**Number**

Challenge Task 1 of 1

Let's practice type conversion. Define a Divide method that takes two int values as parameters, and returns the result of dividing them as a double value. For example, Divide(5, 2) should return 2.5. (Normally, a method like this should probably take double parameters, but we want to practice converting values from int to double.)

You'll need to convert each of the int values to double before performing the division operation, so you don't lose the fractional portion of the result.

**Solution:**

using System;

class Program

{

// YOUR CODE HERE: Define a Divide method!

static double Divide(int firstNumber, int secondNumber){

double result = (double)firstNumber/(double)secondNumber;

return result;

}

static void Main(string[] args)

{

// This should print "2.5".

Console.WriteLine(Divide(5, 2));

// This should print "3.3333333333..."

// (Or a value close to that, since it can't be

// infinitely precise.)

Console.WriteLine(Divide(10, 3));

}}

**Method**

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.

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.

**Class/Objects**

// 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!");

}

static void Main(string[] args)

{

Car **myObj** = new Car();

myObj.fullThrottle(); // Call the method

}

}

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**

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.

// 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.

## **Access Modifiers**

We are quite familiar with the public, private, protected, internal keyword.

## **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 a private field

**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 (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)

{

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);

}

}

## **The sealed Keyword**

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

## **Polymorphism and Overriding Methods**

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

[**Inheritance**](https://www.w3schools.com/cs/cs_inheritance.asp) 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.

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();

}

}

#### **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.

**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 otherhand, Instance class assign the value first and then called from other class.