C# programming in ten easy steps

by Huw Collingbourne

INTRODUCTION

This is the course book to accompany *"C# Programming (in ten easy steps)"*, a programming course available on Udemy. 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.

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

ASampleProgram.sln

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

Step One

Getting Ready

C# (pronounced *C-Sharp*) is Microsoft's most popular language for .NET development. .NET is a 'framework' comprising 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 older C language. This syntax has much in common with other 'C-like' languages including Java, C++, Objective-C, PHP, ActionScript 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 a free edition – *Visual C# Express* – which can be downloaded by following the links from:

http://www.microsoft.com/express

There are many 'Express' products available. Be sure that you download the latest version of 'Visual C# Express'. Once you have a version of Visual Studio installed you are all ready to get started.

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.

Each Visual Studio project is included in a 'Solution'. A Solution may optionally contain more than one project but in this course each Solution will contain just 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. C# is a 'case-sensitive' programming language so 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*.

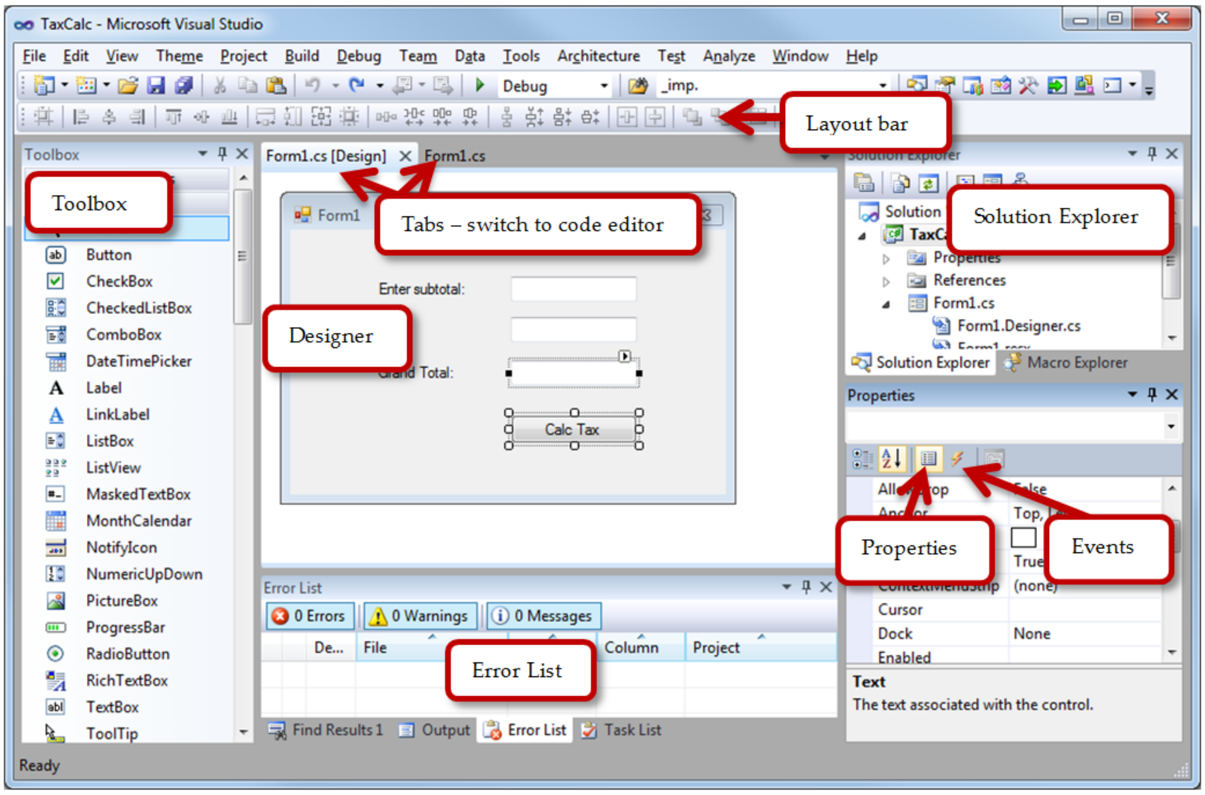
"Hello World" should now appear as the text inside *textBox1*.

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 screen-shot below which identifies some important elements of Visual Studio.



Step Two

Language Elements

DataTypes.sln

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 number 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:

```
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;
```

If you want to make sure that a value cannot be changed, you should declare a constant using the keyword **const**, like this:

```
const double taxrate = 0.2;
```

You can change the value of a variable. So this is OK:

```
mydouble = 500.75;
```

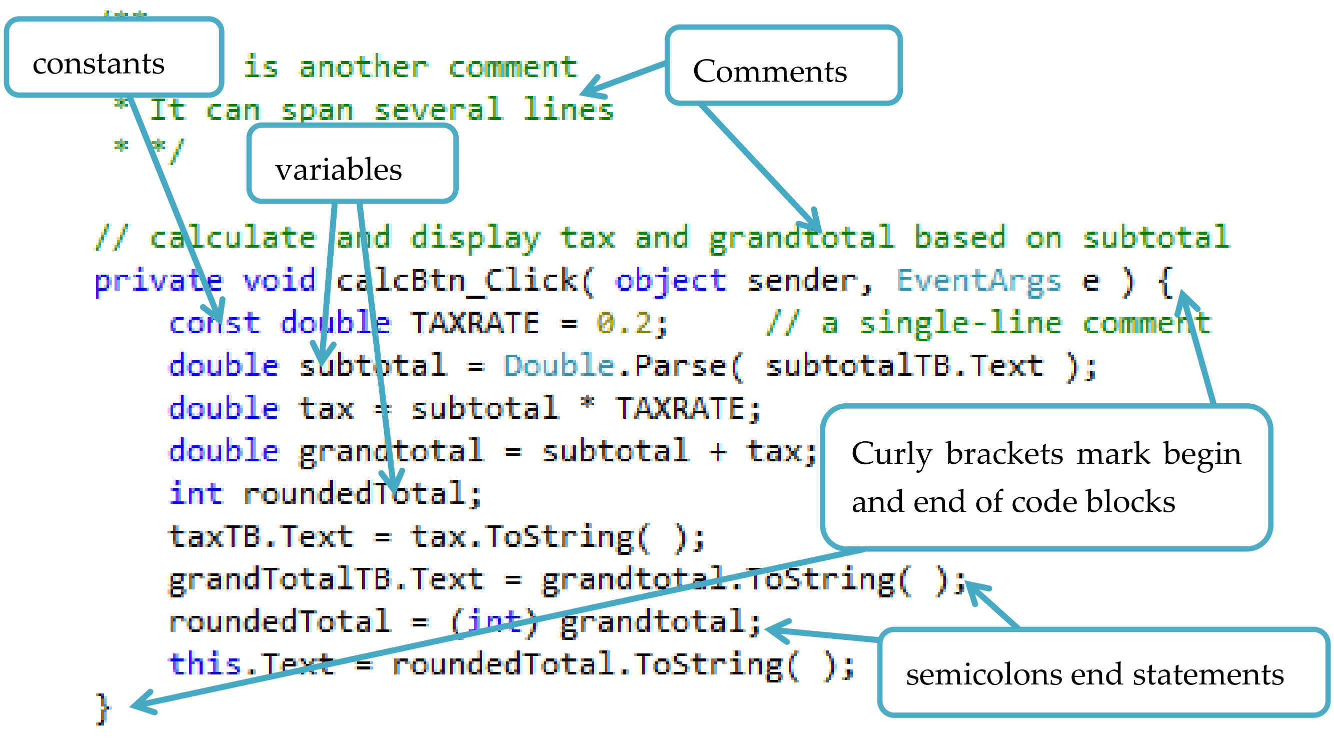
You cannot change the value of a *const* so this is *not* allowed:

```
taxrate = 0.4;
```

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"`)

SUMMARY OF COMMON C# LANGUAGE ELEMENTS


constants
Comments
variables
Curly brackets mark begin and end of code blocks
semicolons end statements

```
/**
          is another comment
 * It can span several lines
 * */

// calculate and display tax and grandtotal based on subtotal
private void calcBtn_Click( object sender, EventArgs e ) {
    const double TAXRATE = 0.2;     // a single-line comment
    double subtotal = Double.Parse( subtotalTB.Text );
    double tax = subtotal * TAXRATE;
    double grandtotal = subtotal + tax;
    int roundedTotal;
    taxTB.Text = tax.ToString( );
    grandTotalTB.Text = grandtotal.ToString( );
    roundedTotal = (int) grandtotal;
    this.Text = roundedTotal.ToString( );
}
```

Namespaces and Usings

At the top of a code file such as *Form1.cs* in the *DataTypes.sln* solution you will see a number of lines beginning with the keyword **using**. The **using** directive gives C# code access to classes and types inside another code module or 'namespace'. Here, for example, we are accessing the System namespace:

```
using System;
```

When a namespace is used in thus way, without further qualification, any classes inside it must be specifically qualified in the code itself. This is because *TextBox* is a class that is defined inside the *Forms* namespace (don't worry about exactly what a 'class' is – we'll come to that later). The *Forms* namespace is defined inside the *Windows* namespace. And the *Windows* namespace is defined inside the *System* namespace.

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.

You can specify the namespaces you wish to use in the **`using`** section at the top of your code like this:

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

When you design a user interface using Visual Studio's form designer, the necessary **using** statements are added automatically.

Type Conversion (Number/String)

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.

TaxCalc.sln

Example. This converts the string **"15.8"** to the Double **15.8** and assigns the result to the variable, **d**:

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

This converts the string **"15"** to the Int16 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**:

```
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 );
```

This is the finished version of the function that computes and displays the values when the button is clicked:

```
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;
	subtotalTB.Text = subtotal.ToString( );
	taxTB.Text = tax.ToString( );
	grandTotalTB.Text = grandtotal.ToString( );
}
```

STEP THREE

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. 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) textBox2.Text = "yes";`

Tests and Comparisons

C# can perform tests using the **if** statement. The test itself must be contained within round brackets and should be capable of evaluating to true or false. If true, the statement following the test executes. A single-expression statement is terminated by a semicolon. A multi-line statement may be enclosed within curly brackets. Optionally, an **else** clause may follow the **if** clause and this will execute if the test evaluates to false. 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.sln

```
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!";
}
```

Note that test conditions must be placed between parentheses. You may use other operators to perform other tests. For example, 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:

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

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
```

If you need to perform any tests, it is often quicker to write them as 'case statements' instead of multiple **if..else if** tests. In C# a case statement begins with the keyword **switch** followed by a test value. Then you put a value after one or more **case** tests. The **case** value is compared with the test value and, if a match is made, the code following the **case** value and a colon (e.g. **case "hi":**) executes. 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:

```
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;
}
```

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.

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.

Operators.sln

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 will probably already have a preference. Many C/C++ programmers prefer the terser 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
```

Step Four

Functions or Methods

Functions provide ways of dividing your code into named ‘chunks’. 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 method returns some value to the code that called it, that return value is indicated by preceding it with the **return** keyword.

Here are some example functions:

Methods.sln

1) A function that takes no arguments and returns nothing:

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

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

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

3) 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;
}
```

Note: In other 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.

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 );
```

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, secomdnumber = 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#.

STEP FIVE

OBJECT ORIENTATION

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).

Each actual 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 – **getStr()** and **setStr()**. 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.sln

```
class MyClass {
        private string _s;

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

        public string getStr( ) {
            return _s;
        }

        public void setStr( 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.sln* solution, 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( );
```

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

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

And you can retrieve the value of the **_s** variable using the **getStr()** method:

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

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.

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:

GameClasses.sln

```
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:

```
public class Thing {
        private string _name;

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

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

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

    public class Treasure : Thing {
        private double _value;

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

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

    }
```

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.

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.sln** project and find the **button1_Click()** method. The first line here creates and initialises an array of three strings:

Arrays.sln

```
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[3]**. This part (the number of items) is optional. If you look at the declaration of **myArray3**, you will see that the number of strings has been omitted:

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

This is 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 **new string[]** when declaring and initializing

myArray3. 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 an 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;
myArray2 = new string[3] { "four", "five", "six" };
```

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 **Arrays.sln** 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 round brackets after the word **for** is used to control the execution of the loop. This code is divided into three parts. The first part initialises an **int** variable, **i**, with the value of 1. The second section contains a test (**i < 100**). 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. 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 **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. 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.sln

The *WordCount* project uses several while loops. These may include a single condition as here:

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

Or multiple conditions, as here:

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

STRINGS AND CHARS

Strings.sln

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.

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.sln* solution.

Dialogs and Message Boxes

This step's projects use several different types of dialog box. You may have noticed that the *WordCount.sln* 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.sln** 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 IO

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#.

In the last step we created a simple editor from a RichTextBox control (*Editor.sln*). This 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?

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.

Steams.sln

Now load up the **Streams.sln** solution. 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**.

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 *Streams.sln*. 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:

```
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 **CreateText()** and **AppendText()** and **Exist()** methods.

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.sln

Load the *Dir.sln* solution 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.sln* solution 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 *Anchor-Styles* 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

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

GENERIC COLLECTIONS

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. 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.

adventure-game.sln

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.

BinaryFormatter

Before leaving tis 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!