Chairbear Digital Learning

C# Essentials

Author: Jade Lei

Created: 19th June, 2020

Table Of Contents

Foreword

This book was written for educational purposes, by Jade Lei ©Chairbear, for use by any individual or organization under the BY-NC-ND creative commons licence (Attribution + Noncommercial + NoDerivatives).

This book is particularly useful for those studying a range of qualifications, including software development and IT.

A breadth of topics are covered in this book, with methods and properties of data types only explored briefly. To see methods and properties of data types in depth, with individual examples for each case, please see our other C# books.

Examples shown in this book are explained in as much depth as necessary for the subject they demonstrate.

Learning code is a rewarding experience, and we encourage you to practice coding fundamentals after you have read about them to help solidify your knowledge.

A special thank you to my father, who always supported me, and to Click for giving me the opportunity to learn software development under their amazing guidance.

We would also like to acknowledge that not everyone learns the same way, and so suggest you find additional resources that accommodate your learning style.

Jade Lei
SEO Technical Specialist,
Junior Software Developer

Prerequisite C#

Here are some of the fundamental C# rules and syntax that you should know before continuing to explore the rest of this book.

```
using System;


namespace fruits
{
 class Program
 {
   static void Main(string[] args)
   {
       var Fruit = "apple";
      var FRUIT = "apple";
      var fruit = "apple";
      Console.WriteLine(FRUIT);
   }
 }
}
```

Case Sensitivity

C# is case sensitive. The variables ‘fruit’, ‘FRUIT’ and ‘Fruit’ are all different variables.

Line Spacing

C# ignores additional spacing, like the above space after `using System;` but just because you can use lots of spaces and tabs doesn't mean you should, the best practise is to include a space on either side of operator and assignment signs (such as the = sign), to make the code more human readable.

For this same reason, avoid code lines longer than 80 characters. If a C# statement does not fit on one line, the best place to break it is after an operator.

Ending Statements

All C# statements should end with a semicolon, as best practice.

Comments

Not all C# statements are executed, code after double slashes (//) or between /* and */ is treated as a comment. Comments are useful for adding notes to bits of code to make it easier to understand when revisiting it later.

```
//this is a single line comment

/* this is

a  multi line comment */
```

Creating Variables

Variables are containers for storing data. There are two ways to declare variables in C#, below we will explore both methods and their syntax.

Var & Const Method

Const works similarly to var (which is short for variable), in which you use the keyword var, or const followed by a name for the variable. Then an equal sign (the assignment operator), and a value you want to put into the variable.

Const is short for constant, meaning that the value of the variable should not change.

Const lets you protect the variable from being overwritten by any accidental or stray assignments in your code.

```
const string firstName = "Jem";

var age = 23;

Console.WriteLine(age);     //expected output: 23
```

```
age = 35;

Console.WriteLine(age);    //expected output: 35 as var let us reassign a value to
the age variable

firstName = "Jack";

Console.WriteLine(firstName);   //expected output: error, as we cannot reassign a
constant variable value
```

Note: You cannot declare a constant variable without assigning the value and data type. If you do, an error will occur.

```
const firstName; //expected output: error
```

When declaring variables using the var keyword but omitting the data type, they also need to have a preliminary value -that is, to assign a value to it immediately. This is because C# will look at the type of the value the variable is initialized with, and use that as the type of the variable - this is known as being 'implicitly-typed'.

```
var age = 23;

Console.WriteLine(age); //expected output: 23 as age has implicitly typed type of int
```

If you declare a variable with the var keyword and data type, you can omit the value, and assign it later.

```
var int age;
```

Type Assignment Method

The other method, which we will use going forward, includes specifying the data type, a name of the variable and then assigning it a value with the assignment operator.

```
string firstName = "Jem";

int age = 23;
```

You don't use the declaration keywords var or const with this method, however you should note that omitting these keywords gives the variable the default of var meaning the variable can be overridden.

Declare Multiple Variables At Once

To declare more than one variable of the **same type**, use a comma-separated list.

```
int x = 5, y = 10, z = 50;

Console.WriteLine(z); //expected output: 50
```

Note: implicitly typed variables cannot have multiple declarations. The below will throw an error.

```
var a = 10, b = 5;
```

Naming A Variable

All C# variables must be identified with unique names (called identifiers). A C# variable name must start with a letter, or underscore (_) and then can be followed by any sequence of letters or numbers, but must never start with a number or contain spaces or special characters.

C#, like other programming languages, has reserved keywords that cannot be used as variable names such as *class, static* and more. To view all current, and potential future reservations, view Microsoft's reference.

It is best practice to give variables descriptive names, so that when someone looks back over the code it is easier to understand what it does. Because of this you may use variable names that contain multiple words. For this, you can use the aforementioned underscore or use camel case. Camel case is the act of capitalizing every word after the first word, like pricePerUnit.

Variable Scope

A variable's scope is the visibility of the variable. This means what classes, methods, ect can see and therefore use the variable. For example, variables declared within a method are only accessible within that method. They are local variables. This allows you to use the same variable name in multiple methods without negatively affecting the outcomes of particular methods. Global variables are those accessible by multiple methods,ect, as they exist outside of a particular method scope.

Display Variables

The most common way to display variables is with the WriteLine() method, but you can also use the Write() method. The difference between them is that WriteLine() prints the output on a new line each time, whilst Write() prints on the same line. With Write() you should add spaces where needed when outputting multiple variables at once, or concatenating them, as omitting spaces will place the variables directly next to each other.

```
Console.WriteLine("Hello!");

Console.WriteLine("Welcome to C#");

//expected output: Hello!

//Welcome to C#



Console.Write("Hello!");

Console.Write("Welcome to C#");

//expected output: Hello!Welcome to C#

Console.Write("Hello! ");

Console.Write("Welcome to C#");

//expected output: Hello! Welcome to C#
```

Data Types

A data type specifies the type of values a variable can accept or are holding, and will be used by C# to determine what operations a variable can do.

```
string firstName = "Jem";           //string

char alphabet = 'a';                //character

int age = 23;                       //integer

long years  = 7000000L;              //big integer
```

```
float BMI = 81.7F;                   //short decimal

double weight = 15.66;               //long decimal

bool onePlusOneEqualsTwo = true;    //boolean
```

Data Type	Meaning
string	**String** Stores a sequence of characters, must be surrounded by double quotes
char	**Character** Stores a single character, must be surrounded by single quotes
int	**Integer** Stores whole numbers. Accepts values from -2,147,483,648 to 2,147,483,647
long	**Long Integer** Stores whole numbers. Accepts values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Note that you should end the value with an "L"
float	**Short Decimal** Stores decimal (fractional) numbers. Can be used for storing 6 to 7 decimal digits. Note that you should end the value with an "F"
double	**Long Decimal** Stores decimal (fractional) numbers. Can be used for storing up to 15 decimal digits
bool	**Boolean** Stores true or false values

If you are ever unsure of a datas type you can use the GetType() method to check.

```
Console.WriteLine(firstName.GetType());

//expected output: System.String
```

Types help prevent errors in code, such as trying to do a math expression with a string, this is known as type safety. This assures that when you have a value of a particular type, you can do particular operations on it, and be certain those operations will work.

Type Casting

Type casting is when you assign a value of one data type to another type. There are two types of casting in C#, these are implicit and explicit.

Implicit

Implicit casting consists of automatically converting a smaller type to a larger type. It is done when passing a smaller size type to a larger size type.

Such as char to int, int to long or float to double.

```
int age = 23;

long years  = age;       //automatic casting of int to long

Console.WriteLine(age);      //expected output: 23

Console.WriteLine(years);   //expected output: 23
```

Explicit

Explicit casting consists of manually converting a larger type to a smaller size type.

For example, double to float, or int to char.

It must be done manually by placing the type in parentheses in front of the value.

```
double decimalNum = 23.78;

int wholeNum = (int) decimalNum;    // manual casting of double to int

Console.WriteLine(decimalNum);   //expected output:  23.78

Console.WriteLine(wholeNum);      //expected output: 23
```

In C# there are also built-in methods for type conversion: Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long).

```
int wholeNum = 23;

int truevar = 1;
int falsevar = 0;

double decimalNum = 23.78;



Console.WriteLine(Convert.ToBoolean(truevar));  //expected output: true

Console.WriteLine(Convert.ToBoolean(falsevar)); //expected output: false

Console.WriteLine(Convert.ToDouble(wholeNum));  //expected output: 23

Console.WriteLine(Convert.ToString(wholeNum));  //expected output: "23"

Console.WriteLine(Convert.ToInt32(decimalNum)); //expected output: 24

Console.WriteLine(Convert.ToInt64(decimalNum)); //expected output: 24
```

Data rarely has to be type converted, but in the instance of user input, the above comes in very useful.

Whilst Console.WriteLine is used to output values, Console.ReadLine is used to get user input.

In the below example we output the text ‘What is your name?’, and the input the user types is stored in a variable called name, meaning we can use this later in some other code.

```
Console.WriteLine("What is your name?");

string name = Console.ReadLine();
```

The Console.ReadLine method returns a string, so if you wanted to collect user input of another type, say int age, you’d have to use one of the conversion methods above to collect the data in the correct type to be able to use it later.

```
Console.WriteLine("What is your age?");

int age = Convert.ToInt32(Console.ReadLine());

if (age < 18){

   Console.WriteLine("You're too young for this site!");

} else

{
    Console.WriteLine("Welcome!");

}
```

The above example asks the user for their age, and assigns their input to the age variable. If the user's age is less than 18, the ‘too young’ message is written to the console, else the ‘welcome’ message is shown.

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on variables and values.

Symbol	Name & Meaning	Example

+	**Addition**
	Adds numbers or variables containing numbers together
		`5 + 5        //10`
		`int x = 10, y = 20;`
		`x + y        //30`

-	**Subtraction**
	Subtracts numbers or variables containing numbers together
		`5 - 5        //0`
		`int x = 10, y = 20;`
		`x - y        //-10`

*	**Multiplication**
	Multiplies numbers or variables containing numbers together
		`5 * 5        //25`
		`int x = 10, y = 20;`
		`x * y        //200`

/	**Division**
	Divides numbers or variables containing numbers
		`10 / 5        //2`
		`int x = 36, y = 6;`
		`x / y        //6`

%	**Modulus**
	Returns the division remainder of numbers or variables containing numbers
		`10 % 5        //0`
		`int x = 20, y = 6;`
		`x % y        //2`

++	**Increment**
	Increases the value of a variable by 1
		`int x = 20;`
		`x++; //21`

--	**Decrement**
	Decreases the value of a variable by 1
		`int x = 20;`
		`x--; //19`

Math Methods

C# also comes built in with a Math class which has many methods that allow you to perform mathematical tasks on numbers. Below are a few examples of the Math Class methods, to see all view our C# Math book.

Method	Name & Meaning	Example

Math.Max(A,B)	**Math.Max**
	Can be used to find the highest value of A and B
		`int A = 10, B = 20;`
		`Math.Max(A,B); //20`

Math.Min(A,B)	**Math.Min**
	Can be used to find the lowest value of A and B
		`int A = 10, B = 20;`
		`Math.Min(A,B); //10`

Math.Sqrt(A)	**Math.Sqrt**
	Returns the square root of A
		`int A = 10, B = 36;`
		`Console.WriteLine(Math.Sqrt(A));`
		`//3.16227766016838`
		`Console.WriteLine(Math.Sqrt(B));`
		`//6`

Math.Abs(A)	**Math.Abs**
	Returns the absolute (positive) value of A
		`int A = -10, B = -36;`
		`Console.WriteLine(Math.Abs(A));`
		`//10`
		`Console.WriteLine(Math.Abs(B));`
		`//36`

Math.Round(A)	**Math.Round**
	Rounds a number to the nearest whole number
		`double A = 10.99;`
		`Console.WriteLine(Math.Round(A));`
		`//11`
		`Console.WriteLine(Math.Round(36.4));`
		`//36`

Assignment Operators

Assignment operators are used to assign values to variables.

Symbol	Example

=	`int x = 10;       //10`
	`int y = 20;      //20`

+=	`int x = 10;`
	`x += 5; //15`

-=	`int x = 10;`
	`x -= 5; //5`

*=	`int x = 10;`
	`x *= 5; //50`

/=	`int x = 10;`
	`x /= 5; //2`

%=	`int x = 10;`
	`x %= 5; //0`

Below are some more advanced assignment operators, to understand them you first must understand bitwise.

Bitwise & Bitwise Operators

Bitwise is a level of operation that involves working with individual bits (the smallest units of data in a computer). Each bit has a binary value of 0 or 1. Bits are usually executed in bit multiples called bytes, with most programming languages manipulating groups of 8, 16 or 32 bits.

Bits can be complex for humans to understand, but are far easier to computers, as each bit (aka binary digit) can be represented by an electrical signal which is either on (1) or off (0).

See the below binary conversion table. The binary example below (11010) is equal to 26 in the decimal system (16+8+2=26).

128	64	32	16	8	4	2	1
			1	1	0	1	0

Here is another example:

00001010 is the equivalent of 10 in the decimal system.

128	64	32	16	8	4	2	1
0	0	0	0	1	0	1	0

11110 would be the equivalent of 30.

Note: With binary notation you can perform some math operations very quickly:

- To half any number - simply move the digits 1 place to the right. Ei 11110 (30) -> 1111 (15)
- To double a number - simply add a zero on the end (shifts digits to the left). Ei 11010(26) -> 110100 (52)

Bitwise operators are useful to perform bit by bit operations.

Symbol	Name & Meaning	Example

&	**Bitwise AND** This compares each individual bit of the first operand with the corresponding bit of the second operand. If both bits are 1, then the result bit will be 1 otherwise it will be 0	`int a = 10; //(00001010)`
		`int b = 20; //(00010100)`
		`Console.WriteLine(a & b); //(00000000) equal to 0`
		`Console.WriteLine(a); //(00001010) equal to 10`

&=	The &= operator compares the bits of the operand on its right to the operand on its left, and assigns the result to the operand on its left	`int a = 10; //(00001010)`
		`int b = 20; //(00010100)`
		`a &= b; //assigns result to a`
		`Console.WriteLine(a); //(00000000) equal to 0`

|	**Bitwise OR** This compares each individual bit of the first operand with the	`int a = 10; //(00001010)`
		`int b = 20; //(00010100)`

	corresponding bit of the second operand. If either of the bits is 1, then the result bit will be 1 otherwise the result will be 0	`Console.WriteLine(a | b); //(00011110) equal to 30`
		`Console.WriteLine(a);     //(00001010) equal to 10`

|=	The |= operator compares the bits of the operand on its right to the operand on its left, and assigns the result to the operand on its left	`int a = 10; //(00001010)`
		`int b = 20; //(00010100)`
		`a |= b;`
		`Console.WriteLine(a); //(00011110) equal to 30`

^	**Bitwise Exclusive OR (XOR)**
	It compares each individual bit of the first operand with the corresponding bit of the second operand. If one of the bits is 0 and the other is 1, then the result bit will be 1 otherwise the result will be 0
		`int a = 10;               //(00001010)`
		`int b = 20;               //(00010100)`
		`Console.WriteLine(a ^ b); //(00011110) equal to 30`
		`Console.WriteLine(a);     //(00001010) equal to 10`

^=	The ^= operator compares the bits of the operand on its right to the operand on its left, and assigns the result to the operand on its left	`int a = 10; //(00001010)`
		`int b = 20; //(00010100)`
		`a ^= b;`
		`Console.WriteLine(a); //(00011110) equal to 30`

<<	**Bitwise Shift Left**
	This shifts bits to the left by the amount specified on the right.
	Ei shift of 1, moves the bits left 1 position each (end will have a zeo added to it). This is equivalent to multiplication by 2
		`int b = 20;                   //(00010100)`
		`Console.WriteLine(b << 2 );        //(1010000) equal to 80`
		`(2 is equivalent of multiplying by 4)`
		
		`int b = 20;                   //(00010100)`
		`Console.WriteLine(b << 1 ); //(101000) equal to 40`

<<=	The <<= operator shifts the bits of the operand on its left by the amount specified on the right, and assigns the result to the operand on its left	`int b = 20;              //(00010100)`
		`b <<= 1;`
		`Console.WriteLine(b); //(101000) equal to 40`

>>	**Bitwise Shift Right**
	This shifts bits to the right by the amount specified on the right.
	Ei shift of 1, moves the bits right 1 position each. This is equivalent to
		`int b = 20;                   //(00010100)`
		`Console.WriteLine(b >> 1 ); //(00001010) equal to 10`

	division by 2	

>>=	The >>= operator shifts the bits of the operand on its left by the amount specified on the right, and assigns the result to the operand on its left	`int b = 20;   //(00010100)`
		`b >>= 1;`
		`Console.WriteLine(b); //(00001010) equal to 10`

The below operand is used to flip bits, ei 0 becomes 1 and vis vera.

Symbol	Name & Meaning	Example

~	**Bitwise Complement**
	This operates on only one operand, and it will invert each bit of the operand (ei it will change 1 to 0 and 0 to 1)
		`int b = 20;   //(00010100)`
		`b =~(b);`
		`Console.WriteLine(b); // -21 (11101011)`

Strings

A string in C# is treated as an object, meaning there are properties and methods that can perform certain operations on strings.

For example the .Length property or .ToUpper() and .ToLower() methods.

```
string name = "Jem";

Console.WriteLine(name.Length);      //3

Console.WriteLine(name.ToUpper());   //JEM

Console.WriteLine(name.ToLower());   //jem
```

.ToUpper() and .ToLower() methods return a copy of the string converted to uppercase or lowercase, they do not change the original string. To view all String Class Methods, view our C# Strings book. Below we explore a few more examples of string methods and properties.

You can access the individual characters in a string by referring to the characters index number inside square brackets. C# is a zero-based indexing system, meaning that it starts counting from zero.

```
string name = "Jem";

Console.WriteLine(name[1]);      //e
```

If you want to find the index position of a character, you can use the IndexOf() method.

```
string name = "Lorenzo";

Console.WriteLine(name.IndexOf("z")); //5
```

String Combination

C# provides many ways to combine two+ strings into a single string.

String concatenation is the process of combining two or more strings together to create a bigger string. The best method to do this is called string interpolation, which uses placeholders in a string, which will then be replaced with values of variables later.

With interpolation the string is prepended with a dollar sign($), and the placeholders within the string are surrounded by curly braces.

```
string name = "Jem";

string greeting = $"Hello {name}! Welcome to C#";

Console.WriteLine(greeting);  //Hello Jem! Welcome to C#
```

An additional benefit of the interpolation is that you don't have to be mindful of spaces like the below methods.

Another way to concatenate strings in C# is with the + operator.

```
string name = "Jem";

string greeting = "Hello " + name + "! Welcome to C#";

Console.WriteLine(greeting);  //Hello Jem! Welcome to C#
```

Did you notice that we had to add a space after 'hello', if we didn't the outcome would have been 'HelloJem! Welcome to C#'. When you combine strings with the concatenation(+) method, you're literally placing them next to each other. So if you need a space, you have to add it.

Another way is with the addition assignment method.

```
string name = "Jem";

string greeting = "Hello " + name;
```

```
greeting += "! Welcome to C#";

Console.WriteLine(greeting); //Hello Jem! Welcome to C#
```

The final method is with the string.Concat() method.

```
string name = "Jem";

string greeting = "Hello ";

string greeting2 = "! Welcome to C#";

Console.WriteLine(string.Concat(greeting, name, greeting2));

//Hello Jem! Welcome to C#
```

In C#, the plus (+) operator is used for both addition and concatenation. The difference between the usage is that when used with numbers it is interpreted as addition, whilst when used with strings it is concatenation.

```
int x = 10, y = 20;

int z = x + y;  //expected output : 30

Console.WriteLine(z);

string a = "10", b = "20";

string c = a + b; //expected output : 1020

Console.WriteLine(c);
```

Typically, if you add a number and a string, the result will be a string concatenation. However, as C# interprets from left to right, it depends on the placement of numbers in the expression.

```
int x = 10;

int y = 20;

var z = "X and Y added is: " + x + y;

Console.WriteLine(z);

/*expected output : 'X and Y added is: 1020' x and y are not added
```

```
as C# hits the string first, and so treats numbers after it as strings too*/
```

```
var a = 10;

var b = 20;

var c = "10";

var d = a + b + c;

Console.WriteLine(d);

/*expected output : 3010 as C# hits the numbers first, and so adds them correctly,

however, once it hits the string, it changes from addition to concatenation,

and outputs the string '3010'. */
```

Strings & Escape Characters

Because strings must be written within quotes, C# will misunderstand additional double quote marks within, and cut the string short. The solution to avoid this problem, is to use the backslash escape character, which turns special characters into string characters.

Escape Sequence	***Meaning***
\’	Single Quote
\”	Double Quote
\\	Backslash
\b	Backspace
\n	New Line
\t	Horizontal Tabulator

```
string mood = “I\’m ok”;
```

Arrays

Arrays are used to store multiple values in a single variable.

Create An Array

There are many ways to create an array, one way is to declare the variable type, followed by a pair of square brackets and then the array name.

```
string[] animals;  //creates a string array called animals

int[] years;       //creates a int array called years
```

Values can then be added to this array declaration type later by creating an array literal (a comma separated list of values inside a pair of curly braces).

```
string[] animals = {"bear", "bat", "bee", "beetle"};
```

Other ways to create arrays include:

```
// Creates a string array called animals,which can hold four elements. These
elements can be added later

string[] animals = new string[4];



// Creates a string array of four elements and initializes it with the values

string[] animals = new string[4] {"bear", "bat", "bee", "beetle"};



// Creates an array, initialized with four elements, but without specifying the
size

string[] animals = new string[] {"bear", "bat", "bee", "beetle"};
```

Access Elements In An Array

We can access elements in an array using the index of the element. This includes writing the array name, followed by a pair of square brackets, and in said brackets writing the index number of the element you want to retrieve - starting at 0 for the first element in the array, 1 for the second, and so forth. If you try to access an index that does not exist in the array, you'll get a 'index out of range' exception.

```
string[] animals = {"bear", "bat", "bee", "beetle"};

Console.WriteLine(animals[0]);
```

```
//expected output: bear

Console.WriteLine(animals[2]);

//expected output: bee
```

Change Array Elements

We can also change elements in an array using indexes.

```
string[] animals = {"bear", "bat", "bee", "beetle"};

animals[0] = "dog";

foreach ( string i in animals) {

  Console.WriteLine(i); }

/*expected output:

dog

bat

bee

beetle

*/
```

Find Array Length

We can find an array's length using the .Length property. This property can also be used to select the last element in an array - demonstrated below.

```
Console.WriteLine(animals[animals.Length-1]); //expected output: beetle
```

Looping Through Arrays

You can loop through array elements with the for loop, by using the .Length property to specify how many times the loop should run. Demonstrated below.

```
string[] animals = {"bear", "bat", "bee", "beetle"};

for (int i = 0; i < animals.Length; i++)

{
  Console.WriteLine(animals[i]);

}

/*expected output:

bear

bat

bee

beetle

*/
```

However it is better to use the foreach loop for looping through arrays.

Conditional Statements - If Else & Switch

With conditional statements you'll often use logical and or comparison operators to test a condition to be able to output(run) particular code blocks if the condition is true, and another if it is false.

Symbol	Name & Meaning	Example

==	**Equal To**
	Tests whether expression on it's left is equal to that on it's right
		`bool z = 5 == 6;`
		`Console.WriteLine(z);`
		`//expected output: false`

!=	**Not Equal To**
	Tests whether expression on it's left is not equal to that on it's right
		`bool z = 5 != 6;`
		`Console.WriteLine(z);`
		`//expected output: true`

>	**Greater Than**
	Tests whether expression on it's left is greater than that on it's right
		`bool z = 5 > 6;`
		`Console.WriteLine(z);`
		`//expected output: false`

<	**Less Than**
	Tests whether the expression on it's left is less than that on it's right. A way to remember the difference between greater than and less than is that less thans symbol looks like an 'L' (for less than)
		`bool z = 5 < 6;`
		`Console.WriteLine(z);`
		`//expected output: true`

>=	**Greater Than Or Equal To**
	Tests whether expression on it's left is greater than, or equal to that on it's right
		`bool z = 5 >= 6;`
		`Console.WriteLine(z);`
		`//expected output: false`

<=	**Less Than Or Equal To**
	Tests whether expression on it's left is less than, or equal to that on it's right
		`bool z = 5 <= 6;`
		`Console.WriteLine(z);`
		`//expected output: true`

&&	**Logical AND**
	Returns true if both statements are true, else returns false
		`bool z = (5 == 5) && (5 < 6);`
		`Console.WriteLine(z);`
		`//expected output: true`

||	**Logical OR**
	Returns true if one of the statements are true, else returns false
		`bool z = (5 == 5) || (5 < 6);`
		`Console.WriteLine(z);`
		`//expected output: true`

!	**Logical NOT**
	Reverses the result - it returns false if the result is true, and true if the result is false
		`bool z = !(5 == 5);`
		`Console.WriteLine(z);`
		`//expected output: false`

The If Statement

The if statement is used to specify a block of code to be executed if the condition within it's parentheses () is true.

```
if (condition){

   code to execute if condition is true

}
```

The below example asks the user for their age, and assigns their input to the age variable. If the user's age is less than 18, the ‘too young’ message is written to the console.

```
Console.WriteLine("What is your age?");

int age = Convert.ToInt32(Console.ReadLine());

if (age < 18){

   Console.WriteLine("You're too young for this site!");

}
```

We can then use the else statement to specify a code block to execute if the condition is false.

```
if (condition){

    code to execute if condition is true

} else

{

    code to execute if condition is false

}
```

```
Console.WriteLine("What is your age?");

int age = Convert.ToInt32(Console.ReadLine());

if (age < 18){

   Console.WriteLine("You’re too young for this site!");

} else

{

   Console.WriteLine("Welcome!");

}
```

The above example asks the user for their age, and assigns their input to the age variable. If the user's age is less than 18, the ‘too young’ message is written to the console, else the ‘welcome’ message is shown.

If we wanted to test another condition if the first evaluates to false, we can add an else if statement after the initial if statement, with a new condition to test in it’s parentheses.

```
if (condition){

    code to execute if condition is true

} else if (second_condition){

    code to execute if condition is true

} else {

    code to execute if condition and second_condition are false

}
```

You can add as many else if statements as you want, however for a conditional statement that has multiple conditions to test, it is best to use the switch statement instead.

The Ternary Operator

The ternary operator is used as a shorthand for if else statements. It is known as the ternary operator because it consists of three operands- see the syntax below.

```
variable = (condition) ? outputForTrue_Expression_Evaluation :
outputForFalse_Expression_Evaluation;
```

The above syntax assigns the ternary expression outcome to a variable, but you can just as easily use it with methods by replacing the ‘variable =’ part with the return keyword.

The below is the previous if else example reconstructed in the ternary operator format.

```
int age = Convert.ToInt32(Console.ReadLine());

string greeting = (age < 18) ? "You're too young for this site!" : "Welcome";

Console.WriteLine(greeting);
```

The Switch Statement

The switch statement is used to select one of many code blocks to be executed based on cases.

```
switch(expression)

{

  case A:

    code to execute if case A is true

    break;

  case B:

    code to execute if case B is true

    break;

  default:

    code to execute if NON of the cases above are true

    break;

}
```

With the switch statement, the expression is evaluated once and the value of the expression is compared with the values of each case. If a case value matches, then it's associated code block will be executed.

The break keyword is important because when C# reaches a break keyword, it *breaks* out of the switch block, which stops the execution of more code and case testing inside the switch block, and stops unexpected behaviour from happening.

The default keyword is optional. It is used to specify some code to run if there is no case match.

The below example of a switch statement first asks the user for their favourite animal, to which their input is stored in a variable called 'favAnimal'. As you can see, their input is also converted to all lowercase letters, so that even if they type 'DOG' or 'DoG', they would still get the correct switch case of 'dog'.

The switch evaluates the users input from the 'favAnimal' variable and compares it to it's available cases. Ei if the user input 'cat' then this matches the 'cat' case, and it's code block is executed.

You may also notice that there is one code block for multiple 'bear' type cases. This is fine. If you want one particular code block to run for more than one case, instead of copy and pasting the code block over and over again, you can just place the cases one after another, with the last case holding the code to execute if any of the cases defined match.

```
Console.WriteLine("What\'s your favourite animal?");

string favAnimal = Console.ReadLine().ToLower();

switch (favAnimal){

  case "dog":

  Console.WriteLine("I love dogs too!");

  break;

  case "cat":

  Console.WriteLine("I love cats too!");

  break;

  case "panda":

  case "bear":

  case "red panda":

  case "polar bear":

  Console.WriteLine("I love all species of bears!");

  break;

  default:

  Console.WriteLine("Animals are great!");

  break;

}
```

Loops

Loops are used to repeatedly execute a block of code, so long as a condition is true. Once it becomes false the execution stops. When working with loops it's important to create a condition or expression that will eventually become false, otherwise the loop will never end. You can do this by incrementing or decrementing a variable until it reaches a point where the expression using it becomes false.

While

The while loop loops through a code block as long as it's condition is true.

```
while (condition)

{

   code to execute if condition is true

}
```

The below example prints the numbers 0-4.

```
int i = 0;

while (i < 5) {

    Console.WriteLine(i);

    i++;}

/*expected output:

0

1

2

3

4 */
```

Do While

The do while loop is a variant of the while loop. The difference between them is that the do while loop will execute the code block once before testing the condition. Then it will repeat the loop as long as the condition is true.

```
do {

  code to execute

}

while (condition);
```

With do while, the loop will always be executed at least once- even if the condition is false- because the code block is executed before the condition is tested.

```
int i = 5;

do{

    Console.WriteLine(i);

    i++;

}

while (i < 5);

//expected output: 5
```

For

The for loop is useful when you know exactly how many times you want to loop through a code block. It's syntax is as below.

```
for (statement1; condition; statement3)

{

   code to execute

}
```

`statement1` is evaluated once before the execution of the code block.

`condition` is the condition to be evaluated, it defines the condition required to execute the code block.

`statement3` is evaluated every time, after the code block has been executed. This is the statement that is usually used to increment or decrement a variable to ensure that the condition will eventually evaluate to false.

```
for (int i = 1; i < 5; i++)

{

  Console.WriteLine(i);

}

/*expected output:

1

2

3

4 */
```

In the example above, a local variable called i is created and initialized with the value of 1. The value is incremented by 1 every time the loop runs, and prints the increasing variable each time to the console. The loop is set to run as long as i is less than 5, once it reaches 5 the loop stops.

Foreach

Foreach is used exclusively to loop through elements in an array.

```
foreach (type variableName in arrayName)

{

  code to execute

}
```

`variableName` here is a local variable, meaning that you can name it anything you want - as long as it follows standard variable naming convention.

The below example prints all elements in the animals array to the console. The foreach statement below can be read as ‘ for each string element called i (local variable) in the animals array, print the value of i.’

```
string[] animals = {"bear", "bat", "bee", "beetle"};

foreach ( string i in animals) {

  Console.WriteLine(i); }

/*expected output:

bear

bat

bee

beetle

*/
```

Break & Continue

As well as being used in the switch statement to break out of the switch, break can also be used to jump out of loops.

```
for (int i = 1; i < 8; i++) {

  if (i == 4)   {

       break;

}

  Console.WriteLine(i);

}

/*expected output:

1

2
```

```
3
*/
```

The above example breaks out of the loop once the value of the local variable i reaches 4.

The continue keyword on the other hand, is used to break one iteration in the loop, if a specified condition occurs. It then continues with the next iteration in the loop.

```
for (int i = 1; i < 8; i++) {
  if (i == 4)   {
    continue;
  }
  Console.WriteLine(i);
}
/*expected output:
1
2
3
5
6
7
*/
```

The above example, ‘skips’ over printing the value of 4, and continues on with the loop until it’s condition becomes false.

Methods

A method is a group of code statements that are put together to perform a particular task (they are also called functions).

Create A Method

To create a method define the name of the method, followed by parentheses () and then a pair of curly braces. These curly braces are the method body and hold one+ lines of code that will be run when the method is called.

Method names follow the same naming conventions as naming variables, however programmers typically start with an uppercase letter for method names.

```
static void AMethod()

{

}
```

Method names will be prepended with keywords such static and return values such as void.

The `static` keyword means you can call the method by itself - it doesn't belong to an object, GetType() is an example of a method you can only call on an object, where you must place a dot operator after an object, then call the method.

Static VS Instance Methods

An example of a static method is the WriteLine() method. Console is the name of the class, and we call the WriteLine() method directly on the Console class without constructing a console object.

```
      Console.WriteLine(FRUIT);
```

Compare this to the instance method below: The sound method is not static. And so we had to construct an animal class object (called dog), to then be able to call sound().

```
...

public void sound(string sound)    // method

  {

    Console.WriteLine($"The sound I make is: {sound}!");

  }}

public class Program

{
```

```
public static void Main(string[] args)

     {

          animal dog = new animal();

          dog.sound("woof");  //The sound I make is: woof!

     }
```

It’s important to remember that methods that are called directly on a class name are static whilst instance methods are called on unique instances(objects) of the class.

Return Values

`void` means that the method has no return value. If the method should return a value, use the data type of the value it should return instead of the void keyword. The return keyword should also be used within the method.

```
static int AMethod(int x, int y)

{

    return x*y;

}

public static void Main(string[] args)

{

    Console.WriteLine(AMethod(5,2));

}

//expected output: 10
```

The example above returns an int value.

Call A Method

To call (run) a method, write the method's name followed by two parentheses () and a semicolon.

```
AMethod(5,2);
```

Parameters & Methods

Information can be passed to methods as parameters - which act as variables inside the method. Parameters are declared inside the method parentheses, and follow variable naming convention.

You can add as many parameters as you want, but they must be separated by a comma.

```
static void AMethod(string name, int age)

         {

              Console.WriteLine($"Hello {name}! You are {age} years old." );

         }

 public static void Main(string[] args)

        {

            AMethod("Jem",23);

        }

//expected output: Hello Jem! You are 23 years old.
```

Note: you may hear people use parameter and argument interchangeably, this is because when a parameter value is passed to the method, this is called an argument. In the above example, name and age are parameters whilst 'Jem' and '23' are arguments.

Parameters can accept default values by using the assignment sign after the parameter.

For example, below gives name the default of 'friend'.

```
static void AMethod(string name = "friend", int age)
```

When using multiple parameters it is important to remember that the method call must have the same number of arguments as there are parameters. Arguments should also be passed in the same order as they are declared, however it is possible to pass a method arguments in key:value pairs which means the order doesn't matter.

```
AMethod(age:23,name:"Jem");

//expected output: Hello Jem! You are 23 years old.
```

Named arguments like above are useful when you have multiple parameters with default values, and you only want to pass one, or a few, none default values when you call the method.

Overloading

Method overloading is the act of one method accepting different parameters.

Consider the following methods, both which add numbers, but different data types of numbers.

```
static int IntPlus(int x, int y)

{

  return x + y;

}

static double DoublePlus(double x, double y)

{

  return x + y;

}
```

Instead of having two methods that do the same thing, it's better to overload one.

The below method is overloaded. It is one method called ‘plus’ but it accepts different parameters and returns different data types.

```
static int Plus(int x, int y)

{

  return x + y;

}

static double Plus(double x, double y)

{

  return x + y;

}
```

Objects

C# is an object oriented programming language (OOP), this means that it creates objects that hold both data and methods. Unlike procedural programming that focuses on writing procedures or methods that perform operations on data.

Classes and objects are key aspects of any OOP.

A class is a template for objects, whilst an object is an instance of a class. When an object is created, it inherits all variables and methods of the class.

Take a look at some of the below examples to help put this into perspective.

Class	Object
Animal	Bear
	Dog
	Cat

Class	Object
Fruit	Apple
	Peach
	Banana

Programming objects , like those in the physical world, have their own attributes, characteristics, abilities and behaviours.

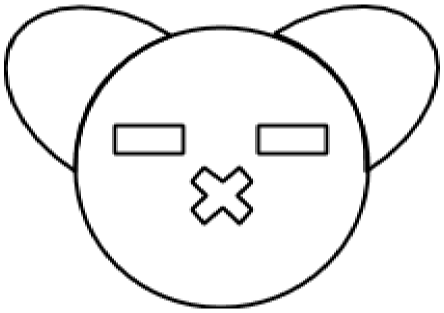
Class	Object	Attributes	Behaviours
Candy	PEZ Dispenser	Color	Dispense
		# Of PEZ Inside	
		Head Figurine	

Classes

In C# every value that a variable can take on is an object, for example 5 and 23 are objects of type int, and “Jem” and “bear” are objects of type string. These are types built into C#. But we can make our own types, with classes.

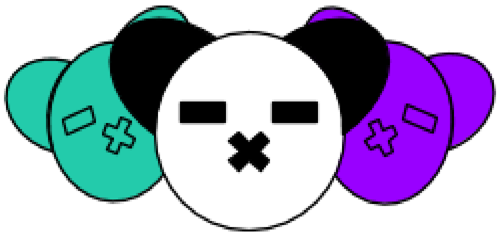
A class is a template for making individual objects of a particular type.

For example, think of a cookie cutter. The classes are the cookie cutter itself whilst objects are the cookies (remember - objects are instances of a class).



This cookie cutter is bear shaped, and so every cookie it produces will be of that type. Each cookie created is a distinct (individual) object. Even though they will all be shaped like bears.

And just because we are using the same cutter, doesn’t mean the objects will all be the same. We could use different ingredients or add coloring. In this instance color, and ingredients would be attributes of our cookie object.



We can create different objects from a class by changing things like the attributes of the class.

Create A Class

To create a class, use the class keyword, followed by the name of the class and a pair of curly braces.

```
class animal
```

```
{
  int eyes = 2;
}
```

The above creates an ‘animal’ class, with an int variable inside of it called eyes. When a variable is declared directly in a class, like above, it is referred to as a field or attribute.

Create A Object

Since an object is created out of a class, we can use the animal class created previously to create a new object.

To create an object of a class, specify the class name, followed by the object name, an assignment operator, and the new keyword, followed then by the class name again with parentheses.

```
class animal
{
  int eyes = 2;
  static void Main(string[] args)
  {
    animal myAnimal = new animal();
    Console.WriteLine(myAnimal.eyes); //2
  }
}
```

The above example creates one new animal object. We could create many by repeating the creation steps.

```
 animal myAnimal2 = new animal();
    Console.WriteLine(myAnimal2.eyes); //2
 animal bear = new animal();
    Console.WriteLine(bear.eyes); //2
```

Fields and methods inside of classes are often referred to as class members. We can access fields through the dot(.) notation, demonstrated above. But we can also leave fields blank, and assign values to them when creating the objects.

```
class animal

{

      string color;

      int eyes;

  static void Main(string[] args)

  {

    animal bear = new animal();

    bear.color = "brown";

    bear.eyes = 2;

    Console.WriteLine(bear.color); //brown

    Console.WriteLine(bear.eyes);  //2

  }

}
```

Methods, which normally belong to classes and define how an object of a class behaves, can also be accessed by the dot(.) notation. However class methods must have the public access modifier instead of static to do so, as a static method can be accessed *without* creating an object of the class, while public methods can *only* be accessed by objects.

```
class animal

{

  string color;                      // field

  int eyes;                          // field

  public void sound(string sound)    // method

  {
    Console.WriteLine($"The sound I make is: {sound}!");

  }}

public class Program

{

   public static void Main(string[] args)

        {

            animal dog = new animal();

            dog.sound("woof");  //The sound I make is: woof!

        }

    }
```

Multiple Classes

We can also create an object of a class and access it in other classes. This practice is often used by programmers for better organization of classes, where one class will have all the fields and methods, whilst another class will hold the Main() method.

```
class animal

{

 public int eyes = 2;

}

public class Program

{

    public static void Main(string[] args)

        {

          animal myAnimal = new animal();
          Console.WriteLine(myAnimal.eyes);

        }

}
```

The `public` keyword is vital here, as it is the access modifier that specifies that the eyes variable(field) of the animal class is accessible for other classes as well, such as program in this instance.

Access Modifiers

An access modifier is used to set the access level (visibility) for classes, fields, methods and properties. The default modifier is private.

Access Modifier	Meaning
public	Code is accessible for all classes
private	Code is only accessible within the existing (same) class
protected	Code is accessible within the same class, or in a class that is inherited from that class
internal	Code is only accessible within its own assembly

Encapsulation

Encapsulation is the process of making sure that "sensitive" data is hidden from users. To encapsulate data we must declare fields(variables) as private, and provide public *get* and *set* methods through properties, to access and update the value of a private field.

Private variables can be accessed through properties, which are a combination of a variable and method.

See the below example, in which the Address **property** is associated with the private address **field**. Note: programmers typically use the same name for both the property and the private field, but with an uppercase first letter for the property.

The **get** method, here, returns the value of the variable address. Whilst the **set** method assigns a value to the address field.

```
class Person
{
  private string address;   // field
  public string Address    // property
  {
    get { return address; }   // get method
    set { address = value; }  // set method
  }
}
```

Below is an example of encapsulation in action.

```
class Person
{
  private string address;   // field

  public string Address    // property
  {
    get { return address; }   // get method
    set { address = value; }  // set method
  }
}

public class Program
{
  public static void Main(string[] args)

  {

    Person Jem = new Person();

    Jem.Address = "23 bear street";

    Console.WriteLine(Jem.Address); //23 bear street

  }

}
```

The encapsulation allows us to change the value of the protected variable whilst still securing the data.

```
    Person Jem = new Person();

    Console.WriteLine(Jem.Address); //blank - as no value assigned yet

    Jem.Address = "23 bear street";

    Console.WriteLine(Jem.Address); //23 bear street

    Jem.Address = "43 cat grove";

    Console.WriteLine(Jem.Address); //43 cat grove
```

Encapsulation also gives better control of class members by reducing the possibility of people messing up the code, by allowing us to make fields read-only - by only using the get method - or write-only - by only using the set method.

Properties Short-Hand

C# also has a short-hand available for creating properties, in that you do not define the field for the property, instead you only have to write get; and set; inside the property.

The below example works the same as the previous.

```
class Person

{

  private string address;   // field
  public string Address    // property
  {
    get; set;
  }
}
public class Program
{
  public static void Main(string[] args)
  {
    Person Jem = new Person();
    Jem.Address = "23 bear street";
    Console.WriteLine(Jem.Address); //23 bear street
  }
}
```

Constructors

A constructor is a special method of instantiating an object. Programmers use constructors because when one is called when an object of a class is created, it can be used to set initial values for fields.

```
using System;
class animal // animal class
{
 public string color;                      // field
 public int eyes;                          // field

 public string animalType;                 // field


  public animal(string animalColor, string animalAnimalType) //animal class
constructor with multiple parameters

  {

    color = animalColor;

    animalType = animalAnimalType;

  }

}

public class Program

{

    public static void Main(string[] args)

    {

      animal dog = new animal("brown", "dog");

      Console.WriteLine(dog.color);      //brown

      Console.WriteLine(dog.animalType); //dog

     }

}
```

The above example constructor assigns the arguments passed to it at creation of the object to the animal class fields.

Ei `new animal("brown", "dog");` is `color = “brown”;      animalType = “dog”;` in the animal class. This is a much cleaner way to give individual objects different values for fields at instantiation.

With Constructor

```
class animal // animal class
{
 public string color;                    // field
 public int eyes;                        // field
 public string animalType;               // field
 public animal(string animalColor, string
animalAnimalType) //animal class constructor
with multiple parameters
  {
    color = animalColor;
    animalType = animalAnimalType;
  }
}
public class Program
{
    public static void Main(string[] args)
    {
      animal dog = new animal("brown", "dog");
      animal cat= new animal("orange", "cat");
      animal bat= new animal("black", "bat");
     }
}
```

Without Constructor

```
class animal // animal class
{
 public string color;                // field
 public int eyes;                    // field
 public string animalType;           // field
}

public class Program
{
    public static void Main(string[] args)
    {
      animal dog = new animal();
      dog.color = "brown";
      dog.animalType = "dog";

      animal cat = new animal();
      dog.color = "orange";
      dog.animalType = "cat";

      animal bat = new animal();
      bat.color = "black";
      bat.animalType = "bat";

     }
}
```

As you can see above, the constructor makes it easier to create objects and assign their fields with values during creation.

A few things to note about constructors. They must have the same name as the class they construct objects for, and they must not contain a return type in their declaration.

Inheritance

C# allows classes to inherit fields and methods from another class where the base (parent) class is the class being inherited from, and the derived (child) class is the class that inherits from another. To inherit from a class the : (colon) symbol is used, with the parent class on the right-hand side.

```
class animal                              // base (parent) class

{

  public string color;                    // parent field

  public void sound(string sound)        // parent method

  {

    Console.WriteLine($"The sound I make is: {sound}!");

  }

}

class dog : animal  // derived (child) class

{

  public string breed;  // child field

}



public class Program

{

 public static void Main(string[] args)

  {
   dog fudge = new dog(); // create new dog class object called fudge

   fudge.color ="brown";     // brown

  }

}
```

To prevent classes from being able to inherit from a class, use the sealed keyword on the class you want to protect.

```
sealed class animal

{

  ...

}

class dog : animal

{

  ... // error will occur as cannot inherit from 'sealed' animal

}
```

Note: if a class constructor is inheriting from another class constructor, you must also satisfy the parameters of the parent class.

Take for example the below constructors for creating points on a map.

```
class Point

    {

        public int X;

        public int Y;

        public Point(int x, int y)

        {

            X = x;
            Y = y;
        }


class MapLocation : Point
    {
        public MapLocation(int x, int y) : base(x, y)
        {}
    }
```

Here, when a map location object is created, a point object is created too. They are not two distinct objects - a map location is also a point.

Think of species family trees (phylum - if you want to get sciency). Like the polar bears.

Kingdom	**Animalia**
Phylum	**Chordata**
Class	**Mammalia**
Order	**Carnivora**
Family	**Ursidae**
Genus	**Ursus**
Species	**Ursus Maritimus**

When a bear is created, a chordata (vertebrate) and an animalia (animal) are also created because a bear is both a vertebrate and an animal. If you continue to think of inheritance this way, also note that the classification below the previous inherits from the above but not the other way around. Ei carnivora inherits from mammalia (going down the hierarchy of inheritance) but mammalia does not inherit from carnivora (going up the hierarchy).

The point constructor requires two parameters (x and y coordinates) to create object, and so when a map location object is created it first needs to call the point class constructor and pass it it's parameters. That's why above, the map location constructor also takes x and y coordinates. To tell it to call the point class constructor and pass the x and y parameters we typed : base followed by a pair of opening and closing parentheses which list the parameters or arguments we want to pass to the base constructor (parent class constructor). So what the above map location constructor is doing is taking x and y parameters, then immediately passing these to the point constructor, where the point constructor then assigns these to the corresponding fields in it's constructor method.

Note: C# doesn't support multiple inheritance, instead you can have a class implement multiple interfaces.

Polymorphism

Polymorphism (poly = many, morphism = forms) occurs when we have many classes that are related to each other by inheritance. Polymorphism uses inherited methods to perform different tasks.

Take for instance our animal example again, where our animal class has a method called sound, which takes the string parameter of sound. It is currently dynamic as it stands:

```
using System;


class animal                             // base (parent) class

  {

      public string color;                   // parent field

      public void sound(string sound)        // parent method

      {

      Console.WriteLine($"The sound I make is: {sound}!");

      }

  }


class dog : animal  // derived (child) class

  {
```

```
  public string breed;  // child field

  }


class bear : animal  // derived (child) class

  {

  public string species;  // child field

  }


public class Program

  {

    public static void Main(string[] args)

    {

      dog fudge = new dog(); // create new dog class object called fudge

      fudge.color ="brown";  // assigns color field the value of brown

      Console.WriteLine(fudge.color); // brown

      fudge.sound("woof"); // assigns parent method sound value of 'woof'

      fudge.sound("bork"); //The sound I make is: bork!


      dog terry = new dog();

      terry.sound("oof"); //The sound I make is: oof!


      bear pudgy = new bear();

      pudgy.sound("grr"); //The sound I make is: grr!

    }
  }
```

But what about for methods that don’t take a parameter, who’s output you do want to change for every subclass? This is where polymorphism comes in, with its keywords virtual and override.

Without specifying ‘virtual’ in the superclass method, and ‘override’ in the child classes methods, the base class method will override the derived class methods that share the same name, and output only the base class output. ei , without using the polymorphism keywords below, all our object outputs would be `All animals make a noise!`’.

```
using System;


class animal                              // base (parent) class

  {

      public string color;                        // parent field

      public virtual void noise()                        // parent method

      {

      Console.WriteLine("All animals make a noise!");

      }

  }


class dog : animal  // derived (child) class

  {

  public string breed;            // child field

  public override void noise()    // child override of parent method

      {

      Console.WriteLine("Dogs go woof!");

      }
  }


class bear : animal  // derived (child) class

  {

  public string species;           // child field

   public override void noise()    // child override of parent method

      {

      Console.WriteLine("Bears go grr!");

      }

  }


public class Program

  {

    public static void Main(string[] args)

    {

      dog fudge = new dog(); // create new dog class object called fudge

      fudge.noise();  //Dogs go woof!


      dog terry = new dog();

      terry.noise(); //Dogs go woof!


      bear pudgy = new bear();

      pudgy.noise(); //Bears go grr!

    }
  }
```

Abstraction

An abstract base class is a base class that defines what it means to be an object of that class, but isn't actually a class to which an object can be created.

Abstraction can be achieved with either abstract classes or interfaces, and consists of abstract classes - which are restricted classes that cannot be used to create objects and to access code within it, it must be inherited from another class- and abstract methods - which can only be used in an abstract class, and do not contain a body, as the body is provided by the derived class.

Continuing on with the previous animal example, we can make an object from the animal class.

```
  animal anAnimalObject = new animal();

  anAnimalObject.noise(); //All animals make a noise!
```

But what if we only wanted the animal class to hold methods, properties and fields that derived classes can assign values to, to create unique objects of their own class, and not have any object to be able to be created from the animal class itself? That's where the abstract keyword comes into play.

See the below example now. There are two important things to note about the example.

1. The ‘abstract' keyword is not valid on fields. Therefore you define fields as private, and create properties to be able to access and or edit them.
2. Abstract methods cannot be marked as ‘virtual', instead we just ensure that the derived class has the ‘overrides' keyword.

```
using System;


abstract class animal                             // base (parent) class

  {

      private string color;                       // parent "abstract" field

      public string Color {get;set;}


      private bool wings;
      public bool Wings {get;set;}


      public abstract void noise();                   // parent abstract method

      public virtual void isAnimal()                  // parent regular method

      {

      Console.WriteLine("I am an animal!");

      }

  }


class dog : animal  // derived (child) class

  {

  public string breed;               // child field

   public override void noise()       // child override of abstract parent method
- child provides body to method

      {

      Console.WriteLine("Dogs go woof!");

      }

  public override void isAnimal()    // child override of parent method

      {

      Console.WriteLine("I am a dog!");

      }

  }


class bat : animal  // derived (child) class
```

```
  {

     public override void noise()    // child override of parent method

      {

      Console.WriteLine("Bats go Sqree!");

      }

  }


public class Program

  {

    public static void Main(string[] args)

    {

      dog fudge = new dog(); // create new dog class object called fudge

      fudge.noise();  //Dogs go woof!


      dog terry = new dog();

      terry.isAnimal(); //I am a dog!


      bat angel = new bat();

      angel.noise(); //Bats go Sqree!

      angel.Wings = true;

      Console.WriteLine(angel.Wings); //true

    }

  }
```

If we now tried to create an object from the animal class we'd throw errors.

```
animal animalObject = new animal();
```

```
Cannot create an instance of the abstract class or interface 'animal'
```

Interfaces

Interfaces are another way to instigate abstraction. An interface is a completely abstract class, in which it can only contain abstract methods and properties. To create an interface, use the interface keyword followed by a name for the interface and a pair of curly braces. The curly braces will contain all the interface methods and properties.

Programmers typically name interfaces with a starting capital I (for interface), as this helps differentiate between abstract classes, interfaces and classes.

```
interface Ianimal

{

    string Color {get;set;}

    bool Wings {get;set;}

    void noise();

    void isAnimal();

}
```

See our previous abstract class example below, now changed into an interface.

There are a few important things to note about the example.

1. Interfaces cannot contain fields.
2. By default, members of an interface are abstract and public (notice we do not write these in the example, as it's not necessary to).
3. Methods must have empty bodies in an interface definition.
4. You do not use the override keyword when implementing an interface method.
5. On implementation of an interface, you must override all of its methods.
6. An interface cannot contain a constructor.

```
using System;
```

```
interface Ianimal

  {

      string Color {get;set;}

      void noise();

      void isAnimal();

  }


class dog : Ianimal

  {

    private string _color;

    public string Color

    {

        get => _color;

        set => _color = value;

    }


    public void noise()

      {

      Console.WriteLine("Dogs go woof!");

      }

    public void isAnimal()

      {

      Console.WriteLine("I am a dog!");

      }
  }


public class Program

  {

    public static void Main(string[] args)

    {

      dog fudge = new dog(); // create new dog class object called fudge

      fudge.noise();  //Dogs go woof!

      fudge.isAnimal(); //I am a dog!

      fudge.Color = "brown";

      Console.WriteLine(fudge.Color); // brown

    }

  }
```

Note: we named the private field in dog _color, as it is typical naming convention to name private fields with an underscore as the first character.

Multiple Interfaces

To implement multiple interfaces, separate them with a comma.

```
class Example : IExampleOfConvention, IExampleOfMultiInterface{

   //

  }
```

Namespaces

Namespaces are groupings of similar classes. They provide a "named space" in which the application resides. The purpose of namespaces is to prevent conflict in classes names. For example, without namespaces you wouldn't be able to make a class named Console, as the .NET framework already uses one in its pre-existing System namespace.

Here's another example:

```
namespace anExample {

  class System {}

  class program{

    static void Main(string[] args){

      System.Console.WriteLine("Welcome to C#!");

    }

  }

}
```

The above outputs the error `'System' does not contain a definition for 'Console'` this is because C# doesn't look at the global system namespace that exists in the .NET framework as it's now looking at our system class instead. And our system class indeed doesn't contain a definition for console. One way to mitigate this issue is to tell the interpreter to look at the global scope rather than our one here. We do this by adding the keyword global followed by two colons.

```
global::System.Console.WriteLine("Welcome to C#!");
```

Note: it's not recommended to create your own name spaces that use pre-existing names in the .NET framework, as it can lead to confusion and potential errors like above.

Another example of class name conflict mitigation with namespaces:

```
namespace anExample {

  class program{

    static void Main(string[] args){

      System.Console.WriteLine("Welcome to C#!");

    }

  }
}
namespace myNamespace{
  class program {
   //code
  }
}
```

The class 'program' in myNamespace doesn't conflict with the class 'program' in the anExample namespace because they exist in different scopes. If we wanted to reference the class 'program' in myNamespace, we simply call it by typing the namespace name dot(.) the name of the class.

```
using System;
namespace anExample
{
   public class program
    {
        public void hello()
       {
            Console.WriteLine("Welcome to C#!");
        }
    }

    namespace myNamespace // myNamespace is now a nested namespace within
anExample
    {
       public class program
        {
            public void greeting()

            {

              Console.WriteLine("I am also in a class called program.");

            }

        }

    }


   public class scopeExample

    {

       public static void Main(string[] args)

        {

            // creates an object of the anExample program class

            anExample.program outer = new anExample.program();

            outer.hello(); //Welcome to C#!


            // creates an object of the nested myNamespace program class

            myNamespace.program inner1 = new myNamespace.program();

            inner1.greeting(); //I am also in a class called program.

        }

    }

  }
```

Namespaces are also useful because writing the name of a namespace over and over prepended onto the type name can clutter code, C# offers a shortcut with the using directive.

Take the prerequisite C# example for instance, if we were to remove the ‘using System’ statement, then for every ‘Console.WriteLine’ statement we would have to prepend ‘system.’ to it, and our code would quickly become hard to read, especially with more complex code.

```
var Fruit = "apple";

var FRUIT = "apple";

var fruit = "apple";

System.Console.WriteLine(FRUIT);

System.Console.WriteLine(Fruit);

System.Console.WriteLine(fruit);
```

```
using System;

namespace fruits

{

 ...

      var FRUIT = "apple";

      ...

      Console.WriteLine(FRUIT);

   } }}
```

Create A Namespace

You can create your own namespaces by declaring the namespace keyword, followed by the name of your namespace and a pair of curly braces. Within these curly braces, you place your code, ei classes, fields, ect.

```
namespace myNamespace {}
```

To create nested namespaces you can use on of the two methods shown below.

```
 namespace anExample
```

```
{

  namespace myNamespace

  {

    //code

  }

}
```

```
 namespace anExample.myNamespace

{

    //code

}
```

Aliases

You can also create aliases for namespaces with the using directive in C#. to give a namespace an alias use the using keyword followed by the alias, then the assignment operator and the namespace the alias is for. To access the member of the alias, use double colons (::).

```
using sy = System;


namespace anExample

{

   public class program

    {

         public void hello()

    {

             sy::Console.WriteLine("Welcome to C#!");

        }
}
```

Naming Conventions

You may notice some examples of our C# book do not follow the typical naming conventions of C#. This is fine. Some software development companies have their own documentation on how they want projects formatted and named, and so it is not uncommon to see a variety of naming conventions. However, we will briefly explore below the most common naming conventions for C# for you to use at your own discretion. The below is not by any means an exhaustive list.

Example	C# Fundamental	Convention
`public class ExampleOfConvention {` `//` `}`	Classes	Title (Pascal) Case
`public static void ExampleOfConvention() {` `//` `}`	Methods	Title (Pascal) Case
`string exampleOfConvention;` `bool exampleOfCamelCase;`	Variables & Parameters	Camel Case
`interface IExampleOfConvention{` `//` `}`	Interfaces	Prefix With Capital I And Then Title Case
`public bool isExample;`	Boolean Values	Prefix Name With Is
`private bool _isExample;`	Private Values	Prefix Name With Underscore