**Method**

**Thread.Sleep(3000);**

**GetType();**

**DateTime.now**

**Static method:** are directly call from class.

Main(), Console.WriteLine()

**Instance method:** are call from class with instance object. Like>>>Game game = new Game();

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

Example:

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.
* void means that this method does not have a return value.

## **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, int y)

{

return x + y;

}

static void Main(string[] args)

{

int z = MyMethod(5, 3);

Console.WriteLine(z);

}

// Outputs 8 (5 + 3)

## **Method Overloading**

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

**Method overriding** is multiple method with same name and same parameter.

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.

**String interpolation**

Console.WriteLine($ “Total {1+2} numbers.”);

**Why did we declare the method as public, and not static.**

**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.**

## **Constructors**

1. 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.

2. It can be used to set initial values for fields.

3. Constructor name is same as **the class name**, and it cannot have a **return type** (like void or int).

4.Constructor are two types: a) **Implicit:** are created by compiler are known as default constructor, also known as parameter less constructor. b) **Explicit:** are created by programmer, also known as parameterized constructor.

5. **Static constructor** is used to initialize static variable and perform action only once.

// 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"

## **Access Modifiers**

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.

Internal: The code is only accessible within its own assembly, but not from another assembly.

## **Properties and Encapsulation**

**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 a private field
* A property is like a combination of a variable and a method, and it has two methods: a get 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
* }
* }
* public string Name // property shorthand
* { get; set; }

#### **Example explained**

The Name property is associated with the name 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 variable name.

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

## **Inheritance**

1. It is possible to inherit **fields** and **methods** from one class to another.

2. 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/sub class) inherits the fields and methods from the Vehicle class (parent/super class).

**Example:**

class Vehicle // base class (parent)

{

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

public void honk() // Vehicle method

{

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

}

}

class Car : Vehicle // Car is derived class (child class/sub class)

{

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);

}

}

3. For change the output of parent class method in child class, we use **virtual** in parent class’s method and **override** in child class’s method and write the code for new output.

4. If we don’t allow class to inherit use **sealed** keyword in front of class name.

## **Polymorphism and Overriding Methods**

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

**Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Add 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)

{

public **virtual** void animalSound()

{

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

}

}

class Pig : Animal // Derived class (child)

{

public **override** void animalSound()

{

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

}

}

class Dog : Animal // Derived class (child)

{

public **override** void animalSound()

{

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

}

}

class Program

{

static void Main(string[] args)

{

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

**Static methods and instance methods**

**Static methods** are directly call from class.

Console.WriteLine() here WriteLine() method called from Console class directly without assigned value before.

On the other hand, Instance class assign the value first and then called from other class.

**Exception (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:

try

{

// *Block of code to try*

}

catch (Exception e)

{

// *Block of code to handle errors*

}

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.'

**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"};

## **The foreach Loop**

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

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

foreach (string i in cars)

{

Console.WriteLine(i);

}

## **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);

}

## **System.Linq Namespace**

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

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

}

}

}

**Reading file from text file**

string path = @"C:\Users\eyahy\Desktop\c#\Scoccerstats\Scoccerstats\data.txt";

//file read

//use "using" to skip reader.Close();

using (FileStream file = new FileStream(path, FileMode.Open,FileAccess.Read))

{

using (StreamReader reader = new StreamReader(file))

{

//only read first line from text file

string line = reader.ReadLine();

Console.WriteLine(line);

//reading all text line

string line = "";

while ((line = reader.ReadLine()) != null)

{

Console.WriteLine(line);

}

//read each character

foreach (var item in reader.ReadLine())

{

Console.WriteLine(item);

}

//read first 4 line

string[] names = new string[4];

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

{

names[i] = reader.ReadLine();

Console.WriteLine(names[i]);

}

//read all line at a time

string line = reader.ReadToEnd();

Console.WriteLine(line);

//Peek() methods return if there is chracter in file or not

if (reader.Peek() > 0)

{

Console.WriteLine(true);

}

else

{

Console.WriteLine(false);

}

//convert unicode to UTF-8

char lowerH = '\u0068';

byte[] unicodeBytes = System.Text.UnicodeEncoding.Unicode.GetBytes(new char[] {lowerH});

string unicodeString = System.Text.UnicodeEncoding.Unicode.GetString(unicodeBytes);

Console.WriteLine(unicodeString);

//display degree sign

char degree = '\u00B0';

Console.WriteLine("The temperature is 10" + degree + "C");

//take the path of text file automaticaly

string path1 = Directory.GetCurrentDirectory();

DirectoryInfo dirInfo = new DirectoryInfo(path1);

var fileName = Path.Combine(dirInfo.FullName, "data.txt");

var file1 = new FileInfo(fileName);

using (var reader1 = new StreamReader(file1.FullName))

{

Console.WriteLine(reader1.ReadToEnd());

}

//take the path of CSV file automaticaly

string path1 = Directory.GetCurrentDirectory();

DirectoryInfo dirInfo = new DirectoryInfo(path1);

var fileName = Path.Combine(dirInfo.FullName, "SoccerGameResults.csv");

var fileContent = ReadFile(fileName);

Console.WriteLine(fileContent);

static string ReadFile(string fileName)

{

using (var reader = new StreamReader(fileName))

{

return reader.ReadToEnd();

}

}

}

}

Console.ReadLine();

**Value type and Reference type**

**Value type:** Int, char, var, bool, double, struct

**Reference type:** string, Abstract, Delegate, Class, Object.

# Method Parameters

The keywords you can use when declaring method parameters:

* [params](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params) specifies that this parameter may take a variable number of arguments.
* [in](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier) specifies that this parameter is passed by reference but is only read by the called method.
* [ref](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref) specifies that this parameter is passed by reference and may be read or written by the called method.
* [out](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier) specifies that this parameter is passed by reference and is written by the called method. The out parameter modifier keyword is used to pass a value type argument by reference, and while the argument doesn’t have to be initialized before it’s passed to the method parameter, the method does have to assign a value to the argument before it returns..

**Enum**

public class WeatherForecast

{

public string WeatherStationId { get; set; }

public DateTime TimeOfDay { get; set; }

public Condition Condition { get; set; }

}

public enum Condition

{

Rain,

Cloudy,

PartlyCloudy,

PartlySunny,

Sunny,

Clear

}

Use enum in another class:

Condition condition;

If (Enum.TryParse(values[1], out conditon))

{

weatherForecast.Condition = condition;

}

## **Characteristics of the integral types**

C# supports the following predefined integral types:

| **C# type/keyword** | **Range** | **Size** | **.NET type** |
| --- | --- | --- | --- |
| sbyte | -128 to 127 | Signed 8-bit integer | [System.SByte](https://docs.microsoft.com/en-us/dotnet/api/system.sbyte) |
| byte | 0 to 255 | Unsigned 8-bit integer | [System.Byte](https://docs.microsoft.com/en-us/dotnet/api/system.byte) |
| short | -32,768 to 32,767 | Signed 16-bit integer | [System.Int16](https://docs.microsoft.com/en-us/dotnet/api/system.int16) |
| ushort | 0 to 65,535 | Unsigned 16-bit integer | [System.UInt16](https://docs.microsoft.com/en-us/dotnet/api/system.uint16) |
| int | -2,147,483,648 to 2,147,483,647 | Signed 32-bit integer | [System.Int32](https://docs.microsoft.com/en-us/dotnet/api/system.int32) |
| uint | 0 to 4,294,967,295 | Unsigned 32-bit integer | [System.UInt32](https://docs.microsoft.com/en-us/dotnet/api/system.uint32) |
| long | -9,223,372,036,854,775,808  to 9,223,372,036,854,775,807 | Signed 64-bit integer | [System.Int64](https://docs.microsoft.com/en-us/dotnet/api/system.int64) |
| ulong | 0 to 18,446,744,073,709,551,615 | Unsigned 64-bit integer | [System.UInt64](https://docs.microsoft.com/en-us/dotnet/api/system.uint64) |
| nint | Depends on platform | Signed 32-bit or 64-bit integer | [System.IntPtr](https://docs.microsoft.com/en-us/dotnet/api/system.intptr) |
| nuint | Depends on platform | Unsigned 32-bit or 64-bit integer | [System.UIntPtr](https://docs.microsoft.com/en-us/dotnet/api/system.uintptr) |

**Serialization**

The process of storing the state of objects to another format, such as XML,JSON is called Serialization