Introduction to C#?

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.

The first version was released in 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

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.

```
using System;



namespace HelloWorld

{
```

```
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("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 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: Every C# statement ends with a semicolon `;`.

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

C# Namespaces

Namespaces in C# are used to organize too many classes so that it can be easy to handle the application.

In a simple C# program, we use System.Console where System is the namespace and Console is the class. To access the class of a namespace, we need to use namespacename.classname. We can use **using** keyword so that we don't have to use complete name all the time.

In C#, global namespace is the root namespace. The global::System will always refer to the namespace "System" of .Net Framework.

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):

```
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:

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!");
```

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!");
```

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);
```

Example

```
int myNum;

myNum = 15;

Console.WriteLine(myNum);
```

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
```

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";
```

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);
```

Example

```
string firstName = "John ";

string lastName = "Doe";

string fullName = firstName + lastName;

Console.WriteLine(fullName);
```

Example

```
int x = 5;

int y = 6;

Console.WriteLine(x + y); // Print the value of x + y
```

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 naming variables 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# Data Types

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

Example

```
int myNum = 5;               // Integer (whole number)

double myDoubleNum = 5.99D;  // Floating point number

char myLetter = 'D';         // Character

bool myBool = true;          // Boolean

string myText = "Hello";     // String
```

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`.

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 3.4e−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 1.7e−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);
```

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);   // Outputs True

Console.WriteLine(isFishTasty);   // Outputs False
```

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);
```

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:

```
                                                                                
```

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

Example

```
Console.WriteLine("Enter your age:");

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

Console.WriteLine("Your age is: " + age);
```

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);      // Outputs 9
Console.WriteLine(myDouble);   // 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;    // Manual casting: double to int

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

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++)
{
  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)
{
  // 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)
{
  Console.WriteLine(i);
}
```

The example above can be read like this: for each `string` element (called i - as in index) 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
    }
  }
}
```

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

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

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

A method can be called multiple times:

Example

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

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

```
{
  MyMethod();
  MyMethod();
  MyMethod();
}

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

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");
```

```
}

// 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)
{
  Console.WriteLine(fname + " is " + age);
}

static void Main(string[] args)
{
  MyMethod("Liam", 5);
  MyMethod("Jenny", 8);
  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)
{
  return 5 + x;
}

static void Main(string[] args)
{
  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)
{
  return x + y;
}

static void Main(string[] args)
{
  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)
{
  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)
{
  Console.WriteLine("The youngest child is: " + child3);
}

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

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

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

```
  MyMethod("Liam");
  MyMethod("Jenny");
  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.

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
{
  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
{
  string color = "red";

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

```
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }

}
```

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
{
  public string color = "red";
}
```

prog.cs

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

}
```