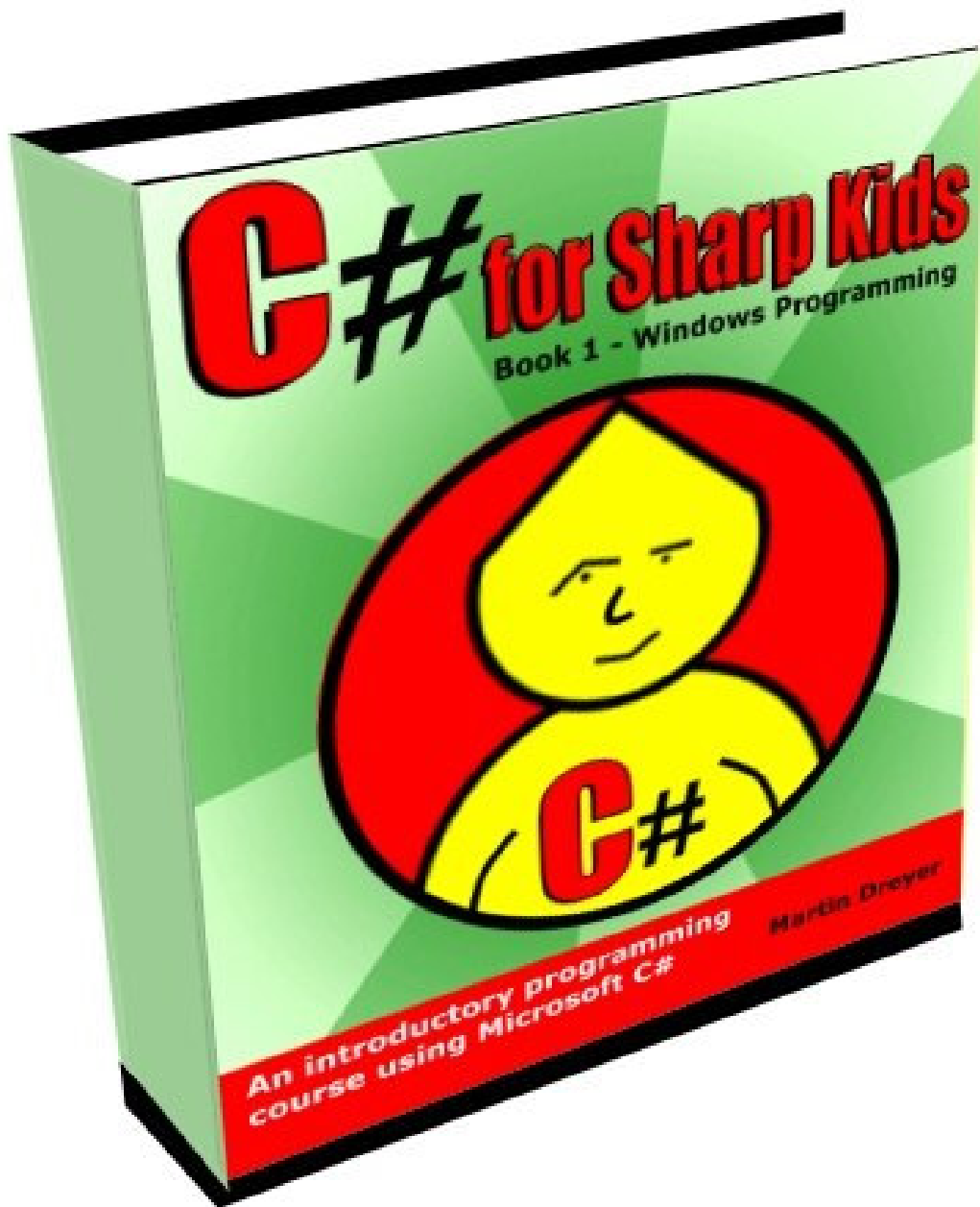
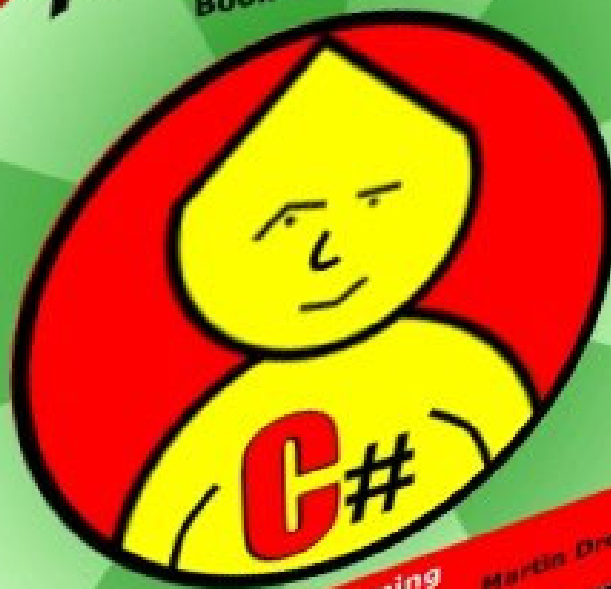




# C# for Sharp Kids

Book 1 - Windows Programming



An introductory programming  
course using Microsoft C#

Martin Dreyer

## Table of Contents

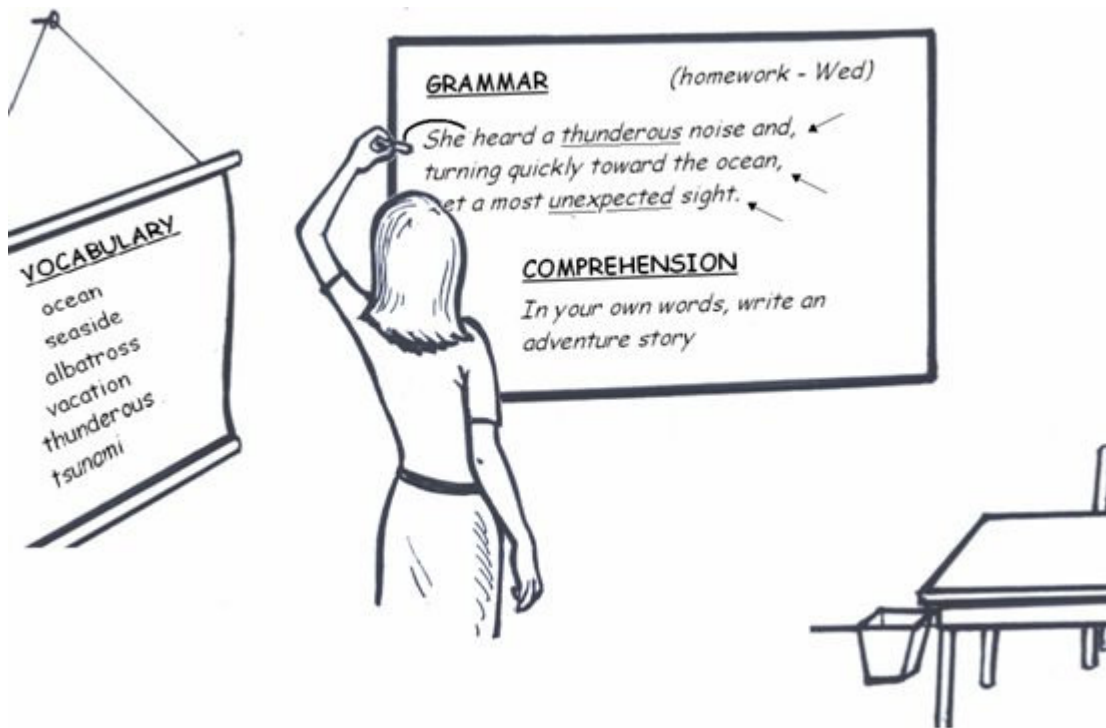
|                                                  |    |
|--------------------------------------------------|----|
| Part III – The C# Language .....                 | 3  |
| Vocabulary.....                                  | 4  |
| Some General Grammar Rules .....                 | 5  |
| Capital and Small Letters.....                   | 5  |
| Statements.....                                  | 6  |
| Code Blocks - Grouping Statements Together ..... | 7  |
| Comments - Helping the Humans Out.....           | 8  |
| Indenting Your Code .....                        | 10 |
| Variables .....                                  | 11 |
| Operators.....                                   | 13 |
| Converting Between Types.....                    | 17 |
| Branching.....                                   | 19 |
| Looping.....                                     | 21 |
| The For Loop .....                               | 21 |
| The While Loop.....                              | 22 |
| Whole Program Structure .....                    | 25 |
| Using Class Libraries.....                       | 30 |

## Part III – The C# Language

---

When one learns to read and write a new natural language, one would usually cover at least:

- **Vocabulary** - the *words* of the language
- **Grammar** - how to *put the words together* correctly, in the accepted way. This includes things like proper sentence structure and punctuation.
- **Comprehension** - understanding the *meaning* behind what is written down. To *comprehend* means to *understand*.



The same is roughly true of learning to read and write a computer language. There are some basic words and other bits which are the pieces to be used in your programs. You put these together in specific ways to make something similar to sentences, with punctuation marks. All of this is for the purpose of conveying some meaning to the computer.

We won't break up our lessons into precisely the same headings above, but you will see those aspects come through in the following chapters.

## Vocabulary

---

Just as in natural languages, C# has a certain number of words that are directly a part of the language, but there are additional words that can be legally included from outside the language.

In English you may write "*There is a Gorilla named Juju that lives in the Ugandan jungle.*" Most of these words are English words - you'll find them in a dictionary of the English language. But the words "Juju" and "Ugandan" are not - they are proper nouns. Does that mean the sentence is not correct? No - it is quite legal to use words from:

- the list of all names of humans and animals
- the official list of country names
- ... and lots of other word lists

Similarly, we will mention words that are directly part of the C# language. But that doesn't mean these are the only words that you'll use in your programs. Depending upon what it is that you're talking about, you will most likely use many extra words that are borrowed from other lists, libraries, etc. In particular, Microsoft has written a vast library of code called the .NET Framework Class Library, which you will definitely be using in your C# programs. It has many class names, field names and method names that are not internally part of C#, but you can certainly use these names in your C# programs.

You yourself will also define new classes, fields and methods and give them your own names - these then become words that you can legally use in C# programs. This is like you making up a new name, "SpangleFoot" for your pet - you're then legally allowed to use that name in English sentences.

Here are some of the built-in words of the language - some you need to know in order to write basic programs. These are all mentioned in this book.

- class
- do
- else
- false
- for
- if
- int
- new
- null
- private
- protected
- public
- return
- static
- string
- this

- true
- using
- void
- while

This is not a complete list, but don't worry - there are not many C# words beyond this list. Computer languages tend to have far less words than natural languages.

What do you do with the words of a language? You put them together in well-structured sentences, according to certain rules. That's called grammar. That's what the rest of the chapters in this section are about.

## Some General Grammar Rules

---

In this chapter, we cover some general information about grammar in C#. In technical terms, this is often referred to as syntax.

### Capital and Small Letters

The C# language is case-sensitive. In other words you must use small letters (also called lower case letters ) and capital letters (upper case letters) in the right places.

#### Built-in Words

The keywords listed in the previous chapter always have to be written exactly as they are shown there. You may have noticed that they're all in lower case.

So if you ever write the word "If" or "IF" instead of "if", the computer will simply refuse to carry on - it does not understand things unless you write them according to the rules.

*Don't be too hard on it – it has some strengths where you have weaknesses, like being able to remember absolutely everything you've ever told it over the last 10 years and doing so in a fraction of a second! Can you do that? Mmm, not so chirpy now are you? Anyway, we're digressing, let's get on with it.*

#### Names That You Make Up Yourself

Now what about words that you've made up yourself? Suppose for example, you have created a class called "myClass" which has a field called "numberOfSpots".

```
class myClass
{
    int numberOfSpots;
}
```

Do small versus capital letters matter? The answer is quite obvious really, and that is Yes. Whenever you talk about these names in a C# program you must write them exactly as you originally defined them. The following, for example, will fail.

```
class myClass
{
    // Field

    int numberOfSpots;

    // Method

    public void Main()
    {
        MyClass x = new MyClass();
        x.NUMBEROFSPOTS = 365;
    }
}
```

But when written as follows, the code will run correctly:

```
class myClass
{
    // Field

    int numberOfSpots;

    // Method

    public void Main()
    {
        myClass x = new myClass();
        x.numberOfSpots= 365;
    }
}
```

So although you may choose to write a class name/field name/method name/etc. with a mixture of small and big letters (NUMBEROFSPOTS or numberofspots or NuMbErOfSpOtS, etc), it is very important that you then continue to write that name the *very same way throughout your program*.

## Statements

In English, we speak of sentences, statements, phrases, etc. All of these are different ways of putting words together. Each of these is ended with a particular punctuation mark. A sentence is ended with a period (also called a full stop). A phrase may be ended with a semi-colon (;).

In C#, we keep it a bit simpler. We speak mainly of statements, and statements are generally ended with a semi-colon.



Here, for example, is a piece of code with 4 statements:

```
string x;  
x = "Nobody wins in war";  
myClass m = new myClass();  
m.numberOfSpots = 3;
```

You may say that's obvious - there are 4 lines. But the computer doesn't really see things in lines like we do. It will be just as happy if we write the four statements in one long line.

```
string x; x = "Nobody wins in war"; myClass m = new myClass(); m.numberOfSpots = 3;
```

And that's where the semi-colons become important. The computer knows we've said 4 different things in this line because we ended each thing with a semi-colon. By the way, please don't write many statements in one long line like that - humans are used to separate lines - and you should try to keep your code as human-readable as possible.

I'm willing to bet that many errors that you encounter in your programs will be because you forgot to end a statement with a semi-colon; but after forgetting about 5000 times, you'll probably start to get into the habit. Luckily, Visual C# Express is quite smart with realizing you've left out things like that and will tell with a squiggly red line or some other gadget to show you how stupid you're being.

## Code Blocks - Grouping Statements Together

In English, we use paragraphs to group together statements that are closely related.

In C#, we use the curly braces "{" and "}" in a similar way to paragraphs. These braces are used to start and end a block that groups a number of statements together.

Let's look at an example. Suppose we have a program that asks the user questions and checks their answers. Suppose we want to do 2 things only if somebody answers correctly. We may write code as follows :

```
if ( userAnswer == correctAnswer )
{
    Console.WriteLine( "That is correct" );
    score = score + 1;
}

AskNextQuestion();
```

The two statements inside the curly braces will both be run if the user's answer is the correct answer - they have been grouped together into a block.

You have already seen many other examples with curly braces in this book. Every time a class is defined, you put all it's code inside curly braces :

```
class MyClass
{
    int numberOfSpots;
}
```

And, when you write a method in your class, you group it's statements together with curly braces too :

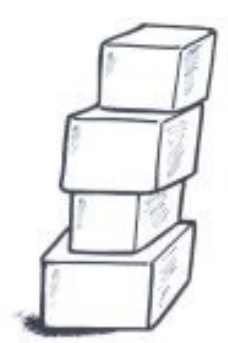
```
class MyClass
{
    int numberOfSpots;

    public void Main()
    {
        MyClass x = new MyClass();
        x.numberOfSpots = 365;
    }
}
```

## Comments - Helping the Humans Out

You will have noticed that many books have notes or comments written at the sides or at the bottom of the pages. These comments are not part of the core text but are usually *extras* that explain something further about the core text.

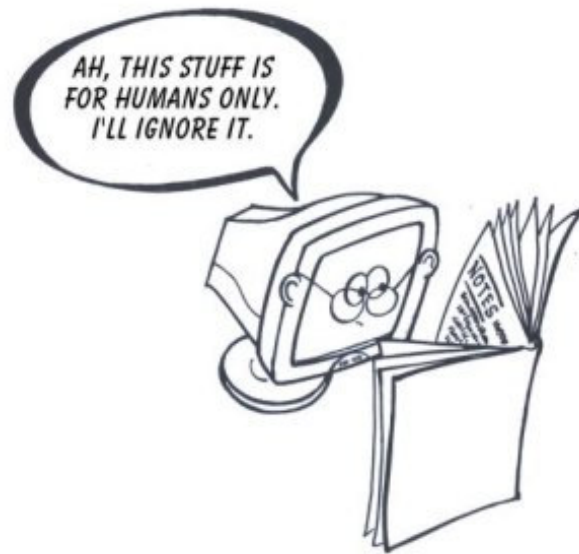
You may wonder whether you can do the same in your computer programs, and you can. You will often want to write some comments for either yourself or other programmers, to explain what's going on in your code. In fact, this is considered good behaviour by programmers. It's true that a programmer should be capable of



figuring out what's going on in a program, but why waste people's time when a short English comment will quickly explain the basics about what your code is doing.

And if you're thinking "I'm the only one who will read my code", trust me, when you come back to your program in a few month's time you'll wish that you'd commented your own code too.

Since these comments are for people and not the computer, you need some way to tell the computer to ignore whatever you write in the comments sections. You need to "hide" these comments from digital eyes.



There are two ways to do this:

- For comments that you write on a single line, you can simply put a double slash `//` at the beginning of the line.
- Comments that span more than one line can be hidden from the computer by starting with `/*` and ending with `*/`.

The following piece of code shows both kinds of comments, used in different ways:

```
void DrawCircleInSquare( int left, int top, int size)
/*
    This is a method for drawing a red circle.
    Pass the size of the circle in the 'size' parameter.
    Pass the position to draw at in the 'left' and 'top' parameters.
*/
{
    Graphics g = this.CreateGraphics();    // Prepare drawing surface
    Pen redPen = new Pen(Color.Red, 3);    // Use a red pen

    // Draw a square (a rectangle with equal width & height)
    g.DrawRectangle( redPen, left, top, size, size );

    // Draw a circle (an ellipse with equal width & height)
```

```

        g.DrawEllipse( redPen, left, top, size, size );

        // Clean up
        g.Dispose();
    }

```

When the computer encounters `//` it ignores anything on the rest of that line. But when it sees `/*` it ignores everything until the next `*/`.

## Indenting Your Code

By now you've probably noticed that we tend to *indent* code like this:

```

if ( userAnswer == correctAnswer )
{
    indent ➡ Console.WriteLine( "That is correct" );
    score = score + 1;
    if ( score > 10 )
    {
        indent ➡ Console.WriteLine( "You have qualified!" );
    }
    else
    {
        indent ➡ Console.WriteLine( "Bugger. Keep going ..." );
    }
}

```

rather than writing it all mashed together like this:

```

if ( userAnswer == correctAnswer )
{
    Console.WriteLine( "That is correct" );
    score = score + 1;
    if ( score > 10 )
    {
        Console.WriteLine( "You have qualified!" );
    }
    else
    {
        Console.WriteLine( "Bugger. Keep going ..." );
    }
}

```

This is in order to make it clear where a particular block of code begins and ends. As your programs get longer, it gets harder and harder to see the beginning and end of a block. Indenting makes it a lot easier to read the code with meaning.

Failing to indent does not break the code - indenting is not a grammatical requirement - but it is considered "good practice" and you should get into the habit.

To indent code when writing within C# Express, you can select one or more lines of code and click the TAB key on your keyboard. If you need to do the opposite, hold the SHIFT key *and* press TAB. Alternatively, you can click the  icons on the toolbar.

## Variables

---

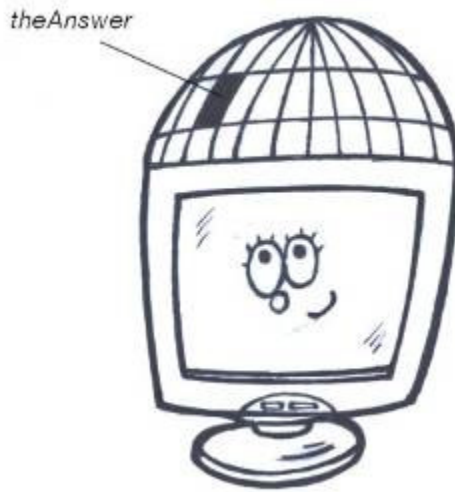
The word "variable" means "changeable" - a variable is a thing whose value can change.

Here's a typical way we might use a variable. We choose to call the variable "*theAnswer*" in this case, because we want it to hold the answer of a sum, although we could have chosen any other name.

```
int addPositiveNumbers( int x, int y )
{
    int theAnswer;
    if ( (x>0) && (y>0) )           // If x and y are positive numbers
    {
        theAnswer = x + y;         // Add them & put the sum into answer
    }
    else                             // Otherwise
    {
        theAnswer = 0;             // Put zero into answer
    }
    return theAnswer ;
}
```

In this example, the variable "*theAnswer*" is used to hold the sum of whatever is passed in the parameters x and y. At the end, we then send back whatever value is being held in *theAnswer*.

It's interesting to note that "theAnswer" is really just a place in the computer's memory. When the computer sees the line "int theAnswer", it finds a free spot in it's memory and names it "theAnswer". When it reaches the line "return theAnswer", it goes and fetches whatever number it had stored in the place called "theAnswer". In the example above, since it's now finished all it was asked to do, it would then free up that memory space for other purposes.



### It's Not That Strange Really

When we humans work with things in our brains, we also use places in our memory to hold answers; but we do it almost without realizing it. Suppose you're asked to

- add the numbers 10, 20 and 50
- then multiply that answer by 2

One likely way you'd approach this is

- Work out  $10 + 20$ . Hold this answer (30) in your head.
- Work out  $30 + 50$ . Now hold this answer (80) in your head.
- Work out  $80 \times 2$ . Hold that in your head until you're ready to give it to the person who asked you to work this out.
- Once you've done what you needed to do, stop trying so hard to remember it - that answer is of no value anymore - rather free up your memory to be used for other things.

So we're not always that different from our digital friends.

### Variables Are Always of a Particular Type

A variable is really a lot like a field or a parameter except that it does not have anything directly to do with a particular class or object - we just need a temporary place to store something so we can work with it.

But variables, just like fields (which are actually a kind of variable), are always of a particular type. E.g.

```
string myString;  
int theAnswer;  
bool isFinished;
```

(If you haven't read the chapter "Fields and Their Types", consider going and reading it so you understand more about "types")

## Operators

---

Suppose you write the following in one of your programs:

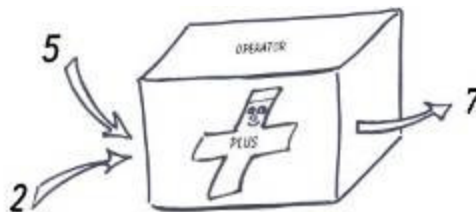
```
int theAnswer = x + y;
```

and in another place you write

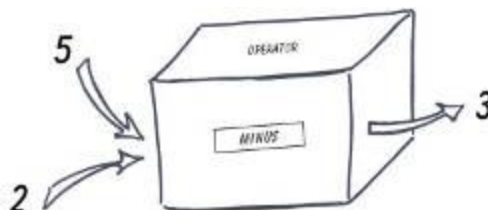
```
int theAnswer = x - y;
```

In the first case you're using the symbol for *adding* and in the second case *subtracting*. These actions you're performing on numbers are called *operations* and their symbols + and - are called *operators* - they *operate* on things. An operator takes one or more things in, operates on them and pushes some resulting value out.

This picture illustrates the **plus operator**, working on 2 numbers. In this example it takes *in* the values 5 and 2 in and puts *out* the value 7



And here is the **minus operator**, taking the same values in but putting a different value out:



All computer languages have at least a few operators and it is very important to be familiar with them. So let's cover the one's you're likely to see often in C#.

| Symbol | Operation             | Discussion                                                                                                                                                                                                                                                                                                                                                                      |
|--------|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| +      | add 2 things          | <ul style="list-style-type: none"> <li>When using number types, this is just as in math. E.g. <math>25 + 5</math> gives the number 30.</li> <li>When using strings, it sticks the strings together.</li> </ul> <p>E.g. "how" + " are" + " you" produces the longer string "how are you".</p>                                                                                    |
| -      | subtract two things   | <ul style="list-style-type: none"> <li>Just as in mathematics. E.g. <math>25 - 5</math> gives the number 20</li> </ul>                                                                                                                                                                                                                                                          |
| /      | divide                | <ul style="list-style-type: none"> <li>Since keyboards don't have a <math>\div</math> key, the slash is used to mean "divide". E.g. <math>20 / 5</math> gives the number 4.</li> </ul>                                                                                                                                                                                          |
| *      | multiply              | <ul style="list-style-type: none"> <li>Multiplication works just as in mathematics. E.g. <math>25 * 5</math> gives the number 125.</li> </ul>                                                                                                                                                                                                                                   |
| >      | Greater than          | <ul style="list-style-type: none"> <li>Example 1: Writing <math>5 &gt; 2</math> will give the answer "true"</li> <li>Example 2: Writing <math>5 &gt; 17</math> will give the answer "false".</li> </ul>                                                                                                                                                                         |
| <      | Less than             | <ul style="list-style-type: none"> <li>Example 1: Writing <math>2 &lt; 3</math> will give the answer "true"</li> <li><math>12 &lt; 11</math> will give the answer "false"</li> </ul>                                                                                                                                                                                            |
| =      | Set equal to          | <ul style="list-style-type: none"> <li>This is used to set a variable equal to some value.</li> <li>Example 1: <code>int x = 512;</code></li> <li>Example 2: <code>string y = "Hello how are you";</code></li> </ul>                                                                                                                                                            |
| ==     | Test if equal to      | <ul style="list-style-type: none"> <li>This is used to check whether two things are equal.</li> <li>Example 1: <math>13 == 70</math> will obviously give the answer "false"</li> <li>Example 2: <math>x == 15</math> will give the answer "true" if x has the value 15</li> <li>Example 3: <code>"HELLO" == "hello".ToUpperCase()</code> will give the answer "true"</li> </ul> |
| <=     | Less than or equal to | <ul style="list-style-type: none"> <li>Example 1: Writing <math>2 \leq 3</math> will give the answer "true"</li> <li>Example 2: Writing <math>2 \leq 2</math> will give the answer "true"</li> </ul>                                                                                                                                                                            |

|                         |                          |                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-------------------------|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                         |                          | <ul style="list-style-type: none"> <li>Example 3: Writing <code>2 &lt;= 1</code> will give the answer “false”</li> </ul>                                                                                                                                                                                                                                                                                                                   |
| <code>&gt;=</code>      | Greater than or equal to | <ul style="list-style-type: none"> <li>Example 1: Writing <code>23 &gt;= 23</code> will give the answer “true”</li> <li>Example 2: Writing <code>23 &gt;= 2</code> will give the answer “true”</li> <li>Example 3: Writing <code>23 &gt;= 33</code> will give the answer “false”</li> </ul>                                                                                                                                                |
| <code>!</code>          | Not                      | <ul style="list-style-type: none"> <li>Example: <code>!( 5 &gt; 17 )</code> means “is it not the case that <code>5 &gt; 17</code> ?<br/>The answer will be “true” since it is not true that 5 is greater than 17.</li> </ul>                                                                                                                                                                                                               |
| <code>!=</code>         | Not equal to             | <ul style="list-style-type: none"> <li>Makes it easy to check whether something is “not equal to”.</li> <li>Example: <code>5 != 12</code> will give the answer “true” since 5 is not equal to 12.</li> </ul>                                                                                                                                                                                                                               |
| <code>&amp;&amp;</code> | And                      | <ul style="list-style-type: none"> <li>This allows you to test whether more than one thing is true. The following example checks whether both the password <i>and</i> username are correct :</li> </ul> <pre>if ( ( pwd == 'Pass') &amp;&amp; (user == 'jim') ) return true;</pre> <p>You could just as easily test more than two conditions – there is no limit to how many times you can use <code>&amp;&amp;</code> in a statement.</p> |
| <code>  </code>         | Or                       | <ul style="list-style-type: none"> <li>The symbol for “or” allows you to check whether <i>any one</i> of a number of things is true. The following example checks whether favourite color is green OR the age is greater than 12.</li> </ul> <pre>if ( ( favcolor == 'green')    (age &lt; 12) ) return true;</pre>                                                                                                                        |
| <code>++</code>         | increment                | <ul style="list-style-type: none"> <li>To increment means to increase by 1</li> <li>This example will first write 15 to the screen, followed by 16 and then 17. This is because <code>x</code> starts out with a value of 15 but then gets increased by 1 each time we hit a line that says <code>x++</code>.</li> </ul> <pre>int x = 15; Console.WriteLine( x ); x++; Console.WriteLine( x ); x++;</pre>                                  |

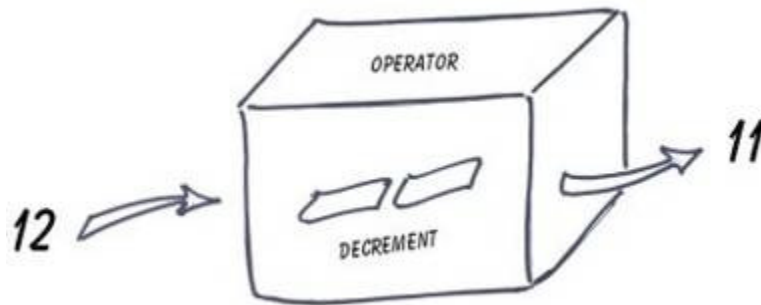
|    |           |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|----|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    |           | <pre>Console.WriteLine( x );</pre> <ul style="list-style-type: none"> <li>Using "++" is the same as using "+ 1". The following two lines have the same general effect as each other : <pre>x++; x = x + 1;</pre> </li> </ul> <p>You can also write ++x (called prefix notation) instead of x++ (called postfix notation) but these two notations can have different effects. Here is an example that reveals how they differ :</p> <pre>int x = 15; Console.WriteLine(x++); // writes x <b>then</b> increments it Console.WriteLine(x);   // writes x Console.WriteLine(++x); // increments x and <b>then</b> writes it Console.WriteLine(x);   // writes x</pre> <p>The results displayed by the above code would be:</p> <pre>15 16 17 17</pre>                                                                             |
| -- | decrement | <ul style="list-style-type: none"> <li>To decrement means to decrease by 1</li> <li>This example will first write 15 to the screen, followed by 14 and then 13. This is because x starts out with a value of 15 but then gets decreased by 1 each time we hit a line that says x--.</li> </ul> <pre>int x = 15; Console.WriteLine( x ); x--; Console.WriteLine( x ); x--; Console.WriteLine( x );</pre> <ul style="list-style-type: none"> <li>Using "--" is the same as using "-1". The following two lines have the same effect as each other :</li> </ul> <pre>x--; x = x - 1;</pre> <p>As with the ++ operator, you can also write --x instead of x--. Here is an example to show how the two notations differ in their effects.</p> <pre>int x = 15; Console.WriteLine(x--); // writes x <b>then</b> decrements it</pre> |

```
Console.WriteLine(x);    // writes x
Console.WriteLine(--x);  // decrements x and then writes it
Console.WriteLine(x);    // writes x
```

The results displayed by the above code would be:

```
15
14
13
13
```

Let's wrap up with one more picture, showing how an operator such as "--" could be illustrated. In this case, the operator takes only one number in and also gives one number out. The number coming out is 1 less than the number that was sent in.



## Converting Between Types

---

Suppose we included the following code in a program:

```
// Declare some variables

int x;
string y;
int z;

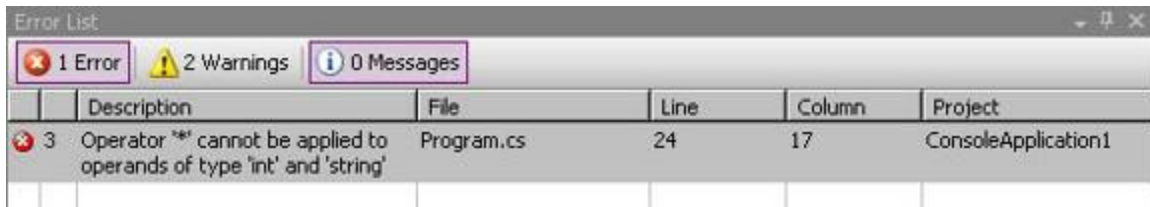
// Put values into x and y

x = 5;
y = "17";

// Calculate the product of x and y and put the answer into z

z = x * y;
```

The last line above would cause an error - the computer complains because you cannot multiply strings - and *y* is a string type. As it stands above, that's as bad as saying "What is 17 multiplied by *funny*?" It simply doesn't compute. Strings are not number types and you can't perform any math on them.



Fortunately, it is possible to convert strings to numbers. The following shows how you could get around the above problem, since the string "5" can be sensibly converted to the number 5. It just takes a small change to the last line.

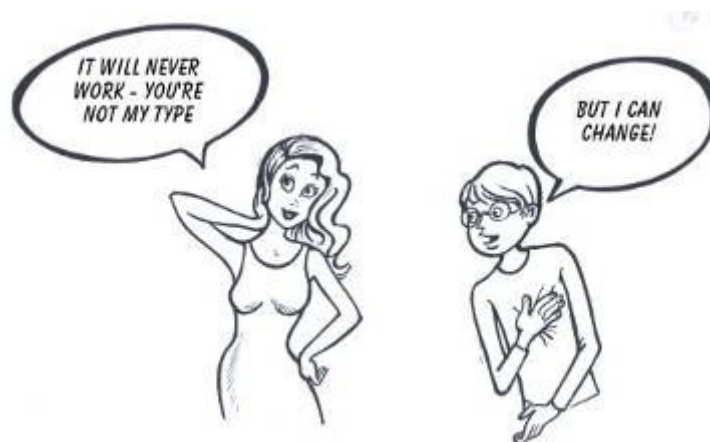
```
z = x * Convert.ToInt32( y );
```

What we've done is to use Microsoft's class called "convert" and call its method "ToInt32", which converts to one of the integer types. Now that *y* has been converted to an integer, the computer will happily multiply the two numbers.

The "convert" class has many other methods but the ones you'll most often need are:

- Convert.ToInt32()
- Convert.ToString()
- Convert.ToBoolean()

It's important to remember how fussy computers are about types.



## Casting

Another way that values can be changed from one type to another is through something called casting. To cast something means to convert it to a different type, but unlike the "convert" examples above, casting only works when the type you're changing from is very similar to the type you're changing to. It is useful, for example, for converting between two *number* types, but is useless for converting between a number and a string. The way you *write* a cast is also very different to the above examples; before the variable that you wish to convert you must write, in parenthesis, the type that you want to convert to.

To show an example, we need to introduce you to another number type - let's use the type called "double". For those of you who have covered scientific notation in school Math or Science, the "double" and "float" types are very similar to that - they allow you to work with decimal numbers, allowing the decimal point to "float". E.g.  $5.0 \times 10^3$  is the same number as  $50.0 \times 10^2$  or  $0.50 \times 10^4$  and so on.

```
int x;
double y;
int theAnswer;

x = 10;
y = 15;
theAnswer = x + (int) y;
```

The code `(int) y` converts y to an integer.

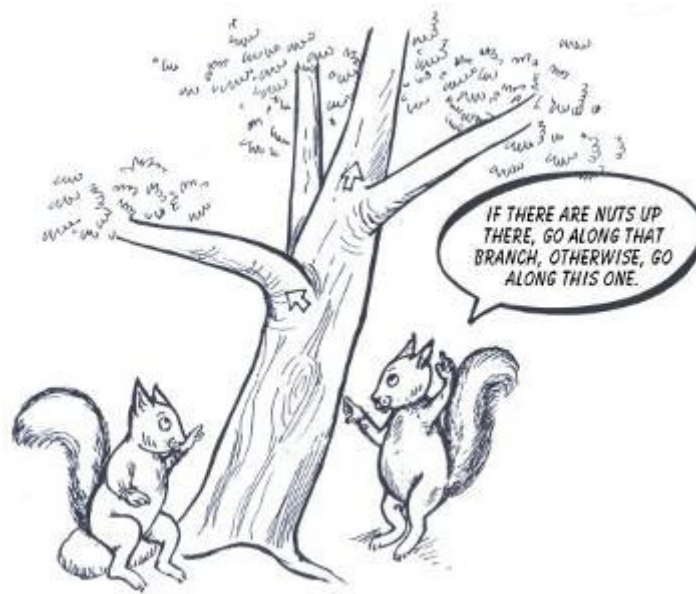
By the way, this particular cast will only work if the number held in y is something that can be properly converted into an integer. So the example above will be fine, but if y had a value of 15.5 instead this code would fail. In a real program you would probably add some more code to first round off y to a whole number using `Math.Round(y)`.

You probably won't need to use casting in your early programs, but you may well see it being used in other people's examples that you read through. Now at least you'll know what's going on.

## Branching

---

Often when you want the program to do something, the line it must run next depends on something. You want the program to "branch off" one way in one case but branch off another way in another case. C# has two branching structures to allow this, the **if** statement and the **switch** statement. We won't discuss the switch statement here since it's more complicated and you can achieve the same result using the **if** statement.



The **if** statement is easy to understand. You check something to see if it's true and if the computer finds it to be true it runs the code in that block. Otherwise it falls down to the next block and tests that one. It keeps falling through until it either finds something true or reaches the end of the entire **if**.

```
if ( UserAnswer == "Bald Eagle" )
{
    Console.WriteLine("Yup, you got it right !");
}
else if ( UserAnswer == "Eagle" )
{
    Console.WriteLine("Almost - the answer is 'Bald Eagle'");
}
```

You will often want to have a final “else” block at the end too – so if none of the things higher up were true, the code in the “else” block will be run.

```
if ( UserAnswer == "Bald Eagle" )
{
    Console.WriteLine("Yup, you got it right !");
}
else if ( UserAnswer == "Eagle" )
{
    Console.WriteLine("Almost - the answer is 'Bald Eagle'");
}
else
{
    Console.WriteLine("Sorry, you got it wrong. Try again.");
}
```

## Looping

---

What if you need to do something over and over again? Take, for example, writing the numbers 1 to 10. It would be foolish to do it this way:

```
Console.WriteLine("Number " + 1 );
Console.WriteLine("Number " + 2 );
Console.WriteLine("Number " + 3 );
Console.WriteLine("Number " + 4 );
Console.WriteLine("Number " + 5 );
Console.WriteLine("Number " + 6 );
Console.WriteLine("Number " + 7 );
Console.WriteLine("Number " + 8 );
Console.WriteLine("Number " + 9 );
Console.WriteLine("Number " + 10 );
```

And it would be REALLY foolish to do it that way if you needed to write numbers from 1 to 15000!

That's where loops come in – we can do things over and over again with very little code. C# has a few kinds of loops, but we'll just cover two of them.

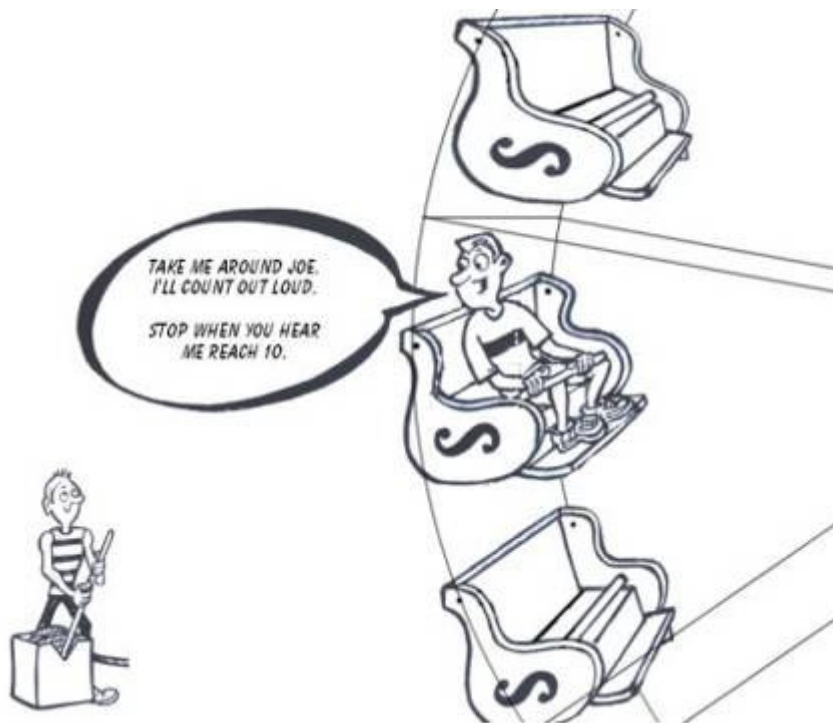
### The For Loop

For loops are usually used in a counting fashion. You can think of a FOR loop like this:

- start with a counter set to some value (E.g. 0)
- keep going around until that counter gets to some specific value (E.g. 11)
- each time we go around, do something specific to that counter (E.g. add 1 to the counter)

The following piece of code - only 3 lines - writes out exactly the same as the 10 lines of code above.

```
for ( int x = 0; x < 11; x = x + 1 )
{
    Console.WriteLine( "Number " + x );
}
```



By the way, it's more common to see a loop like the one above written like this:

```
for ( int x = 0; x < 11; x++ )  
{  
    Console.WriteLine( "Number " + x );  
}
```

The "x++" bit is a lazy (but nice and short) way to write  $x = x + 1$ .

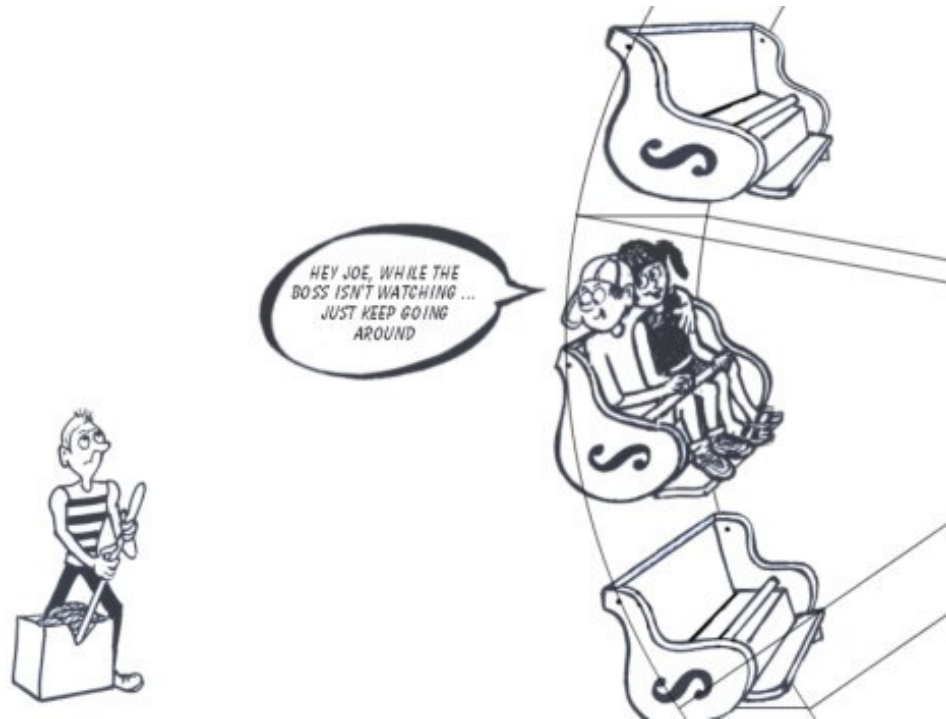
You can also have "for" loops count downwards. This is simply done by having them subtract from the counter instead of add.

```
for ( int x = 11; x > 0; x-- )  
{  
    Console.WriteLine( "Number " + x );  
}
```

This says "starting with x equal to 11, and as long as x is greater than zero, subtract 1 from x".

## The While Loop

Think of a WHILE loop like this: “while something is true, keep going round and doing this block of code again and again”.



"While" loops can be used to achieve exactly the same as the "for" loop example above, but here is an example that has nothing to do with counting. This code keeps asking the user to type the word "smokey" in at the keyboard. Only once they type it correctly does it jump out of the loop and say "Thank You". Notice that the condition we check each time we go around is "`s != smokey`", which means while the word typed in is *not equal* to 'smokey'.

```
string s = "";

while ( s != "smokey" )
{
    Console.WriteLine("Please type the word 'smokey' : ");
    s = Console.ReadLine();
}

Console.WriteLine("Thank You");
```

Here is an example which uses a while loop to do exactly the same as the "for" loop example we showed further above. We simply introduce a variable (x in this case) to use as a counter.

```
int x = 0;
while ( x < 11 )
{
    WriteLine( "Number " + x );
```

```
    x = x + 1;  
}
```

You can see that  $x$  starts out being zero but gets 1 added to it each time we go around the loop. Each time we go around again, the computer checks again "is  $x$  still less than eleven?" Eventually  $x$  gets to be equal to 11 and then the computer jumps out of the loop.

### Never-ending Loops

Hey, want to try something really stupid? Leave out the line  $x = x + 1$ . That way,  $x$  will NEVER get any bigger and so it will ALWAYS be less than 11 and we'll go around and around the loop FOREVER – or at least until the computer runs out of memory, throwing up an ugly error message. Whenever you write an endless loop, you should feel suitably embarrassed.



# Whole Program Structure

---

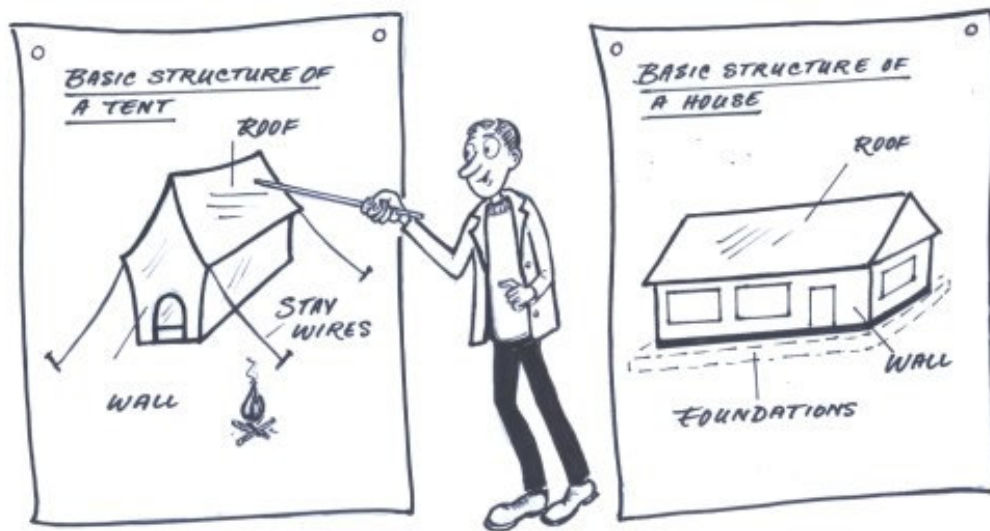
## Introduction

In the previous chapters, we've been looking at specific concepts and how to explain each of those ideas to the computer. But we have not considered what a "whole program" should look like. How do we start the program and what are the parts that have to be there for it to work.

So in this chapter we discuss the basic structure that a C# program must have. The idea is not to confuse ourselves with all the detail - just to understand the basic outline of a program - the skeleton.

## There are Several Different Types of C# Programs

If you went camping in the mountains for the weekend, you would use a structure such as a tent - not a *house* that needs brick foundations. You use a structure that suits the environment you're in. Tents and houses are basically the same, but tents are specifically structured for camping purposes.



In a comparable way, C# and the .NET framework can be used to create applications for different environments. The language is the same in each case, but there are some specific differences in how you structure the program - depending on the environment you're working in. Here are some examples of different types of C# programs.

- Windows Forms applications
- Class libraries
- Windows services
- Console applications
- ASP.NET web applications

This book focuses on only two of these types of programs - Console applications and Windows Forms applications. But the knowledge you gain in these two environments is also directly useful in the other types of applications.

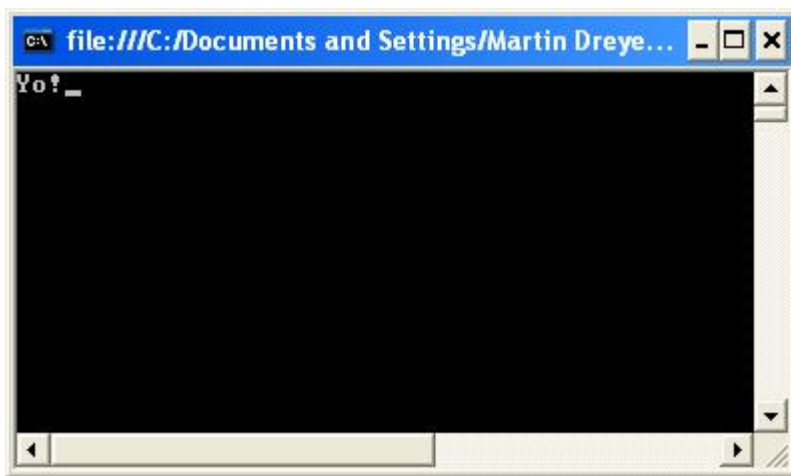
### The Structure of Console Applications

A console application is one which can display and read text only - no pictures, graphs, etc. If you've ever opened a "command prompt" and typed a command like DIR to see a list of files, then you've already used a console application. When you want to try something out quickly, or want to write a program that doesn't need anything other than text, you will often use a console application, because that's the simplest type of program you can write with C#.

Here is an example of a simple, but complete, console application:

```
1. using System;
2.
3. class PleaseSayYo
4. {
5.     static void Main()
6.     {
7.         Console.WriteLine("Yo!");
8.         Console.ReadLine();
9.     }
10. }
```

And this is what you'll see on the screen when you run it. It simply writes a line of text to the screen:



The program's pretty short isn't it? But let's break it down and discuss the structure.

- Line 1 tells the computer that something later on in our program is going to be using some class from the namespace called "System". The particular class whose code we'll be using is "Console". This is a class, written by Microsoft, which has all sorts of things that console applications need. Instead of us writing lots of lines of code to do something simple, like write a word to the screen, we simply call the method "WriteLine" from the System.Console class.

- Between lines 3 and 9 we "define the class". In the class we write our program by writing down properties and methods. This particular class has no properties (it just doesn't need any) but does have one method.
- Between lines 5 and 8, we define a method. This method is very much like the methods we discussed earlier in this book, but it is a special one - it **MUST** be called "Main" - we're not allowed to choose our own name. When the computer is asked to run a C# program, it needs to know where to start. It has been taught that, for C# programs, the programmer will always write a method called "Main" and that's where it must start - that's the entry point. You are allowed to put this method anywhere in your class - top, bottom or anywhere else - the computer will find it as long as you've called it "Main".
- In line 7, we call the "WriteLine" method from the System.Console class, which writes out the text we specify.

Console applications can, of course, do way more than what we've shown here, and they will typically have a lot more code, but generally they start out with a basic structure like the two examples shown above.

### Static Methods

The one strange word you'll see in the program above is "**static**". To explain this, let's first write a slightly more complex example of a console application. Read the "Main" method through carefully - do you understand what it's doing ?

```

1. using System;
2.
3. class SimpleConsoleApp
4. {
5.     public void MySimpleMethod()
6.     {
7.         Console.WriteLine("Yo!");
8.     }
9.
10.    static void Main()
11.    {
12.        SimpleConsoleApp s;
13.        s = new SimpleConsoleApp();
14.        s.MySimpleMethod();
15.    }
16. }
```

This example really shows the object-oriented approach better than the previous one. Instead of just writing our code directly in the "Main" method, we have, in lines 12 and 13, declared and constructed an object - an instance of this class. Then, in line 14, we have called a method of that object.

You'll notice that the method "MySimpleMethod" does not have the word "static" in front of it, while the method "Main" does. This is why:

- The word "static" is added to the beginning of a method definition if that method is **not meant for operating on a specific object**.
- Now since the "Main" method is the "entry point" method, it is called **before any objects are constructed** - so it has to be defined as "static".

Here is a further example to illustrate the “static” concept. We draw your attention particularly to the bold text.

```
using System;

class SchoolTest
{
    public string testName;
    public decimal total;
    public decimal myScore;

    // A constructor method for this class
    public SchoolTest(string tn, int s, int tot)
    {
        this.testName = tn;
        this.myScore = s;
        this.total = tot;
    }

    // A STATIC method to calculate the percentage from ANY two numbers
    public static decimal CalculateAnyPercentage(decimal numerator, decimal denominator)
    {
        decimal percentage = (numerator / denominator) * 100;
        return percentage;
    }

    // An INSTANCE method to calculate the percentage of an instance of SchoolTest
    public decimal CalculateThisPercentage()
    {
        decimal percentage = (this.myScore / this.total) * 100;
        return percentage;
    }

    static void Main()
    {
        // Use the static method to calculate a percentage without any instance
        Console.WriteLine("123/200 = " + SchoolTest.CalculateAnyPercentage(123, 200));

        // Create a new instance of a SchoolTest, passing in the scores
        SchoolTest t = new SchoolTest("Geography", 12, 60);

        // Use the instance method to calculate the percentage of this instance
        Console.WriteLine("Percentage for test = " + t.CalculateThisPercentage() );

        // Wait until ENTER is pressed
        Console.ReadLine();
    }
}
```

```
}
```

The important thing to notice is that we have two very similar methods – they both calculate percentages – but their usage is quite different. The one with the word “static” in it can be used without any need to declare a *new SchoolTest()* instance, while the other one can ONLY work on an instance. In the above example, our instance is named “t”.

So static methods have a kind of “**I’m above all this – I work where I want to!**” feel to them, while instance methods acknowledge “I only come into being when an actual object arises – and then **I only work inside that object**”.

### The Structure of Windows Forms Applications

Once you know the general structure of a Console application, it's easy to graduate to more complex program types, because the basic structure remains the same. Here is a simple Windows forms application.

```
using System.Windows.Forms;

class SimpleWindowsApp : Form
{
    // Constructor method

    public SimpleWindowsApp()
    {
        this.Text = "A really Simple Form";
    }

    // Entry point for the program

    static void Main()
    {
        Application.Run( new SimpleWindowsApp() );
    }
}
```

The two "new" things here are:

1. The ": Form" piece

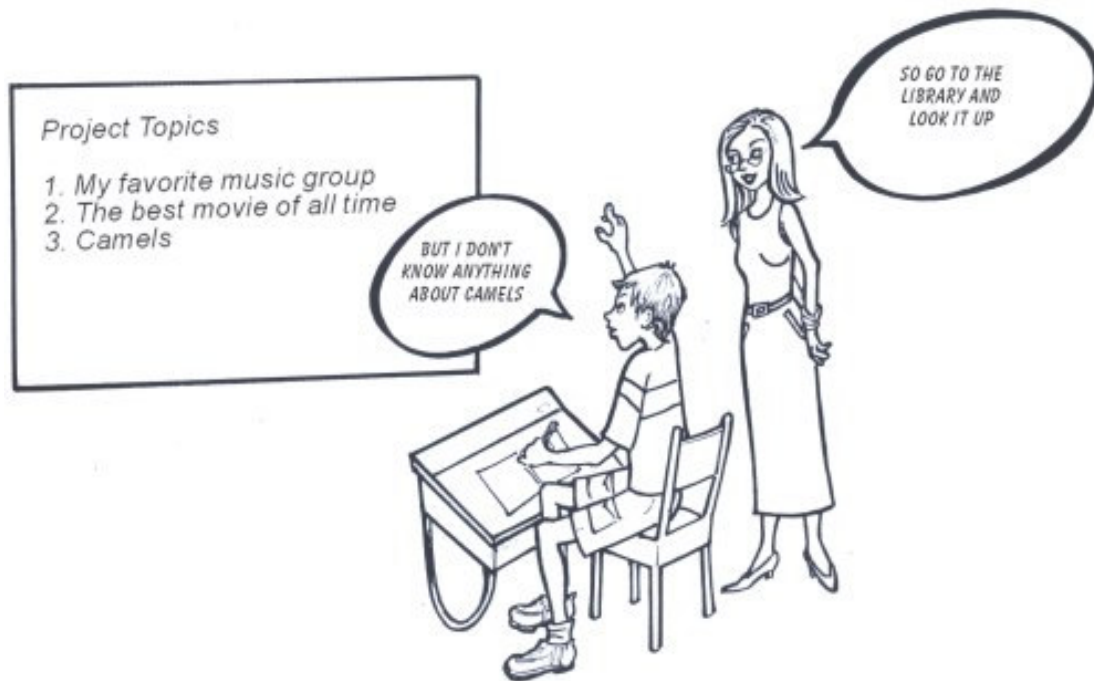
This is saying "my class is going to inherit everything from the class called 'Form'". This is quite an important secret, discussed in the chapter named “Inheritance”. Microsoft has written a large set of classes which know how to create buttons, menus, picture boxes, checkboxes and all sorts of other things to put in applications. Simply by adding ": Form", your own class can suddenly have access to all those things.

2. The "Application.Run" bit. This is the standard way to start a Windows forms application. It is a method which "runs" your forms application. You pass to that method a new instance of your class.

## Using Class Libraries

---

A library, in the real world, is a place that contains a large number of books and other sources of information. When people need to know something, they don't have to go and discover it all over again and write their own book - they can simply refer to the correct books, saving a lot of time and effort.



In the same way, people all across the world have written huge libraries of C# code. So, instead of us writing *everything* ourselves, it often makes sense to use someone else's libraries for parts of our programs.

This still takes some effort - you need to look up what's in the library and understand which parts are useful to you - but it allows you to concentrate more on doing what your program needs to do, rather than wasting a lot of time writing the background code.

By far the most important library for C# is Microsoft's ".NET Framework Class Library". You can't really write any useful program that doesn't use something from it. Part III of this book is all about that library. When you install the .NET Framework on your computer, the library is automatically loaded and is therefore available to your programs. You just need to know how to refer to it - how to use it from your programs.

### An Example

The "System.Drawing" part of the .NET Framework Class Library contains many useful classes for working with pictures. Let's use it to illustrate how to use other people's code in your own program.

The `System.Drawing.Image` class has a method called **RotateFlip**, which will rotate (turn around) or flip (reverse) any picture that you pass to it. Suppose you want to use that method in your own program - you want to load a picture file from disk, flip it horizontally, and save a copy of the flipped picture back to disk. Here's one way to do so. This is a complete, working program. (If you're experimenting, note that program 9 in the collection of example files provided with this book, is a similar program).

```
class PhotoSizer
{
    public PhotoSizer()
    {
        // Load a photograph from disk into the computer's memory
        System.Drawing.Image img;
        img = new System.Drawing.Bitmap(@"d:\samples\myDog.jpg");

        // Flip the image horizontally
        img.RotateFlip( System.Drawing.RotateFlipType.RotateNoneFlipX );

        // Save the flipped photo to a different disk file
        img.Save( @"d:\samples\myDogFlipped.jpg" );
    }

    static void Main()
    {
        PhotoSizer p = new PhotoSizer();
    }
}
```

A quick note about the @ symbol :

The C# @ symbol used above can be added at the beginning of a string to “escape” or ignore backslash characters if the path includes special characters like backslashes. Backslashes can confuse C# as they have special meaning. So usually you would have to write such a path with double backslashes, such as `d:\\samples\\myDog.jpg`. The @ symbol avoids the need to do that.)

What we've done in the example above is to use a class from a library. The class is called `Image` and it is in the section of the library called `System.Drawing`.

After creating an *Image* object, we called two methods of the image class. The methods' names are *RotateFlip* and *Save*.

### An Easier Way

But you can see that each time you need to talk about something in the `System.Drawing` space, you need to write those long words again. This is a waste of energy and there is a way to avoid it. Instead, you can tell your program once at the top that you'll be "using" some things from the `System.Drawing` namespace as well as some things from the `System.Drawing.Imaging` namespace specifically.

```
using System.Drawing;
using System.Drawing.Imaging;
```

```
class PhotoSizer
```

```

{
    public PhotoSizer()
    {
        // Load a photograph from disk into the computer's memory
        Image img = new Bitmap(@"d:\samples\myDog.jpg");

        // Flip the image horizontally
        img.RotateFlip( RotateFlipType.RotateNoneFlipX );

        // Save the flipped photo to a different disk file
        img.Save( @" d:\samples\myDogFlipped.jpg" );
    }

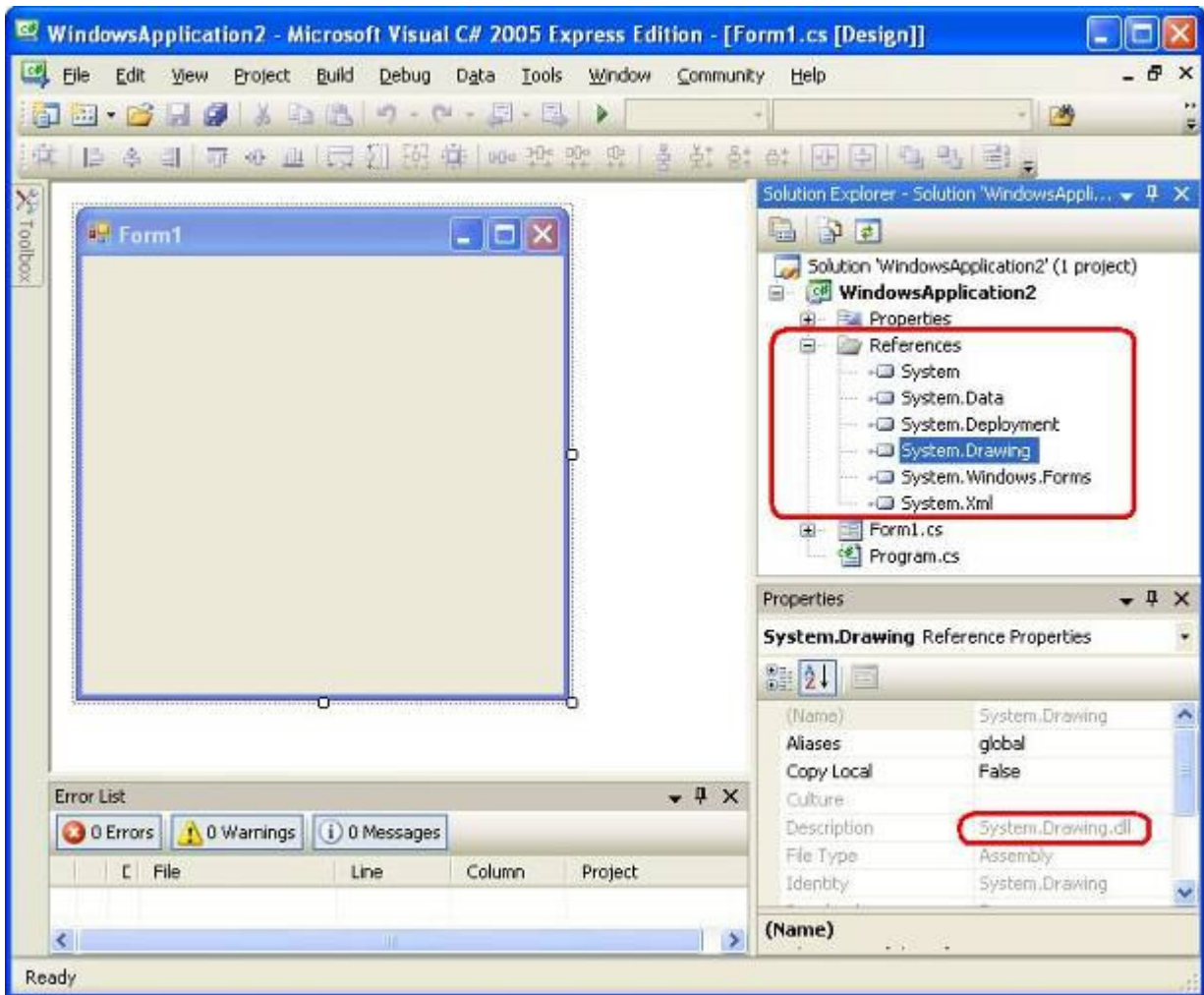
    static void Main()
    {
        PhotoSizer p = new PhotoSizer();
    }
}

```

### A Note About References in C# Express

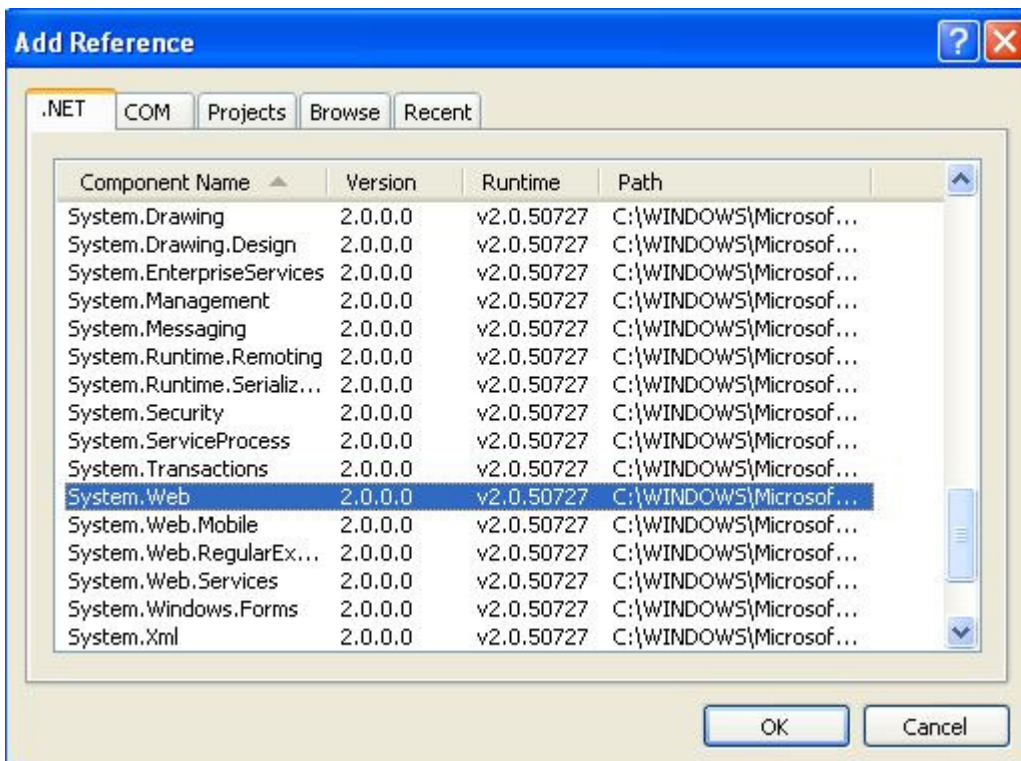
When creating a Visual C# Express project, it is also necessary to create something called a "reference" pointing to the Class Library file which holds the library code you want to use (these files have a ".dll" extension that stands for "Dynamic Link Library"). Fortunately, the most common references are usually added for you automatically.

The picture below, for example, shows the references that are added when you create a "Windows Application" project. Visual C# Express guesses the parts of the library that it thinks you might use for an application like that.



In the example above, you can see that the classes of the System.Drawing namespace are held in a file called System.Drawing.dll.

If the part of the library you want to use has not been automatically referenced for you, you need to add a reference yourself. Suppose, for example, you find a code example that uses the System.Web.HttpRequest class to retrieve some information from a web server, and you want to try it. You would right-click on “References”, select “Add Reference” and select the System.Web part of the library. Now you will find you can use its classes in your code.



## Conclusion

So it's quite simple to include code from a library. The hard part is knowing what classes, methods, etc. are available for you to use in this vast library.



That's why part IV of this book has been written - to expose you to some of the useful parts of Microsoft's .NET Framework class library.

## **About the Book**

Microsoft's newest programming language, C#, (pronounced "c-sharp") is both powerful and easy to use. It presents a great opportunity for the new generation of developers to start out with a language that is highly respected in the modern workplace.

This text introduces object-oriented programming to the young developer (core target age is 12-16) in a lightweight fashion, allowing them to get started with real programs in a Windows environment.

## **About the Author**



Martin Dreyer is an ex-high school teacher who now heads a team of software developers in South Africa.

His formal qualifications are a Higher Diploma in Education : Physical Science and a Bachelor of Science Degree : Computer Science and Information Systems.