In the process, we've learned a lot about the syntactical differences between the languages. In fact, we have now learned most of the basic syntax that C# uses to piece statements together. We have also had our 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, 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 actually a lot more code in the C# sample than simply an event handler. For the VB version of the `SquareRoot` sample, the code for the event handler that we have presented here represents the complete total of all the sourcecode in the project that is visible to us. However, in the C# version of the project, this event handler is just one method in a large sourcecode file that contains a much other code.

The reason why there is so much additional code in the C# project is to do with the fact that the VB IDE hides a lot of what's going on in your program from you. In VB, all we 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 VB, you don't have access to any of the code that does this. By contrast, C# takes a completely different philosophy, and leaves all this code in the open. That might make your sourcecode 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 there will be an identifiable location in the executable code that the computer knows is where it has to start running the code from. In other words, an instruction that it executes first. 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, perhaps depending on the values contained in certain variables. 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 hits a command that tells it to terminate 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) but 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 VB executable program. In VB 6, what you write is essentially a set of event handlers – a set of subs, each of which you know will be 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 if and when the client code chooses to call it.

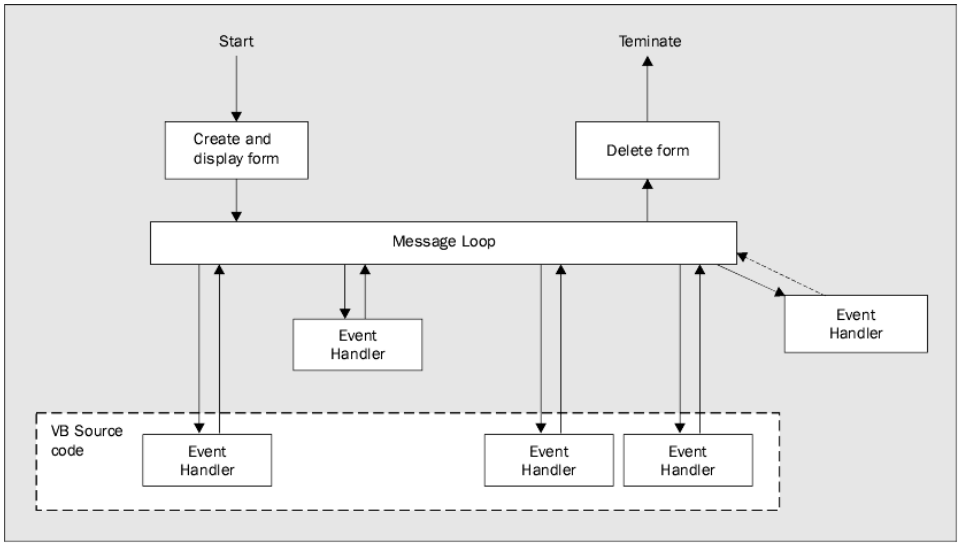
> *Note also that in VB, `Sub Main` does exist, and acts as the entry point to a program, but unlike the `Main()` method of C#, `SubMain` is optional.*

In order to see how we can relate the two programming ideas, let's 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 we mentioned before, as now we are confining our attention to Windows GUI apps (in other words, not consoles, services, and so on).

As usual, execution starts at some well-defined point. The commands executed will 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 VB 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 sourcecode for this in VB, but a VB application is still able to respond correctly when the user attempts to resize the form because the VB IDE has invisibly added event handlers to your project to correctly handle this situation.

Whenever an event occurs, Windows wakes the application up 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 will put itself to sleep again, once again telling Windows to wake it up when another interesting event happens. Finally, assuming nothing goes disastrously wrong, at some point Windows will wake up the application and inform it that it needs to shut down. At that point, the application will take any appropriate action – for example, displaying a message box asking the user if they want to save a file – and will then quietly terminate itself. Again, most of the code to do this has been quietly added to your project behind the scenes by the VB IDE, and you never get to see it.

The thread of execution in a typical Windows GUI application looks rather like this:



In this diagram, the box with a dashed border indicates the part of execution that the VB IDE lets you get access to, and for which you can write sourcecode: some of the event handlers. The rest of the code is inaccessible to you, though you can to some extent specify it through your choice of type of application when you first ask VB to create a project. Recall that when you create a new project in VB, you get a dialog box asking you what type of application you want to create – Standard EXE, ActiveX EXE, ActiveX DLL, and so on. When you make that selection the VB IDE uses your choice to generate all the appropriate code for the part of the program that is outside the dashed box in the above diagram. 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.

Now, I said earlier that in C# you get access to all the code that does everything but I should qualify that. All the nitty-gritty details of things, like what's going on inside the message loop, are well 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 the whole program running, 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 sourcecode 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. One other point I should stress is that you don't need to write any of this code yourself. When you use Visual Studio .NET 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 .NET will still write all the background code for you. The difference is that Visual Studio .NET writes this background code as source C# code, which then becomes code that you can edit directly.

Doing things this way does, as we've remarked, bloat your sourcecode. 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 project in C#. Whereas in VB, 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 program that run on Windows, and we see these different types of program throughout the book. In this appendix however, we will concentrate exclusively on classic Windows GUI applications.

The C# Code for the Rest of the Program

In this section, we will examine the rest of the code for the `SquareRoot` sample. In the process we will

learn a bit more about classes in C#.

> *The C# `SquareRoot` sample was created in Visual Studio .NET, and the VB one was created in the VB 6 IDE. However, the code presented here isn't quite what Visual Studio .NET generated for us. Apart from adding the event handler, I've made a couple of other tweaks 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 .NET does when it creates a project for you.*

The full text of the sourcecode is quite long. We will not present it here – it can be found in the accompanying document, `VBToCSharp_CSharpSource.pdf`, and can be browsed simultaneously with the text.

Namespaces

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

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

The `SquareRootForm` class is going to hold almost all the code – all the methods and so on, with a small amount of code being contained in another class called `MainEntryClass`. Remember that it's easiest here to think of a class as being like a VB class object – though we can see here one difference is that we actually see the sourcecode that begins the declaration of the class. In VB, the VB IDE just 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 VB, 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 have almost certainly got a large 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, then 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 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 formally do anything to create a namespace, other than just declare it in your code in the way we've done in our sample above. The code presented above means that the full name of the class we have defined is not `SquareRootForm`, but `Wrox.ProCSharp.VbToCSharp.SquareRootForm`. It is extremely unlikely that anyone else will write a class with that full name. On the other hand, if we 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 VB 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, were making 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

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, the 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

            }
         }
      }
   }
}
```

In this code we have added the closing curly braces 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 we will be defining a couple of textboxes. A textbox is represented by the class `System.Windows.Forms.TextBox`. If we had to write that in our code every time we wanted to refer to a `TextBox`, our 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 we have not specified a namespace. Now we can simply write `TextBox` whenever we want to refer to that class. It is usual to start any C# program with a number of `using` directives that bring in all the namespaces we are going to be using into the set of namespaces searched by the compiler. The namespaces specified in the above code are all namespaces that cover various parts of the .NET base-class library, and so allow us to conveniently use various .NET base classes.

The Class Definition: Inheritance

Now we 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 we 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 we need to bring in that other important concept that we mentioned earlier you need to learn in order to understand C# programming: i**nheritance**.

What the above syntax does is tell the compiler that our `SquareRootForm` class inherits from the class `Form` (actually `System.Windows.Forms.Form`). What this means is that our class gets not only any methods, properties, etc. that we define; it also gets 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, our own class gets all these features straight away, 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 used interfaces much before, inheritance will not be new to you, since you will know that interfaces can inherit from each other. What we have here, however, is much more powerful than interface inheritance. When a COM interface inherits from another interface, all it gets is the names and signatures of the methods and properties. That is, after all, all that an interface contains. However, a class contains all the code that implements these methods and so on as well, just as in VB 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, normally use the concept extensively, but it was not supported in VB. (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 we 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 sourcecode of the base class in order to derive from it. For obvious commercial reasons, Microsoft is keeping the sourcecode of `Form` to itself. The fact that the compiled library is available in the form of an assembly is sufficient for us to be able to inherit from the class, taking those methods we want and overriding those that we don't.

Program Entry Point

We will now jump to near the bottom of the code for the example, to examine the main program entry point. That is the `Main()` function, reproduced below:

```
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 sourcecode – 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).

Since `Main()` has to be in a class, we've put it in one: a class called `MainEntryClass`. This class doesn't contain anything else, but that's not necessarily the case – it's quite 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 we said earlier that static methods can be run without actually creating an object of the class first. Since the very first thing that happens when the program is run is that `Main()` is be 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 we examined. However, it is prefixed by the word `[STAThread]` in square brackets. `STAThread` is an example of an **attribute** – another concept that has no parallel in VB sourcecode.

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. We won't go into details of threading models here, but we will say that writing `[STAThread]` in the C# sourcecode has a similar effect to selecting the threading model under Project Properties in the VB IDE, although in VB you can only do this for ActiveX DLL and ActiveX Control projects. Also, note that the analogy is only approximate, because the C# attribute selects a .NET threading model, not a COM threading model.

Incidentally, this comparison shows once again the different philosophy of C# compared to VB. In VB, the threading model is there and needs to be specified, but it is all but hidden by the VB IDE, so you can't get to it in the VB sourcecode – hence in VB you have to access it through the project settings.

Instantiating Classes

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

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

We obviously can't compare this with the corresponding VB code, since the corresponding VB commands aren't available as sourcecode, but we can do a comparison – if we imagine that in some VB code we are going to create a dialog box. In VB, the way you would do that would look something like this:

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

In this VB code, we first declare a variable that is an object reference: `SomeDialog` will refer to a `MyDialogClass` instance. Then, we actually instantiate an object using VB's `New` keyword, and set our variable to refer to it.

That's exactly what is going on in the C# code too: we declare a variable called `TheMainForm`, which is a reference to a `SquareRootForm` object, then we use the C#'s `new` keyword to create an instance of `SquareRootForm`, and set our variable to refer to it. The main syntactical difference is that C# allows us to combine both operations into one statement, in the same way that we were previously able to declare and initialize the `NumberInput` variable in one go. You will also notice 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, we don't pass in any parameters, but we still need the parentheses.

C# Classes

Up until now we've said that C# classes are similar to class modules in VB. We've already seen one difference in that C# classes allow static methods. The code for the `Main()` method above now highlights another difference. If we were doing something like this in VB, we would also need to set the object created to `Nothing` when we'd finished with it. However, nothing like that appears in our C# code, because in C# it is not necessary to do this.

The reason for this difference is to do with the fact that C# classes are more efficient and lightweight than their corresponding VB counterparts are. VB class objects are really COM objects, which means they each 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 VB, if you don't set your object reference to `Nothing` when you've 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 hang around 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 something called 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 means the same thing as `Nothing` in VB. Hence where in VB you would write:

```
Set SomeDialog = Nothing
```

in C# you would write something like:

```
TheMainForm = null;
```

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

Entering the Message Loop

We will now consider the final statement in the `Main()` method:

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

This statement is the one that enters the message loop. What we 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 will handle all events. `Run()` exits when an event instructing the form to terminate has occurred and been handled.

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

One piece of syntax in the above statements that you might find surprising is that we use parentheses when calling the `Run()` method, even though we are not using any return value from this method, and hence we are doing the equivalent of calling a VB sub. In this situation, VB 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

We have now seen how C# enters a message loop, but we have not yet seen the process of displaying and creating the form itself, and we have also been rather vague about the calling of the event handlers. We have indicated that Windows calls event handlers, such as our `OnClickButtonResults()` method. But how does Windows know that that is the method to be called? We can find the answers to

those questions in the SquareRootForm class definition, and in its base class, Form.

First we 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 VB variable that has form scope, or alternatively as being like a VB 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. We can see clearly the three textboxes and the button. There are also four labels, corresponding to the areas of text on the form. We won't be doing anything with these labels so we haven't bothered 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 something called a constructor. A constructor in C# is a bit analogous to such VB subs 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.

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

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

Notice that because this is a constructor rather than 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 (recall that earlier we said that 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, we don't have any parameters here – we will see examples of constructors that take parameters in the Employee code sample later in the appendix.

In this case the constructor just calls a method, InitializeComponent(). This is really because of Visual Studio .NET. Visual Studio .NET has all the same kinds of features as the VB 6 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 sourcecode, Visual Studio .NET has to be able to read the sourcecode 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 we 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 above code is a set of calls to actually instantiate all the controls on the form. This bit of code doesn't really contain any new pieces of C# syntax that we haven't already encountered. 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 and the textbox 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 above 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 we need to create and initialize a `Point` object that holds the correct coordinates. This is the first time that we've seen a constructor that takes parameters – in this case the horizontal and vertical coordinates of the `Point`, and hence of the control. If we'd wanted to translate one of these lines into VB, assuming we'd defined some VB class module called `Point`, and we had a class that had such a property, the best we 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 we will look at the same commands for the button. In this case, we see the same kinds of properties being set up, but here there is one other thing that needs to be done: we need to tell Windows to call our event handler when the button is clicked. The line that does this is highlighted:

```
            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`, which will be raised when the user clicks on it. We need to add our event handler to this event. Now C# doesn't allow us to pass names of methods around directly – instead we have to wrap them up into something called a delegate. The details of this are beyond the scope of this appendix – they are covered in Chapter 4 of the book, but this is done to ensure type safety, and is the reason for the new `System.EventHandler()` text in the code. Once we've wrapped the name of the event handler up, we add it to the event using an operator `+=`, which we will discuss 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 VB 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;
```

Although we won't go into the details in this appendix, 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. Details of these operators are in Chapter 2.

In the `SquareRootForm` sample, we 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". You might be a bit surprised at seeing an operator like += being applied to something that isn't a simple numeric type like `int` or `float`, but this actually illustrates an important point about operators in C# compared to ones in VB:

> **Operators like +, –, *, and so on in VB only really have a meaning when applied to numeric data. But in C#, they can be applied to any type of object.**

The above statement needs to be qualified a bit. In order to be able to apply these operators to other types of object, you have to first tell the compiler what these operators mean for other types of object – a process known as **operator overloading**. It works a bit like this. Suppose you wanted to write a class that represented a mathematical vector. Something that in VB you would code up as a class module, and would then let you write:

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

In mathematics, it's possible to add vectors together, which is where operator overloading would come in. But VB 6 doesn't support operator overloading, so instead in VB 6 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 VB, that's the best you could do. However, in C#, if you defined a `Vector` class you could 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 would be able to write:

```
// V1, V2 and V3 are Vectors

V3 = V1 + V2;
```

In Chapter 3 of Professional C# 2nd Edition, we detail the code for overloading such a `Vector` class, and discuss 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 towards 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 us do this using the intuitive syntax using the `+` (and `+=`) operators. You can also use – or `-=` to remove a handler from an event.

Summing Up

We've really got as far as we can go with the `SquareRootForm` code samples. There is a lot more C# code that we haven't examined in the C# version of this application, but this extra code is largely to do with setting up the various other controls on the form, and doesn't introduce any new principles, so we're not going to examine it further.

Up to now, we've got a flavor of the syntax of C#. We've seen how it lets you write statements in a way that is often much shorter than the corresponding VB code. We have also seen the way that C# places all the code in the source file, unlike VB, 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. We've also had our first hints at the concepts behind inheritance.

However, what we 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 VB code to do the same sort of thing. We are going to see an example of this in the next code sample, in which we write a couple of classes that illustrate the kinds of things we can do with inheritance.

Example: Employees and Managers

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

The VB Employee Class Module

The following code represents our attempt to code up an `Employee` class module in VB. The class module exposes two public properties, the `EmployeeName` and the `Salary`, as well as a public method, `GetMonthlyPayment`, which 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 we want to allow for the possibility of other things getting added on to what the company pays its employees (like 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 we'd probably be writing something more complex than this, but this class will be sufficient to illustrate the concepts I want to cover. In fact, we already have a problem with this VB class module. Most people's names do not change very often, which is why we have made the `EmployeeName` property read-only. That still leaves us needing to set up the name in the first place. For that, we've had to add a `Create` method, which sets the name and the salary. That means that the process of creating an employee object is going to look like this:

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

This is workable but messy. The problem is that we 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 – but that is exactly what we have in the above code for the brief instant between instantiating `Britney` and initializing the object in the code above. As long as you always remember to call `Create`, you are fine, but there is a potential source of bugs here.

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

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

which is a lot neater and less prone to bugs. Of course, we could overload the constructor to only supply a name, for example. Notice by the way, the '`M`' appended to the salary. This is because the C# equivalent to VB's `Currency` type is something called `decimal`, and '`M`' appended to a number in C# indicates we want the number interpreted as a `decimal`. We don't have to supply it, but it makes for a useful extra compile-time check.

The C# Employee Class

Bearing in mind the above comments, we can now present our first definition of the C# version of `Employee` (note that here we have just shown 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, we first see a couple of private variables, the so-called member fields corresponding to the member variables in the VB class module. The field `name` is marked `readonly`. We'll see what exactly this means soon. Roughly speaking, it 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 we prefix the names of variables with some letters that indicate their type (`mStr`, `mCur` and so on). It's not considered as 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.

There is also a constructor in our `Employee` class, a couple of properties, `Name` and `Salary`, and two methods, `GetMonthlyPayment()` and `ToString()`. We will examine all of these next.

> *Notice by the way, 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 we've named the properties and fields here corresponds to the usual convention in C# and shows how we can actually take advantage of case sensitivity.*

The Employee Constructor

After the field declarations in the above code, we get to a "method" that has the same name as the class, `Employee`, which tells us that it is a constructor. However, this constructor takes parameters, and does the same thing as the `Create` method in the VB 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 we've resolved this using the `this` reference to mark the fields. We could have given the parameters different names instead, but the way we've done it is still clear enough, and means that the parameters keep the obvious simple names that correspond to their meanings. It's also the conventional way of dealing with this situation in C#.

We can now explain the precise meaning of the `readonly` qualifier on the name field:

```
         private readonly string name;
```

If a field is marked as `readonly` then 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 we 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 once it has been set. It wouldn't be possible to do anything like this in VB because VB doesn't have constructors that take parameters, so class-level variables in VB have to be initialized via methods or properties that are called after the object has been instantiated.

Incidentally, this constructor doesn't just allow us to supply parameters to initialize an `Employee` object: it actually forces us to do so. If we tried to write code like this:

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

then this code would actually not compile. The compiler would raise an error because, in C#, a constructor must always be called when a new object is created. However, we 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. We'll see how to do this later in the chapter. However, for this particular class, our one constructor is quite adequate.

Properties of Employee

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

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

In VB, the compiler knows that we are defining a property, because we 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 we were defining a method, this would be an opening parenthesis signaling the start of the parameter list, while for a field this would be a semicolon, marking the end of the definition.

One other point to note is that the definitions of the get and set accessors do not contain any parameter lists. There's no point – we know from the fact that `Salary` is a `decimal` 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!*

Just as with VB, if you want to make a property read-only, you simply omit the set accessor, as we have done for the `Name` property:

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

Methods of Employee

There are also two methods: `GetMonthlySalary()` and `ToString()`.

`GetMonthlySalary()` needs little comment, since we have covered most of relevant C# syntax already. It simply takes the salary, divides it by 12 to convert from annual to monthly salary, and returns the result:

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

The only new piece of syntax here is the `return` statement. In VB, we 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#, we achieve the same result by appending a parameter to a `return` statement (without parentheses). `return` in C# also specifies that we are exiting from the function, so the C# statement:

```
      return salary/12;
```

is actually equivalent to the following VB 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 us to do the same thing in our 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();
}
```

Our override here simply displays the name and the salary of the employee. One new piece of syntax is that we have specifically declared the method as `override`. C# requires that you explicitly mark method overrides in this way, and will raise a compilation error if you don't. This eliminates the risk of any potential bugs where, for example you accidentally override a method without realizing it, perhaps because you didn't realize that a method of that name was already present in the base class.

We have now completed writing the `Employee` class in both VB and C# – and so far, although there is a bit of awkwardness about constructing and initializing an `Employee` instance in the VB version, both languages have coped reasonably well with our requirements. However, one of the aims of this appendix is to show you why C# can be so much more powerful than VB 6 in some situations. So it's about time we started demonstrating some useful C# code where it would be very difficult or impossible to do the same thing in VB 6. That's what's going to happen from now on. We're going to start adding some features to the C# version of our sample, which are going to leave the VB version way behind and quite unable to catch up. Let's start with a `static` field and property.

Static Members

We have 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 VB. 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 VB.**

To illustrate how static members work and why you would use them we are going to imagine that we would like our `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 VB, 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 we 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. Lets declare such a field, `companyName`:

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

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

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

Of course, merely declaring this field isn't quite enough. We also need to make sure it is initialized with the correct data. Where should we do that? Well, clearly not in our constructor – the constructor gets

called every time we create an `Employee` object, whereas we 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, then it will be executed just once. It's not guaranteed exactly when it will execute, but it will be before the first time any client code attempts to access the class. This will normally be when the program first starts up. Let's add a static constructor to the `Employee` class:

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

As usual, we 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 not marked as either `public` or `private` because it will not be called by any other C# code, but by the .NET runtime. So, just for the static constructor, we don't need any access modifier.

In our example, we have implemented the static constructor by hard-coding in a company name. More realistically, we 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 we can legally assign a value to it. We have one last thing to do, which is to define a public property that lets us access the company name.

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

The `CompanyName` property has also been declared as static, and we 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 we 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, we 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 VB 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.

Other examples where you might use static class members are:

- You might choose to implement a `MaximumLength` property for our `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, in order 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

We are now going to look in a bit more detail at how implementation inheritance works, using an example. We are going to suppose that, a year or so after we have shipped our software package it's time for the next version. One point that our 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 our `GetMonthlyPayment()` method doesn't give the complete information for managers. The practical upshot of this is that there is a need to add some way of dealing with managers, too.

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

If we were coding in VB, how would we set about upgrading our software? There are two possible approaches, but both of them have severe disadvantages.

- We could write a new class, `Manager`
- We could modify the `Employee` class

Writing a new class is probably the approach that what would give us the least amount of work, since we'd probably start by simply copying and pasting all the code for the `Employee` class module and then modifying our copy of the code. The trouble is that `Employee` and `Manager` have an awful lot of code in common, such as all of the code around 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, for some reason. 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 us with two separate objects, as well as an awkward, indirect, syntax for calling employee methods and properties (`objManager.objEmployee.Name` and so on).

If we opt for modifying the `Employee` class module, then we could, for example, add an extra field, a `Boolean`, that indicates whether this `Employee` is a manager or not. Then, at relevant parts of the code, we would test this `Boolean` in an `If` statement, to check what to do. This solves the problem of having two unrelated classes – we are back to just one class. However, it introduces a new difficulty: I specifically said earlier that we 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 we really want to have to dive in and start pulling working code to bits, with all the associated risk of introducing new bugs?

Bluntly, we have reached a point at which VB cannot offer any satisfactory solutions. From the title of this section you won't be surprised to learn that C# does offer a way round this problem, through inheritance.

We have already seen that inheritance involves adding or replacing features of classes. In our previous example, our `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. In the `Employee` example we are going to demonstrate both adding and replacing features of a base class. We will define a `Manager` class, which is derived from `Employee`. We will add a field and property that represent the bonus, and replace the `GetMonthlyPayment()` method (for completeness, we'll also replace `ToString()` so that it displays the bonus as well as the name and salary). This all means that we will get a separate class. But we won't need to duplicate any code, and we won't need to make any big changes to the `Employee` class either. You might think that we still have a problem of two different classes – which makes it more difficult to write client code, but as we will see later, C# has got an answer for this is as well.

Inheriting from the Employee Class

Before we define the `Manager` class we need to make one small change to `Employee`:

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

The change we have made is to make the `GetMonthlyPayment()` method `virtual`. Roughly speaking, this is C#'s way of saying that this is a method that in principle can be overridden.

> *You might think that this means we are changing the base class, which knocks out our 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 VB approach we 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 what methods 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 we really would have been able to add the `Manager` class without making any changes whatsoever to the `Employee` class.*

The Manager Class

We 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 we 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, we will look in detail at the overridden methods and the new constructors, as these will illustrate important language features of C#.

Method Overrides

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

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

Our override also contains a call to the base-class version of this method. This uses a new keyword, `base`. `base` works in the same way as `this`, except that it specifically indicates that we want to grab a method, or property etc, from the definition in the base class. If we had wanted, we could alternatively have implemented our override of `GetMonthlyPayment()` like this:

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

but I wanted to show off the `base` keyword, so I chose not to do it this way! Incidentally, one thing that we could not do is this:

```
        public override decimal GetMonthlyPayment()
```

```
        {
           return (salary + bonus)/12;   // wrong
        }
```

This code looks almost exactly like the previous version, except that we 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 we 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 then 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 VB class modules: because 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

We 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 we need to specify when we create a `Manager` instance
- Unlike methods, properties, and fields, constructors are not inherited by derived classes

In fact, we have added two constructors. This is because we have decided to assume that the manager's bonus normally defaults to $100,000 if it is not explicitly specified. In VB we can specify default parameters to methods, but C# doesn't allow us to do that directly. Instead, C# offers a more powerful technique that can achieve the same effect, **method overloads**. Defining two constructors here will allow us to 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 we notice about this constructor is a call to the base class constructor using a slightly strange syntax. The syntax is known as a **constructor initializor**. What happens is that any constructor is allowed to call one other constructor before it executes. This call is made in a constructor initializor with the syntax shown above. 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. The syntax for the constructor initializor requires a colon, followed by one of the keywords `base` or `this` to specify which class we are calling the second constructor from, followed by the parameters we are passing on to the second constructor.

The constructor shown above 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 really the responsibility of the Employee class rather than the Manager class, so what we 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 we saw earlier, the Employee constructor will simply use these parameters to initialize the name and salary fields. Finally, we 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 we'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 we set up the value of the default parameter, and then pass everything on to the three-parameter constructor. Of course, in turn, the three-parameter constructor will call the base-class constructor to deal with the name and salary parameters. You might wonder why we 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 we change how we 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 would be considered better.

Method Overloading

The fact that we have supplied two constructors for the Manager class illustrates the principle of method overloading in C#. All that method overloading involves is that a class has more than one method that have the same name, but have different numbers of parameters. We have illustrated overloading for constructors, but exactly the same principles apply for all methods.

> **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 that should be called. In the case of creating a manager object, since one constructor takes three parameters and the other only takes two, the first thing the compiler will examine is the number of parameters. 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 called – 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 there are several overloads available, but the compiler is unable to find one that is suitable, it will raise a compilation error. For example, if you wrote:

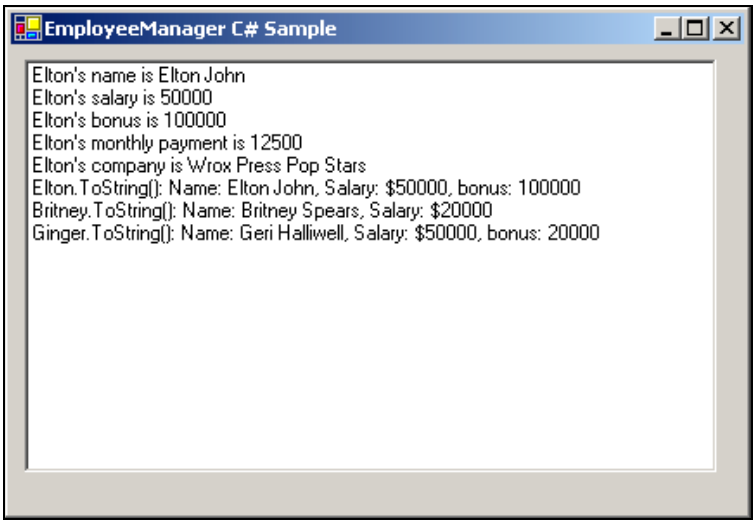
```
Manager SomeManager = new Manager (100, 300000.00M, 50000.00M);   // wrong
```

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.

Finally, we will note that, as previously remarked, C# does not allow methods to take default parameters in the way VB does. However, it is very easy to achieve the same effect using method overloads, as we have 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 we have completed defining the `Employee` and `Manager` classes, we can write some code that uses them. In fact, if you download the sourcecode for this project from the Wrox Press web site, you will find that we defined these two classes as part of a standard Windows Forms project, quite similar to the `SquareRoot` sample. In this case, however, the main form just has one control – a listbox. We use the constructor of the main form class (a class called `MainForm`) to instantiate a couple of instances of `Employee` and `Manager` objects, then display data for these objects in the listbox. The results look like this:



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 we have learned up to now, apart from one little oddity, that one of the `Manager` objects, `Elton`, is being referred to by an `Employee` reference instead of a `Manager` reference. Will explain how this works next.

References to Derived Classes

Let's have a closer look at that `Manager` class that is referenced by a variable declared as a reference to `Employee`:

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

This is actually perfectly legal C# syntax. The rule is quite simple: if you declare a reference to some 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 above example, we call `Elton.Name`, `Elton.Salary`, and `Elton.GetMonthlyPayment()`. This is fine. The fact that `Employee` implements all these members guarantees that any class derived from `Employee` will too. So it doesn't matter if a reference points to a derived class – we 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 we use when we call the `Bonus` property against `Elton`: `((Manager)Elton).Bonus`. In this case, we need to explicitly convert `Elton` to a `Manager` reference because `Bonus` is not implemented by `Employee`. The compiler knows this, and would raise a compilation error if we tried to call `Bonus` through an `Employee` reference. That line of code is really a shorthand way of writing:

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

As in VB, conversion between data types in C# is known as casting. We can see from the code above that the syntax for casting involves placing the name of the destination data type in parentheses before the name of the variable we are attempting to cast. Of course, the object being referred to must be of the correct type in the first place. If with this example, we wrote:

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

the code would compile correctly, but when we ran it, we would get an error, because the .NET runtime would see that `Britney` is just an `Employee` instance, not a `Manager`. 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. (We can't permit it because if we did, what would happen if we attempted to call the `Bonus` property through such a reference?)

Incidentally I've not said anything about what raising an error at run time involves. In fact, C# has a very sophisticated mechanism for this, known as exceptions, which we will briefly look at later in this appendix.

Because VB doesn't support implementation inheritance, there is no direct parallel in VB for C#'s support for references referring to objects of derived classes. However, there is some similarity with the fact that in VB 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 we were coding the `Employee` and `Manager` classes in VB, we might well have done so by defining an `IEmployee` interface that both class modules implement, and then accessing the `Employer` features through this interface.

Arrays of Objects

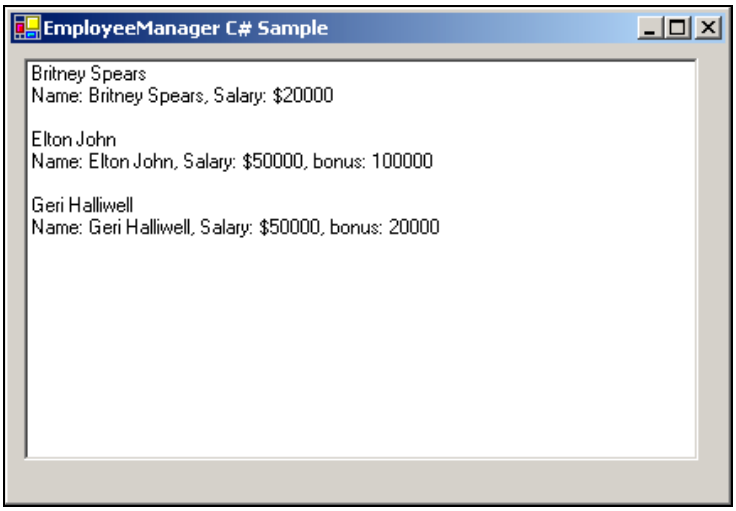
One important benefit of having references able to refer to derived class instances is that we can form arrays of object references, where the different objects in the array might be of different types. This is analogous to the situation in Visual Basic where we could form arrays of interface references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

We haven't yet seen how C# deals with arrays, so we will take a chance now and rewrite the test harness code for the `Employee` and `Manager` classes so that it forms an array of object references. This revised code is also downloadable from the Wrox Press web site as the `EmployeeManagerWithArrays` sample. 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("");
   }
}
```

We simply call up the `Name` property and the `ToString()` method of each element of the array. Running this code gives these results:



From this we can see that C# uses square brackets for dealing with arrays. This means that, unlike in VB, there is no danger of any confusion about whether we'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];
```

We see that we 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 we could have split the above line of code up like this:

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

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

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

There are a couple of other slightly different syntaxes for declaring arrays but we will stick to this one here. Once we have the array, we simply assign values to its elements in the usual way.

> **Note that one difference between C# and VB is that arrays in C# always start at element 0.**

In VB we have the option to change this to element 1 using the `Option Base` statement. Also in VB, you can specify any lower boundary you like for any particular array. But this feature doesn't really add any benefits, and it can hurt performance, because it means that whenever you access an element in an

array in VB, the code has to do some extra checking to find out what of the lower bound of that array is. C# does not support changing the base of an array in this way.

In the above code, once we have initialized the elements of the array, we just loop through them. The strange-looking syntax of the `for` loop may cause you some headache – we will examine that soon.

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

On the other hand, although we are using `Employee` references, we do always pick up the correct version of `ToString()`. If the object we're referring to is a `Manager` object, then when we 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 what reference type this object is accessed through, you will always run the correct method for that object.

The for Loop

Now let's have a look at that strange syntax of the `for` loop. What we are doing is the C# equivalent of this VB code:

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

The idea of the `For` loop in VB is that you start off by initializing some variable – 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 hit some 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 in order to move to the next iteration, and in order to determine when to exit from the loop, you perform some test. Comparing the Visual Basic and C# versions of `for`, the situation looks like this:

	VB	C#
At start of loop	Initialize loop control variable	Do something
To test whether to exit loop	Has loop control variable exceeded some value?	Test some condition
At end of each iteration	Increment 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, for example, 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 loops in VB – `For`, `For Eeach`, `Do`, and `While`, or alternatively the loop can work in some exotic manner for which there is no simple equivalent in VB. The C# for loop really gives you complete freedom to control the loop in whatever manner is appropriate to the task at hand.

> *We should point out, however, that C# also does support* `foreach`, `do`, *and* `while`, *for the situations where that is what you need.*

So let's get down to some precise syntax. Recall that the C# version of the above `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 the associated 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 here we 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 our 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 will not get executed at all.
- In the third item is the statement that will be executed at the end of each iteration of the loop. Visual Basic loops always works by incrementing some number, and this is exactly what we do in this case.

The syntax looks unfamiliar, but once you've got used to it, it means that you can use the `for` loop in very powerful ways. For example, suppose we wanted to display all the integer powers of 2 that are less than 4000 in a ListBox. We could write this:

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

This kind of effect is possible to achieve in VB, but slightly harder – for this particular loop, we would probably opt for a `while` loop in VB.

Other C# Features

We have now completed examining the code samples. The remainder of his appendix will briefly examine a couple of features of C# that you need to be aware of when making the transition from VB to C#, and which

we haven't yet discussed – in particular some of the C# concepts relating to data types and operators.

Data Types

As we have indicated, the data types available in C# do differ in detail from those available in Visual Basic. Not only that, all data types in C# have features that you would normally associate with an object. For example, as we have seen, 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 VB types, most of the types that you are familiar with in VB do have direct equivalents types in C#. For example, in place of VB's `Double`, C# has `double`. Where VB has the `Date`, C# has a .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 direct equivalent in C#. The `Variant` type in VB 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 explicitly indicate the data type you are referring to, at least one major source of run-time 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 a very much analogous to `Object` in VB. However, in VB, `Object` specifically refers to a COM object, and therefore can only be used to refer to objects, which in VB terms effectively means to reference data types. You cannot use an object reference, for example, to refer to an `Integer` or to a `Single`. In C#, by contrast, an `object0` can be used to refer to any .NET data type, and since in C# 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 VB.

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# allows a little more flexibility 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 we have 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 we can call methods such as `ToString()` against them. The data type `string`, however, is a reference type and so is really a class.

Operators

We need to say a couple of words about operators in C#, because they do work in a somewhat different way from VB operators, and this can catch you out if you are used to the VB way of working. In VB there are really two types of operator:

- 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 have an additional side effect of assigning some value to a variable as well. In fact we have already seen an example of this when we examined the addition-assignment operator, +=:

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

In fact, += returns a value as well as assigning the value. It returns the new value that has been assigned. Because of this we 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 is just 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`, 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 VB's `Boolean`) to the result of this condition, so we 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, since the assignment operator also returns the value that it is just assigned, 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` gets tested to determine whether the conditional statements should be executed, and at the same time we have stored the results of this test in the variable `B`.

The Ternary Operator

We do not have space in this appendix to go over all the various operators are available in C#. They are detailed in Chapters 2-5 of Professional C# 2nd Edition. However, we 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 VB. 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, we could write:

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

which would achieve exactly the same effect as:

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

With the VB `Iif` function, this would be achieved with:

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

Summary

In this appendix, we have presented a brief introduction to C# from the point of view of comparing it with Visual Basic. We have found quite a few differences in syntax. In general, C# syntax allows most statements to be expressed in a more compact way. We have also found many similarities between the languages – for example in their use of classes (or class modules in VB), value and reference types, and many of the syntactical structures. However, we have also seen how C# supports many powerful features, particularly those related to inheritance and classic object-oriented programming, that are not available in VB.

Appendix A of Professional C# 2nd Edition contains an introduction to object-oriented programming, a strong grasp of which is important to any serious C# development.

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