C# Tutorial

C# (C-Sharp) is a programming language developed by Microsoft that runs on the .NET Framework.

C# is used to develop web apps, desktop apps, mobile apps, games and much more.

Examples in Each Chapter

Our "Try it Yourself" tool makes it easy to learn C#. You can edit C# code and view the result in your browser.

Example

```
using System;

namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

Click on the "Run example" button to see how it works.

We recommend reading this tutorial, in the sequence listed in the left menu.

C# Exercises

Learn by Examples

Learn by examples! This tutorial supplements all explanations with clarifying examples.

See All C# Examples

C# Quiz

Learn by taking a quiz! The quiz will give you a signal of how much you know, or do not know, about C#.

Start C# Quiz

C# Introduction

What is C#?

C# is pronounced "C-Sharp".

It is an object-oriented programming language created by Microsoft that runs on the .NET Framework.

C# has roots from the C family, and the language is close to other popular languages likeC++ and Java.

The first version was released in year 2002. The latest version,**C# 8**, was released in September 2019.

C# is used for:

- Mobile applications
- Desktop applications

- Web applications
- Web services
- Web sites
- Games
- VR
- Database applications
- And much, much more!

Why Use C#?

- It is one of the most popular programming language in the world
- It is easy to learn and simple to use
- It has a huge community support
- C# is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs.
- As C# is close to C, C++ and Java, it makes it easy for programmers to switch to C# or vice versa

---

Get Started

This tutorial will teach you the basics of C#.

It is not necessary to have any prior programming experience.

Get Started Â»

C# Get Started

C# IDE

The easiest way to get started with C#, is to use an IDE.

An IDE (Integrated Development Environment) is used to edit and compile code.

In our tutorial, we will use Visual Studio Community, which is free to download from https://visualstudio.microsoft.com/vs/community/.

Applications written in C# use the .NET Framework, so it makes sense to use Visual Studio, as the program, the framework, and the language, are all created by Microsoft.

C# Install

Once the Visual Studio Installer is downloaded and installed, choose the .NET workload and click on the **Modify**/**Install** button:

After the installation is complete, click on the **Launch** button to get started with Visual Studio.

On the start window, choose **Create a new project**:

Then click on the "Install more tools and features" button:

Choose "Console App (.NET Core)" from the list and click on the Next button:

Enter a name for your project, and click on the Create button:

Visual Studio will automatically generate some code for your project:

The code should look something like this:

Program.cs

```
using System;

namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

Don't worry if you don't understand the code above - we will discuss it in detail in later chapters. For now, focus on how to run the code.

Run the program by pressing the **F5** button on your keyboard (or click on "**Debug**" -> "**Start Debugging**"). This will compile and execute your code. The result will look something to this:

```
Hello World!
C:\Users\Username\source\repos\HelloWorld\HelloWorld\bin\Debug\netcoreapp3.0\HelloWorld.exe (process 13784) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Congratulations! You have now written and executed your first C# program.

Learning C# At W3Schools

When learning C# at W3Schools.com, you can use our "Try it Yourself" tool, which shows both the code and the result. This will make it easier for you to understand every part as we move forward:

Program.cs

```
using System;

namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

Result:

```
Hello World!
```

C# Syntax

C# Syntax

In the previous chapter, we created a C# file called Program.cs, and we used the following code to print "Hello World" to the screen:

Program.cs

```
using System;
```

```
namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

Result:

```
Hello World!
```

Example explained

Line 1: `using System` means that we can use classes from the `System` namespace.

Line 2: A blank line. C# ignores white space. However, multiple lines makes the code more readable.

Line 3: `namespace` is a used to organize your code, and it is a container for classes and other namespaces.

Line 4: The curly braces `{}` marks the beginning and the end of a block of code.

Line 5: `class` is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.

Don't worry if you don't understand how `using System`, `namespace` and `class` works. Just think of it as something that (almost) always appears in your program, and that you will learn more about them in a later chapter.

Line 7: Another thing that always appear in a C# program, is the `Main` method. Any code inside its curly brackets `{}` will be executed. You don't have to understand the keywords before and after Main. You will get to know them bit by bit while reading this tutorial.

Line 9: `Console` is a class of the `System` namespace, which has a `WriteLine()` method that is used to output/print text. In our example it will output "Hello World!".

If you omit the `using System` line, you would have to write `System.Console.WriteLine()` to print/output text.

Note: Every C# statement ends with a semicolon `;`.

Note: C# is case-sensitive: "MyClass" and "myclass" has different meaning.

Note: Unlike Java, the name of the C# file does not have to match the class name, but they often do (for better organization). When saving the file, save it using a proper name and add ".cs" to the end of the filename. To run the example above on your computer, make sure that C# is properly installed: Go to the Get Started Chapter for how to install C#. The output should be:

```
Hello World!
```

WriteLine or Write

The most common method to output something in C# is `WriteLine()`, but you can also use `Write()`.

The difference is that `WriteLine()` prints the output on a new line each time, while `Write()` prints on the same line (note that you should remember to add spaces when needed, for better readability):

Example

```
Console.WriteLine("Hello World!");
Console.WriteLine("I will print on a new line.");

Console.Write("Hello World! ");
Console.Write("I will print on the same line.");
```

Result:

```
Hello World!
I will print on a new line.
Hello World! I will print on the same line.
```

In this tutorial, we will only use `WriteLine()` as it makes it easier to read the output of code.

C# Exercises

C# Comments

C# Comments

Comments can be used to explain C# code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line Comments

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is ignored by C# (will not be executed).

This example uses a single-line comment before a line of code:

Example

```
// This is a comment
Console.WriteLine("Hello World!");
```

This example uses a single-line comment at the end of a line of code:

Example

```
Console.WriteLine("Hello World!");  // This is a comment
```

C# Multi-line Comments

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be ignored by C#.

This example uses a multi-line comment (a comment block) to explain the code:

Example

```
/* The code below will print the words Hello World
to the screen, and it is amazing */
Console.WriteLine("Hello World!");
```

Single or multi-line comments?

It is up to you which you want to use. Normally, we use// for short comments, and /* */ for longer.

C# Exercises

C# Variables

C# Variables

Variables are containers for storing data values.

In C#, there are different **types** of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, you must specify the type and assign it a value:

Syntax

type variableName = value;

Where *type* is a C# type (such as int or string), and *variableName* is the name of the variable (such as **x** or **name**). The **equal sign** is used to assign values to the variable.

To create a variable that should store text, look at the following example:

Example

Create a variable called **name** of type string and assign it the value "**John**":

```
string name = "John";
Console.WriteLine(name);
```

To create a variable that should store a number, look at the following example:

Example

Create a variable called **myNum** of type int and assign it the value **15**:

```
int myNum = 15;
Console.WriteLine(myNum);
```

You can also declare a variable without assigning the value, and assign the value later:

Example

```
int myNum;
myNum = 15;
Console.WriteLine(myNum);
```

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

Example

Change the value of myNum to 20:

```
int myNum = 15;
myNum = 20; // myNum is now 20
Console.WriteLine(myNum);
```

Constants

However, you can add the const keyword if you don't want others (or yourself) to overwrite existing values (this will declare the variable as "constant", which means unchangeable and read-only):

Example

```
const int myNum = 15;
myNum = 20; // error
```

The const keyword is useful when you want a variable to always store the same value, so that others (or yourself) won't mess up your code. An example that is often referred to as a constant, is PI (3.14159...).

Note: You cannot declare a constant variable without assigning the value. If you do, an error will occur: A const field requires a value to be provided.

Other Types

A demonstration of how to declare variables of other types:

Example

```
int myNum = 5;
double myDoubleNum = 5.99D;
char myLetter = 'D';
bool myBool = true;
string myText = "Hello";
```

You will learn more about data types in the next chapter.

Display Variables

The WriteLine() method is often used to display variable values to the console window.

To combine both text and a variable, use the + character:

Example

```
string name = "John";
Console.WriteLine("Hello " + name);
```

You can also use the + character to add a variable to another variable:

Example

```
string firstName = "John ";
string lastName = "Doe";
string fullName = firstName + lastName;
Console.WriteLine(fullName);
```

For numeric values, the + character works as a mathematical operator (notice that we use int (integer) variables here):

Example

```
int x = 5;
int y = 6;
Console.WriteLine(x + y); // Print the value of x + y
```

From the example above, you can expect:

- x stores the value 5
- y stores the value 6
- Then we use the WriteLine() method to display the value of x + y, which is **11**

Declare Many Variables

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

Example

```
int x = 5, y = 6, z = 50;
Console.WriteLine(x + y + z);
```

C# Identifiers

All C# **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code:

Example

```
// Good
int minutesPerHour = 60;

// OK, but not so easy to understand what m actually is
int m = 60;
```

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits and the underscore character (_)
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names are case sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like C# keywords, such as `int` or `double`) cannot be used as names

---

C# Exercises

C# Data Types

C# Data Types

As explained in the variables chapter, a variable in C# must be a specified data type:

Example

```
int myNum = 5;Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â  // Integer (whole number)
double myDoubleNum = 5.99D;Â  // Floating point number
char myLetter = 'D';Â Â Â Â Â Â Â Â Â Â  // Character
bool myBool = true;Â Â Â Â Â Â Â Â Â Â Â  // Boolean
string myText = "Hello";Â Â Â Â Â  // String
```

A data type specifies the size and type of variable values. It is important to use the correct data type for the corresponding variable; to avoid errors, to save time and memory, but it will also make your code more maintainable and readable. The most common data types are:

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
bool	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes per character	Stores a sequence of characters, surrounded by double quotes

Numbers

Number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `int` and `long`. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. Valid types are `float` and `double`.

Even though there are many numeric types in C#, the most used for numbers are `int` (for whole numbers) and `double` (for floating point numbers). However, we will describe them all as you continue to read.

Integer Types

Int

The int data type can store whole numbers from -2147483648 to 2147483647.In general, and in our tutorial, the int data type is the preferred data type when we create variables with a numeric value.

Example

```
int myNum = 100000;
Console.WriteLine(myNum);
```

Long

The long data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

Example

```
long myNum = 15000000000L;
Console.WriteLine(myNum);
```

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

Float

The float data type can store fractional numbers from 3.4eâˆ’038 to 3.4e+038. Note that you should end the value with an "F":

Example

```
float myNum = 5.75F;
Console.WriteLine(myNum);
```

Double

The double data type can store fractional numbers from 1.7eâˆ’308 to 1.7e+308. Note that you can end the value with a "D" (although not required):

Example

```
double myNum = 19.99D;
Console.WriteLine(myNum);
```

Use float or double?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of float is only six or seven decimal digits, while double variables have a precision of about 15 digits. Therefore it is safer to use double for most calculations.

Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

Example

```
float f1 = 35e3F;
double d1 = 12E4D;
Console.WriteLine(f1);
Console.WriteLine(d1);
```

Booleans

A boolean data type is declared with the bool keyword and can only take the values true or false:

Example

```
bool isCSharpFun = true;
```

```
bool isFishTasty = false;
Console.WriteLine(isCSharpFun);Â Â Â // Outputs True
Console.WriteLine(isFishTasty);Â Â  // Outputs False
```

Boolean values are mostly used for conditional testing, which you will learn more about in a later chapter.

Characters

The char data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

Example

```
char myGrade = 'B';
Console.WriteLine(myGrade);
```

Strings

The string data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
string greeting = "Hello World";
Console.WriteLine(greeting);
```

C# Exercises

C# Type Casting

C# Type Casting

Type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

- **Implicit Casting** (automatically) - converting a smaller type to a larger type size
  char -> int -> long -> float -> double
- **Explicit Casting** (manually) - converting a larger type to a smaller size type
  double -> float -> long -> int -> char

Implicit Casting

Implicit casting is done automatically when passing a smaller size type to a larger size type:

Example

```
int myInt = 9;
double myDouble = myInt;       // Automatic casting: int to double

Console.WriteLine(myInt);Â Â Â Â Â  // Outputs 9
Console.WriteLine(myDouble);Â Â  // Outputs 9
```

Explicit Casting

Explicit casting must be done manually by placing the type in parentheses in front of the value:

Example

```
double myDouble = 9.78;
int myInt = (int) myDouble;Â Â Â Â // Manual casting: double to int

Console.WriteLine(myDouble);Â Â Â // Outputs 9.78
Console.WriteLine(myInt);Â Â     // Outputs 9
```

Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such as `Convert.ToBoolean`, `Convert.ToDouble`, `Convert.ToString`, `Convert.ToInt32` (`int`) and `Convert.ToInt64` (`long`):

Example

```
int myInt = 10;
double myDouble = 5.25;
bool myBool = true;

Console.WriteLine(Convert.ToString(myInt));    // convert int to string
Console.WriteLine(Convert.ToDouble(myInt));    // convert int to double
Console.WriteLine(Convert.ToInt32(myDouble));  // convert double to int
Console.WriteLine(Convert.ToString(myBool));   // convert bool to string
```

Why Conversion?

Many times, there's no need for type conversion. But sometimes you have to. Take a look at the next chapter, when working with user input, to see an example of this.

C# User Input

Get User Input

You have already learned that `Console.WriteLine()` is used to output (print) values. Now we will use `Console.ReadLine()` to get user input.

In the following example, the user can input his or hers username, which is stored in the variable `userName`. Then we print the value of `userName`:

Example

```
// Type your username and press enter
Console.WriteLine("Enter username:");

// Create a string variable and get user input from the keyboard and store it in the variable
string userName = Console.ReadLine();

// Print the value of the variable (userName), which will display the input value
Console.WriteLine("Username is: " + userName);
```

User Input and Numbers

The `Console.ReadLine()` method returns a `string`. Therefore, you cannot get information from another data type, such as `int`. The following program will cause an error:

Example

```
Console.WriteLine("Enter your age:");
int age = Console.ReadLine();
Console.WriteLine("Your age is: " + age);
```

The error message will be something like this:

```
Cannot implicitly convert type 'string' to 'int'
```

Like the error message says, you cannot implicitly convert type 'string' to 'int'.

Luckily, for you, you just learned from the previous chapter (Type Casting), that you can convert any type explicitly, by using one of the Convert.To methods:

Example

```
Console.WriteLine("Enter your age:");
int age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Your age is: " + age);
```

Note: If you enter wrong input (e.g. text in a numerical input), you will get an exception/error message (like System.FormatException: 'Input string was not in a correct format.').

You will learn more about Exceptions and how to handle errors in a later chapter.

C# Exercises

C# Operators

C# Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + **operator** to add together two values:

Example

```
int x = 100 + 50;
```

Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

Example

```
int sum1 = 100 + 50;Â Â Â Â Â Â Â  // 150 (100 + 50)
int sum2 = sum1 + 250;Â Â Â Â Â Â  // 400 (150 + 250)
int sum3 = sum2 + sum2;Â Â Â Â  // 800 (400 + 400)
```

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations:

Operator	Name	Description	Example	Try it
+	Addition	Adds together two values	x + y	
-	Subtraction	Subtracts one value from another	x - y	
*	Multiplication	Multiplies two values	x * y	
/	Division	Divides one value by another	x / y	
%	Modulus	Returns the division remainder	x % y	
++	Increment	Increases the value of a variable by 1	x++	
--	Decrement	Decreases the value of a variable by 1	x--	

C# Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Example

```
int x = 10;
```

The **addition assignment** operator (+=) adds a value to a variable:

Example

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

A list of all assignment operators:

Operator	Example	Same As	Try it
=	x = 5	x = 5	
+=	x += 3	x = x + 3	
-=	x -= 3	x = x - 3	
*=	x *= 3	x = x * 3	
/=	x /= 3	x = x / 3	
%=	x %= 3	x = x % 3	
&=	x &= 3	x = x & 3	
|=	x |= 3	x = x | 3	
^=	x ^= 3	x = x ^ 3	
>>=	x >>= 3	x = x >> 3	
<<=	x <<= 3	x = x << 3	

C# Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example	Try it
==	Equal to	x == y	
!=	Not equal	x != y	
>	Greater than	x > y	
<	Less than	x < y	
>=	Greater than or equal to	x >= y	
<=	Less than or equal to	x <= y	

C# Logical Operators

Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example	Try it
&&Â	Logical and	Returns true if both statements are true	x < 5 &&Â x < 10	
||Â	Logical or	Returns true if one of the statements is true	x < 5 || x < 4	
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)	

You will learn more about comparison and logical operators in theBooleans and If...Else chapters.

C# Exercises

C# Math

The C# Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.Max(*x,y*)

The Math.Max(*x*,*y*) method can be used to find the highest value of*x* and *y*:

Example

```
Math.Max(5, 10);
```

Math.Min(*x,y*)

The Math.Min(*x,y*) method can be used to find the lowest value of of*x* and *y*:

Example

```
Math.Min(5, 10);
```

Math.Sqrt(*x*)

The Math.Sqrt(*x*) method returns the square root of *x*:

Example

```
Math.Sqrt(64);
```

Math.Abs(*x*)

The Math.Abs(*x*) method returns the absolute (positive) value of*x*:

Example

```
Math.Abs(-4.7);
```

Math.Round()

Math.Round() rounds a number to the nearest whole number:

Example

```
Math.Round(9.99);
```

C# Exercises

C# Strings

C# Strings

Strings are used for storing text.

A string variable contains a collection of characters surrounded by double quotes:

Example

Create a variable of type string and assign it a value:

```
string greeting = "Hello";
```

String Length

A string in C# is actually an object, which contain properties and methods that can perform certain operations on strings. For example, the length of a string can be found with the Length property:

Example

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
Console.WriteLine("The length of the txt string is: " + txt.Length);
```

Other Methods

There are many string methods available, for example `ToUpper()` and `ToLower()`, which returns a copy of the string converted to uppercase or lowercase:

Example

```
string txt = "Hello World";
Console.WriteLine(txt.ToUpper());Â Â  // Outputs "HELLO WORLD"
Console.WriteLine(txt.ToLower());Â Â  // Outputs "hello world"
```

String Concatenation

The `+` operator can be used between strings to combine them. This is called **concatenation**:

Example

```
string firstName = "John ";
string lastName = "Doe";
string name = firstName + lastName;
Console.WriteLine(name);
```

Note that we have added a space after "John" to create a space between firstName and lastName on print.

You can also use the `string.Concat()` method to concatenate two strings:

Example

```
string firstName = "John ";
string lastName = "Doe";
string name = string.Concat(firstName, lastName);
Console.WriteLine(name);
```

String Interpolation

Another option of string concatenation, is **string interpolation**, which substitutes values of variables into placeholders in a string. Note that you do not have to worry about spaces, like with concatenation:

Example

```
string firstName = "John";
string lastName = "Doe";
string name = $"My full name is: {firstName} {lastName}";
Console.WriteLine(name);
```

Also note that you have to use the dollar sign ($) when using the string interpolation method.

String interpolation was introduced in C# version 6.

Access Strings

You can access the characters in a string by referring to its index number inside square brackets `[]`.

This example prints the **first character** in **myString**:

Example

```
string myString = "Hello";
Console.WriteLine(myString[0]);Â  // Outputs "H"
```

Note: String indexes start with 0: [0] is the first character. [1] is the second character, etc.

This example prints the **second character** (1) in **myString**:

Example

```
string myString = "Hello";
Console.WriteLine(myString[1]);Â  // Outputs "e"
```

You can also find the index position of a specific character in a string, by using the `IndexOf()` method:

Example

```
string myString = "Hello";
Console.WriteLine(myString.IndexOf("e"));Â  // Outputs "1"
```

Another useful method is `Substring()`, which extracts the characters from a string, starting from the specified character position/index, and returns a new string. This method is often used together with `IndexOf()` to get the specific character position:

Example

```
// Full name
string name = "John Doe";

// Location of the letter D
int charPos = name.IndexOf("D");

// Get last name
string lastName = name.Substring(charPos);

// Print the result
Console.WriteLine(lastName);
```

Special Characters

Because strings must be written within quotes, C# will misunderstand this string, and generate an error:

```
string txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

The sequence `\"Â ` inserts a double quote in a string:

Example

```
string txt = "We are the so-called \"Vikings\" from the north.";
```

The sequence `\'Â ` inserts a single quote in a string:

Example

```
string txt = "It\'s alright.";
```

The sequence `\\Â ` inserts a single backslash in a string:

Example

```
string txt = "The character \\ is called backslash.";
```

Other useful escape characters in C# are:

Code	Result	Try it
\n	New Line	
\t	Tab	
\b	Backspace	

Adding Numbers and Strings

WARNING!

C# uses the + operator for both addition and concatenation.

Remember: Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

Example

```
int x = 10;
int y = 20;
int z = x + y;Â Â // z will be 30 (an integer/number)
```

If you add two strings, the result will be a string concatenation:

Example

```
string x = "10";
string y = "20";
string z = x + y;Â  // z will be 1020 (a string)
```

C# Exercises

C# Booleans

C# Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- TRUE / FALSE

For this, C# has a `bool` data type, which can take the values `true` or `false`.

Boolean Values

A boolean type is declared with the `bool` keyword and can only take the values `true` or `false`:

Example

```
bool isCSharpFun = true;
bool isFishTasty = false;
Console.WriteLine(isCSharpFun);Â Â Â // Outputs True
Console.WriteLine(isFishTasty);Â Â  // Outputs False
```

However, it is more common to return boolean values from boolean expressions, for conditional testing (see below).

Boolean Expression

A **Boolean expression** is a C# expression that returns a Boolean value: `True` or `False`.

You can use a comparison operator, such as the **greater than** (`>`) operator to find out if an expression (or a variable) is true:

Example

```
int x = 10;
```

```
int y = 9;
Console.WriteLine(x > y); // returns True, because 10 is higher than 9
```

Or even easier:

Example

```
Console.WriteLine(10 > 9); // returns True, because 10 is higher than 9
```

In the examples below, we use the**equal to** (==) operator to evaluate an expression:

Example

```
int x = 10;
Console.WriteLine(x == 10); // returns True, because the value of x is equal to 10
```

Example

```
Console.WriteLine(10 == 15); // returns False, because 10 is not equal to 15
```

The boolean value of an expression is the basis for all C# comparisons and conditions.

You will learn more about conditions in the next chapter.

C# Exercises

C# If ... Else

C# Conditions and If Statements

C# supports the usual logical conditions from mathematics:

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to a == b
- Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

C# has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

---

The if Statement

Use the `if` statement to specify a block of C# code to be executed if a condition is `True`.

Syntax

```
if (condition)
{
Â  // block of code to be executed if the condition is True
}
```

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is `True`, print some text:

Example

```
if (20 > 18)
{
Â  Console.WriteLine("20 is greater than 18");
}
```

We can also test variables:

Example

```
int x = 20;
int y = 18;
if (x > y)
{
Â  Console.WriteLine("x is greater than y");
}
```

Example explained

In the example above we use two variables,**x** and **y**, to test whether x is greater than y (using the > operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

The else Statement

Use the else statement to specify a block of code to be executed if the condition is False.

Syntax

```
if (condition)
{
Â  // block of code to be executed if the condition is True
}
else
{
Â  // block of code to be executed if the condition is False
}
```

Example

```
int time = 20;
if (time < 18)
{
Â  Console.WriteLine("Good day.");
}
else
{
Â  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

Example explained

In the example above, time (20) is greater than 18, so the condition is False. Because of this, we move on to the else condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

The else if Statement

Use the else if statement to specify a new condition if the first condition is False.

Syntax

```
if (condition1)
{
Â  // block of code to be executed if condition1 is True
}
else if (condition2)
{
Â  // block of code to be executed if the condition1 is false and condition2 is True
}
else
```

```
{
Â  // block of code to be executed if the condition1 is false and condition2 is False
}
```

Example

```
int time = 22;
if (time < 10)
{
Â  Console.WriteLine("Good morning.");
}
else if (time < 20)
{
Â  Console.WriteLine("Good day.");
}
else
{
Â  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

Example explained

In the example above, time (22) is greater than 10, so the**first condition** is False. The next condition, in the else if statement, is also False, so we move on to theelse condition since **condition1** and **condition2** is both False - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the**ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

Syntax

```
variable = (condition) ? expressionTrue :  expressionFalse;
```

Instead of writing:

Example

```
int time = 20;
if (time < 18)
{
Â  Console.WriteLine("Good day.");
}
else
{
Â  Console.WriteLine("Good evening.");
}
```

You can simply write:

Example

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
Console.WriteLine(result);
```

C# Exercises

C# Switch

C# Switch Statements

Use the switch statement to select one of many code blocks to be executed.

Syntax

```
switch(expression)
{
Â  case x:
Â Â Â  // code block
Â Â Â  break;
Â  case y:
Â Â Â  // code block
Â Â Â  break;
Â  default:
Â Â Â  // code block
Â Â Â  break;
}
```

This is how it works:

- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The break and default keywords will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

Example

```
int day = 4;
switch (day)
{
Â  case 1:
Â Â Â  Console.WriteLine("Monday");
Â Â Â  break;
Â  case 2:
Â Â Â  Console.WriteLine("Tuesday");
Â Â Â  break;
Â  case 3:
Â Â Â  Console.WriteLine("Wednesday");
Â Â Â  break;
Â  case 4:
Â Â Â  Console.WriteLine("Thursday");
Â Â Â  break;
Â  case 5:
Â Â Â  Console.WriteLine("Friday");
Â Â Â  break;
Â  case 6:
Â Â Â  Console.WriteLine("Saturday");
Â Â Â  break;
Â  case 7:
Â Â Â  Console.WriteLine("Sunday");
Â Â Â  break;
}
// Outputs "Thursday" (day 4)
```

The break Keyword

When C# reaches a break keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

The default Keyword

The default keyword is optional and specifies some code to run if there is no case match:

Example

```
int day = 4;
switch (day)
```

```
{
Â  case 6:
Â Â Â  Console.WriteLine("Today is Saturday.");
Â Â Â  break;
Â  case 7:
Â Â Â  Console.WriteLine("Today is Sunday.");
Â Â Â  break;
Â  default:
Â Â Â  Console.WriteLine("Looking forward to the Weekend.");
Â Â Â  break;
}
// Outputs "Looking forward to the Weekend."
```

C# Exercises

C# While Loop

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

C# While Loop

The `while` loop loops through a block of code as long as a specified condition is `True`:

Syntax

```
while (condition)
{
Â  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0;
while (i < 5)
{
  Console.WriteLine(i);
  i++;
}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do
{
Â  // code block to be executed
}
while (condition);
```

The example below uses a `do/while` loop. 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:

Example

```
int i = 0;
do
{
Â Â Console.WriteLine(i);
Â Â i++;
}
while (i < 5);
```

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

C# Exercises

C# For Loop

C# For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

Syntax

```
for (statement 1; statement 2; statement 3)
{
Â  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example

```
for (int i = 0; i < 5; i++)
{
Â Â Console.WriteLine(i);
}
```

Example explained

Statement 1 sets a variable before the loop starts (`int i = 0`).

Statement 2 defines the condition for the loop to run (`i` must be less than `5`). If the condition is `true`, the loop will start over again, if it is `false`, the loop will end.

Statement 3 increases a value (`i++`) each time the code block in the loop has been executed.

Another Example

This example will only print even values between 0 and 10:

Example

```
for (int i = 0; i <= 10; i = i + 2)
{
Â Â Console.WriteLine(i);
}
```

The foreach Loop

There is also a `foreach` loop, which is used exclusively to loop through elements in an**array**:

Syntax

```
foreach (type variableName in arrayName)
{
Â  // code block to be executed
}
```

The following example outputs all elements in the**cars** array, using a foreach loop:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars)
{
Â  Console.WriteLine(i);
}
```

Note: Don't worry if you don't understand the example above. You will learn more about Arrays in the[C# Arrays chapter](#).

C# Exercises

C# Break and Continue

C# Break

You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch statement.

The break statement can also be used to jump out of a **loop**.

This example jumps out of the loop wheni is equal to 4:

Example

```
for (int i = 0; i < 10; i++)
{
Â  if (i == 4)
  {
Â Â Â  break;
  }
Â  Console.WriteLine(i);
}
```

C# Continue

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of4:

Example

```
for (int i = 0; i < 10; i++)
{
Â  if (i == 4)
  {
Â Â Â  continue;
  }
Â  Console.WriteLine(i);
}
```

Break and Continue in While Loop

You can also use break and continue in while loops:

Break Example

```
int i = 0;
while (i < 10)
{
Â Â Console.WriteLine(i);
Â Â i++;
Â Â if (i == 4)
  {
Â Â Â Â break;
  }
}
```

Continue Example

```
int i = 0;
while (i < 10)
{
Â Â if (i == 4)
  {
Â Â Â  i++;
Â Â Â Â continue;
Â Â }
Â Â Console.WriteLine(i);
Â Â i++;
}
```

C# Arrays

Create an Array

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with**square brackets**:

```
string[] cars;
```

We have now declared a variable that holds an array of strings.

To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in**cars**:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
Console.WriteLine(cars[0]);
// Outputs Volvo
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
Console.WriteLine(cars[0]);
// Now outputs Opel instead of Volvo
```

Array Length

To find out how many elements an array has, use the `Length` property:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
Console.WriteLine(cars.Length);
// Outputs 4
```

Loop Through an Array

You can loop through the array elements with the `for` loop, and use the `Length` property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.Length; i++)
{
Â  Console.WriteLine(cars[i]);
}
```

The foreach Loop

There is also a `foreach` loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
foreach (type variableName in arrayName)
{
Â  // code block to be executed
}
```

The following example outputs all elements in the **cars** array, using a `foreach` loop:

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars)
{
Â  Console.WriteLine(i);
}
```

The example above can be read like this: **for each** `string` element (called **i** - as in **i**ndex) in **cars**, print out the value of **i**.

If you compare the `for` loop and `foreach` loop, you will see that the `foreach` method is easier to write, it does not require a counter (using the `Length` property), and it is more readable.

Sort Arrays

There are many array methods available, for example `Sort()`, which sorts an array alphabetically or in an ascending order:

Example

```
// Sort a string
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
Array.Sort(cars);
foreach (string i in cars)
{
  Console.WriteLine(i);
}

// Sort an int
int[] myNumbers = {5, 1, 8, 9};
Array.Sort(myNumbers);
foreach (int i in myNumbers)
{
  Console.WriteLine(i);
}
```

System.Linq Namespace

Other useful array methods, such as `Min`, `Max`, and `Sum`, can be found in the `System.Linq` namespace:

Example

```
using System;
using System.Linq;

namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] myNumbers = {5, 1, 8, 9};
      Console.WriteLine(myNumbers.Max());  // returns the largest value
      Console.WriteLine(myNumbers.Min());  // returns the smallest value
      Console.WriteLine(myNumbers.Sum());  // returns the sum of elements
    }
  }
}
```

You will learn more about other namespaces in a later chapter.

Other Ways to Create an Array

If you are familiar with C#, you might have seen arrays created with the `new` keyword, and perhaps you have seen arrays with a specified size as well. In C#, there are different ways to create an array:

```
// Create an array of four elements, and add values later
string[] cars = new string[4];

// Create an array of four elements and add values right away
string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size
string[] cars = new string[] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements, omitting the new keyword, and without specifying the size
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

It is up to you which option you choose. In our tutorial, we will often use the last option, as it is faster and easier to read.

However, you should note that if you declare an array and initialize it later, you have to use the `new` keyword:

```
// Declare an array
string[] cars;

// Add values, using new
cars = new string[] {"Volvo", "BMW", "Ford"};

// Add values without using new (this will cause an error)
cars = {"Volvo", "BMW", "Ford"};
```

C# Methods

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method is defined with the name of the method, followed by parentheses **()**. C# provides some pre-defined methods, which you already are familiar with, such as `Main()`, but you can also create your own methods to perform certain actions:

Example

Create a method inside the Program class:

```
class Program
{
  static void MyMethod()
  {
Â    // code to be executed
  }
}
```

Example Explained

- `MyMethod()` is the name of the method
- `static` means that the method belongs to the Program class and not an object of the Program class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Note: In C#, it is good practice to start with an uppercase letter when naming methods, as it makes the code easier to read.

Call a Method

To call (execute) a method, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, `MyMethod()` is used to print a text (the action), when it is called:

Example

Inside `Main()`, call the `myMethod()` method:

```
static void MyMethod()
{
Â  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
Â  MyMethod();
}

// Outputs "I just got executed!"
```

A method can be called multiple times:

Example

```
static void MyMethod()
```

```
{
Â  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
Â  MyMethod();
Â  MyMethod();
Â  MyMethod();
}

// I just got executed!
// I just got executed!
// I just got executed!
```

C# Exercises

C# Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a string called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
static void MyMethod(string fname)
{
Â  Console.WriteLine(fname + " Refsnes");
}

static void Main(string[] args)
{
Â  MyMethod("Liam");
Â  MyMethod("Jenny");
Â  MyMethod("Anja");
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: fname is a **parameter**, while Liam, Jenny and Anja are **arguments**.

Default Parameter Value

You can also use a default parameter value, by using the equals sign (=). If we call the method without an argument, it uses the default value ("Norway"):

Example

```
static void MyMethod(string country = "Norway")
{
Â  Console.WriteLine(country);
}

static void Main(string[] args)
{
Â  MyMethod("Sweden");
Â  MyMethod("India");
  MyMethod();
```

```
A  MyMethod("USA");
}

// Sweden
// India
// Norway
// USA
```

A parameter with a default value, is often known as an "**optional parameter**". From the example above, `country` is an optional parameter and `"Norway"` is the default value.

Multiple Parameters

You can have as many parameters as you like:

Example

```
static void MyMethod(string fname, int age)
{
Â  Console.WriteLine(fname + " is " + age);
}

static void Main(string[] args)
{
Â  MyMethod("Liam", 5);
Â  MyMethod("Jenny", 8);
Â  MyMethod("Anja", 31);
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The `void` keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as `int` or `double`) instead of `void`, and use the `return` keyword inside the method:

Example

```
static int MyMethod(int x)
{
Â  return 5 + x;
}

static void Main(string[] args)
{
Â  Console.WriteLine(MyMethod(3));
}

// Outputs 8 (5 + 3)
```

This example returns the sum of a method's**two parameters**:

Example

```
static int MyMethod(int x, int y)
{
Â  return x + y;
}

static void Main(string[] args)
{
Â  Console.WriteLine(MyMethod(5, 3));
}

// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example

```
static int MyMethod(int x, int y)
{
Â  return x + y;
}

static void Main(string[] args)
{
  int z = MyMethod(5, 3);
  Console.WriteLine(z);
}

// Outputs 8 (5 + 3)
```

Named Arguments

It is also possible to send arguments with the *key: value* syntax.

That way, the order of the arguments does not matter:

Example

```
static void MyMethod(string child1, string child2, string child3)
{
Â  Console.WriteLine("The youngest child is: " + child3);
}

static void Main(string[] args)
{
Â  MyMethod(child3: "John", child1: "Liam", child2: "Liam");
}

// The youngest child is: John
```

Named arguments are especially useful when you have multiple parameters with default values, and you only want to specify one of them when you call it:

Example

```
static void MyMethod(string child1 = "Liam", string child2 = "Jenny", string child3 = "John")
{
  Console.WriteLine(child3);
}

static void Main(string[] args)
{
  MyMethod("child3");
}

// John
```

C# Method Overloading

Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

Example

```
int MyMethod(int x)
float MyMethod(float x)
double MyMethod(double x, double y)
```

Consider the following example, which have two methods that add numbers of different type:

Example

```
static int PlusMethodInt(int x, int y)
{
  return x + y;
}

static double PlusMethodDouble(double x, double y)
{
  return x + y;
}

static void Main(string[] args)
{
  int myNum1 = PlusMethodInt(8, 5);
  double myNum2 = PlusMethodDouble(4.3, 6.26);
  Console.WriteLine("Int: " + myNum1);
  Console.WriteLine("Double: " + myNum2);
}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the`PlusMethod` method to work for both `int` and `double`:

Example

```
static int PlusMethod(int x, int y)
{
  return x + y;
}

static double PlusMethod(double x, double y)
{
  return x + y;
}

static void Main(string[] args)
{
  int myNum1 = PlusMethod(8, 5);
  double myNum2 = PlusMethod(4.3, 6.26);
  Console.WriteLine("Int: " + myNum1);
  Console.WriteLine("Double: " + myNum2);
}
```

Note: Multiple methods can have the same name as long as the number and/or type of parameters are different.

C# OOP

C# - What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

C# - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

class

Fruit

objects

Apple

Banana

Mango

Another example:

class

Car

objects

Volvo

Audi

Toyota

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

You will learn much more about classes and objects in the next chapter.

C# Classes and Objects

Classes and Objects

You learned from the previous chapter that C# is an object-oriented programming language.

Everything in C# is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the `class` keyword:

Create a class named "`Car`" with a variable `color`:

```
class Car
{
Â  string color = "red";
}
```

When a variable is declared directly in a class, it is often referred to as a **field** (or attribute).

It is not required, but it is a good practice to start with an uppercase first letter when naming classes. Also, it is common that the name of the C# file and the class matches, as it makes our code organized. However it is not required (like in Java).

Create an Object

An object is created from a class. We have already created the class named `Car`, so now we can use this to create objects.

To create an object of Car, specify the class name, followed by the object name, and use the keyword new:

Example

Create an object called "myObj" and use it to print the value of color:

```
class Car
{
Â  string color = "red";

Â  static void Main(string[] args)
  {
Â Â Â  Car myObj = new Car();
Â Â Â  Console.WriteLine(myObj.color);
Â  }
}
```

Note that we use the dot syntax (.) to access variables/fields inside a class (myObj.color). You will learn more about fields in the next chapter.

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of Car:

```
class Car
{
  string color = "red";
  static void Main(string[] args)
  {
    Car myObj1 = new Car();
    Car myObj2 = new Car();
    Console.WriteLine(myObj1.color);
    Console.WriteLine(myObj2.color);
  }
}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the fields and methods, while the other class holds the Main() method (code to be executed)).

- prog2.cs
- prog.cs

prog2.cs

```
class Car
{
Â  public string color = "red";
}
```

prog.cs

```
class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }
}
```

Did you notice the public keyword? It is called an **access modifier**, which specifies that the color variable/field of Car is accessible for other classes as well, such as Program.

You will learn much more about **access modifiers** and **classes**/**objects** in the next chapters.

C# Class Members

Class Members

Fields and methods inside classes are often referred to as "Class Members":

Example

Create a Car class with three class members:**two fields** and **one method**.

```
// The class
class MyClass
{
  // Class members
Â  string color = "red";        // field
Â  int maxSpeed = 200;          // field
  public void fullThrottle()   // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

Fields

In the previous chapter, you learned that variables inside a class are called fields, and that you can access them by creating an object of the class, and by using the dot syntax (.).

The following example will create an object of theCar class, with the name myObj. Then we print the value of the fields color and maxSpeed:

Example

```
class Car
{
Â  string color = "red";
  int maxSpeed = 200;

Â  static void Main(string[] args)
  {
Â Â Â  Car myObj = new Car();
Â Â Â  Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
Â  }
}
```

You can also leave the fields blank, and modify them when creating the object:

Example

```
class Car
{
Â  string color;
  int maxSpeed;

Â  static void Main(string[] args)
  {
Â Â Â  Car myObj = new Car();
    myObj.color = "red";
    myObj.maxSpeed = 200;
Â Â Â  Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
Â  }
}
```

This is especially useful when creating multiple objects of one class:

Example

```
class Car
```

```
{
  string model;
Â  string color;
  int year;

Â  static void Main(string[] args)
  {
Â Â Â  Car Ford = new Car();
    Ford.model = "Mustang";
    Ford.color = "red";
    Ford.year = 1969;

Â Â Â  Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;

Â Â Â  Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
Â  }
}
```

Object Methods

You learned from the C# Methods chapter that methods are used to perform certain actions.

Methods normally belongs to a class, and they define how an object of a class behaves.

Just like with fields, you can access methods with the dot syntax. However, note that the method must be `public`. And remember that we use the name of the method followed by two parantheses `()` and a semicolon `;` to call (execute) the method:

Example

```
class Car
{
Â  string color;                // field
  int maxSpeed;                  // field
  public void fullThrottle()    // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }

Â  static void Main(string[] args)
  {
Â Â Â  Car myObj = new Car();
    myObj.fullThrottle();  // Call the method
Â  }
}
```

Why did we declare the method as `public`, and not `static`, like in the examples from the C# Methods Chapter?

The reason is simple: a `static` method can be accessed without creating an object of the class, while `public` methods can only be accessed by objects.

Use Multiple Classes

Remember from the last chapter, that we can use multiple classes for better organization (one for fields and methods, and another one for execution). This is recommended:

prog2.cs

```
class Car
{
  public string model;
Â  public string color;
  public int year;
  public void fullThrottle()
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

prog.cs

```
class Program
{
  static void Main(string[] args)
  {
Â Â Â  Car Ford = new Car();
    Ford.model = "Mustang";
    Ford.color = "red";
    Ford.year = 1969;

Â Â Â  Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;

Â Â Â  Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
Â  }
}
```

The public keyword is called an **access modifier**, which specifies that the fields of Car are accessible for other classes as well, such as Program.

You will learn more about Access Modifiers in a later chapter.

Tip: As you continue to read, you will also learn more about other class members, such as constructors and properties.

C# Constructors

Constructors

A constructor is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

Example

Create a constructor:

```
// Create a Car class
class Car
{
  public string model;  // Create a field

  // Create a class constructor for the Car class
  public Car()
  {
    model = "Mustang"; // Set the initial value for model
  }

  static void Main(string[] args)
  {
    Car Ford = new Car();  // Create an object of the Car Class (this will call the constructor)
    Console.WriteLine(Ford.model);  // Print the value of model
  }
}

// Outputs "Mustang"
```

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void or int).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.

Constructors save time! Take a look at the last example on this page to really understand why.

Constructor Parameters

Constructors can also take parameters, which is used to initialize fields.

The following example adds a string modelName parameter to the constructor. Inside the constructor we set model to modelName (model=modelName). When we call the constructor, we pass a parameter to the constructor ("Mustang"), which will set the value of model to "Mustang":

Example

```
class Car
{
  public string model;

  // Create a class constructor with a parameter
  public Car(string modelName)
  {
    model = modelName;
  }

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang");
    Console.WriteLine(Ford.model);
  }
}

// Outputs "Mustang"
```

You can have as many parameters as you want:

Example

```
class Car
{
  public string model;
  public string color;
  public int year;

  // Create a class constructor with multiple parameters
  public Car(string modelName, string modelColor, int modelYear)
  {
    model = modelName;
    color = modelColor;
    year = modelYear;
  }

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang", "Red", 1969);
    Console.WriteLine(Ford.color + " " + Ford.year + " " + Ford.model);
  }
}


// Outputs Red 1969 Mustang
```

Tip: Just like other methods, constructors can be**overloaded** by using different numbers of parameters.

Constructors Save Time

When you consider the example from the previous chapter, you will notice that constructors are very useful, as they help reducing the amount of code:

Without constructor:

rog.cs

```
lass Program

static void Main(string[] args)
{
 Â Â  Car Ford = new Car();
  Ford.model = "Mustang";
  Ford.color = "red";
  Ford.year = 1969;

 Â Â  Car Opel = new Car();
  Opel.model = "Astra";
  Opel.color = "white";
  Opel.year = 2005;

 Â Â  Console.WriteLine(Ford.model);
  Console.WriteLine(Opel.model);
 }
```

With constructor:

rog.cs

```
lass Program

static void Main(string[] args)
{
 Â Â  Car Ford = new Car("Mustang", "Red", 1969);
  Car Opel = new Car("Astra", "White", 2005);

 Â Â  Console.WriteLine(Ford.model);
  Console.WriteLine(Opel.model);
 }
```

C# Access Modifiers

Access Modifiers

By now, you are quite familiar with the public keyword that appears in many of our examples:

public string color;

The public keyword is an **access modifier**, which is used to set the access level/visibility for classes, fields, methods and properties.

C# has the following access modifiers:

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the same class

protected	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter
internal	The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter

There's also two combinations: protected internal and private protected.

For now, lets focus on public and private modifiers.

Private Modifier

If you declare a field with a private access modifier, it can only be accessed within the same class:

Example

```
class Car
{
  private string model = "Mustang";

  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
Mustang
```

If you try to access it outside the class, an error will occur:

Example

```
class Car
{
  private string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
'Car.model' is inaccessible due to its protection level
The field 'Car.model' is assigned but its value is never used
```

Public Modifier

If you declare a field with a public access modifier, it is accessible for all classes:

Example

```
class Car
{
  public string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
Mustang
```

Why Access Modifiers?

To control the visibility of class members (the security level of each individual class and class member).

To achieve "**Encapsulation**" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as private. You will learn more about this in the next chapter.

Note: By default, all members of a class areprivate if you don't specify an access modifier:

Example

```
class Car
{
  string model;  // private
  string year;   // private
}
```

C# Properties (Get and Set)

Properties and Encapsulation

Before we start to explain properties, you should have a basic understanding of **Encapsulation**".

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare fields/variables as private
- provide public get and set methods, through **properties**, to access and update the value of aprivate field

---

Properties

You learned from the previous chapter thatprivate variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

A property is like a combination of a variable and a method, and it has two methods: aget and a set method:

Example

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

  public string Name   // property
  {
    get { return name; }   // get method
    set { name = value; }  // set method
  }
}
```

Example explained

The Name property is associated with thename field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.

The get method returns the value of the variablename.

The set method assigns a value to the name variable. The value keyword represents the value we assign to the property.

If you don't fully understand it, take a look at the example below.

Now we can use the Name property to access and update the private field of the Person class:

Example

```
class Person
{
  private string name; // field
  public string Name   // property
  {
    get { return name; }
    set { name = value; }
  }
}

class Program
{
  static void Main(string[] args)
  {
    Person myObj = new Person();
    myObj.Name = "Liam";
    Console.WriteLine(myObj.Name);
  }
}
```

The output will be:

```
Liam
```

Automatic Properties (Short Hand)

C# also provides a way to use short-hand / automatic properties, where you do not have to define the field for the property, and you only have to write `get;` and `set;` inside the property.

The following example will produce the same result as the example above. The only difference is that there is less code:

Example

Using automatic properties:

```
class Person
{
  public string Name  // property
  { get; set; }
}

class Program
{
  static void Main(string[] args)
  {
    Person myObj = new Person();
    myObj.Name = "Liam";
    Console.WriteLine(myObj.Name);
  }
}
```

The output will be:

```
Liam
```

Why Encapsulation?

- Better control of class members (reduce the possibility of yourself (or others) to mess up the code)
- Fields can be made **read-only** (if you only use the `get` method), or **write-only** (if you only use the `set` method)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

C# Inheritance

Inheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- **Derived Class** (child) - the class that inherits from another class
- **Base Class** (parent) - the class being inherited from

To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

Example

```
class Vehicle  // base class (parent)
{
Â  public string brand = "Ford";  // Vehicle field
Â  public void honk()             // Vehicle method
  {Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â Â 
Â Â Â  Console.WriteLine("Tuut, tuut!");
Â  }
}

class Car : Vehicle  // derived class (child)
{
Â  public string modelName = "Mustang";  // Car field
}

class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
    Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The sealed Keyword

If you don't want other classes to inherit from a class, use the sealed keyword:

If you try to access a sealed class, C# will generate an error:

```
sealed class Vehicle
{
  ...
}

class Car : Vehicle
{
  ...
}
```

The error message will be something like this:

```
'Car': cannot derive from sealed type 'Vehicle'
```

C# Polymorphism

Polymorphism and Overriding Methods

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit fields and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
class Animal  // Base class (parent)
{
Â  public void animalSound()
  {
Â Â Â  Console.WriteLine("The animal makes a sound");
Â  }
}

class Pig : Animal  // Derived class (child)
{
Â  public void animalSound()
  {
Â Â Â  Console.WriteLine("The pig says: wee wee");
Â  }
}

class Dog : Animal  // Derived class (child)
{
Â  public void animalSound()
  {
Â Â Â  Console.WriteLine("The dog says: bow wow");
Â  }
}
```

Remember from the Inheritance chapter that we use the : symbol to inherit from a class.

Now we can create Pig and Dog objects and call the animalSound() method on both of them:

Example

```
class Animal  // Base class (parent)
{
Â  public void animalSound()
  {
Â Â Â  Console.WriteLine("The animal makes a sound");
Â  }
}

class Pig : Animal  // Derived class (child)
{
Â  public void animalSound()
  {
Â Â Â  Console.WriteLine("The pig says: wee wee");
Â  }
}

class Dog : Animal  // Derived class (child)
{
Â  public void animalSound()
  {
Â Â Â  Console.WriteLine("The dog says: bow wow");
Â  }
}

class Program
{
Â  static void Main(string[] args)
  {
Â Â Â  Animal myAnimal = new Animal();Â  // Create a Animal object
Â Â Â  Animal myPig = new Pig();Â  // Create a Pig object
Â Â Â  Animal myDog = new Dog();Â  // Create a Dog object

Â Â Â  myAnimal.animalSound();
Â Â Â  myPig.animalSound();
Â Â Â  myDog.animalSound();
```

```
A  }
}
```

The output will be:

```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

Not The Output I Was Looking For

The output from the example above was probably not what you expected. That is because the base class method overrides the derived class method, when they share the same name.

However, C# provides an option to override the base class method, by adding the `virtual` keyword to the method inside the base class, and by using the `override` keyword for each derived class methods:

Example

```
class Animal  // Base class (parent)
{
Â  public virtual void animalSound()
  {
Â Â Â  Console.WriteLine("The animal makes a sound");
Â  }
}

class Pig : Animal  // Derived class (child)
{
Â  public override void animalSound()
  {
Â Â Â  Console.WriteLine("The pig says: wee wee");
Â  }
}

class Dog : Animal  // Derived class (child)
{
Â  public override void animalSound()
  {
Â Â Â  Console.WriteLine("The dog says: bow wow");
Â  }
}

class Program
{
Â  static void Main(string[] args)
  {
Â Â Â  Animal myAnimal = new Animal();Â  // Create a Animal object
Â Â Â  Animal myPig = new Pig();Â  // Create a Pig object
Â Â Â  Animal myDog = new Dog();Â  // Create a Dog object

Â Â Â  myAnimal.animalSound();
Â Â Â  myPig.animalSound();
Â Â Â  myDog.animalSound();
Â  }
}
```

The output will be:

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

Why And When To Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

C# Abstraction

Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The abstract keyword is used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal
{
Â  public abstract void animalSound();
Â  public void sleep()
  {
Â Â Â  Console.WriteLine("Zzz");
Â  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // Will generate an error (Cannot create an instance of the abstract class or interface 'Animal')
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class.

Remember from the Inheritance chapter that we use the : symbol to inherit from a class, and that we use the override keyword to override the base class method.

Example

```
// Abstract class
abstract class Animal
{
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep()
  {
    Console.WriteLine("Zzz");
  }
}

// Derived class (inherit from Animal)
class Pig : Animal
{
  public override void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The pig says: wee wee");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();  // Call the abstract method
    myPig.sleep();  // Call the regular method
  }
}
```

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Note: Abstraction can also be achieved withInterfaces, which you will learn more about in the next chapter.

C# Interface

Interfaces

Another way to achieve abstraction in C#, is with interfaces.

An interface is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies):

Example

```
// interface
interface Animal
{
Â  void animalSound(); // interface method (does not have a body)
Â  void run(); // interface method (does not have a body)
}
```

It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.

By default, members of an interface are abstract and public.

Note: Interfaces can contain properties and methods, but not fields.

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class. To implement an interface, use the : symbol (just like with inheritance). The body of the interface method is provided by the "implement" class. Note that you do not have to use the override keyword when implementing an interface:

Example

```
// Interface
interface IAnimal
{
Â  void animalSound(); // interface method (does not have a body)
}

// Pig "implements" the IAnimal interface
class Pig : IAnimal
{
Â  public void animalSound()
  {
Â Â Â  // The body of animalSound() is provided here
Â Â Â  Console.WriteLine("The pig says: wee wee");
Â  }
}

class Program
{
Â  static void Main(string[] args)
  {
Â Â Â  Pig myPig = new Pig();Â  // Create a Pig object
Â Â Â  myPig.animalSound();
Â  }
}
```

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interfaces can contain properties and methods, but not fields/variables
- Interface members are by default abstract and public
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface IFirstInterface
{
Â  void myMethod(); // interface method
}

interface ISecondInterface
{
Â  void myOtherMethod(); // interface method
}

// Implement multiple interfaces
class DemoClass : IFirstInterface, ISecondInterface
{
Â  public void myMethod()
  {
Â Â Â  Console.WriteLine("Some text..");
Â  }
Â  public void myOtherMethod()
  {
Â Â Â  Console.WriteLine("Some other text...");
Â  }
}

class Program
{
Â  static void Main(string[] args)
  {
Â Â Â  DemoClass myObj = new DemoClass();
Â Â Â  myObj.myMethod();
Â Â Â  myObj.myOtherMethod();
Â  }
}
```

C# Enum

C# Enums

An enum is a special "class" that represents a group of **constants** (unchangeable/read-only variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma:

Example

```
enum Level
{
Â  Low,
Â  Medium,
Â  High
}
```

You can access enum items with the **dot** syntax:

```
Level myVar = Level.Medium;
Console.WriteLine(myVar);
```

Enum is short for "enumerations", which means "specifically listed".

Enum inside a Class

You can also have an enum inside a class:

Example

```
class Program
{
  enum Level
```

```
  {
    Low,
    Medium,
    High
  }
  static void Main(string[] args)
  {
    Level myVar = Level.Medium;
    Console.WriteLine(myVar);
  }
}
```

The output will be:

```
Medium
```

Enum Values

By default, the first item of an enum has the value 0. The second has the value 1, and so on.

To get the integer value from an item, you must explicitly convert the item to an `int`:

Example

```
enum Months
{
  January,    // 0
  February,   // 1
  March,      // 2
  April,      // 3
  May,        // 4
  June,       // 5
  July        // 6
}

static void Main(string[] args)
{
  int myNum = (int) Months.April;
  Console.WriteLine(myNum);
}
```

The output will be:

```
3
```

You can also assign your own enum values, and the next items will update the number accordingly:

Example

```
enum Months
{
  January,    // 0
  February,   // 1
  March=6,    // 6
  April,      // 7
  May,        // 8
  June,       // 9
  July        // 10
}

static void Main(string[] args)
{
  int myNum = (int) Months.April;
  Console.WriteLine(myNum);
}
```

The output will be:

```
7
```

Enum in a Switch Statement

Enums are often used in `switch` statements to check for corresponding values:

Example

```
enum Level
{
Â  Low,
Â  Medium,
Â  High
}

static void Main(string[] args)
{
  Level myVar = Level.Medium;
Â  switch(myVar)
  {
Â Â Â  case Level.Low:
Â Â Â Â Â  Console.WriteLine("Low level");
Â Â Â Â Â Â Â break;
Â Â Â Â case Level.Medium:
Â Â Â Â Â Â Â Console.WriteLine("Medium level");
Â Â Â Â Â Â Â break;
Â Â Â Â case Level.High:
Â Â Â Â Â Â Console.WriteLine("High level");
Â Â Â Â Â Â break;
Â Â }
}
```

The output will be:

```
Medium level
```

Why And When To Use Enums?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

C# Files

Working With Files

The File class from the System.IO namespace, allows us to work with files:

Example

```
using System.IO;  // include the System.IO namespace

File.SomeFileMethod();  // use the file class with methods
```

The File class has many useful methods for creating and getting information about files. For example:

Method	Description
AppendText()	Appends text at the end of an existing file
Copy()	Copies a file
Create()	Creates or overwrites a file
Delete()	Deletes a file
Exists()	Tests whether the file exists
ReadAllText()	Reads the contents of a file
Replace()	Replaces the contents of a file with the contents of another file
WriteAllText()	Creates a new file and writes the contents to it. If the file already exists, it will be overwritten.

For a full list of File methods, go to Microsoft .Net File Class Reference.

Write To a File and Read It

In the following example, we use the WriteAllText() method to create a file named "filename.txt" and write some content to it. Then we use the ReadAllText() method to read the contents of the file:

Example

```
using System.IO;  // include the System.IO namespace

string writeText = "Hello World!";  // Create a text string
File.WriteAllText("filename.txt", writeText);  // Create a file and write the content of writeText to it

string readText = File.ReadAllText("filename.txt");  // Read the contents of the file
Console.WriteLine(readText);  // Output the content
```

The output will be:

```
Hello World!
```

C# Exceptions - Try..Catch

C# Exceptions

When executing C# code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C# will normally stop and generate an error message. The technical term for this is: C# will throw an **exception** (throw an error).

C# try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

Syntax

```
try
{
Â Â //Â  Block of code to try
}
catch (Exception e)
{
Â Â //Â  Block of code to handle errors
}
```

Consider the following example, where we create an array of three integers:

This will generate an error, because **myNumbers[10]** does not exist.

```
int[] myNumbers = {1, 2, 3};
Console.WriteLine(myNumbers[10]); // error!
```

The error message will be something like this:

```
System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'
```

If an error occurs, we can use `try...catch` to catch the error and execute some code to handle it.

In the following example, we use the variable inside the catch block (`e`) together with the built-in `Message` property, which outputs a message that describes the exception:

Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine(e.Message);
}
```

The output will be:

```
Index was outside the bounds of the array.
```

You can also output your own error message:

Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
```

The output will be:

```
Something went wrong.
```

Finally

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
finally
{
  Console.WriteLine("The 'try catch' is finished.");
}
```

The output will be:

```
Something went wrong.
The 'try catch' is finished.
```

The throw keyword

The `throw` statement allows you to create a custom error.

The `throw` statement is used together with an **exception class**. There are many exception classes available in C#: `ArithmeticException`, `FileNotFoundException`, `IndexOutOfRangeException`, `TimeOutException`, etc:

Example

```
static void checkAge(int age)
{
  if (age < 18)
  {
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");
  }
  else
  {
    Console.WriteLine("Access granted - You are old enough!");
  }
}

static void Main(string[] args)
{
  checkAge(15);
}
```

The error message displayed in the program will be:

System.ArithmeticException: 'Access denied - You must be at least 18 years old.'

If age was 20, you would **not** get an exception:

Example

```
checkAge(20);
```

The output will be:

```
Access granted - You are old enough!
```

C# How To Add Two Numbers

Add Two Numbers

Learn how to add two numbers in C#:

Example

```
int x = 5;
int y = 6;
int sum = x + y;
Console.WriteLine(sum); // Print the sum of x + y
```

C# Examples

C# Syntax

Create a simple "Hello World" program
Syntax Explained

C# Comments

Single-line comment before a line of code Single-line comment at the end of a line of code Multi-line comment
Comments Explained

C# Variables

Create a string variable Create an integer variable Create a variable without assigning the value, and assign the value later Overwrite an existing variable value Combine text and a variable on display Add a variable to another variable Declare many variables of the same type with a comma-separated list
Variables Explained

C# Data Types

A demonstration of different data types in C# Create an int type Create a long type Create a float type Create a double type Create a bool type Create a char type Create a string type
Data Types Explained

C# Type Casting

Implicit casting Explicit casting Type conversion methods
Type Casting Explained

C# User Input

Get user input text Get user input with numbers
User Input Explained

C# Operators

Addition operator Subtraction operator Multiplication operator Division operator Modulus operator Increment operator Decrement operator Assignment operator Addition assignment operator
Operators Explained

C# Math

Math.Max(x,y) - return the highest value of x and y Math.Min(x,y) - return the lowest value of x and y Math.Sqrt(x) - return the square root of x Math.Abs(x) - return the absolute (positive) value of x Math.Round() - round a number to the nearest whole number
Math Explained

C# Strings

Create a string Find the length of a string Using methods to convert strings to uppercase and lowercase String concatenation String concatenation with the Concat() method String interpolation Access characters in a string by referring to its index number Find the index position of a specific character in a string, by using the IndexOf() method Use the Substring() method together with the IndexOf() method Use quotes in a string
Strings Explained

C# Booleans

Create a bool (boolean) type Find out if an expression is true or false Use the "equal to" operator to evaluate a boolean expression
Booleans Explained

C# If...Else (Conditions)

The if statement The else statement The else if statement
If...Else Explained

C# Switch

The switch statement The switch statement with a default keyword
Switch Explained

C# Loops

While loop Do while loop For loop foreach loop Break a loop Continue a loop
Loops Explained

C# Arrays

Create and access an array Change an array element Find the length of an array Access an array Loop through an array Loop through an array with foreach Sort an array Using System.Linq
Arrays Explained

C# Methods

Create and call a method Call a method multiple times Method with parameters Default parameter value Multiple parameters Return value Return the sum of a method's two parameters Named arguments Overload a method
Methods Explained

C# Classes and Objects

Create a class and an object of a class Create multiple objects of a class Use multiple classes for better organization Access fields and methods Create a class constructor Constructor with parameters Private modifier Public modifier Properties (get and set) Automatic (short-hand) properties Inheritance Polymorphism Abstraction Interface Multiple interfaces Enums Working with files
Classes and Objects Explained

C# Exceptions (Try...Catch)

The try...catch statement The finally statement
Exceptions Explained

C# Online Compiler

C# Compiler (Editor)

With our online C# compiler, you can edit C# code, and view the result in your browser.

Example

```
using System;

namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

```
Hello World!
```

Try it Yourself Â»

Click on the "Try it Yourself" button to see how it works.

C# Compiler Explained

The window to the left is editable - edit the code and click on the "Run" button to view the result in the right window.

The icons are explained in the table below:

Icon	Description
	Go to www.w3schools.com
	Menu button for more options
	Change orientation (horizontally or vertically)
	Change color theme (dark or light)

If you don't know C#, we suggest that you read ourC# Tutorial from scratch.

C# Exercises

You can test your C# skills with W3Schools' Exercises.

Exercises

We have gathered a variety of C# exercises (with answers) for each C# Chapter.

Try to solve an exercise by editing some code, or show the answer to see what you've done wrong.

Count Your Score

You will get 1 point for each correct answer. Your score and total score will always be displayed.

Start C# Exercises

Good luck!

If you don't know C#, we suggest that you read ourC# Tutorial from scratch.

C# Quiz

You can test your C# skills with W3Schools' Quiz.

The Test

The test contains 25 questions and there is no time limit.

The test is not official, it's just a nice way to see how much you know, or don't know, about C#.

Count Your Score

You will get 1 point for each correct answer. At the end of the Quiz, your total score will be displayed. Maximum score is 25 points.

Start the Quiz

Good luck!

If you don't know C#, we suggest that you read ourC# Tutorial from scratch.
