C# DOCUMENTATION

**Introduction :**

Know as C-Sharp created by Microsoft that runs on .NET Framework.

**NameSpace:**

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.

Syntax: using System;

**Variables in C-Sharp:**

**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

**Variable Syntax:**

type variableName = value;

Example:

int myNum = 15;

Console.WriteLine(myNum);

## 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
  + int myInt = 9; double
  + myDouble = myInt;
* **Explicit Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char
  + double myDouble = 9.78;
  + int myInt = (int) myDouble;

## 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):

int myInt = 10;

Console.WriteLine(Convert.ToString(myInt)); Console.WriteLine(Convert.ToDouble(myInt));

**User Input:**

string userName = Console.ReadLine();

**Operators:**

* **C# Arithmetic Operators:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Description | Example |
| + | 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:**

|  |  |  |
| --- | --- | --- |
| 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**

## Math.Max(x,y)

## Math.Min(x,y)

## Math.Sqrt(x)

## Math.Abs(x)

## Math.Round()

# **C# Strings**

## String Length

## ToUpper() and ToLower()

## String Concatenation - string.Concat()

## String Interpolation

## string firstName = "John";

## string lastName = "Doe";

## string name = $"My full name is: {firstName} {lastName}";

# **C# Booleans**

## C# Conditions and If 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

## C# Switch Statements

## switch(expression)

## { case x:

## // code block break;

## case y:

## // code block break;

## default:

## // code block break;

## }

# **C# While Loop**

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

## C# For Loop

## for (statement 1; statement 2; statement 3) { // 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.

# **C# Break and Continue**

## C# Break

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

## 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.

# C# Arrays

## Example: int[] myNumbers = {5, 1, 8, 9};

# C# Methods:

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

Example Syntax:

class Program {

static void MyMethod() {

// code to be executed

}

}

## Parameters and Arguments

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

## Method Overloading

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

int MyMethod(int x)

float MyMethod(float x)

double MyMethod(double x, double y)

**OOPS in C#:**

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.

class Car

{

string color = "red";

static void Main(string[] args)

{

Car myObj = new Car();

Console.WriteLine(myObj.color);

}

}

## Constructors

A constructor is a **special method** that is used to initialize objects.

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](https://www.w3schools.com/cs/cs_inheritance.asp) 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 |

## Properties

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

class Person

{

private string name; // field

public string Name // property

{

get { return name; } // get method

set { name = value; } // set method

}

}

# **C# Inheritance**

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

class Vehicle // base class (parent)

{

public string brand = "Ford"; // Vehicle field

public void honk() // Vehicle method

{

Console.WriteLine("Tuut, tuut!");

}

}

class Car : Vehicle // derived class (child)

{

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);

} }

# **C# Polymorphism**

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

class Animal // Base class (parent)

{

public void animalSound()

{

Console.WriteLine("The animal makes a sound");

}

}

class Pig : Animal // Derived class (child)

{

public void animalSound()

{

Console.WriteLine("The pig says: wee wee");

}

}

class Dog : Animal // Derived class (child)

{

public void animalSound()

{

Console.WriteLine("The dog says: bow wow");

}

}

## 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**](https://www.w3schools.com/cs/cs_interface.asp)

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).

abstract class Animal

{

public abstract void animalSound();

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");

}

}

## Interfaces

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

interface IAnimal

{

void animalSound(); // interface method (does not have a body)

}

// Pig "implements" the IAnimal interface

class Pig : IAnimal

{

public void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

**C# Enums:**

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

enum Level

{

Low,

Medium,

High

}

Level myVar = Level.Medium;

Console.WriteLine(myVar);

# **C# Files**

using System.IO;

string writeText = "Hello World!";

File.WriteAllText("filename.txt", writeText);

string readText = File.ReadAllText("filename.txt");

Console.WriteLine(readText);

## 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:

try

{

// Block of code to try

}

catch (Exception e)

{

// Block of code to handle errors

}

~ L. N. S. S. Ravi Teja