The Little Book Of C#

by Huw Collingbourne

(Bitwise Courses)

http://bitwisecourses.com/

The Little Book Of C#

written by

Huw Collingbourne

First edition: (*'C# Programming in Ten Easy Steps'*) March 2013
Second edition: April 2016
Third edition: (revision 3): June 2017
(revision 4): November 2017

A revised and expanded (152 page) edition of **The Little Book Of C#** is also available as a paperback or Kindle book from Amazon (US), Amazon (UK) and worldwide (ISBN: 978-1-913132-06-4).

INTRODUCTION

This is the course book to accompany *"C# Programming (in ten easy steps)"*, an online programming course. It is aimed at newcomers to C# programming and it is suitable for people who have never programmed before or who may need to revise the basics of programming before going on to more complex programming topics.

The core lessons in this course are the online videos. This eBook is a secondary resource which summarizes and expands upon the topics in the videos. You should also be sure to download the source code archive which contains all the sample programs described in the videos and in this eBook.

MAKING SENSE OF THE TEXT

In **The Little Book Of C#**, any C# source code is written like this:

```
private void greet( string aName ) {
    textBox1.AppendText( "Hello, " + aName + "!" );
}
```

Any output that you may expect to see on screen when a program is run is shown like this:

```
Hello, Fred!
```

When the text refers to code in a sample program, the name of the Visual Studio 'project' or 'solution' (the file that contains the program) will be shown like this:

ASampleProgram

To follow along with the tutorial, you can load the named project or solution into your copy of Visual Studio.

When an important concept is introduced, it may be highlighted in the left margin like this:

FUNCTIONS

Explanatory notes (which generally provide some hints or give a more in-depth explanation of some point mentioned in the text) are shown in a shaded box like this:

> This is an explanatory note. You can skip it if you like – but if you do so, you may miss something of interest…!

STEP ONE - GETTING READY

C# (pronounced *C-Sharp*) is Microsoft's most popular language for .NET development. The .NET 'framework' comprises a large library of programming code and tools to assist in creating and running programs.

The C# language uses a syntax that derives from the C language. This syntax has much in common with other 'C-like' languages including Java, C++, Objective-C and JavaScript.

In this course I'll assume that you will be using Microsoft's Visual Studio programming environment on Windows. You may either use a commercial edition of Visual Studio or the free 'Community' edition. The Community edition has most of the same features as the commercial edition including code editor, a visual designer and an integrate debugger. Visual Studio can be downloaded here:

https://www.visualstudio.com/

> There are also other editors and IDEs that support C# programming on Windows, Mac and Linux. For more information, please refer to the **FAQ** document provided with this course.

YOUR FIRST PROGRAM

If this is the first time you've used C#, follow these instructions to create your first program.

Start Visual Studio.

Select *File, New, Project*

In the left-hand pane, make sure Visual C# is selected.

In the right-hand pane, select *Windows Forms application*

In the *Name* field at the bottom of the dialog, enter a name for the project. I suggest:

```
HelloWorld
```

Click the *Browse* button to find a location (a directory on disk) in which to save the project.

> You may want to store all your projects beneath a specific directory or 'folder'. Or you may click 'New Folder' at the top of the dialog to create a new directory. For example, you might create a directory on your C: drive called *\CSharpProjects*. Once you have located a suitable directory, click the 'Select Folder' button.

In the *New Project* dialog, verify that the *Name* and *Location* are correct then click OK.

Visual Studio will now create a new Project called *Hello World* and this will be shown in the Solution Explorer panel.

PROJECTS AND SOLUTIONS

> Each Visual Studio project is included in a 'Solution'. A Solution may optionally contain more than one project.

You will now see a blank form in the centre of Visual Studio. This is where you will design the user interface. Make sure the Toolbox (containing ready-to-use 'controls' such as Button and Checkbox) is visible. Normally the Toolbox is shown at the left of the screen. If you can't see it, click the *View* menu then *Toolbox*.

In the Toolbox, click *Button*. Hold down the left mouse button to drag it onto the blank form. Release the mouse button to drop it onto the form. The form should now contain a button labelled '*button1*'.

In a similar way, drag and drop a *TextBox* from the Toolbox onto the form.

On the form, double-click the *button1* Button.

This will automatically create this block of C# code:

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

}
```

Don't worry what this all means for the time being. Just make sure your cursor is placed between the two curly brackets and type in this code:

```
textBox1.Text = "Hello world";
```

Make sure you type the code exactly as shown. In particular, make sure that the case of the letters is correct.

CASE-SENSITIVITY

C# is a 'case-sensitive' programming language. It considers an uppercase letter such as 'T' to be different from a lowercase letter such as 't'. So if you entered *TextBox1.Text* you won't be able to run the program because the name of the TextBox control, *textBox1*, begins with a lowercase 't' and C# will fail to find a control called *TextBox1* with an uppercase 'T'.

The code above tells C# to display the words "Hello World" as the text inside *textBox1*. The complete code of this code block should now look like this:

```
private void button1_Click(object sender, EventArgs e)
{
       textBox1.Text = "Hello world";
}
```

This code block is called a 'function' or 'method' and its name is *button1_click*. It will be run when the *button1* control is clicked.

Let's try it out….

Press *CTRL+F5* to run the application (you can also run it by selecting the *Debug* menu then, *'Start Without Debugging*').

The form you designed will pop up in its own window.

Click *button1*.

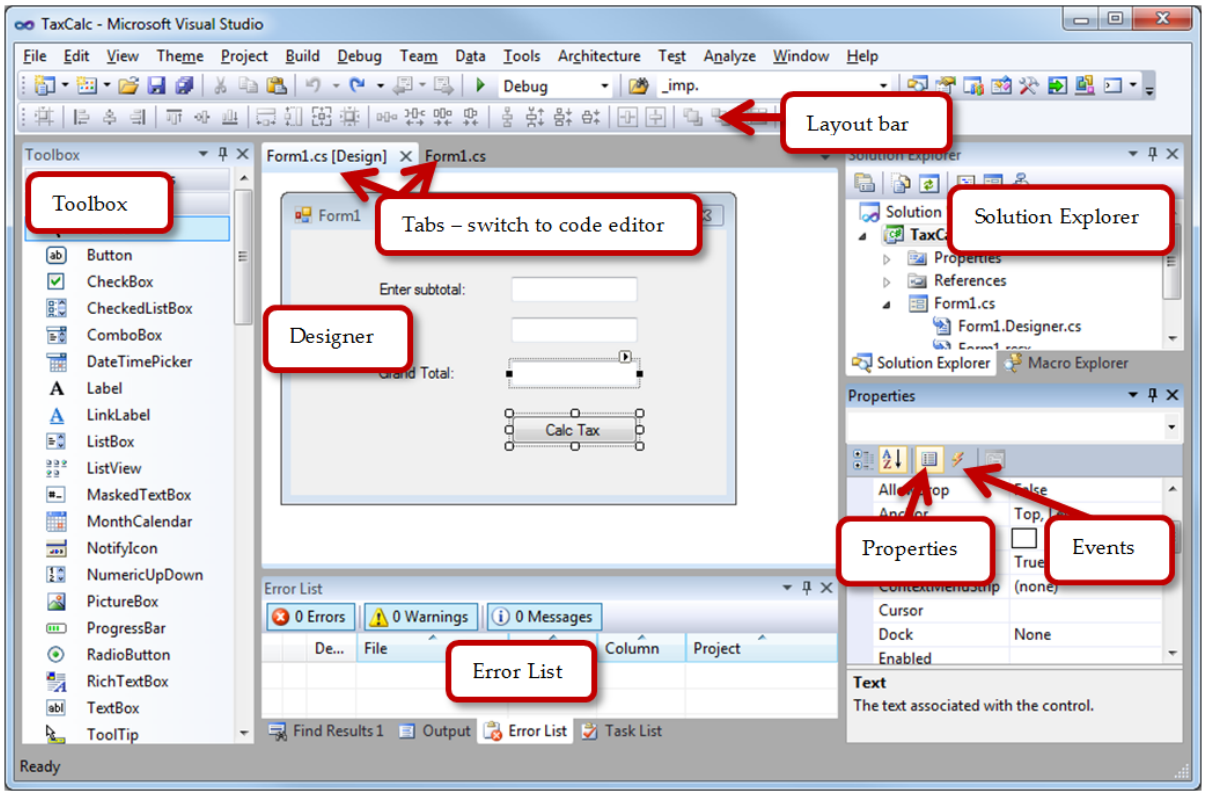
"Hello World" should now appear as the text inside *textBox1*.

Click the *close-button* in the right of the caption bar to close the program and return to Visual Studio.

VISUAL STUDIO OVERVIEW

Visual Studio is Microsoft's 'integrated development environment' (IDE) for programming using C# and other programming languages.

Visual Studio is a huge and complicated IDE and, even after many years of use, many programmers barely scratch the surface of its capabilities. In this step, I guide you through the essential features. Watch the videos and refer to the screenshot below which identifies some important elements of Visual Studio.



Project Types

In Visual Studio, you are able to create a variety of different types of project using C#. These include:

Universal Windows Platform (UWP)

This provides support for creating applications across all Windows 10 devices including PCs, tablets and phones.

Windows Presentation Foundation (WPF)

This provides support for highly configurable visual applications with styled components.

Windows Forms Application

This is used to create 'traditional' Windows applications with common controls such as buttons and text boxes.

Console Application

This is a 'non-visual' application that is intended to be run from the system prompt.

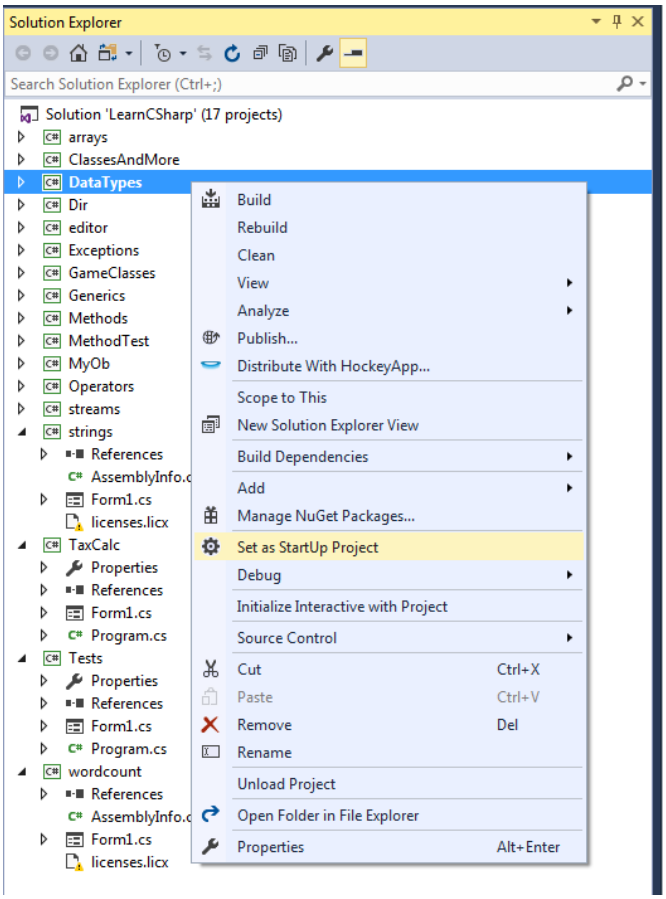
Visual Studio also provides options to create various types of 'class libraries' and components. These are more specialized sorts of project and will not be discussed in this course.

In this course, we will usually be creating Windows Forms applications. That is because they are much easier to create (simply by dragging and dropping components onto a 'form') than either WPF or UWP applications. Moreover, Windows Forms applications are supported by all versions of Visual Studio and Windows. UWP applications require Windows 10.

SOLUTIONS CONTAINING MULTIPLE PROJECTS

As I mentioned earlier, a Visual Studio Solution may contain more than one project. You can think of a Solution as being a convenient way to group together related projects so that you don't have to keep loading projects one by one each time you need them. The file that contains a Solution ends with the extension '*.sln*'. The file that contains a project ends with the extension '*.csproj*'.

When a Solution contains multiple projects, these are shown as 'nodes' in the Solution Explorer. Before compiling or running a specific project you need to set it as the startup project. You can do that by right-clicking the project node and selecting 'Set as Startup project' from the menu.



Here I have loaded the *LearnCSharp* solution. It contains numerous projects called *arrays*, *ClassesAndMore*, *DataTypes*, *Dir* and many others. Each project is shown as a 'node' in the solution. To activate a specific project I have to set it to be the 'startup project'. I am here activating the *DataTypes* project.

USING THE CODE ARCHIVE

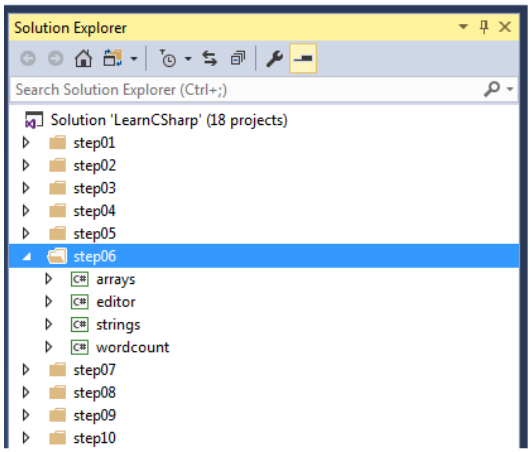
You may either load the projects from the code archive one by one (by selecting either the Solution or Project files from the subdirectories containing each project) or, as I recommend, you may load a Solution that groups together all the sample programs.

LearnCSharp.sln

This solution is called **LearnCSharp.sln** and it is found in the 'top-level' directory – that is the directory containing the subdirectories for each step in the course (*\Step02*, *\Step03* and so on). Remember that if you load the *LearnCSharp* solution you must be sure to set a startup project, as explained previously, before compiling and running a project.

SOLUTION FOLDERS

The *LearnCSharp* solution contains 'folders' of projects relating to each setp of this course. The folders are shown as icons labeled 'step01', 'step02' and so on. These 'solution folders' do not necessarily have the same structure as folders (directories) on your disk. However, in the *LearnCSharp* solution I have arranged the solution folders to match the disk folders. When you are working on a specific step of the course just double-click a solution folder to show the projects it contains…



STEP TWO – FEATURES OF THE C# LANGUAGE

Every program is made up of some comment elements such as keywords, functions and variables In this chapter, we look at the building blocks of C# programs.

TYPES

When your programs do calculations or display some text, they use data. The data items each have a data type. For example, to do calculations you may use integer numbers such as 10 or floating point numbers such as 10.5. In a program you can assign values to named variables. Each variable must be declared with the appropriate data type. This is how to declare a floating-point variable named `mydouble` with the double data-type:

DataTypes

```
double mydouble;
```

You can now assign a floating-point value to that variable:

```
mydouble = 100.75;
```

Alternatively, you can assign a value at the same time you declare the variable:

```
double mydouble = 100.75;
```

Common data types include:

`int`	an integer such as 10
`double`	a floating point number such as 10.5
`char`	a character between single quotes such as 'a'
`string`	a sequence of characters between double quotes such as "abc"

VARIABLES

The value of a variable can be changed. So, for example, if you were to write a tax calculator that allowed the user to enter a subtotal and then calculated both the tax and also the grand total (that is, the subtotal plus the tax), you might declare three floating point (`double`) variables named `subtotal`, `tax` and `grandtotal`.

TaxCalcTest

```
double subtotal = 0.0;
double tax = 0.0;
double grandtotal = 0.0;
```

Here I have assigned the value 0.0 to each variable. But the whole point of a variable is that its value may vary – that is, once it has been given one value, it may be given a different vale later on. So my code could assign new values like this:

```
double subtotal = 12.5;
double tax = 2.5;
double grandtotal = 15.0;
```

This simple tax calculator is limited to displaying the same values each time it's run which is, of course, not very useful. At the end of this chapter, we'll see how to create a tax calculator that calculates tax based on values entered by the user.

CONSTANTS

But sometimes you may want to make sure that a value *cannot* be changed. For example, if the tax rate is 20% you might declare a variable like this:


```
double TAXRATE = 0.2;
```

But now someone might accidentally set a different tax rate (this is not a problem in a very small program – but this sort of error can easily crop up in a real-world program that may contains tens or hundreds of thousands of lines of C# code). For example, it would be permissible to assign a new value of 30% like this:

```
TAXRATE = 0.3;
```

If you want to make sure that this cannot be done, you need to make `TAXRATE` a constant rather than a variable. A constant is an identifier whose value can only be assigned once. In C#, I can create a constant by placing the `const` keyword before the declaration like this:

```
const double TAXRATE = 0.2;
```

COMMENTS

You can document your code by adding comments to it. For example, you might add comments to describe what each function is supposed to do. C# lets you insert multi-line comments between pairs of `/*` and `*/` delimiters, like this:

```
/*
     This function calculates
        tax at a rate of 20%
*/
```

In addition to these multi-line comments, you may also add 'line comments' that begin with two slash characters `//` and extend to the end of the current line. Line comments may either comment out an entire line or any part of a line which may include code before the `//` characters. This is an examples of a full-line comment:

```
// calculate and display tax and grand total based on a subtotal
```

This is a line comment placed after a piece of code:

```
const double TAXRATE = 0.2;     // Define the tax rate at 20%
```

KEYWORDS

C# defines a number of keywords that mean specific things to the language. You cannot use these keywords as the names of variables, constants or functions. Keywords include words such as `if`, `else`, `for`, `while` and the names of data types such as `char`, `double`, `int` and `string`. You can find a full list of C# keywords in Microsoft's C# reference guide which can be found online:

https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/

STATEMENTS

In C#, single statements are usually terminated with a semicolon `;` character. A statement is a small piece of code such as an assignment or a method call. Here, for example, are four statements, each terminated with a semicolon:

```
textBox1.Clear( );
sayHello( );
greet( "Mary" );
someName = "Fred";
```

When you need several statements to execute one after another – for example, when some test evaluates to *true* (here I test if the `num` variable has a value of 100), you may enclose the statements between a pair of curly brackets, like this:

```
if( num == 100 ){
        textBox1.Clear( );
        sayHello( );
        greet( "Mary" );
        someName = "Fred";
}
```

FUNCTIONS

Typically we will divide our code into small named blocks or 'functions'. The names of functions (or 'methods' as they are often called in Object Oriented languages such as C#) must be followed by a pair of parentheses like this:

```
private void sayHello( )
```

The start and end of a function is indicated with a pair of curly brackets and the code of the function is placed between those brackets, like this:

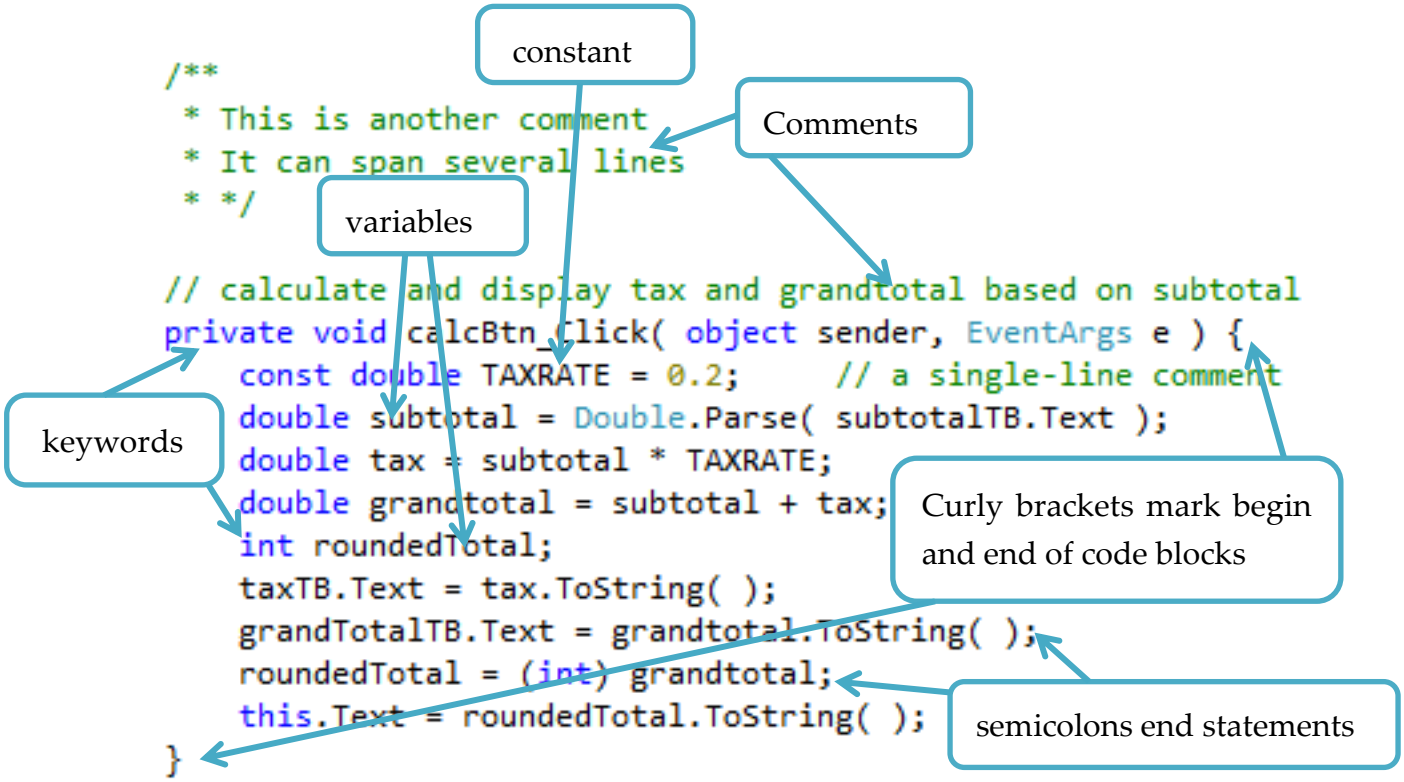
```
private void sayHello( ) {
    textBox1.AppendText( "Hello\n" );
}
```

If you want to be able to pass some kind of data to a function, you can add one or more 'arguments' between the parentheses. And a function may return a value to the code that called the function after the keyword `return`. Here I have a function called `addNumbers()` which takes two `int` (integer) arguments called `num1` and `num2`. It returns a string that includes the sum of `num1` plus `num2`:

```
private string addNumbers( int num1, int num2 ) {
     return "The total is " + num1+num2;
}
```

We will look at functions, or methods, in much more detail in Chapter 4.

SUMMARY OF COMMON C# LANGUAGE ELEMENTS



NAMESPACES AND USINGS

At the top of a code file such as *Form1.cs* in the *DataTypes* project you will see these lines beginning with the keyword `using`:

DataTypes

```
using System;
using System.Windows.Forms;
```

The `using` directive gives C# code access to classes and types inside another code module or 'namespace'. When you create an application, C# automatically adds the `using` statements you will need, at least initially.

CLASSES

Classes are the building blocks of object oriented programming. A class is a blueprint for an object. For example, we could have a `Dog` class which defines all the features and behaviour of a dog. From that `Dog` class we could create individual dog objects called Fido, Bonzo and Cujo each of which shares the features (fur, tail, paws etc.) and behaviour (eating and woofing) defined by the `Dog` class. We'll come back to Classes when we look at Object Orientation in Chapter 5 .

This provides access to the `System` namespace:

```
using System;
```

When a namespace is used in this way, any classes inside it can be used in your code. The `System` namespace contains a `String` class that has a `Format()` method. Since my code uses the `System` namespace, I can use `String.Format()` like this:

Namespaces

```
textBox1.Text = String.Format("Value of x = {0}", x);
```

If you regularly need to use classes in a namespace, you should add that namespace to the `using` section. If you don't add the namespace, you may still be

able to use a class it contains. But you will need to add the namespace and a dot before the class name in your code, like this:

```
textBox1.Text = System.String.Format("Value of x = {0}\n", x);
```

SCOPE

A namespace precisely defines the '*scope*' or 'visibility' of the code it contains. Namespaces can be nested inside other namespaces to add finer levels of 'scoping'. You cannot access any class without explicitly referencing its namespace. This has the advantage of enforcing clear, unambiguous code. Two classes with the same name but declared inside separate namespaces will not conflict with one another.

Some namespaces are nested inside other namespaces. For example, the `IO` namespace, which provides lots of useful code for dealing with disk files and directories, is nested inside the `System` namespace. One of the classes in the `IO` namespace is called `Directory`. This class provides various useful methods including one called `GetCurrentDirectory()`. Assuming I am using the `System` namespace, this is how I could use that method:

```
System.IO.Directory.GetCurrentDirectory();
```

But if I want to use classes from the `IO` namespace frequently, I can simplify my code by adding this nested namespace to the `using` section like this:

```
using System.IO;
```

When I do that, I can use the `Directory` class without needing to qualify it by placing the namespace names in front of it, like this:

```
Directory.GetCurrentDirectory();
```

TYPE CONVERSION

Many classes and structures (or *structs*) in the .NET framework include built-in conversion routines. For example, number structs such as *Double* and *Int16* include a method called `Parse()` which can convert a string into a number.

STRUCTS

> A struct is a data structure that encloses one or more 'fields' containing data items. For example, a struct might contain an employee record with fields to store each employee's first name, last name and salary. In some programming languages, structs are called 'records'.

In this example, I convert the string `"15.8"` to the Double `15.8` and assign the result to the variable, `d`:

```
double d = Double.Parse( "15.8" );
```

This converts the string `"15"` to the Int16 (integer) value, `15`, and assigns the result to the variable, `i`:

```
Int16 i = Int16.Parse( "15" );
```

But beware. If the string cannot be converted, an error occurs! This will cause an error:

```
Int16 i = Int16.Parse( "hello" );
```

Numeric types can be converted to Strings using the `ToString()` method placed after the variable name followed by a dot. Assuming that the *double* variable, `grandtotal`, has the value `16.5`, the following code will convert it to a string, `"16.5"`, and that string will be displayed as the Text of the text box named `grandtotalTB`:

TaxCalc

```
grandTotalTB.Text = grandtotal.ToString( );
```

Note that .NET also has a class called `Convert` which can be used to convert between a broad range of different data types. For example, this converts, the *Text* (a string) in the `subtotalTB` text box to a double:

```
double subtotal = Convert.ToDouble( subtotalTB.Text );
```

And this converts the value of the `subtotal` variable (a double) to a string and displays it in the text box, `subtotalTB`:

```
subtotalTB.Text = Convert.ToString( subtotal );
```

TYPE CASTING

I've decided that I want my tax calculator to display an integer version of the grand total in the title bar of the window. In order to 'round' a double value (dropping the floating point part) to an integer, I can 'cast' the double to an integer.

A cast is an instruction to the compiler to let you treat one type as another type. The types must be compatible (for example, you can cast a floating point number to a decimal number but you cannot cast a number to a string).

To cast one data type to another, you need to precede the variable name with the name of the target data type between parentheses. Here is how I would cast the double variable, `grandtotal` to an integer and assign the resulting value to the `int` variable, `roundedtotal`:

```
roundedTotal = (int)grandtotal;
```

In general, converting data using a casts is less 'safe' than converting data using a method that has been specifically designed to do the conversion. That is because a method can try to deal with potential problems and return information if an error occurs. In the present case, therefore, instead of casting `grandtotal` to `int` I will once again use a method of the `Convert` class:

```
roundedTotal = Convert.ToInt32(grandtotal);
```

And I convert this back to a string in order to display it in the title bar of the current program :

```
this.Text = roundedTotal.ToString();
```

The `this` keyword refers to the current object. Here the current object is the form – the main window – of my calculator program, so `this.Text` sets the text shown in the title bar of the window.

SAMPLE PROGRAM: TAX CALCULATOR

Here is a complete version of the function that computes and displays the values when the button is clicked:

TaxCalc

```
private void calcBtn_Click( object sender, EventArgs e ) {
      const double TAXRATE = 0.2;
      double subtotal = Double.Parse(subtotalTB.Text);
      double tax = subtotal * TAXRATE;
      double grandtotal = subtotal + tax;
      int roundedTotal;
      subtotalTB.Text = subtotal.ToString();
      taxTB.Text = tax.ToString();
      grandTotalTB.Text = grandtotal.ToString();
      roundedTotal = (int)grandtotal;
      this.Text = roundedTotal.ToString();
}
```

Notice that I have assigned values to the variables as I declare them. In some cases, the assigned value is the result of executing some code to do a calculation like this:

```
double tax = subtotal * TAXRATE;
```

Declaring and initializing variables all in one go like this has the benefit of making your code concise. As an alternative, I could declare the variables first, like this:

```
double tax;
double grandtotal;
```

And then, later on, I can calculate and assign their values like this:

```
tax = subtotal * TAXRATE;
grandtotal = subtotal + tax;
```

Even though this is more verbose (I have to write more lines of code), this is generally my preferred style of programming. In short and simple programs, there is nothing wrong with initializing variables as they are declared. But in long and complex programs, it may be clearer if variables are declared all in one place, at the start of a function, and their values are then assigned later on in that function.

PROGRAMMING STYLE

Different programmers have differing views on what is 'good' or 'bad' programming style. There are no hard and fast rules. However, you should generally try to be clear and consistent. At the very least, I would strongly encourage you to declare all variables in one place, prior to using them. C# allows you to declare variables in the middle of executable code as I declare the variable `grandtotal` here:

```
subtotal = Double.Parse(subtotalTB.Text);
tax = subtotal * TAXRATE;
double grandtotal = subtotal + tax;
taxTB.Text = tax.ToString();
```

This can be confusing, however, in long code blocks. It is much clearer to declare all variables *before* the executable code that uses them.

Using uninitialized variables in C# is not allowed. This code, for example, won't compile (because the `tax` variable is not initialized with a value):

```
double tax;
double grandtotal = 0.0;
subtotal = Double.Parse(subtotalTB.Text);
grandtotal = subtotal + tax;       // ERROR: tax is uninitialized
```

Finally, this is my rewritten version of the function. This time, I have initialized the constant when it is declared, then initialized the variables with default (zero) values and assigned calculated values to those variables later in the function. You don't have to assign default values but if you need to debug your program, it is sometimes useful to be able to check the value of a variable at different points and an explicitly defined default value may then be useful. In fact, if you don't assign default values, C# assigns its own default values anyway so this step is not strictly needed here. Assigning default values may be more useful with more complex datatypes as we will see later in the course:

```
private void calcBtn_Click(object sender, EventArgs e) {
        const double TAXRATE = 0.2;
        double subtotal = 0.0;
        double tax = 0.0;
        double grandtotal = 0.0;
        int roundedTotal = 0;
        subtotal = Double.Parse(subtotalTB.Text);
        tax = subtotal * TAXRATE;
        grandtotal = subtotal + tax;
        taxTB.Text = tax.ToString();
        grandTotalTB.Text = grandtotal.ToString();
        roundedTotal = Convert.ToInt32(grandtotal);
        this.Text = roundedTotal.ToString();
}
```

IMPLICITLY TYPED VARIABLES

C# allows you to declare and initialize local variables without specifying a type. You must do this using the `var` keyword instead of a defined type, like this:

InferredTypes

```
var num = 100;
var num2 = 200.5;
var str = "Hello world";
```

C# infers the actual type from the data that is assigned to each variable. So if you assign a string, the variable will be typed to the string datatype. If you assign a double it will be typed to double and so on. If you are used to a language such as Ruby where it is normal for variables to infer a datatype, this may seem like an attractive option in C#. Be aware, though, that unlike Ruby, C# does not allow the type of a variable to be changed. Once it is inferred to be a string, a variable's type becomes fixed as a string and it cannot be assigned any other type. This is not permitted:

```
var str = "Hello world";
str = 2;
```

Type inference in C# has no real benefit for most programming tasks. In most programs, it is usually better to declare variables with explicit data types. Type inference may be useful for certain specialist types of data querying or looping operations which may be capable of returning more than one type. This is a very simple example of a `foreach` loop that iterates over an array (a sequential list) containing a string, an int and a double. The loop defines the `thing` variable using the `var` keyword. At each turn through the loop, the `thing` variable infers the type of an array element:

```
private void button1_Click(object sender, EventArgs e) {
        var num = 100;
        var num2 = 200.5;
        var str = "Hello world";
        object[] somethings = { str, num, num2 };
        textBox1.Text = str + " is a: " + str.GetType() + "\n";
        textBox1.AppendText(num + " is a: " + num.GetType() + "\n");
        textBox1.AppendText(num2 + " is a: " + num2.GetType() + "\n");
        textBox1.AppendText("--- Let's see what is in the somethings array ----\n");
        foreach (var thing in somethings) {
              textBox1.AppendText(thing + " is a: " + thing.GetType() + "\n");
        }
}
```

This is the output:

```
Hello world is a: System.String
100 is a: System.Int32
200.5 is a: System.Double
--- Let's see what is in the somethings array ----
Hello world is a: System.String
100 is a: System.Int32
200.5 is a: System.Double
```

STEP THREE – TESTS AND OPERATORS

In this chapter we look at ways of running different bits of code depending on the results of tests. And we also look at operators which special symbols that are used to perform a number of different operations in C#.

TESTS AND COMPARISONS

Most computer programs have to make decisions and take different actions according to whether some condition is or is not true. C# can perform tests using the `if` statement.

IF...

The test itself must be contained within parentheses and should be capable of evaluating to true or false. If true, the statement, or block of statements, following the test executes. A single-expression statement is terminated by a semicolon. A multi-line block of statements may be enclosed within curly brackets. Here is an example:

ifelse

```
double someMoney;
someMoney = Double.Parse(textBox1.Text);
if (someMoney > 100.00) {
       MessageBox.Show("You have lots of money!");
}
```

Here the statement enclosed between curly brackets (the one that shows the message box) is only run if the test (`someMoney > 100.00`) is true - that is, if the value of the `someMoney` variable is greater than 100.00.

TEST OPERATORS

You may use other operators to perform other tests. Here is use the 'greater than' operator (`>`) in the test. If I wanted to test if the variable was less than 100.00 I would use the 'less than' operator (`<`). I'll take a closer look at operators later in this chapter.

ELSE...

Optionally, an `else` section may follow the `if` section and this will execute if the test evaluates to false. In this example, if the value of `someMoney` is greater than 100.00 the message "You have lots of money!" is shown. If it is 100.00 or less, the message, "You need more money…" is shown:

```
if (someMoney > 100.00) {
       MessageBox.Show("You have lots of money!");
 } else {
       MessageBox.Show("You need more money...");
 }
```

When only one statement needs to be run after a test, you may omit the enclosing curly brackets. So the code above could be rewritten like this:

```
if (someMoney > 100.00) {
       MessageBox.Show("You have lots of money!");
else
       MessageBox.Show("You need more money...");
```

But be careful. If you omit the curly brackets around a set of statements, only the first statement is associated with the test condition. For example, let's assume you want to test if `x` is greater than 10. If the test succeeds you want to display the message "Great news!, then assign 100 to `x` and display the message "You've won $100". If the test fails, you want to do nothing. This code works as expected because when `x` is 100 or less all the statements between curly brackets are skipped:

```
if (x > 10) {
       MessageBox.Show("Great news!");
       x = 100;
       MessageBox.Show("You've won $" + x);
}
```

But this code does not work as expected:

```
if (x > 10)
       MessageBox.Show("Great news!");
       x = 100;
       MessageBox.Show("You've won $" + x);
```

That's because only *one* statement – the first line that follows the test – is skipped if the test fails. The other two lines are *not* associated with the test and they

will, therefore, run both if the test succeeds and if it fails. That means that in all cases `x` is set to 100 and the message "You've won $100" is shown. For reasons of clarity and consistency, I generally prefer to use curly brackets to enclose blocks of code to be executed following a test even if those blocks only contain a single statement.

IF...ELSE IF

You may also 'chain together' multiple *if..else if* sections so that when one test fails the next condition following `else if` will be tested. Here is an example:

Tests

```
if( userinput.Text == "" ) {
          output.Text = "You didn't enter anything";
      } else if( ( userinput.Text == "hello" ) || ( userinput.Text == "hi" ) ) {
          output.Text = "Hello to you too!";
      } else {
          output.Text = "I don't understand that!";
}
```

This code tests if the text box named `userinput` is empty (that is, if it contains an empty string `""`). If that test succeeds the message "You didn't enter anything" is displayed. If, however, it fails because there *is* some text in the text box, the next test after `else if` is made. This uses the 'or' operator `||` to test if the text in `userinput` is either "hello" or "hi". If this test succeeds, "Hello to you too!" is displayed. But if that test too fails then the code following the final `else` is run and "I don't understand that!" is displayed.

SWITCH STATEMENTS

If you need to perform many tests, it is often quicker to write them as 'switch statements' instead of multiple `if..else if` tests. In C# a switch statement begins with the keyword `switch` followed by a 'match expression' between parentheses. This is going to be tested against values in a number of 'switch sections' or 'case statements' that follow it. The match expression needn't be a string. It could be some other data type such as an integer or a single character. Each switch section begins with the `case` keyword followed by a value – such as an empty string, the string "Hello" and so on. The value in each switch section is compared with the match expression and, if a match is made, the code following the `case` keyword, a test value and a colon executes:

```
case "hi":
       output.Text = "Hello to you too!";
```

When you want to exit the `case` block you need to use the keyword `break`. If `case` tests are not followed by `break`, sequential `case` tests will be done one after the other until `break` is encountered. You may specify a `default` which will execute if no match is made by any of the `case` tests. Here's an example:

Tests

```
switch( userinput.Text ) {
     case "":
       output.Text = "You didn't enter anything";
       break;
     case "hello":
     case "hi":
       output.Text = "Hello to you too!";
       break;
     default:
       output.Text = "I don't understand that!";
       break;
}
```

The entire switch block, including all its switch (the `case` and `default`) statements is enclosed by a pair of curly brackets, with the opening bracket immediately following the match expression.

In this example, if `userinput.Text` is an empty string (`""`), it matches the first `case` test and "You didn't enter anything" will be displayed. Then the keyword `break` is encountered so no more tests are done. If `userinput.Text` is either "hello" or "hi", then "Hello to you too!" is displayed. This matches "hello" because there is no `break` after that test. If `userinput.Text` is anything else, the code in the `default` section is run, so "I don't understand that!" is displayed.

Operators

Operators are special symbols that are used to do specific operations such as the addition and multiplication of numbers or the concatenation (adding together) of strings.

Assignment Operator

One of the most important operators is the assignment operator, `=`, which assigns the value on its right to a variable on its left. Note that the type of data assigned must be compatible with the type of the variable.

This is an assignment of an integer (`10`) to an integer (`int`) variable:

```
int myintvariable = 10;
```

Beware. While one equals sign `=` is used to assign a value, two equals signs, `==`, are used to test a condition. This is a simple test:

`=`	this is the assignment operator. e.g. `x = 1;`
`==`	this is the equality operator. e.g. `if (x == 1)`

ARITHMETIC OPERATORS

We've already used some arithmetic operators in previous programs – for example, in the tax calculator from *Chapter 2* I calculated values by adding and subtracting using the addition `+` and subtraction `-` operators. The other operators you need to know about are the division `/` and multiplication `*` operators which divide and multiply one value by the other and the remainder (or 'modulus') operator `%` which calculates the remainder after a division.

Here are some examples of these operators:

ArithmeticOps

```
100 + 7
100 - 7
100 / 7
100 % 7
```

+ THE STRING CONCATENATION OPERATOR

Incidentally, bear in mind that the `+` operator is also commonly used to concatenate strings. C# works out whether it is an addition operator or a concatenation operator based on the context. If it occurs between two strings, it's obviously a concatenation operator. If it's between two numbers it's an addition operator. But what about if it's between a string and a number as in this code?

```
textBox1.AppendText("Result =" + 100 );
```

Well, C# assumes you want to concatenate the string version of the number and automatically converts it before adding it onto the end of the string. So the output from the above code would be the string: `"Result = 100"`.

Be careful though. Look at this:

```
textBox1.AppendText("155+3=" + 155 + 3 + "\n");
```

You might assume that here the integer 155 will be added to 3 and the result, 158, will be concatenated onto the string. In fact, that's not what happens. What actually happens is that 155 is converted to a string, "155", 3 is converted to a string

"3" and those two strings are concatenated onto the preceding string, so that this is the output:

```
155+3=1553
```

If you want the + between the two integers to be treated as an arithmetic operator, you should put the complete arithmetic expression inside parentheses like this:

```
textBox1.AppendText("155+3=" + (155 + 3) + "\n");
```

This ensures that the addition operation is evaluated as a single unit and the result, 158, will then be converted to a string and concatenated onto the preceding string. Now, this is what will be displayed when the string is appended to the text box:

```
155+3=158
```

Comparison Operators

These are the most common comparison operators that you will use in tests:

```
==          // equals
!=          // not equals
>           // greater than
<           // less than
<=          // less than or equal to
>=          // greater than or equal to
```

This code tests if the value of `i` is less than or equal to the value of `j`. If it is, then the conditional evaluates to *true* and "Test is true" is displayed. Otherwise the condition evaluates to *false* and "Test is false is displayed:

Tests

```
int i = 100;
int j = 200;
if( i <= j ) {
       output.Text = "Test is true";
} else {
       output.Text = "Test is false";
}
```

You can alter the nature of the test shown above by substituting different comparison operators. For example, this test returns *true* if the value of `i` is greater than the value of `j`:

```
if( i > j )
```

And this test returns *true* if the value of `i` is *not equal to* the value of `j`:

```
if( i != j )
```

LOGICAL OPERATORS

In some of the code examples in this chapter, I have used the `&&` operator to mean 'and' and the `||` operator to mean 'or'. You'll find this example in the *Tests* project:

Tests

```
if( ( msglen == 0 ) && ( namelen == 0 ) ) {
    output.Text = "You haven't entered anything!";
} else if( ( msglen == 0 ) || ( namelen == 0 ) ) {
    output.Text = "You must enter something into both text boxes";
} else if( ( msg != "hi" ) && ( msg != "hello" ) ) {
    output.Text = "Please enter a friendly greeting ('hi' or 'hello' would be nice)";
} else {
    output.Text = "Well, " + msg + " to you too, " + name;
}
```

The `&&` and `||` operators are called 'logical operators'. Logical operators can help you chain together conditions when you want to take some action only when *all* of a set of conditions are true or when *any one* of a set of conditions is true. For example, you might want to offer a discount to customers only when they have bought goods worth more than 100 dollars *and* they have also bought the deal of the day. In code these conditions could be evaluated using the logical *and* (`&&`) operator, like this:

```
if ((valueOfPurchases > 100) && (boughtDealOfTheDay))
```

But if you are feeling more generous, you might want to offer a discount *either* if a customer has bought goods worth more than 100 dollars *or* has bought the deal of the day. In code these conditions can be evaluated using the logical *or* (`||`) operator, like this:

```
if ((valueOfPurchases > 100) || (boughtDealOfTheDay))
```

When a test includes multiple parts or 'conditions', each condition enclosed by a pair of parentheses can evaluate to *true* or *false*. When multiple conditions are separated by `&&` ('and') operators *all* those conditions must evaluate to *true* in order for the entire test to evaluate to *true*. When multiple conditions are separated by `||` ('or') operators *any one* of those conditions must evaluate to *true* in order for the entire test to evaluate to *true*.

! The Logical NOT Operator

Just as the test for equality can be negated using the not-equals operator `!=` instead of the equals operator `==`, so too other tests can be negated by preceding the test with a single exclamation mark `!` which is called the 'logical not operator'. So if `age` is less than or equal to 30 and `salary` is greater than or equal to 30000.00, this test succeeds:

```
if ((age <= 30) && (salary >= 30000.00)) {
```

But this test fails (due to the `!`), fails:

```
if (!((age <= 30) && (salary >= 30000.00))) {
```

You will find some examples of using these operators in the *LogicalOperators* sample program:

LogicalOperators

```
int age;
int number_of_children;
double salary;

age = 25;
number_of_children = 1;
salary = 20000.00;

if ((age <= 30) && (salary >= 30000.00)) {
    textBox1.AppendText("You are a rich young person\n");
} else {
    textBox1.AppendText("You are not a rich young person\n");
}
// Negate this test with a !
if (!((age <= 30) && (salary >= 30000.00))) {
    textBox1.AppendText("You are a rich young person\n");
} else {
    textBox1.AppendText("You are not a rich young person\n");

}

if ((age <= 30) || (salary >= 30000.00)) {
    textBox1.AppendText("You are either rich or young or both\n");
} else {
    textBox1.AppendText("You are not neither rich nor young\n");
}

if ((age <= 30) && (salary >= 30000.00) && (number_of_children != 0)) {
    textBox1.AppendText("You are a rich young parent\n");
} else {
    textBox1.AppendText("You are not a rich young parent\n");
}
```

Notice that I place each test condition between parentheses. This helps to resolve any possible ambiguities.

OPERATOR PRECEDENCE

> If you don't put test conditions between parentheses, C# evaluates the elements of a test in a predefined order – evaluating some operators before other operators. The default order of evaluation is described as 'operator precedence'. This is explained in more detail in the Microsoft document: https://msdn.microsoft.com/en-us/library/2bxt6kc4.aspx

When run, this program produces this output:

```
You are not a rich young person
You are either rich or young or both
You are not a rich young parent
```

Try changing some of the operators to understand how they work. For example, if I change the && to || in the last if test like this:

```
if ((age <= 30) || (salary >= 30000.00) && (number_of_children != 0))
```

The output now shows that the rest succeeds:

```
You are a rich young parent
```

That's because the test no longer requires that all conditions are met – only that either the first conditions or the last two conditions are met.

KEEP IT SIMPLE!

As a general rule, try not to have too many conditions linked with && and || as the more complex the conditions the more likely you are to make a mistake of logic that could result in hard-to-find program bugs. For example, take a look at this code:

```
age = 25;
number_of_children = 1;
salary = 20000.00;
bonus = 500.00;

if (age > 20 && salary > 10000.00 || number_of_children == 1 && bonus > 8000.00 ) {
    textBox1.AppendText("You've won the star prize!");
} else {
    textBox1.AppendText("Sorry, you are not a winner\n");
}
```

Can you work out which of the two specified strings will actually be printed out? It's difficult because the test is so complicated. Let's run it and see:

This is what it shows:

```
"You've won the star prize!"
```

Does it mean that there are two necessary conditions of winning? Namely:

[1] that your age is greater than twenty AND your salary is greater than 10,0000 – OR, alternatively, that you have one child…

…and:

[2] that your bonus must be greater than 8000?

If that is what you intend, then this is the first condition:

```
age > 20 && salary > 10000.00 || number_of_children == 1
```

And here's the second condition:

```
bonus > 800.00
```

We can clarify this by putting parentheses around the separate conditions like this so that condition [1] above is now enclosed within a pair of parentheses which, for clarity, I show in red below:

```
if ( (age > 20 && salary > 10000.00) || number_of_children == 1) && bonus > 800.00 )
```

But this time this is the output:

```
"Sorry, you are not a winner\n"
```

In my original code, without the conditions placed between parentheses, the test was evaluated according to the rules of operator precedence. As I said earlier, operator precedence can be quite hard to understand, especially in a test such as the one above that includes both logical and comparison operators. By adding parentheses to ensure that a particular set of conditions is evaluated as a single 'sub-test' (which returns *true* or *false*), I have changed how the entire test is evaluated. You could try adding parentheses around different parts of this test to see how the end result may change. But really the lesson to be learnt here is that the test itself is far too complicated. It is simply too hard to understand what exactly it all means. If you plan to use tests like this, keep them simple!

Compound Assignment Operators

Some assignment operators in C#, and other C-like languages, perform a calculation prior to assigning the result to a variable. This table shows some examples of common 'compound assignment operators' along with the non-compound equivalent.

operator	example	equivalent to
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`

It is up to you which syntax you prefer to use in your own code. If you are familiar with other C-like languages, you may already have a preference. Many C and C++ programmers prefer the short form as in `a += b`. Basic and Pascal programmers may feel more comfortable with the slightly longer form as in `a = a + b`.

Increment ++ and Decrement -- Operators

When you want to increment or decrement by 1 (add 1 to or subtract 1 from) the value of a variable, you may also use the `++` and `--` operators. Here is an example of the increment (`++`) operator:

```
int num = 100;
num++;  // num is now 101
```

This is an example of the decrement (`--`) operator:

```
int num = 100;
num--;  // num is now 99
```

`++` and `--` are called 'unary operators' because they only require one value, one 'operand', to work upon (e.g. `a++`) whereas binary operators such as `+` and `-` require two (e.g. `a + b`).

PREFIX AND POSTFIX OPERATORS

You may place the `++` and `--` operators either before or after a variable like this: `a++` or `++a`. When placed *before* a variable, the value is incremented before any assignment is made:


```
int a;
a = 10;
b = ++a;
```

After the above code executes, `a` has the value 11; and `b` also has the value 11.

When placed *after* a variable, the assignment of the existing value is done before the variable's value is incremented:

```
int a;
a = 10;
b = a++;
```

After the above code executes, `a` has the value 11; but `b` has the value 10.

Mixing prefix and postfix operators in your code can be confusing and may lead to hard-to-find bugs. So, whenever possible, keep it simple and keep it clear. In fact, there is often nothing wrong with using the longer form, which may seem more verbose but is at least completely unambiguous – e.g. `b = a + 1` or `b = a - 1`.

STEP FOUR – FUNCTIONS AND ARGUMENTS

Functions provide ways of dividing your code into named 'chunks'. In this chapter I explain how to write functions, pass arguments to them and return values from a function to the code that called it.

FUNCTIONS OR METHODS

A function is declared using a keyword such as `private` or `public` (which controls the degree of visibility of the function to the rest of the code in the program) followed by the data type of any value that's returned by the function or `void` if nothing is returned. Then comes the name of the function, which may be chosen by the programmer. Then comes a pair of parentheses.

The parentheses may contain one or more 'arguments' or 'parameters' separated by commas. The argument names are chosen by the programmer and each argument must be preceded by its data type. When a function returns some value to the code that called it, that return value is indicated by preceding it with the `return` keyword.

PARAMETERS AND ARGUMENTS

Parameters are the named variables declared between parentheses in the method itself, like this:

```
private string addNumbers( int num1, int num2 )
```

Arguments are the values passed to the method when that method is called, like this:

```
addNumbers( 100, 200 );
```

While computer scientists may make a clear distinction between the terms 'parameter' and 'argument', it is often the case that programmers use these two words interchangeably. So, informally, a method's 'parameter list' may be referred to as its 'argument list'.

When you call a method, you must pass the same number of arguments as the parameters declared by the method. Each argument in the list must be of the same data type as the matching parameter.

Here are some example functions:

A function that takes no arguments and returns nothing:

```
private void sayHello( ) {
    textBox1.AppendText( "Hello\n" );
}
```

A function that takes a single string argument and returns nothing:

```
private void greet( string aName ) {
    textBox1.AppendText( "Hello, " + aName + "\n" );
}
```

A function that takes two `int` arguments and returns a string:

```
private string addNumbers( int num1, int num2 ) {
    return "The total of " + num1 + " plus " + num2 + " is " + num1+num2;
}
```

To execute the code in a method, your code must 'call' it by name. In C#, to call a method with no arguments, you must enter the method name followed by an empty pair of parentheses like this:

```
sayHello( );
```

To call a method with arguments, you must enter the method name followed by the correct number and data-type of values or variables, like this:

```
addNumbers( 100, 200 );
```

If a method returns a value, that value may be assigned (in the calling code) to a variable of the matching data type. Here, the `addNumbers()` method returns a string and this is assigned to the `calcResult` string variable:

```
string calcResult;
calcResult = addNumbers( 100, 200 );
```

FUNCTIONS OR METHODS?

In some other programming languages, functions that return nothing may be called 'subroutines' or 'procedures'. In Object Oriented terminology, a function is generally called a 'method' and, in this course, the terms 'function' and 'method' will regarded as synonymous.

VALUE, REFERENCE AND OUT PARAMETERS

By default, when you pass variables to methods these are passed as 'copies'. That is, their values are passed but any changes made within the method affect only the copies that have been passed to the method. The original variable (outside the method) retains its original value.

Sometimes, however, you may want any changes made within a method to affect the original variables in the code that called the method. If you want to do that you can pass the variables 'by reference'. When variables are passed by reference, the original variables themselves (or, to be more accurate, the references to the location of those variables in your computer's memory) are passed to the function. So any changes made to the arguments inside the method will also change the variables that were used when the method was called. To pass *by reference*, both the arguments defined by the method and the variables passed when the method is called must be preceded by the keyword `ref`.

These examples should clarify the difference between 'by value' and a 'by reference' arguments:

Example 1: By Value arguments (the default in C#)

```
private void byValue( int num1, int num2 ) {
    num1 = 0;
    num2 = 1;
}

private void paramTestBtn_Click( object sender, EventArgs e ) {
    int firstnumber;
    int secondnumber;
    firstnumber = 10;
    secondnumber = 20;
    byValue( firstnumber, secondnumber );
       // firstnumber now = 10, secondnumber = 20
}
```

Example 2: By Reference arguments

```
private void byReference( ref int num1, ref int num2 ) {
    num1 = 0;
    num2 = 1;
}

private void paramTestBtn_Click( object sender, EventArgs e ) {
    int firstnumber;
    int secondnumber;
    firstnumber = 10;
    secondnumber = 20;
    byValue( firstnumber, secondnumber );
       // firstnumber now = 0, secondnumber = 1
     }
```

You may also use *out* parameters which must be preceded by the `out` keyword instead of the `ref` keyword. *Out* parameters are similar to *ref* parameters. However, it is not obligatory to assign a value to an *out* variable before you pass it to a method (it is obligatory to assign a value to a *ref* variable). However, it is obligatory to assign a value to an *out* argument within the method that declares that argument (this is not obligatory with a *ref* argument).

While you need to recognise *ref* and *out* arguments when you see them, you may not have any compelling reason to use them in your own code. As a general rule, I suggest that you normally use the default 'by value' arguments for C#.

Local Functions

You can use functions inside other functions. This feature was introduced in version 7 of C#. In the code show below, `addBonus()` is a local function:

LocalFunctions

```
private string ShowSalary(string aName, int earnings) {
    string msg;
    double bonus;

    double addBonus()
    {
        return earnings + (earnings * 0.05);
    }

    bonus = addBonus();
    msg = aName + " has a salary of " + bonus + "\n";
    return msg;
}
```

Here, the `addBonus()` function is 'local' to the `ShowSalary()` function because it is declared inside that function. Here `addBonus()` returns a double value. Local functions can return other types of data too, just like regular functions, or they can be declared using the keyword `void` which means they don't return any data.

Notice that I haven't used the keyword `public` or `private` as I would with a regular function. That's because those keywords define the visibility or 'scope' of a function. The scope of `addBonus()` is already defined. It is only visible inside the `ShowSalary()` function. That means that only code in the `ShowSalary()` function is able to call the `addBonus()` function. Meanwhile the `addBonus()` function can access the variables or parameters that are visible inside the `ShowSalary()` function. Here, you can see that `addBonus()` uses the `earnings` parameter.

You may call a local function from within the function that contains it, as I have done here:

```
bonus = addBonus();
```

You cannot call the local function from code outside the function that contains it, however. Local functions may be useful when you need to do some operation repeatedly within the context of some other function but nowhere else in your code.

STEP FIVE – OBJECT ORIENTATED PROGRAMMING

C# is an object oriented programming language. What this means is that everything you work with in C# – from a string such as "Hello world" to a file on disk – is wrapped up inside an object that contains the data itself (for example, the characters in a string) and the functions or 'methods' that can be used to manipulate that data – such as, for example, a string object's `ToUpper()` method, which returns the string in upper case.

OBJECT ORIENTATION

Each object that you use is created from a 'class'. You can think of a class as a blueprint that defines the structure (the data) and the behaviour (the methods) of an object.

Let's look at an example of a very simple class definition. You define a class by using the `class` keyword. Here I have decided to call the class `MyClass`. It contains a string, `_s`. I've made this string 'private' which means that code outside the class is unable to access the string variable. In order to access the string they must go via the methods which I've written – `GetS()` and `SetS()`. It is generally good practice to make variables private. By using methods to access data you are able to control how much access is permitted to your data and you may also write code in the methods to test that the data is valid. This is the complete class:

MyOb

```
class MyClass {
        private string _s;

        public MyClass( ) {
            _s = "Hello world";
        }

        public string GetS( ) {
            return _s;
        }

        public void SetS( string aString ) {
            _s = aString;
        }
}
```

At this point, the class doesn't actually do anything. It is simply a definition of objects that can be created from the `MyClass` 'blueprint'. Before we can use a `MyClass` object, we have to create it from the `MyClass` class. In the *MyOb* project, I declare an object variable of the `MyClass` type:

```
MyClass ob;
```

Before I can use the object, I need to create it. I do that by using the `new` keyword. This calls the `MyClass` constructor method. This returns a new object which I here assign to the `ob` variable:

```
ob = new MyClass( );
```

CONSTRUCTORS

When you create a class with `new`, a new object is created and any code in the constructor is executed. A *constructor* is a special method which, in C#, has the same name as the class itself.

This is the `MyClass` constructor:

```
public MyClass( ) {
       _s = "Hello world";
}
```

The code in my constructor assigns the string "Hello world" to the variable `_s`. You aren't obliged to write any code in a constructor. However, it is quite common – and good practice – to assign default values to an object's variables in the constructor. You can change the default value of the objects `_s` string variable using the `MyClass` object's `SetS()` method:

```
ob.SetS( "A new string" );
```

And you can retrieve the value of the `_s` variable using the `GetS()` method:

```
textBox1.Text = ob.GetS( );
```

CLASSES, OBJECTS AND METHODS

A 'class' is the blueprint for an object. It defines the data an object contains and the way it behaves. Many different objects can be created from a single class. So you might have one `Cat` **class** but three cat **objects**: *tiddles*, *cuddles* and *flossy*. A **method** is like a function or subroutine that is defined inside the class.

ONE CLASS PER CODE FILE?

In C#, you may define several different classes in a single code file. For example, the `Thing`, `Treasure` and `Room` classes might all be defined in a file called *GameClasses.cs*. However, many programmers consider it better to define just one class per code file. In large projects this may make it easier to find and maintain the classes you create. In that case, the `Thing` class would be defined in a file called *Thing.cs*, the `Treasure` class would be defined in *Treasure.cs* and the `Room` class would be defined in *Room.cs*.

REFACTORING

Visual Studio has a refactoring tool that can help you to extract classes from a code file containing several classes and place them into their own dedicated code files. Just right-click the class name (for example `Treasure`) in the class definition, select *Quick Actions and Refactorings*, then *Move Type To…* In this case, this would move the definition of the `Treasure` class into a code file named *Treasure.cs*.

CLASS HIERARCHIES

To create a descendent of a class, put a colon `:` plus the ancestor (or 'base') class name after the name of the descendent class, like this:

```
public class Treasure : Thing
```

A descendent class inherits the features (the methods and variables) of its ancestor so you don't need to recode them. In this example, the `Thing` class has two *private* variables, `_name` and `_description` (being private they cannot be accessed from outside the class) and two *public* properties to access those variables.

The `Treasure` class is a descendent of the `Thing` class so it automatically has the `_name` and `_description` variables and the `Name` and `Description` accessor properties and it adds on the `_value` variable and the `Value` property:

GameClasses

```
public class Thing {
    private string _name;
    private string _description;

    public Thing( string aName, string aDescription ) {
        _name = aName;
        _description = aDescription;
    }

    public string Name {
        get {
            return _name;
        }
        set {
            _name = value;
        }
    }


public string Description {
        get {
            return _description;
        }
        set {
            _description = value;
        }
    }
}

public class Treasure : Thing {
    private double _value;

    public Treasure( string aName, string aDescription, double aValue )
        : base( aName, aDescription ) {
            _value = aValue;
    }
```

```
    public double Value {
        get {
            return _value;
        }
        set {
            _value = value;
        }
    }
}
```

ACCESS MODIFERS – PUBLIC, PRIVATE, PROTECTED

When placed before a variable or function name, the keywords `public`, `private` and `protected` limit the scope or visibility of that variable or function. A `public` variable or function can be seen, and accessed, from code outside the object in which it occurs; a `private` variable or function cannot. Typically you will make variables `private` and provide `public` accessor methods when you want to get and set their values. When you want a method to be callable from code that uses an object, you will declare it as `public`. If, for some reason, you want a method to be callable only from other methods inside an object, then you will normally make that `private`.

There is another modifier that you may occasionally encounter called `protected`. A `protected` variable or method can only be accessed from within inside the class in which it is declared or from inside any *descendent* of that class. That's different from a `private` variable or method which can be accessed only from inside the *current* class but *not* from inside one of its *descendent* classes.

While `protected` variables and methods are sometimes useful, the two most common accessor levels are `public` and `private`. It is good practice to make the variables or 'fields' of objects `private`.

PROPERTIES

A *property* is a group of one or two special types of method to get or set the value of a variable. This is a typical definition for a property:

```
    private string _name;
    public string Name {
        get {
            return _name;
        }
        set {
            _name = value;
        }
}
```

Here `_name` is private but `Name` is public. This means that code outside the class is able to refer to the `Name` property but not to the `_name` variable. It is generally better to keep variables private and provide public properties in this way. If you omit the *get* part of a property, it will be impossible for code outside the class to retrieve the current value of the associated variable. If you omit the *set* part it will be impossible for code outside the class to assign a new value to the associated variable.

ALTERNATIVE SYNTAX

An alternative syntax for properties which need to do nothing more than get and set the values of private variables was introduced in version 7 of the C# language. With this syntax you can use an equals sign followed by a right-pointing bracket to get a variable or set it using the keyword `value`. For example, the `Name` property could be written as follows:

```
    private string _name;
    public string Name { get => _name; set => _name = value; }
```

Even though properties are like pairs of methods, they do not use the same syntax as methods when code refers to them. This is how to might get or set a value using a pair of *methods* named `GetName()` and `SetName()`:

```
myvar = ob.GetName( );
ob.SetName( "A new name" );
```

But this is how you would get and set a value using a *property* such as `Name`:

```
myvar = ob.Name;
ob.Name = "A new name";
```

When a descendant class receives arguments intended to match items defined by the ancestor class, it should pass those arguments to its ancestor by putting a colon after the argument list of the constructor then `base` then the arguments to be sent to its ancestor's constructor between parentheses. Notice how the `Treasure` class constructor passes `aName` and `aDescription` to the constructor of its ancestor, `Thing`:

```
public Treasure( string aName, string aDescription, double aValue )
            : base( aName, aDescription )
```

STEP SIX – ARRAYS, STRINGS AND LOOPS

One of the fundamental arts of programming is repetition. Whether you are developing a payroll application or a shoot-em-up game, you will need to deal with numerous objects of the same fundamental type - 100 employees' salaries, perhaps. Or 500 Bugblatter Beasts of Traal. We'll look here at just a few of the ways in which multiple objects can be arranged and manipulated using C#.

ARRAYS

An array is a sequential list. You can think of it as a set of slots in which a fixed number of items can be stored. The items should all be of the same type. As with everything else in C#, an array is an object. Strictly speaking, a C# array is an instance of the .NET class, `System.Array` and it comes with a built-in set of methods and properties. Just like other objects, an array has to be created before it can be used. To see a simple example of this, load up the *Arrays* project and find the `button1_Click()` method. The first line here creates and initialises an array of three strings:

Arrays

```
string[] myArray = new string[3] { "one", "two", "three" };
```

The definition of an array begins with the type of the objects it will hold, followed by a pair of square braces. This array contains strings, so it begins `string[]`. Next comes the name of the array, `myArray`. The `new` keyword is responsible for creating the array object. I specify that this array is capable of storing three strings by appending `string[]`. Optionally you may place an integer representing the number of array items between square brackets. I have done that in the declaration of myArray2:

```
myArray2 = new string[3] { "four", "five", "six" };
```

This integer is not in fact required here because C# can determine the actual number of strings from the items (the three strings) which I have placed between curly brackets. Putting comma-separated array items between curly brackets

provides a quick way of initializing an array. Notice that I have even omitted the `new` keyword when declaring and initializing `myArray3`:

```
string[] myArray3 = { "seven", "eight", "nine" };
```

Again, this is a shorthand alternative provided by C#. As a general rule, however, no harm is done by writing code that explicitly creates typed arrays of a defined length. The shorthand way of creating and initializing an array is only available when you do the process in a single line of code. This is how to declare an array first and initialize it later:

```
string[] myArray2;                                    // declaration
myArray2 = new string[3] { "four", "five", "six" };   // initialization
```

An array can hold any type of object. The `myObjectArray` array, for example, holds three objects of the class, `MyClass`:

```
MyClass[] myObjectArray = new MyClass[3];
```

Loops

In the code of the *Arrays* project you will see several `for` loops. The first `for` loop, fills `MyObjectArray` with three objects of the `MyClass` type:

```
for( int i = 0; i <= 2; i++ ) {
       myObjectArray[i] = new MyClass( "Object #" + i + ". " );
}
```

Arrays in C# are *0-based*, which means that the first item of a three-slot array is at index 0 and the last item is at index 2. A position in an array is indicated by the index number between square brackets.

FOR LOOP SYNTAX

Here is a simple example of a `for` loop:

```
for (int i = 1; i < 100; i++) {
     dosomething();
}
```

The code between the parentheses after the word `for` is used to control the execution of the loop. This code is divided into three parts:

1) `int i = 1` : The first part initialises an `int` variable, `i`, with the value of 1.
2) `i < 100` : The second section contains a test. The loop executes as long as this test remains true. Here the test states that the value of the variable `i` must be less than 100.
3) `i++` : Finally, we have to be sure to increment the value of `i` at each turn through the loop. That happens in the third part of the loop control statement (`i++`) which adds 1 to the value of `i`.

We could paraphrase the code of the loop shown above like this:

Let i equal 1. While i is less than 100 execute the code inside the loop, then add 1 to i.

Note that this loop will execute 99 times, not 100, since it will fail when the value of `i` is no longer less than 100.

Now look at loop 2 in the sample code:

```
for( int i = 0; i <= myArray.Length - 1; i++ ) {
       textBox1.Text = textBox1.Text + myArray[i] + ". ";
}
```

This loop iterates through the list of strings in `myArray` and displays them in `textBox1`. Note the expression in second part of the `for` loop:

```
i <= myArray.Length - 1
```

Instead of using a fixed number, such as 2, to specify final index of the array, I've used the array object's `Length` property. However, since `Length` is always 1 greater than the index (arrays being zero-based), I have subtracted 1 from `Length`.

The third loop uses two of the methods that are available for use with arrays, `GetLowerBound(0)` and `GetUpperBound(0)`, to determine the start and end indices of `myArray2`:

```
for( int i = myArray2.GetLowerBound( 0 ); i <= myArray2.GetUpperBound( 0 ); i++ ) {
       textBox2.Text = textBox2.Text + myArray2[i] + ". ";
}
```

These methods return the actual indices (here 0 to 2) of the array's first and last elements. Using these two methods makes the code more generic since it will work with arrays of any size. The 0 argument here indicates that I am testing the first *dimension* of an array. In fact, this array only has one dimension. C# can work with multi-dimensional arrays (arrays containing other arrays) too.

The fourth loop shows how similar techniques can be used to iterate through an array of user-defined objects and retrieve values (here the `Name` property) from each in turn:

```
for( int i = 0; i <= myObjectArray.GetUpperBound( 0 ); i++ ) {
       textBox3.Text = textBox3.Text + myObjectArray[i].Name;
}
```

FOREACH LOOPS

The fifth loop is more interesting. Instead of using `for` it uses `foreach`. The `foreach` statement iterates through each element in an array without any need for an indexing variable. A `foreach` loop is controlled by specifying the data type (here `MyClass`), an identifier to be used within the loop, here `myob`, and the name of the array or the collection:

```
foreach( MyClass myob in myObjectArray ) {
       textBox4.Text = textBox4.Text + myob.Name;
}
```

At each turn through the loop, the next `MyClass` object in `myObjectArray` is assigned to the variable, `myob`. I can then access this object's methods or properties, such as `Name`.

WHILE LOOPS

Sometimes you may want to execute some code an uncertain number of times – that is, while some test condition remains true. To do that, use a `while` loop.

WordCount

The *WordCount* project uses several `while` loops. These may test a single condition as in this loop which executes the code as long as the value of `charcount` is less than the length of `s`:

```
while( charcount < s.Length )
```

Or the loop may test multiple conditions, as here which executes the code in the loop only as long as *both* of *two* conditions remain true: 1) the value of `charcount` is less than the length of `s` and (`&&`) 2) the boolean value returned from the `IsDelimiter()` method when passed the character at the `charcount` index of the array `s` is *false* - that is, the value it is *not* (`!`) true:

```
while( ( charcount < s.Length ) && !( IsDelimiter( s[charcount] ) ) ) {
```

DO..WHILE LOOPS

In some circumstances, you may want to be certain that the code within a loop runs at least once. For example, you might want to prompt the user to enter some data – and you need to ensure that this is done *at least once*. One way of doing that would be to write a **`do..while`** loop.

A **`do..while`** loop is essentially the same as a **`while`** loop, apart from the fact that the test condition is placed at the *end* of the loop rather than at the beginning. This means that the statements inside the loop are bound to execute *at least once* before the test condition is even evaluated. This is the syntax of a **`do..while`** loop:

```
do {
   // one or more statements
} while (condition);
```

Here is a very simple example of a `do..while` loop in code:

dowhile

```
int[] intarray = { 1, 2, 3, 4, 5 };
int i;
i = 2;

do {
        Console.WriteLine(intarray[i]);
        i++;
} while (i < 2);
```

Here the test condition, `i < 2`, evaluates to **false** the first time it is tested. But since this test is only performed *after* the code in the loop, that code is run *once* and, in this case, this output is shown (the integer at index 2 in `intarray`):

```
3
```

Compare that with this `while` loop:

```
i = 2;
while (i < 2) {
  Console.WriteLine(intarray[i]);
  i++;
}
```

Here the test (which is the same test as in the do..while loop) – this test is evaluated at the *start* of the loop, it evaluates to *false* so the code inside the loop *never* executes. Not even once. That is an important difference to understand between a regular `while` loop with the test condition at the *start* and a `do..while` loop that has the test at the *end* of the code to be run. No matter what happens a `do..while` loop must run at least once. But in some cases, if the test condition evaluate to false right away, then the code inside a regular `while` loop may never run at all.

STRINGS AND CHARS

Strings

In C#, a string can be entered as a series of characters delimited by double quotes. The data type is `string` (in lower case). This is, in fact, a shorthand alias for `System.String` in the .NET framework.

String objects can make use of a broad range of methods belonging to the `System.String` class. These include methods to find substrings, trim blank space and return a copy of the string in upper or lower case. Strings also have a `Length` property.

CHAR

A single character has the `char` data type and is delimited by single quotes:

```
char mychar = 'a';
```

The characters in a string can be accessed by specifying an integer to index into the string. Just like arrays, strings are zero-based so the first character is at position 0 rather than 1.

Assume that the string `s` contains the text, "Hello world!". The expression, `s.Length`, would return a value of 12. This counts the number of characters in the string from 'H' to '!'. However, in order to assign the 1st character, 'H', and the 12th character, '!', to the char variables, `c1` and `c2`, you would need to use these expressions:

```
char c1 = s[0];
char c2 = s[11];
```

To see some working examples, try out the code in the *strings* project.

FORMAT STRINGS

I first mentioned the very useful `String.Format()` method back in Chapter 2. You may recall that the `Format()` method provides a simple way of substituting values from variables or expressions into a string. The string is passed to the method as the first argument; and a comma-delimited list of values is placed after the string. When numbered placeholders between curly brackets are placed into the string, values from the comma-delimited list will be inserted at the positions of the placeholders.

stringinterpolation

```
String.Format("{0} of String.Format(): 1+2 = {1}. {2}. {3}", mystring, 1 + 2, "hello world".ToUpper(), Multiply(10, 5));
```

The first value, which is at index 0 (here that's the variable `mystring`) replaces the `{0}` placeholder, the second value at index 1 (here that's 3 – the value returned from the expression `1+2`) replaces the `{1}` placeholder and so on.

You can see from this example, that I am not limited to displaying a simple variable. I can also show the result of a calculation, I can call a method on an object such as `"hello world".ToUpper()` or I can call a function of my own such as `Multiply(10, 5)` and display its return value.

You don't have to display the arguments in the order in which they appear in the list. Here for example, I display them in reverse order, the 5th argument, that's 6, then the 4th, which has the value 5 and so on down to the first at index 0 which has the value 1, and I also repeat the arguments `{5}`, `{4}` and `{3}`:

```
String.Format("{5} {4} {3} {2} {1} {0} {5} {4} {3} ", 1, 2, 3, 4, 5, 6);
```

STRING INTERPOLATION

As an alternative to using `String.Format()` with placeholders C# also lets you include expressions right inside the string itself. This is called string interpolation. This feature was first introduced in version 6 of the C# language.

To create an interpolated string you need to precede the string itself with the `$` character. Then include any expressions to be evaluated between curly brackets in the string. This is an example:

stringinterpolation

```
s = $"(a) {mystring} of string interpolation: 1+2 = {1 + 2}. {"hello
world".ToUpper()}. {Multiply(10, 5)}";
```

It turns out that the WriteLine method of Console, which I'm using here, will evaluate these expressions even without preceding the string with a dollar symbol. You can see that here. But in most cases, the dollar symbol is needed.

Even ordinary strings evaluate certain special characters, for example, substituting a newline for \n and a tab for \t as shown here. Those embedded characters are also evaluated in interpolated strings as you can see in this next example. But what if you want to prevent any such evaluation?> Well, then you need to create a verbatim string. You do that by putting an ampersand in front of the string. This final example down here is a verbatim string. And as you can see when I display that nothing is evaluated – even \n and\t are shown literally rather than as newline and tab characters.

VERBATIM STRINGS

Even standard C# strings evaluate certain special characters. For example, `\n` is evaluated as a newline character and `\t` is evaluated as a tab character. Consider this standard string:

stringinterpolation

```
s = "\nThis is a \t{mystring}\n";
```

This is now displayed by `Console.WriteLine()` – a function which evaluates both special characters in standard strings and also any values between curly braces in interpolated strings:

```
Console.WriteLine(s);
```

This is the result (note the gap made by the tab character and the newline made by the newline character):

```
This is a       {mystring}
```

Here is an interpolated string (the string "Test" has previously been assigned to the `mystring` variable):

```
s = $"\nThis is a \t{mystring}\n";      // string interpolation
Console.WriteLine(s);
```

And this is the result:

```
This is a       Test
```

But what if you want to prevent any such evaluation? In that case you need to create a verbatim string. You do that by putting @ in front of the string. This is an example of displaying verbatim string:

```
s = @"\nThis is a \t{mystring}\n";      // verbatim string
Console.WriteLine(s);
```

The result in this case is the literal string: nothing is evaluated:

```
\nThis is a \t{mystring}\n
```

DIALOGS AND MESSAGE BOXES

This step's projects use several different types of dialog box. You may have noticed that the *WordCount* project uses the `MessageBox` dialog. This is defined entirely in code. The `MessageBox.Show()` method is passed a number of arguments to display a string, one or more buttons and an icon. Refer to the .NET help system for a full explanation.

Editor.sln

The *Editor* project uses *OpenFile* and *SaveFile* dialogs which are displayed when menu items are clicked in the application's File menu. These dialogs are configured by setting properties in code – such as the default file extension, *DefaultExt*, and the *Filter* (the file extensions available from the dialog's File Type selector.

You can, if you wish, create Open and Save dialogs in code, like this:

```
OpenFileDialog openFile1 = new OpenFileDialog( );
```

But in this project, I simply dropped the two dialog controls from the Toolbox. This has the benefit that their properties can be set using the Properties panel. In fact, if you wish, you can delete the *Filter* and *DefaultExt* assignments from the code and use the Properties panel to set them instead.

STEP SEVEN – FILES AND DIRECTORIES

File-handling is one of the fundamental skills of programming. Whether you are programming a spreadsheet, a web browser, a word processor or a game, you will, at some stage, have to read data from one place and write it to another. In this step we shall be exploring the classes and techniques needed to master all kinds of data reading and writing operations in C#.

FILES AND IO

In the last step we created a simple editor from a `RichTextBox` control (the *Editor* project). This control comes with its own `SaveFile()` and `LoadFile()` methods. So, for example, this is how I was able to save the contents of the control named `richTextBox1` to a file whose name the user had entered into the `SaveFileDialog` named `saveFile1` - that is, as the `FileName` property:

```
richTextBox1.SaveFile( saveFile1.FileName, RichTextBoxStreamType.RichText );
```

Incidentally, the final parameter in the above code specifies that rich text formatting is to be retained. To save it as simple ASCII text (text that does not retain colours, font style and so on), you would need to replace `RichText` with `PlainText`. This is all well and good if you happen to be saving data from a rich text component, which has all this behaviour 'built in'. But what if you happen to be saving data from your own, non-visual objects or loading data from a non-text file?

STREAMS

To do this we need to learn a bit about the IO (Input/Output) features of the .NET framework. In particular, I want to explain the concept of a 'stream'. A stream is a sequence of bytes (a byte is a chunk of data approximately corresponding to a single character) that can flow from one place to another. Often it will flow from the hard disk into memory or vice versa. But a stream could equally flow from one place in memory to another place in memory or from one computer to another.

VARIATIONS ON A STREAM

There are various different of more or less 'specialised' types of stream in .NET. Here I want to concentrate only on the essential features of streams which you will need to read and write data to and from disk. Be warned: the nitty-gritty details of reading and writing streams may be quite hard to understand at first. Here I describe the main options but you may not need to use all of these in your own code. You may want to try out my sample project so that you have a general overview of streaming but you certainly don't need to memorise all the details!

Let's start by seeing how to use basic Stream objects to make a copy of a file. In this project, we will be making copies of the file, *Text.txt*. Our code assumes that this file can be found in the **\Test** directory on the C drive.

> Before continuing, use the Windows Explorer to create the *\Test* directory on your C drive and copy the *Test.txt* file (from *\Step07\streams*) into it. If you fail to do this, the sample program will not work! Be sure to keep the Windows Explorer open on the *C:\Test* directory when you run the program so you can see which files are created.

Streams

Now load up the *Streams* project. Scroll to the top of the code editor. You will see that I have specified the source file and directory here:

```
const string SOURCEFN = "C:\\Test\\test.txt";
```

Note that I have also added `System.IO` to the `using` section at the top of the code. The `System.IO` namespace contains all the essential IO classes that we'll need such as `File`, `StreamWriter`, `StreamReader` and the entire hierarchy of Streams.

Switch to the form designer and double-click the top button, labelled 'Write Stream'. This will take you to the button's event-handling method which is called `WriteStreamBtn_Click ()`. I declare the target output file name, `OUTPUTFN`, and I create two stream objects:

```
const string OUTPUTFN = "C:\\Test\\Stream.txt";
Stream instream = File.OpenRead(SOURCEFN);
Stream outstream = File.OpenWrite(OUTPUTFN);
```

Notice that I use double-slashes '\\' for directory separators. That's because a single slash in a string is used to indicate that the letter following it represents a special character. For example '\n' is a new line and '\t' is a tab. When I want to put an actual '\' character in a string, I need to precede it by another '\' character which is what I've done here.

Here, each stream is created by the `File` class. The `OpenRead()` and `OpenWrite()` methods create files for reading and writing and they each return a `FileStream` object. Note that these methods are `static` which means that you can use them by referring to the `File` class itself rather than to a specific `File` object. You can think of a `static` method as a method that 'belongs' to the class itself rather than to an individual object created from the class. This is why it is not necessary to create an instance (an object) of the `File` class using `new`.

I'll have more to say on `static` methods in *Chapter 8*.

Now I create a 'buffer' into which to read data. This buffer is an array of 1024 bytes (I set the `BUFFSIZE` constant to 1024 at the top of the code):

```
byte[] buffer = new Byte[BUFFSIZE];
```

A `while` loop continually reads bytes from the input stream, `instream`, into this buffer as long as there are more bytes to be read (the reading is all done inside the parentheses at the start of this `while` loop and the bytes read are assigned to the `numbytes` variable until the value is 0 when there are no more bytes to be read). The bytes that have been read (whose total number is stored in the `numbytes` variable) are then written into the output stream, `outstream`:

```
int numbytes;
while ((numbytes=instream.Read(buffer,0,BUFFSIZE)) > 0)
{
       outstream.Write(buffer,0,numbytes);
}
```

If you are interested in finding out more on the `Read()` and `Write()` methods of Stream, refer to the .NET help (press F1 over the method names in the code editor). When all the reading and writing is completed, the two streams are closed:

```
instream.Close();
outstream.Close();
```

The real problem with this code is that it is inefficient since it reads and writes data one byte at a time. You may not notice this when working with a small file. But it might be noticeable when working with very long files. We could make the code more efficient by reading larger blocks of bytes in one go. We can do this by creating a `BufferedStream` object. This is done in the `BuffStreamWriteBtn_Click()` method. This is how we've created a `BufferedStream buffInput` from the `FileStream` object, `instream`:

```
BufferedStream buffInput = new BufferedStream(instream);
```

The rest of the code is much as before. The only difference is that it uses the buffered stream objects rather than the basic stream objects. To ensure that all data is written ('flushed') to disk, the `BufferedStream` class provides the `Flush()` method. However, when the output stream is closed, any buffered data is automatically flushed, so it is not necessary to call the `Flush()` method here.

The `FileStreamBtn_Click()` method implements an equivalent file-copying routine to the one we've just looked at. It combines the efficiency of a `BufferedStream` with the simplicity of an unbuffered Stream. There is, in fact, much more to a `FileStream` than we have space to explore here. It comes with many methods that can assist in file handling. Refer to the .NET help documentation for more information.

Most of the code in this method needs no explanation. The only new feature is the `FileStream` constructor:

```
FileStream instream = new FileStream(SOURCEFN,
                          FileMode.OpenOrCreate,
                          FileAccess.Read );
```

The `FileStream` class has several constructors. The one I use here takes the three arguments: the *file name* parameter, the *file mode* parameter (which is a pre-declared constant that specifies how to open or create a file) and the *file access* parameter which is a constant that determines whether the file can be read from or written to.

The three file copying methods I've created up to now, operate on bytes and can be used to copy data of any type. I've used them to copy text files but they could just as well be used to copy an executable file. If the input file were *wordpad.exe* and the output file were named *mycopy.exe*, all three routines would make a correct copy of the wordpad program.

READING AND WRITING TEXT FILES

There is another way of handling text files. The `StreamReader` and `StreamWriter` classes can read and write a text file one line at a time. This makes it pretty easy to deal with files of plain text. A line is defined as a sequence of characters followed by a line feed ("\n") or a carriage return immediately followed by a line feed ("\r\n"). The string that is returned does not contain the terminating carriage return or line feed. Unlike the classes I've used previously, these classes are not descendants of the `Stream` class. Instead, they operate *upon* Stream objects.

The `File` class provides the methods, `OpenText()` and `CreateText()` which, when passed a file name as an argument, will return a `StreamReader` or a `StreamWriter` object.

An example of this can be seen in the `StreamWriterBtn_Click()` method.

To read a line, use `StreamReader`'s `ReadLine()` method. To write a line use, `StreamWriter`'s `WriteLine()`. In my code, a `while` loop reads through the lines of text from the input file. The lines of text are read in by the `StreamReader` object, `sread`, as long as there are more lines to be read (that is, until `null` is returned). The string variable, `aline`, is initialised with the current line which is then written by the `StreamWriter` object, `swrite`, to the output file:

```
StreamReader sread;
StreamWriter swrite;
String aline;
sread = File.OpenText( SOURCEFN );
swrite = File.CreateText( OUTPUTFN );
while( ( aline = sread.ReadLine( ) ) != null ) {
       swrite.WriteLine( aline );
}
sread.Close( );
swrite.Close( );
```

In a finished application, it would be good practice always to test whether a file exists before attempting to read data from it. The `StreamWriterBtn_Click()` method does this using the test:

```
if( !File.Exists( SOURCEFN ) )
```

In this test, I use the C# '*not*' operator, '`!`', so that the code above could be read as *"if not File.Exists(SOURCEFN)"*.

Appending Data To A File

There may be occasions when you want to modify an existing file. There are various ways in which this can be done using C#. If you are working with plain text files (or other file types such as RTF, which contain ordinary ASCII 'text' characters), there is a very easy way of appending data. Look at the `AppendBtn_Click()` method in the *Streams* project. This creates the `StreamWriter` object, `swrite`, using the `File` class's `AppendText()` method. This cause any text that is subsequently written to be added to the end of the specified file if that file already exists. If the file does not exist, it is created ready for text to be written into it:

Streams

```
sread = File.OpenText( SOURCEFN );
swrite = File.AppendText( OUTPUTFN ); //!!
while( ( aline = sread.ReadLine( ) ) != null ) {
       swrite.WriteLine( aline );
}
```

The File Class

If you are dealing with files in .NET, it is worth getting to know the `File` class. All the methods of this class are `static` so you won't have to create new `File` objects in order to use them. In the example code I have used the methods: `CreateText()`, `AppendText()` and `Exist()`.

The methods of the `File` class need to be passed a string argument indicating the directory path and file name. The directory separators take the form of a double backslash "`\\`". The `File` class has other useful capabilities. Its `GetAttributes()` method retrieve a file's attributes, `Delete()` deletes a file, `Move()` and `Copy()` move or copy a file. I could have used `File.Copy()` to do all the copying which I have laboriously coded in this step. Of course, if I had done that, I would never have had the opportunity to try out the `Stream` and `StreamReader` classes. You will need to use the `Stream` class and its descendants when you want to process data in some way or save data from user-defined objects. So, while the `File` methods are useful to know about, they are no substitute for a full grasp of the inner life of .NET's streams!

THE DIRECTORY CLASS

The `Directory` class can be used to create, move and enumerate through directories and subdirectories. It provides `static` methods which means that (just like the methods of the `File` class) you do not need to create an object from the class in order to use its methods.

Dir

Load the *Dir* project and locate the `dirBtn_Click()` method. This retrieves the names of the drives that are active or mapped on your system. To do this it calls `Directory.GetLogicalDrives()` method to initialise a string array of the drive names:

```
string[] drives = Directory.GetLogicalDrives( );
```

To display those names, the code iterates through the strings using a `foreach` loop. Retrieving the name of the currently active directory is even simpler.

```
string currdir = Directory.GetCurrentDirectory( );
```

In order to find the top-level directory, you can use the `GetDirectoryRoot()` method. This takes a string argument representing the path of a file or directory. My code uses the `currdir` string, which I have already initialised to the current directory:

```
string dirroot = Directory.GetDirectoryRoot( currdir );
```

If you want to retrieve the names of any subdirectories too, use the `GetDirectories()` method to return an arrays of strings. Then use a `foreach` loop to iterate through them. My code iterates through all the directories immediately below the root:

```
string[] subdirs = Directory.GetDirectories( dirroot );
foreach( string sd in subdirs ) {
       richTextBox1.AppendText( sd + "\n" );
}
```

You can also use the `Environment` class to obtain paths to special directories such as the System or Program Files directories. The `Directory` class could then be used to work with the files or subdirectories in those directories.

For example, this code will display the subdirectories in the Program Files folder:

```
string[] subdirs = Directory.GetDirectories( Environment.GetFolderPath(
                            Environment.SpecialFolder.ProgramFiles ) );
foreach( string sd in subdirs ) {
       richTextBox1.AppendText( sd + "\n" );
}
```

THE PATH CLASS

Having obtained a string representing the path to a subdirectory or file, you may want to do certain operations such as parse out the file name or extension or the path (the full directory specification) minus the file name. The `Path` class has these and other capabilities built in.

Load the *Dir* project and find the `pathBtn_Click()` method. This starts by getting the path to the executable file of the `Dir.exe` program itself (the program that is created when you compile or run this project). The `Executable` property of the `Application` class supplies this:

```
string path = Application.ExecutablePath;
```

Now, if you want to get a string initialised with the directory name, minus the file name, you can just pass the `path` variable to the `GetDirectoryName()` method of the `Path` class as follows:

```
Path.GetDirectoryName( path )
```

In a similar manner, you can pass the `path` variable to the `GetExtension()`, `GetFileName()`, `GetFileNameWithoutExtension()` and `GetPathRoot()` methods of the `Path` class. Each of these methods returns a modified version of the original string. The names of the methods are self-explanatory and you can view the results by running the *Dir* application and clicking the 'Path Test' button.

STEP EIGHT - CLASSES AND STRUCTS

In this step, I look at some additional features of classes and structures (*structs*). I also explain *Enums*. The text here summarizes the essential features.

ONE CLASS PER CODE FILE

While you may put more than one class into a single code file, it is often considered better practice to give each class its own code file. For example, if you create a class named MyClass you can write its code in a file named *MyClass.cs* and avoid adding any other classes to that file.

ONE CLASS ACROSS MULTIPLE CODE FILES

In complicated programs, it sometimes happens that you need to write hundreds or even many thousands of lines of code in a single class. This may make your code hard to navigate. In that case, you can divide one class across several different code files. You do this by preceding the class name with the keyword `partial`. For example, in the *ClassesAndMore* solution, MyClass is a partial class and its code is divided between two code files. The *MyClass.cs* file contains most of the code:

ClassesAndMore.sln

```
partial class MyClass {
       private string _str = "";

       public MyClass( ) {
            _str = "A Default String";
       }
       // more code here. . .
}
```

And the same class is 'continued' in the *MyClass2.cs* code file:

```
partial class MyClass {

// more code here. . .
     }
```

STATIC METHODS AND CLASSES

A class may contain both ordinary methods (which are called by referring to an object created from the class) and static methods (which are called by referring to the class itself. Typically, normal methods act upon an object's internal data while static methods may be defined to act upon 'external' data that's passed to them for some sort of processing. For example, the .NET **File.Exists()** method is static. You can pass a file name to it as an argument and it tells you whether or not that file exists on disk. In the MyClass class, I've defined the method **ToUpCase()** as static and **ToLowCase()** as a normal (or 'instance') method. This is how I call the static method:

```
string s;
s = MyClass.ToUppCase( "abc" );
// s now equals "ABC"
```

And this is how I call the normal method:

```
MyClass ob1;
ob1 = new MyClass( "Hello World" );
string s;
s = ob1.ToLowCase( );
// s now equals "hello world"
```

If you want to create a class that contains nothing but static methods you can make the class itself static. I've done this with the MyStaticClass class:

```
static class MyStaticClass {
```

A static class does not permit objects to be created from it. The .NET framework includes several static classes such as File and Directory.

OVERLOADED METHODS

In C# (but not in all other languages!) it is permissible to define multiple methods with the same name inside a single class. You can even create multiple constructors for a class. Each constructor or each similarly-named method must have a different set of arguments, however. It is the argument list which resolves the ambiguity of the repeated names and allows C# to determine which method the programmer intends to call. The MyClass class, for example, defines three alternative methods called **ToLowCase()**. Two of these are static methods (and these return the lowercase version of one or more string arguments), one of them is a normal method that returns the lowercase version of a MyClass object's **_str** variable:

```
public static string ToLowCase( string aString ) {
	return aString.ToLower( );
}

public static string ToLowCase( string aString, string anotherString ) {
	return ( ToLowCase( aString ) + ToLowCase( anotherString ) );
}

public string ToLowCase( ) {
	return ToLowCase( _str );
}
```

This class also has two constructors. The first takes no arguments so it sets a default value for **_str**. The second takes a string argument which is assigned to **_str**:

```
public MyClass( ) {
	_str = "A Default String";
}

public MyClass( string aString ) {
	_str = aString;
}
```

STRUCTS

As an alternative to a class, you may create a *struct*. Unlike a class, a struct cannot have descendants. In addition, its constructor cannot have an empty argument list. You will see an example of a struct in the file *MyStruct.cs*. This implements a structure that contains an **x** and a **y** coordinate, similar to a .NET *Point* which its itself a struct.

ENUMS

You can create a group of constants called an *Enum*. To do this you define a coma-separated list of identifiers between curly braces like this:

```
public enum CardSuits {
        Clubs,
        Spades,
        Hearts,
        Diamonds,
        Unknown
    }
```

By default, each constant is assigned a numeric value from 0 upwards. However, often Enums are used simply to provide descriptive names rather than to supply associated values. You may then assign values to variables of the specific Enum type like this:

```
CardSuits selectedSuit;
selectedSuit = CardSuits.Clubs;
```

If you want to display one of the Enum's identifiers as a string, you can use the `ToString()` method:

```
textBox1.Text = selectedSuit.ToString( );
```

You may also assign specific numeric values to elements of an Enum like this:

```
public enum PictureCards {
        Jack = 11,
        Queen = 12,
        King = 13,
        NotAPictureCard = 0
    }
```

Enums are used in various places throughout the .NET framework. For example, if you select a control in the Form designer, you can change its docking behaviour by clicking the `Dock` property. You can anchor a control (so that its edges are auto-resized when the control containing it is resized) using the `Anchor` property. These two properties are assigned values from the *DockStyle* and *AnchorStyles* Enums like this:

```
textBox1.Dock = DockStyle.Top;
```

```
textBox1.Anchor = AnchorStyles.Top;
```

You can "add" Enums together using the single upright bar | operator. This will not necessarily result in a sensible value for all Enums. However, some Enums expect this sort of operation. The *AnchorStyles* Enum, for example, can anchor a control at the Bottom and Left of its container like this:

```
textBox1.Anchor = AnchorStyles.Bottom | AnchorStyles.Left;
```

STEP NINE – EXCEPTION-HANDLING

EXCEPTIONS

There is one important little problem I have not yet considered in any depth in this course. Namely: *bugs*! Few programs of any ambition are totally bug-free. At least, they aren't at the outset. But, with a little care, and a lot of debugging, it should be possible to squash most of the most troublesome bugs before the end-user is let loose on your application. In some respects you could say that the programmer's greatest enemy is the user. The trouble with users is that they don't play to the rules. Your code expects them to do one thing but they go and do something else altogether.

It is your responsibility, therefore, to assume that the user will, at some time or another, do the most stupid things conceivable. And you have to build into your code some means of recovering from potential disaster. Fortunately, C# and .NET make this fairly easy to do using exception-handling.

Imagine that you've created a calculator of some kind. Obviously, you expect the user to do calculations using numbers. But what happens if, instead of entering the letters 1 and 0 (one and zero), the user enters I and O (that is, the capital *letters* I and O)? The user may be dumb, but that's no excuse. Your program has to be clever enough to cope with it.

In C# and .NET, runtime errors (errors that occur when the program is running) are handled by *exceptions*. As with everything else in C#, an exception is an object. When an error happens, an instance (and object) of the Exception class is created. Your code can make use of this by trapping the exception object and accessing its properties and methods. Load and run the *Exceptions.sln* solution. Enter 'IO' (letters, not numbers) into the subtotal edit box at the top of the form. Now click the first button, labelled 'Calc'. A system error pops up to tell you that there is an unhandled exception. This may be acceptable when you are *developing* an application. But is not the kind of message an end user expects to see when *running* it. Click the 'Continue' button.

As before, make sure the two letters 'IO' are in the subtotal edit box. This time click the second button, 'calc2Btn'. This time you will see quite a different, and altogether more polite, error message. This is because the code that executes when this button is clicked 'catches' the exception object handles the exception.

Take a look at the code of `calc2Btn_Click()` in the editor. This is the exception-handling code:

```
try {
       st = Convert.ToDouble( subTotBox.Text );
} catch( Exception exc ) {
       MessageBox.Show( "Awfully sorry to bother you, but apparently the " +
       exc.Message, "Oops! There has been an error of the type: " + exc.GetType( ),
       MessageBoxButtons.OK, MessageBoxIcon.Error );
}
```

Here the bit of code that could potentially cause a problem is the first line which attempts to convert the text in **subTotBox** to a *double* value. This code has been put between the curly brackets following the **try** keyword. So, during execution, C# tries to run this code and convert the data. If this attempt fails (i.e. if the text cannot be converted to a double) then the code block following the **catch** keyword is run. Note that the **catch** block takes an argument, **exc** (the name here is not important) of the type **Exception**.

When a problem occurs in the **try** block, the variable **exc** is initialised with the exception object which contains all kinds of useful information about the error that has just occurred. For example, it contains a **Message** property which gives access to a string describing the error. In the present case, **Message** equals *"Input string was not in a correct format"*. I display this message in a MessageBox. The exception object also has a **GetType()** method. This returns a string description of the exact error type. Here this is *"System.FormatException"* and I display this in the caption of the message box.

An exception object has many other properties and methods that can provide even more detailed information on an error. Try, for example, editing the code shown above by replacing **exc.Message** with **exc.ToString()**.

EXCEPTION TYPES

It is also possible to catch specific *types* of exception. For an example of this, look at the code in **calc3Btn_Click()**:

```
try {
	st = Convert.ToInt32( subTotBox.Text );
} catch( OverflowException exc ) {
	MessageBox.Show( "Try a smaller number! " + exc.Message,
		"Yikes! Overflow error: " + exc.GetType( ),
		MessageBoxButtons.OK, MessageBoxIcon.Warning );
} catch( Exception exc ) {
	MessageBox.Show( "Awfully sorry to bother you, but apparently the " +
		exc.Message, "Oops! There seems to have been a slight error of the type:
		" + exc.GetType( ),
		MessageBoxButtons.OK, MessageBoxIcon.Error );
}
```

This expects the user to enter a 32-bit integer into the subtotal edit box. This means that if the user enters a double such as 1.5 or an alphanumeric character such as 'X', a *FormatException* will occur. However, another type of exception is also possible. An *int32* value must fall between the range of -2,147,483,648 and 2,147,483,647. If the user enters a number larger or smaller than these values, an *OverflowException* will occur. Try it. Enter 'X' into the subtotal box and click the *Calc3* button. You will see the same error message as previously. Now enter eleven or more digits (e.g. 999999999999) into the subtotal box and once again click *calc3*. This time, you will see a different error message which states 'Try a smaller number!'.

To handle a particular exception type, just specify that type in a **catch** section. Here I have specified *OverflowException*. I could subsequently add separate blocks specifying other exceptions such as *FormatException*. Each block will only execute if the specified exception object is found. If not, the code moves on to the next **catch** block. The final **catch** block should normally specify the base *Exception* class and this will execute if an exception of any type has occurred which has not already been handled by an earlier **catch** block.

You may also nest one **try..catch** block inside another. And you may use a **finally** block instead of a **catch** block or place a **finally** block after one or more **catch** blocks. A **finally** block will execute whether or not an exception has occurred and it may be used to reset values of any data which may be left in an unpredictable state following an exception. You will find examples of a nested **try..catch** and **finally** block in **calc5Btn_Click()**.

DEBUGGING

Visual Studio has one of the best debuggers available anywhere, so be sure to make good use of it. Typically you will start by placing breakpoints in your code to pause execution of your program at one or more points where you want to examine the values of variables.

To place a breakpoint, click in the left-hand shaded margin of a code file. The breakpoint will be shown as a red circle in the margin and a highlighted red line in the code. Now press F5 to run your program in the debugger. The program will stop when a breakpoint is encountered.

Use the Locals window to view variables that are in scope or the Watch window to view selected variables. You can add variables to the Watch window by entering them as text or by dragging them out of a code window. You can also evaluate variables or expressions (for example, mathematical expressions such as *10 * 5*) in the Immediate window. You can view the calls that have been made (that is the sequence of methods that have executed prior to arriving at the current method) in the Call Stack window.

Important Debugging keys:

F5 – Continue Debugging (until a breakpoint is hit)
F10 – Step Over (debug to next line of code but do not step into any methods)
F11 – Step Into (debug to next line of code, step into methods if necessary)
SHIFT-F11 – Step Out (step to next line of code after the code of the current method)
SHIFT-F5 – Stop Debugging

You can find debugging commands on the Debug menu or on the Debug Toolbar which will normally appear above the editing window during a debugging session. If the Debug toolbar is not visible, right-click the Toolbar area and select 'Debug' from the drop-down menu.

STEP TEN – LISTS AND GENERICS

In this step, I plan to start work on creating an adventure game in which the player can move around a map taking and dropping objects. In order to do that I need to manipulate lists of objects so that, for example, an item can be transferred from one list (belonging to a Room) to another list (belonging to the Player) when it is taken. To do this, I shall use some special .NET collection classes.

GENERIC COLLECTIONS

We looked at simple arrays earlier in this course. Arrays are sequential lists of items. The .NET framework also supplies several other collection classes such as *List* and *Dictionary*. These are called 'generic collection' classes.

LISTS

A *List* represents a strongly-typed collection of objects and it comes with lots of useful methods to add, remove and locate objects in the collection. This is called a 'generic' list. The syntax for declaring a generic list is:

```
List<T>
```

Here **T** is the name of the type of the items in the list. In real code you might declare lists of strings or Thing objects like this:

```
List<string>
List<Thing>
```

This is how I declare and construct a List typed to hold Thing objects:

Generics.sln

```
public List<Thing> thingList = new List<Thing>();
```

I can now add an object like this:

```
thingList.Add( new Thing( "Sword", "An Elvish weapon" ) );
```

I can remove an object at index `i` like this:

```
thingList.RemoveAt( i );
```

DICTIONARIES

The .NET Framework also has a *Dictionary* class. A Dictionary is a type of list in which the items are indexed not by a sequential numerical index but by a *key* which may be a unique object of any type. You can think of a Dictionary as the programming equivalent of a real-world dictionary in which each entry has a unique name as a '*key*' – for example, "Dog" – followed by a definition which is its '*value*' – for example: "A furry mammal that woofs and gnaws on bones."

The declaration of a Dictionary is a bit like the declaration of a List but it requires two types (the key and the value) between a pair of angle-brackets. This is how I might declare and construct a Dictionary with a string key and string value and add a single item to it:

```
public Dictionary< string, string > petDictionary = new Dictionary<string, string >();
petDictionary.Add( "Dog", "A furry mammal that woofs and gnaws on bones" );
```

This is how I would declare and construct a Dictionary with a string key and Room value and add a single item to it:

```
     public Dictionary< string, Room > roomDictionary = new Dictionary<string,
Room>();

roomDictionary.Add( "Troll Room", new Room( "A dank cave" ));
```

I can try to get a value associated with a key like this:

```
string value = "";
petDictionary.TryGetValue( "Dog", out value);
```

I can also use other methods to (for example), determine whether a Dictionary object contains a specific key and remove an object if it exists:

```
string searchname = nameTB.Text;
if( roomDictionary.ContainsKey( searchname ) ) {
       roomDictionary.Remove( searchname );
}else {
       // do something else…
}
```

This is an example of iterating over the items in the Dictionary and accessing the key and the value from each item:

```
foreach( KeyValuePair<string,Room> kvp in roomDictionary) {
       s += String.Format( "{0}: {1}\r\n", kvp.Key, kvp.Value.description) ;
}
```

In this case, as the Dictionary has been typed to hold Room items I am able to access the Room objects' **description** property without having to 'cast' the item to a Room type inside the loop.

OVERRIDDEN METHODS

In my adventure game, objects of different sorts need to describe themselves in different ways. That is, each class (such as Room and Player) that descends from the Thing class must have a different implementation on the `describe()` method. But sometimes I will call the `describe()` method for each object in a list. I have to be sure that the correct version of `describe()` is called. My solution to this problem is to make `Thing.describe()` method 'virtual':

```
public virtual string describe( )
```

The descendant classes must then `override` this virtual method. To do this you must add the keyword `override` before the `describe` method name in the descendant classes like this:

```
public override string describe( )
```

To understand why virtual methods are needed, let's consider how a normal 'non-virtual' method works. Let's suppose you have defined a class that has a method with the same name and argument list as a method of its ancestor class. Imagine, for example, that your program contains an ancestor class, **A**, and a descendant class, **B**. A descendant class is compatible with its ancestor. You might say that class B is a *type of* class A. For simplicity, let's suppose that **A.method1()** returns the string, *"class A: method1"* whereas **B.method1()** returns *"class B: method1"*.

Now let's suppose that class A has a method called **method1()** and class B also has a method called **method1()**. Instances of these classes are created as follows:

```
A mya1 = new A( );
B myb1 = new B( );
```

Were my code now to call **mybOb.method1()**, it would, of course, return the string **"class B: method1**". But now consider what would happen if you were dealing with a collection of mixed objects, some of which might be A objects, others B objects and others C, D and E objects.

All of the objects in this collection are either A objects or are descendants of A objects. You add these objects, to a generic List that is typed to be compatible with the A class called **oblist**. Now you write a **foreach** loop that iterates through all the objects in the list, **oblist**. This loop has to know which base type of object it is dealing with (here that's class A). Since all descendant objects are compatible with class A, it is possible to call `method1()` for each object encountered, like this:

```
foreach( A aOb in oblist ) {
{
      aOb.method1( )
      aOb.method2( )
}
```

This time consider what will happen if the **foreach** loop processes the two objects: the A object **mya1** and the B object **myb1**. The **foreach** loop has been told it is dealing with instances of the A class. When **myb1** is processed within the loop, will **b.method1()** or **a.method1()** be called?

In fact, **a.method1()** will be called. So all the objects processed within the loop will return the string **"class A: method1"**, even though some of the objects may actually be B objects with their own unique **method1()** implementations.

How can we get around this problem? The answer is to make **a.method1()** a '*virtual*' method and to make all the **method1()** implementations in descendent objects '*overridden*' methods. You do this by adding the keywords **virtual** and **override** before the method type and name, like this:

```
public virtual String method2( ) {        // class A
public override String method2( ) {       // class B
```

If you call the virtual **method2()** inside the **foreach** loop, C# works out the *exact type* of the object being processed before calling the specified method. So when the B object, **myb1**, goes through the loop, the **A.method1()** *non-virtual* method is executed, but the **B.method2()** *virtual overridden* method is executed.

MethodTest.sln

If virtual methods are new to you, you may find this much easier to understand by running the project in the *MethodTest.sln* solution. Re-read the explanation given above and see how each step has been coded. Note that I have used the **new** keyword in **b.method1()**. This is recommended for clarity when an ancestor class defines a non-virtual method of the same name and argument list. When preceded by the keyword `new` a method is specifically declared to be a *replacement* for a method with the same name in its ancestor class. In fact, if you omit the `new` keyword in a non-virtual method, the method will operate in the same way as if the `new` keyword had been used. However, in that case, Visual Studio will show a warning.

STRING.FORMAT

The String class includes a method called `Format()` that lets you insert values at marked points in a string. This avoid the necessity of concatenating values and variables using the `+` operator. Instead you place index number (from 0 upwards) between a pair of curly brackets into a string, and follow the string with a comma-delimited list of items whose values will replace the placeholders.

For example, assuming you have these variables declared:

```
string toy = "Cuddly Toy";
double price = 20.75;
```

Using concatenation:

```
textBox1.Text = "Item " + 1 + " is a " + toy + " worth $" + price ;
```

Using `String.Format()`:

```
textBox1.Text = String.Format("Item {0} is a {1} worth ${2}", 1, toy, price);
```

In both cases, the output will be:

"Item 1 is a Cuddly Toy worth $20.75"

An Adventure Game

To end this course, I've written a small adventure game. This takes the form of a Map which is a generic List containing Room objects. Each room may itself contain a list of Thing Objects implemented as a ThingList. You'll find my definition of the ThingList class in *ListManagers.cs*. It is a generic List of Things. The player can wander around the game from room to room taking and dropping objects. All that remains is for you to add a few puzzles.

AdventureGame.sln

The game can be loaded in the *AdventureGame.sln* solution. This contains two projects – one uses Windows Forms for the user interface. The other uses WPF. Both projects use the same source code files containing the classes required for the game. This provides an example of how to use your class libraries in different projects.

adventure-game.csproj

The *adventure-game* project is the Windows Forms version of the game. I am not going to explain this game in detail. It illustrates features we've already discussed earlier in this courser including class hierarchies with overridden methods (see `describe()`), Enums (see *Dir*), generic Lists, properties, *switch..case* tests, *foreach* loops and more.

WPF Adventure

WPFAdventure.csproj

To try out the WPF version of this game, set *WPFAdventure* as the startup project. Most of the code in this version is identical to that in the Windows Forms version. However, its user interface uses the Windows Presentation Foundation. You can explore this in the *MainWindows.xaml* file. WPF design is not described in detail in this course. However, there are numerous WPF tutorials online (just Google *"C# WPF"*).

BinaryFormatter

Before leaving this game, I'd like to say a few words about how the 'game state' (the position of the player and treasure objects, for example) is saved and loaded from disk.

As in previous projects, I 'serialize' (deconstruct) objects so that they can be stored in streams. In this case I have decided to use the BinaryFormatter class. This class serializes and deserializes an object, or an entire graph of connected objects, in binary format. That means that I can give it a 'top level' object – one that contains many other objects, each of which may themselves contain objects – and let it work out all the details of saving and restoring them to and from disk. Note too that I have had to mark all the classes I want to save with the `[Serializable]` attribute. An attribute is a special directive placed between square brackets. The BinaryFormatter requires this attribute to be placed before the definition of any class to be serialized.

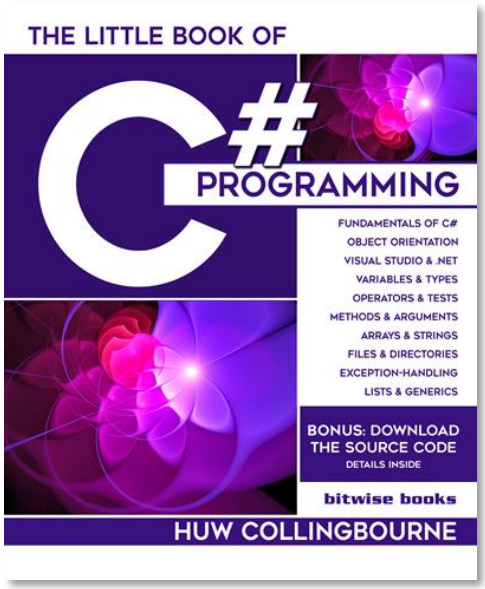
Further Adventures In Coding

This is currently a very simple game and you can try to improve it. See my *TODO* comments for ideas. You may also extend it by creating new rooms and treasures and adding some puzzles – limited only by your imagination.

Have fun!

THE LITTLE BOOK OF C# – PAPERBACK/KINDLE EDITION

If you want a paperback (or Kindle) guide to C# which perfectly complements this course, did you know that a new edition of **The Little Book Of C#** is available from Amazon (US), Amazon (UK) and worldwide (ISBN: 978-1-913132-06-4)? This is a substantially revised, expanded and reformatted 152-page book based on the 92-page eBook provided with this course.



The Little Book Of C#

- Fundamentals of C#
- Object Orientation
- Static Classes and Methods
- Variables, Types, Constants
- Operators & Tests
- Methods & Arguments
- Arrays & Strings
- Loops & Conditions
- Files & Directories
- structs & enums
- Overloaded and overridden methods
- Exception-handling
- Lists & Generics

Bonus: Download the source code

More programming books written by Huw Collingbourne (the author of this course) are available from Bitwise Books. To keep up to date with available titles, go to the Bitwise Books web site: http://www.bitwisebooks.com