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 like c++ and java.

C# is used for:

- Mobile applications
- Desktop applications
- Web applications
- Web services
- Web sites
- Games
- VR
- Database applications

C# Syntax

```
using System;

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

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.

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:
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).

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

Syntax

```
type variableName = value;
```

C# Data Types

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 byte	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

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`

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

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

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

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`):

Get User Input

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

C# Operators

Operators are used to perform operations on variables and values.

Arithmetic Operators:

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

C# Assignment Operators

Operator	Example	Same As
=	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
==	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
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	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)

C# Math

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

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

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

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

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

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

C# Strings

Strings are used for storing text.

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

Example : string greeting = "Hello";

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 = {"Volvo", "BMW", "Ford", "Mazda"};
```

Access the Elements of an Array

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

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

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

Other 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"};
```

C# Method

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()
  {
    // 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.

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)
{
  Console.WriteLine(fname + " Refsnes");
}

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

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")
{
  Console.WriteLine(country);
}
static void Main(string[] args)
{
  MyMethod("Sweden");
  MyMethod("India");
  MyMethod();
  MyMethod("USA");
}
```

Note:

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

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)
{
  Console.WriteLine("The youngest child is: " + child3);
}

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

Method Overloading

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

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);
}
```

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

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:

Ex-1

Class
Fruit
Objects
Apple
Banana
Mango

Ex-2

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.

Create a Class

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

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

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

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

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

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

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:

Note: The constructor name must **match the class name**, and it cannot have a **returntype** (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.

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
  }
}
```

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

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"
```