

About the Course

ProgrammingAdvices.com

مميزات الكورس

- مبادئ ومفاهيم البرمجة الشيئية\الكائنية من ارض الواقع باستخدام سي شارپ
- يقدم هذا الكورس مهارات البرمجة المتقدمة ويركز على المفاهيم الأساسية في البرمجة الشيئية\الكائنية باستخدام لغة سي شارپ
- تمثل البرمجة الموجهة للأشياء \ للكائنات تكامل مكونات البرامج في بنية برمجية واسعة النطاق للبرامج الكبيرة.
- تطوير البرمجيات بهذه الطريقة يمثل الخطوة المنطقية التالية بعد تعلم أساسيات البرمجة الوظيفية ، مما يسمح بامتداد إنشاء برامج كبيرة مترامية الأطراف بكل سلاسة ويسر.
- تركز الدورة على الفهم العملي والعمل العميق للتمكن من المفاهيم والأساليب للبرمجة الشيئية\الكائنية ومناقشة مواضيع متقدمه فيها

OOP As It Should Be In



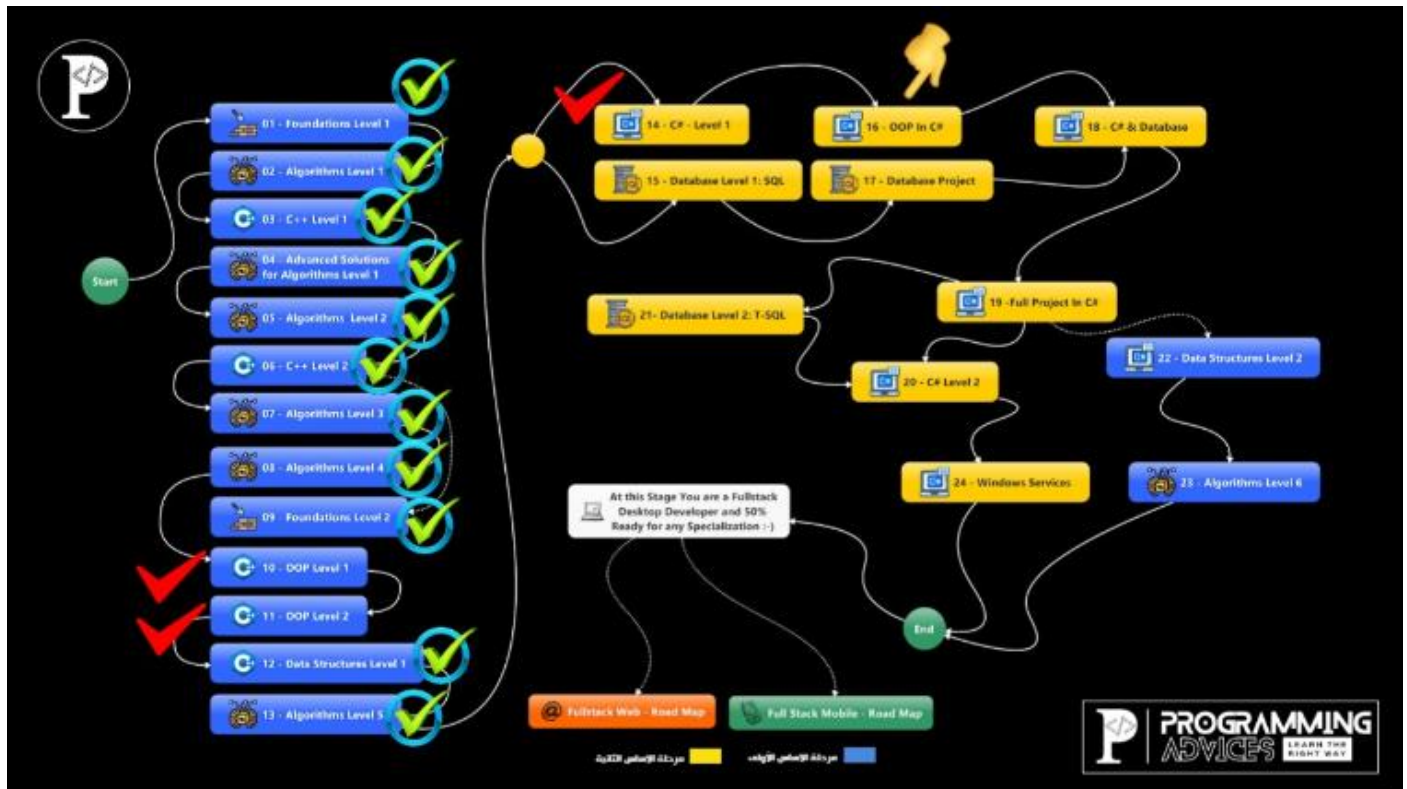
Concepts & Ways

Mohammed Abu-Hadoud

ProgrammingAdvices.com



كل التوفيق للجميع
محمد ابوهدود



Telegram Group for This Course

<https://t.me/+vecOqLMypb9kYzRk>

Important Introduction

راجع المفاهيم بتاعت ال oop بتاعت ال c++ الاول
هناك هنا ال class library اللي هوا عبارة عن مكتب بتقدر تستخدمها في بناء اي تطبيق سواء ويب
او موبايل او desktop بكود واحد يتكتب مره واحده

Class & Object

(Revision) What is Object Oriented Programming? And Why?

<https://programmingadvices.com/courses/16-oop-in-c/lectures/46306928>

مراجعته علي درس ال oop وازاي هيا بتسهل البرمجه

Class & Object

مبادئ ال oop واحده مهما انتقلت من لغة للغة

عندك هنا هوا بيقولك انه ال oop بياخد ال functions وبيصنفها في class بحيث انك ماتتهوش
وانت بتستخدمها وان الكلاس هوا عبارة عن data type لازم عشان تستخدمها بتعرف متغير من نوعها
والمتغير ده اسمه object
نفس الشرح اللي كان في ال c++

Functional (FP) vs Object Oriented (OOP)

University System Example (FP)

- AddStudent
- UpdateStudent
- DeleteStudent
- CalculateAverage
- AddCourse
- UpdateCourse
- DeleteCourse
- EnrollStudentInCourse
- UnEnrollStudentFromCourse
- HowManyStudentsInCourse
- Doctor (Add, Edit, Delete)
- AssignCourseToDoctor
- SendEmailToStudent
- SendTextMessageToStudent
- SendEmailToDoctor
- SendTextMessageToDoctor
- CallStudent
- CallDoctor
- AddEmployee
- UpdateEmployee
- DeleteEmployee
- CalculateSalary
- PaySalary
-

Simply

You can have thousands of
functions in your
System!!!!



What is Class? Why we call it Class?

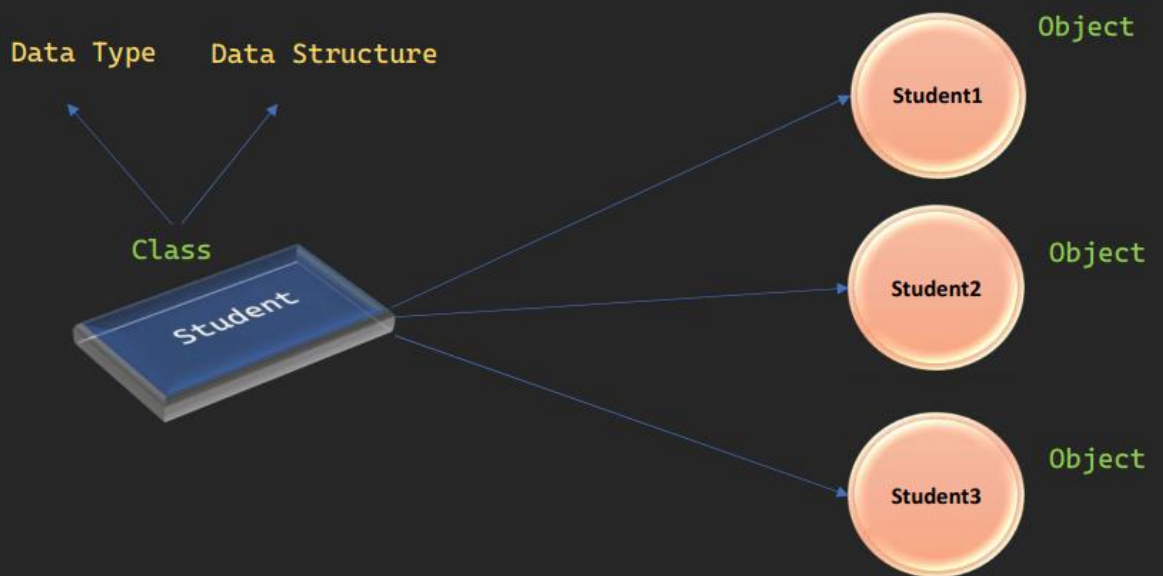
University System Example (OOP)



Simply
Class came from
Classification



What is Object?



تعالیٰ نعمل console application بال c#

Visual Studio 2022

Open recent

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

Get started



Clone a repository

Get code from an online repository like GitHub or Azure DevOps



Open a project or solution

Open a local Visual Studio project or .sln file



Open a local folder

Navigate and edit code within any folder



Create a new project

Choose a project template with code scaffolding to get started

[Continue without code →](#)

Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

Search for templates (Alt+S)

[Clear all](#)

C#

Windows

All project types



WPF User Control Library

A project for creating a user control library for .NET WPF Applications

C# Windows Desktop Library



WPF App (.NET Framework)

Windows Presentation Foundation client application

C# XAML Windows Desktop



JUnit Test Project

A project that contains NUnit tests that can run on .NET on Windows, Linux and MacOS.

C# Linux macOS Windows Desktop Test Web



Console App (.NET Framework)

A project for creating a command-line application

C# Windows Console



Class Library (.NET Framework)

A project for creating a C# class library (.dll)

C# Windows Library



Unit Test Project (.NET Framework)

A project that contains MSTest unit tests.

Back

Next

هنعمل کلاس اسمه cls person

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    class clsPerson {

        //Fields
        public string FirstName;
        public string LastName;

        //Method
        public string FullName() {
            return FirstName+' '+LastName;
        }

        internal class Program
        {
            static void Main(string[] args)
            {
            }
        }
    }
}
```

زي ماقولنا في المستوي اللي فات بتاع ال c# انك عشان تاخذ object من كلاس بتكتب new وبعدين تستدعي ال constructor سواء كان ال constructor ده كان default معمول من قبل ال compiler او constructor انت معرفه

وبعد كده بتشتغل عادي بقي

هنعمل شخصين ونطبعهم

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    class clsPerson {

        //Fields
        public string FirstName;
        public string LastName;

        //Method
        public string FullName() {
            return FirstName+' '+LastName;
        }

        internal class Program
        {
            static void Main(string[] args)
            {
            }
        }
    }
}
```



```

//Create object from class
clsPerson Person1 = new clsPerson();

Console.WriteLine("Accessing Object 1 (Person1):");
Person1.FirstName = "Mohammed";
Person1.LastName = "Abu-Hadhoud";
Console.WriteLine(Person1.FullName());

//Create ANOTHER object from class
clsPerson Person2 = new clsPerson();

Console.WriteLine("Accessing Object 1 (Person1):");
Person2.FirstName = "Ali";
Person2.LastName = "Maher";
Console.WriteLine(Person2.FullName());

Console.ReadLine();
    }
}

```

C# Class and Object

C# is an object-oriented program. In object-oriented programming(OOP), we solve complex problems by dividing them into objects.

To work with objects, we need to perform the following activities:

- create a class
- create objects from the class

C# Class

Before we learn about objects, we need to understand the working of classes. Class is the blueprint for the object.

We can think of the class as a **sketch (prototype) of a house**. It contains all the details about the floors, doors, windows, etc. We can build a house based on these descriptions. **House** is the object.

Like many houses can be made from the sketch, we can create many objects from a class.

Create a class in C#

We use the class keyword to create an object. For example,

```
class ClassName {  
  
}
```

Here, we have created a class named ClassName. A class can contain

- **fields** - variables to store data
- **methods** - functions to perform specific tasks

Let's see an example,

```
class clsPerson  
{  
  
    //Fields  
    public string FirstName;  
    public string LastName;  
  
    //Method  
    public string FullName()  
    {  
        return FirstName + ' ' + LastName;  
    }  
  
}
```

In the above example,

- clsPerson- class name
- FirstName, LastName - fields
- FullName() - method

Note: In C#, fields and methods inside a class are called members of a class.

C# Objects

An object is an instance of a class. Suppose, we have a class clsPerson. Person1, Person2 are objects of the class.

Creating an Object of a class

In C#, here's how we create an object of the class.

```
ClassName obj = new ClassName();
```

Here, we have used the **new** keyword to create an object of the class. And, obj is the name of the object. Now, let us create an object from the clsPerson class.

```
clsPerson Person1= new clsPerson();
```

Now, the Person1 object can access the fields and methods of the clsPerson class.

Access Class Members using Object

We use the name of objects along with the **.** operator to access members of a class. For example,

```
using System;

namespace ConsoleApp1
{

    class clsPerson
    {

        //Fields
        public string FirstName;
        public string LastName;

        //Method
```



```
        public string FullName()
        {
            return FirstName + ' ' + LastName;
        }

    }

    internal class Program
    {
        static void Main(string[] args)
        {
            //Create object from class
            clsPerson Person1= new clsPerson();

            Console.WriteLine("Accessing Object 1 (Person1):");
            Person1.FirstName = "Mohammed";
            Person1.LastName = "Abu-Hadhoud";
            Console.WriteLine(Person1.FullName());

            //Create another object from class
            clsPerson Person2 = new clsPerson();
            Console.WriteLine("\nAccessing Object 2 (Person2):");
            Person2.FirstName = "Ali";
            Person2.LastName = "Maher";
            Console.WriteLine(Person2.FullName());

            Console.ReadKey();

        }
    }
}
```

Output

Accessing Object 1 (Person1):

Mohammed Abu-Hadhoud

Accessing Object 2 (Person2):

Ali Maher

In the above program, we have created two objects named Person1, Person2 from the clsPerson class. Notice that we have used the object name and the `.` (dot operator) to access the fields

```
// access fields of Person1
Person1.FirstName = "Mohammed";
Person1.LastName = "Abu-Hadhoud";
```

and the FullName() method

```
// access method of the Person1
Person1.FullName();
```

Why Objects and Classes?

Objects and classes help us to divide a large project into smaller sub-problems, and have control over the code, dealing with classes and objects will make our life easier and we don't have to remember anything. classes will increase code reusability and will make it easier to maintain.

This helps to manage complexity as well as make our code reusable.

الواجب

What is Class?

Class is the blue-print of object

Class is a Datatype

Class is the same as object

Class is a Data-structure

You can have multiple objects from the same class.

True

False

Any Function or Procedure inside class is called "Method".

True

False

Object is an Instance of class.

True

False

C# is an Object Oriented Language.

True

False

Class members means any variable or function inside the class is called "Member".

True

False

Data Member is any variable inside the class that holds data.

True

False

Function Member is any function or procedure inside a class.

True

False

Class Members are Data Members and Function Members

True

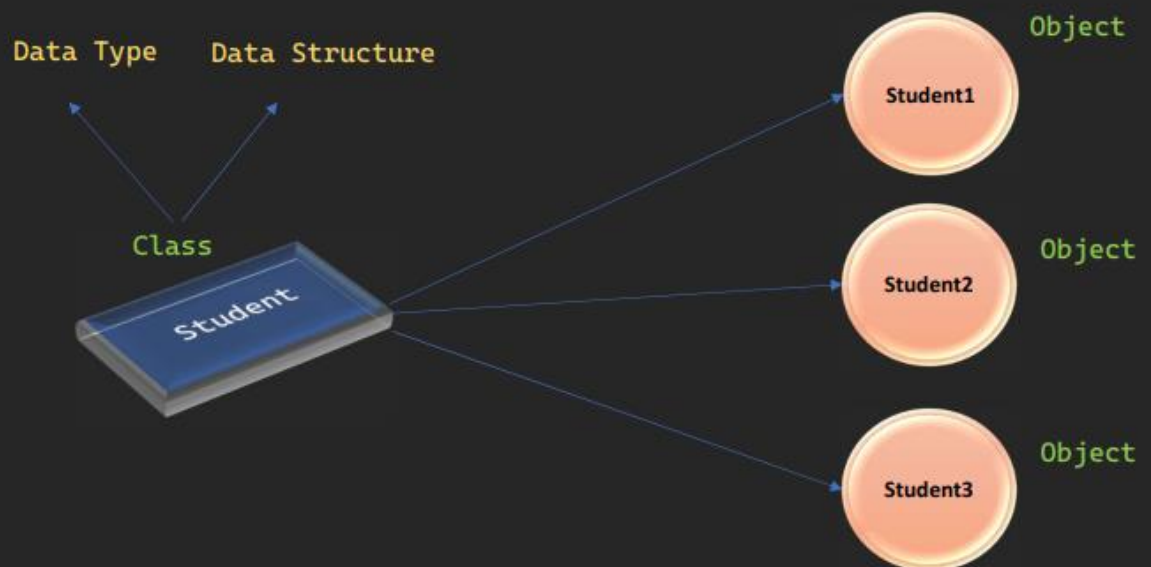
False

Objects In Memory

هنا برضه بيرجع يأكداك انك لما بتيجي تعمل object من كلاس معين بيخزن بس ال data members بتاعت كل object وده لانه كل object ليه القيم بتاعته

انما ال functions بتتخزن مره واحده وبيتم مشاركتها بين كل ال objects لانها عباره عن عمليات بتتم عال objects ومش بتخزن داتا

What is Object?

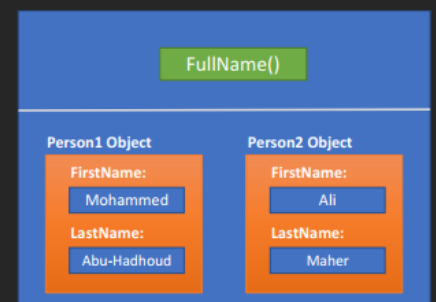


Each instance has its own space in memory, Only Member functions are shared among all objects



```
class clsPerson
{
    //Fields
    public string FirstName;
    public string LastName;

    //Method
    public string FullName()
    {
        return FirstName + ' ' + LastName;
    }
}
```



الواجب

Function Members are shared to all objects in memory and has one memory space for them.

True

False

Every Object has it's own space in memory that hold both Data /
Function Members.

True

False

Every Object has it's own space in memory that holds only Data
Members.

True

False

Access Modifiers

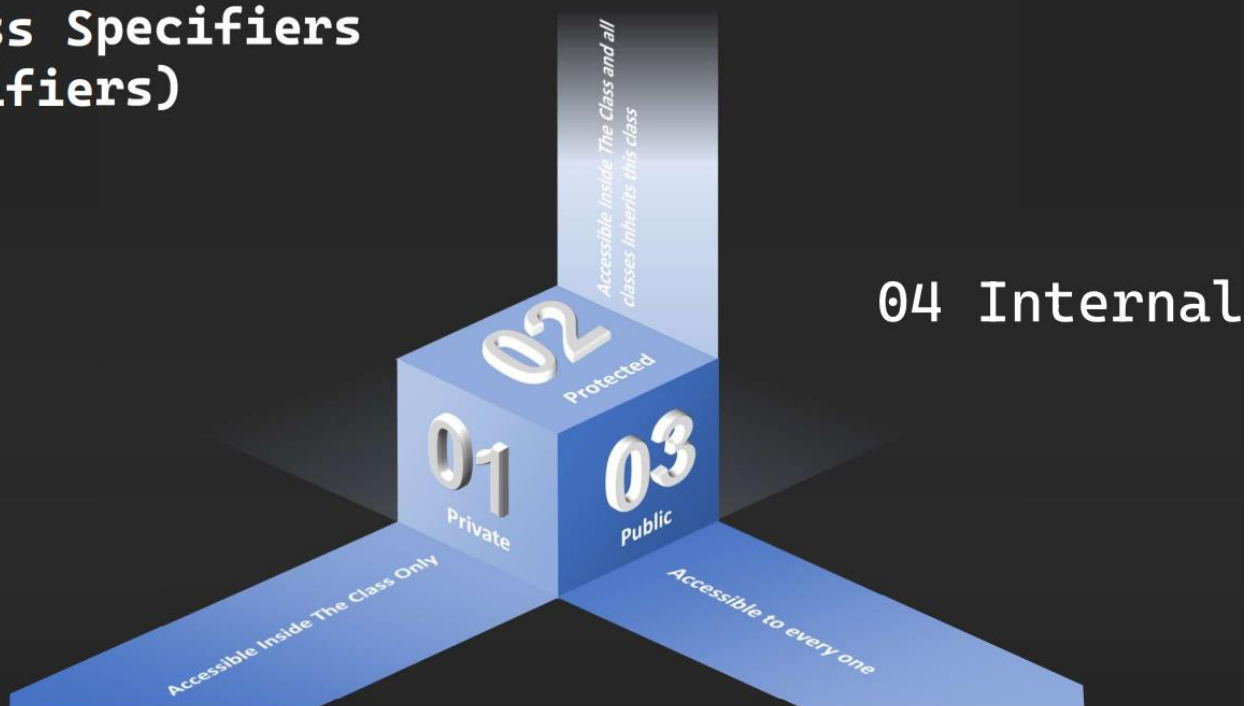
اللي هما ال public وال private و ال protected

وال c# فيها نوع رابع اسمه internal مش هيشرحه دلوقتي

زي ماقولنا قبل كده ال public ده تقدر تشوفه من بره الكلاس وال private بتشوفه من جوه الكلاس
بس وال protected بتشوفه من جوه ومن الوراثة

ال internal بتقدر تشوفه من جوه ال assembly وماتقدرش تشوفه لو انت في assembly ثاني
وال assembly ده مجموعه classes موجودين في dll واحد هنشوفه بعدين

Access Specifiers (Modifiers)




```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    class clsA {

        public int x1 = 10;
        private int x2 = 20;
        protected int x3 = 30;

        public int fun1() { return 100; }
        private int fun2() { return 200; }
        protected int fun3() { return 300; }

    }

    class clsB :clsA{
        public int fun4() { return x1 + x3; }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            clsA A=new clsA();

            Console.WriteLine("all public members are accessible");
            Console.WriteLine("x1={0}", A.x1);
            //Console.WriteLine("x2={0}", A.x2);
            //Console.WriteLine("x3={0}", A.x3);

            Console.WriteLine("result of fun1{0}", A.fun1());
            //Console.WriteLine("result of fun2{0}", A.fun2());
            //Console.WriteLine("result of fun3{0}", A.fun3());

            clsB B = new clsB();
            Console.WriteLine("\nObjects from class B expose all public memembers from class
A");

            Console.WriteLine("x1={0}", B.x1);
            //Console.WriteLine("x2={0}", B.x3);
            //Console.WriteLine("x3={0}", B.x3);

            Console.WriteLine("result of fun1{0}", B.fun1());
            Console.WriteLine("result of fun4{0}", B.fun4());
            // Console.WriteLine("result of fun2{0}", B.fun2());
            // Console.WriteLine("result of fun3{0}", B.fun3());

            Console.ReadLine();
        }
    }
}

```

C# Access Modifiers

In C#, access modifiers specify the accessibility of types (classes, interfaces, etc) and type members (fields, methods, etc).

Access modifiers, are used to set the access level/visibility for classes, fields, methods and properties.

For example,

```
using System;

namespace AccessModifiers
{
    class clsA
    {
        public int x1 = 10;
        private int x2 = 20;
        protected int x3 = 30;

        public int fun1()
        {
            return 100;
        }

        private int fun2()
        {
            return 200;
        }

        protected int fun3()
        {
            return 300;
        }
    }

    class clsB : clsA
    {
        public int fun4()
```

```

    {
        //x1 is public and x3 is protected in the base class so you can access them.
        //You cannot access any private members of the base class.
        return x1 + x3 ;
    }

}

internal class Program
{
    static void Main(string[] args)
    {
        //Create object from class
        clsA A = new clsA();

        //all public members are accessible and internal
        Console.WriteLine("All public members are accessible");
        Console.WriteLine("x1={0}" , A.x1);
        Console.WriteLine("result of fun1={0}", A.fun1());

        //you cannot access private members in the folling line.
        //Console.WriteLine("x2={0}", A.x2);

        //you cannot access protected members in the folling line.
        // Console.WriteLine("x3={0}", A.x3);

        //you cannot access private members in the folling line.
        // Console.WriteLine("result of fun2={0}", A.fun2());

        //you cannot access protected members in the folling line.
        // Console.WriteLine("result of fun3={0}", A.fun3());

        clsB B = new clsB();
        Console.WriteLine("\nObjects from class B expose all public members from the base
class");

        Console.WriteLine("x1={0}", B.x1);
        Console.WriteLine("result of fun1={0}", B.fun1());

        //you cannot access private members in the folling line.

```

```

        //Console.WriteLine("x2={0}", B.x2);
        // Console.WriteLine("result of fun1={0}", B.fun2());

        //you cannot access protected members in the folling line.
        // Console.WriteLine("x3={0}", B.x3);
        //Console.WriteLine("result of fun3={0}", B.fun3());

        Console.ReadKey();
    }
}
}

```

Types of Access Modifiers

In C#, there are 4 basic types of 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. You will learn more about inheritance in a later chapter
- **internal** : The code is only accessible within its own assembly (dll), but not from another assembly. **internal is equivalent to friend in c++.**

Public/Private access modifier

When we declare a type or type member **public**, it can be accessed from anywhere. When we declare a type member with the private access modifier, it can only be accessed within the same class or struct.

Protected access modifier

When we declare a type member as **protected**, it can only be accessed from the same class and its derived classes (the classes that inherits myclass) .

Internal access modifier

When we declare a type or type member as **internal**, it can be accessed only within the same assembly.

An assembly is a collection of types (classes, interfaces, etc) and resources (data). They are built to work together and form a logical unit of functionality.

That's why when we run an assembly all classes and interfaces inside the assembly run together.

Example: internal within the same Assembly

```
using System;

namespace Assembly {

    class Student {
        internal string name = "Mohammed Abu-Hadhoud";
    }

    class Program {
        static void Main(string[] args) {

            // creating object of Student class
            Student theStudent = new Student();

            // accessing name field and printing it
            Console.WriteLine("Name: " + theStudent.name);
            Console.ReadLine();
        }
    }
}
```

Output

Name: Mohammed Abu-Hadhoud

In the above example, we have created a class named Student with a field name. Since the field is **internal**, we are able to access it from the Program class as they are in the same assembly.

If we use **internal** within a single assembly, it works just like the **public** access modifier.

Summary:

Keyword	Description
public	Public class is visible in the current and referencing assembly.
private	Visible inside the current class.
protected	Visible inside the current and derived class.
Internal	Visible inside containing assembly.
Internal protected	Visible inside containing assembly and descendent of the current class.

There's also two combinations: **protected internal** and **private protected**.

For now, let's focus on **public** and **private** and **protected** modifiers.

الواجب

Which of the following is Access Specifiers/Modifiers:

Public

Private

Protected

Internal

Constant

Can you access class members and methods directly?

Yes

No, You have to declare an object of the class first, and access all members and methods through the object not class.

How to access member function of class using Object ?

FunctionName();

Class.FunctionName();

ObjectName.FunctionName();

None of the above.

If you have a private member or method in class, can you access this member or method through Object?

Yes, it can be accessed

No, only public members and methods can be accessed through the object, all private members and methods are for internal use inside the class

Access modifiers (or access specifiers) are keywords in object-oriented languages that set the accessibility of classes, methods, and other members.

True

False

Public Members can be accessed from inside and outside the class.

True

False

Private Members can be accessed from outside the class through object.

True

False

Private Members can be accessed by any class inherits the current class.

True

False

Private Members can be accessed only from inside the class, it cannot be accessed from outside the class nor from the classes inherits the current class..

True

False

If you want to have a member that is private to outside class and public to classes inherits the current class, which access specifier/modifier you use?

Public

Private

Protected

Protected Members can be accessed from outside class through objects.

True

False

Protected Members can be accessed from inside class and from all classes inherits the current class.

True

False

OOP is more secured because you can hide members from developers.

True

False

Inside the class I can access everything including Public, Private , and Protected Members.

True

False

When we declare a type or type member as internal, it can be accessed only within the same assembly.

True

False

If we use internal within a single assembly, it works just like the public access modifier.

True

False

internal in C# is equivalent to friend in c++;

True

False

Static Members

Static اعراف انه معناها مشترك يعني مشترك

فاكر لما قولتلك انه في الذاكرة بيحط المتغيرات بتاعت كل object لوحدها وبيخزن ال functions مر واحده عشان بتكون مشتركه بين كل ال objects ؟

حط المتغيرات اللي من النوع static مع ال functions

يعني ايه ؟

يعني المتغيرات اللي من النوع static بتكون قيمتها مشتركة بين كل ال object ومايقدرش اي object يعدل فيها بعكس ال ++c بتعدلها عن طريق الكلاس نفسه وبالتالي لو جت استدعيتها من اي object تاني هتظهرلك بالقيمة المعدله كانه اكثر من يوزر داخلين علي داتا بيز واحده وبishtglaw عليها

طيب بالنسبة لل function هيا نفسها بتكون مشتركة بين كل ال object لو كتبت static جنبها هيحصل ايه ؟

لو عملت كده اصبحت ال function لايمكن استدعائها من خلال object واستدعائها بيكون عن طريق الكلاس فقط وماتقدرش تعدل غير علي متغيرات من النوع static وبس

بص الكود ده

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    class clsA
    {
        public int x1;
        public static int x2;

        public int Method1() { return x1 + x2; }

        public static int Method2()
        {
            //return clsA.x1+x2;
            return x2;
        }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            clsA objA1 = new clsA();
            clsA objA2 = new clsA();
            objA1.x1 = 7;
            objA2.x1 = 10;

            clsA.x2 = 100;

            Console.WriteLine("objA1.x1={0}", objA1.x1);
            Console.WriteLine("objA2.x1={0}", objA2.x1);
            Console.WriteLine("objA1.method1 result={0}", objA1.Method1());
            Console.WriteLine("objA2.method1 result={0}", objA2.Method1());

            Console.WriteLine("static method2 result={0}", clsA.Method2());

            Console.WriteLine("static x2={0}", clsA.x2);

            Console.ReadLine();
        }
    }
}
```

Static Members

C# supports two types of class methods: static and nonstatic. Any normal method is a nonstatic method.

A static method in C# is a method that keeps only one copy of the method at the Type level, not the object level. The last updated value of the method is shared among all objects of that Type. That means all class instances share the exact copy of the method and its data.

Look at the following example.

Each instance has its own space in memory, Only Member functions are shared among all objects, All static members are shared and can be accessed through the class name

```
class clsA
{
    public int x1;
    public static int x2;

    public int Method1 ()
    {
        //not static methods can always access the static members
        return x1 + x2;
    }

    public static int Method2()
    {
        return x2;
    }
}
```

Class clsA



using System;

```
class clsA
{
    public int x1;
    //x2 is shared for all object because it's on the class level
    public static int x2;

    public int Method1 ()
    {
        //not static methods can always access the static members
    }
}
```



```

        return x1 + x2;
    }

    public static int Method2()
    {
        //static methods cannot access non-static members because there is no object
        //static methods are called at the class level.
        //return clsA.x1 + x2;
        return x2;
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        //Create an object of Employee class.
        clsA objA1 = new clsA();
        clsA objA2 = new clsA();

        objA1.x1 = 7;
        objA2.x1 = 10;
        //x2 is shared for all object because it's on the class level, you can access it
        //using the class name.
        clsA.x2 = 100;

        Console.WriteLine("objA1.x1:={0}", objA1.x1);
        Console.WriteLine("objA2.x1:={0}", objA2.x1);
        Console.WriteLine("objA1.method1 results:={0}", objA1.Method1());
        Console.WriteLine("objA2.method1 results:={0}", objA2.Method1());

        //Method 2 cannot be accessed through object, only through the class itself.
        // Console.WriteLine(objA1.Method2());
        Console.WriteLine("static method2 results:={0}",clsA.Method2());

        Console.WriteLine("static x2:={0}", clsA.x2);
        Console.ReadLine();
    }
}

```

```
}  
  
}
```

x2 is also a static Field that is saved on the class level.

Note: remember that all static methods and properties are shared for all objects because they are saved at the class level not at the object level.

Properties Set and Get

هنا بيقولك انه فيه طريقه مختصره عشان تعمل بيها ال **get** وال **set** للمتغيرات الطريقه باختصار انك بتعرف متغير من النوع **public** وبعدين تفتح قوسين من دول {} وجواهم تكتب **get** وتفتح قوسين وترجع المتغير وبعدين تكتب **set** وتقول ان المتغير اللي عايز تعمله **set** يساوي المتغير اللي عرفته قبل الاقواس زي كده

```
class clsEmployee  
{  
    private int _ID;  
    private string _Name=String.Empty;  
  
    public int ID {  
        get { return _ID; }  
        set { _ID = value; }  
    }  
  
    public string Name {  
        get { return _Name; }  
        set { _Name = value; }  
    }  
}
```

وبيقولك فيه طريقه اسهل من كده كمان هيقلها لك بعدين
وده الكود كله

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace ConsoleApp2  
{  
  
    class clsEmployee  
    {  
        private int _ID;  
        private string _Name=String.Empty;  

```

```

    public int ID {
        get { return _ID; }
        set { _ID = value; }
    }

    public string Name {
        get { return _Name; }
        set { _Name = value; }
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        clsEmployee Employee1 = new clsEmployee();
        Employee1.ID = 7;
        Employee1.Name = "Mohammed Abu-Hadhoud";

        Console.WriteLine("Employee ID ={0}", Employee1.ID);
        Console.WriteLine("Employee Name ={0}", Employee1.Name);

        Console.ReadLine();
    }
}

```

Properties Get and Set

Properties provide a flexible mechanism to read, write, validate or compute a private field. You can also use public fields in properties, but if we use a public field in a property then anybody can access our field in a program. A property makes our field secure, and we can change our rule (property) in one location, and it is easy to use anywhere.

Look at the following example.

```

using System;

class clsEmployee
{
    // Private fields
    private int _ID;
    private string _Name = string.Empty;
}

```

```
//ID Property Declaration
public int ID
{
    //Get is use for Reading field
    get
    {
        return _ID;
    }

    //Set is use for writing field
    set
    {
        _ID = value;
    }
}

//Name Property Declaration

public string Name
{
    //Get is use for Reading field
    get
    {
        return _Name;
    }

    //Set is use for writing field
    set
    {
        _Name = value;
    }
}

static void Main(string[] args)
{
    //Create an object of Employee class.
```

```

        clsEmployee Employee1 = new clsEmployee();

        Employee1.ID = 7;
        Employee1.Name = "Mohammed Abu-Hadhoud";

        Console.WriteLine("Employee Id:={0}", Employee1.ID);
        Console.WriteLine("Employee Name:={0}", Employee1.Name);
        Console.ReadLine();

    }

}

```

In the preceding example, we have the following two private fields:

1. `_ID (int)`
2. `_Name (string)`

And we have two properties, `Id` and `Name`. When a property is created we have the two methods `Get` and `Set`. `Get` is for reading the value and `Set` is for writing the value for a field.

ReadOnly Properties

فكرة ال read only property انك بتعمل المتغير private وبعدين تعمله get من غير set زي كده

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    class clsEmployee
    {
        private int _ID;
        private string _Name=String.Empty;

        public int ID {
            get { return _ID; }
        }
    }
}

```

```

    }

    public string Name {
        get { return _Name; }
        set { _Name = value; }
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        clsEmployee Employee1 = new clsEmployee();
        //Employee1.ID = 7;
        Employee1.Name = "Mohammed Abu-Hadhoud";

        Console.WriteLine("Employee ID ={0}", Employee1.ID);
        Console.WriteLine("Employee Name ={0}", Employee1.Name);

        Console.ReadLine();
    }
}
}

```

ReadOnly Properties

You can define a readonly property by only implementing the get method.

Look at the following example.

```

using System;

class clsEmployee
{
    // Private fields
    private int _ID;
    private string _Name = string.Empty;

    //ID Property Declaration as readonly
    public int ID
    {
        //Get is use for Reading field
        get
        {
            return _ID;
        }
    }
}

```



```

    }
}

//Name Property Declaration

public string Name
{
    //Get is use for Reading field
    get
    {
        return _Name;
    }

    //Set is use for writing field
    set
    {
        _Name = value;
    }
}

static void Main(string[] args)
{
    //Create an object of Employee class.

    clsEmployee Employee1 = new clsEmployee();
    // You cannot modify the id value because it's readonly
    // Employee1.ID = 7;
    Employee1.Name = "Mohammed Abu-Hadhoud";

    Console.WriteLine("Employee Id:={0}", Employee1.ID);
    Console.WriteLine("Employee Name:={0}", Employee1.Name);
    Console.ReadLine();

}
}

```

In the preceding example, we have the following two private fields:

1. `_ID` (int)
2. `_Name` (string)

Note that property `ID` has only get method therefore it's read only.

Auto Implemented Properties

هنا بيقلوك ان الطريقة المختصره لل `get` وال `set` وهيا نفس الطريقة اللي فاتت بس بتكتب `get` و `set` بس ومن غير ماتعرف متغير وتخليه `private` بس بالطريقة دي مش هتقدر تحط عمليات تانيه زي انه يسجل تاريخ عملية التعديل مثلا زي كده

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    class clsEmployee
    {
        public int ID {
            get;
            set;
        }

        public string Name {
            get;
            set;
        }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            clsEmployee Employee1 = new clsEmployee();
            Employee1.ID = 7;
            Employee1.Name = "Mohammed Abu-Hadhoud";

            Console.WriteLine("Employee ID ={0}", Employee1.ID);
            Console.WriteLine("Employee Name ={0}", Employee1.Name);
        }
    }
}
```

```
        Console.ReadLine();  
    }  
}
```

Auto Implemented Properties

You can define a readonly property by only implementing the get method.

Look at the following example.

```
using System;  
  
class clsEmployee  
{  
  
    //ID Property  
    public int ID  
    {  
        get;  
        set;  
    }  
  
    //Name Property Declaration  
    public string Name  
    {  
        get;  
        set;  
    }  
  
    static void Main(string[] args)  
    {  
        //Create an object of Employee class.  
        clsEmployee Employee1 = new clsEmployee();  
  
        Employee1.ID = 7;  
        Employee1.Name = "Mohammed Abu-Hadhoud";  
  
        Console.WriteLine("Employee Id:={0}", Employee1.ID);  
    }  
}
```

```
        Console.WriteLine("Employee Name:={0}", Employee1.Name);
        Console.ReadLine();
    }
}
```

Look at the above example. There are no implementations in the get and set methods. And also you don't need to create private fields. So Auto implemented properties are helpful, when you don't think you need any validation, computation or any implementation.

Static Properties & Static Class

ال static class بتعمله لو عايز تجمع كل ال class members اللي نوعهم static والكلاس اللي من النوع static ماينفعش تاخذ منه object

مثلا عايز اعمل كلاس بيتحط فيه ال settings بتاعت التطبيق

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    static class Settings
    {
        public static int DayNumber {
            get { return DateTime.Today.Day; }
        }

        public static string DayName {
            get { return DateTime.Today.DayOfWeek.ToString(); }
        }

        public static string ProjectPath {
            set;
            get;
        }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Settings.DayNumber);
            Console.WriteLine(Settings.DayName);

            Settings.ProjectPath = @"C:\MyProjects\";
            Console.WriteLine(Settings.ProjectPath);

            Console.ReadKey();
        }
    }
}
```

```
}  
}
```

Static Class

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the new operator to create a variable of the class type.

Static Properties

A **static property** is similar to a static method. It uses the composite name to be accessed. Static properties use the same get and set tokens as instance properties. They are useful for abstracting global data in programs.

Look at the following example.

```
using System;  
  
static class Settings  
{  
    public static int DayNumber  
    {  
        get  
        {  
            return DateTime.Today.Day;  
        }  
    }  
  
    public static string DayName  
    {  
        get  
        {  
            return DateTime.Today.DayOfWeek.ToString();  
        }  
    }  
  
    public static string ProjectPath  
    {
```

```

        get;
        set;
    }
}

class Program
{
    static void Main()
    {
        //
        // Read the static properties.
        //
        Console.WriteLine(Settings.DayNumber);
        Console.WriteLine(Settings.DayName);
        //
        // Change the value of the static bool property.
        //
        Settings.ProjectPath = @"C:\MyProjects\";
        Console.WriteLine(Settings.ProjectPath);
        Console.ReadKey();

    }
}

```

Note: there is no need to have an object from the class to access static properties.

First Principle or Concept of OOP- Encapsulation

مراجعہ علی درس ال encapsulation وان الكلاس اصبح زي الكبسوله بتاعت الدواء جزء فيه ال methods وجزء فيه المتغيرات وبكده ماتقدرش توصل لأي member من غير ماتعمل object

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.

Encapsulation is defined as the process of enclosing one or more items within a physical or logical package'. Encapsulation, in object oriented programming methodology, prevents access to implementation details.

Encapsulation is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member. C# supports the following access specifiers –

- Public
- Private
- Protected
- Internal
- Protected internal

In normal terms Encapsulation is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.

True

False

Second Principle or Concept of OOP - Abstraction

مراجعہ علی درس ال abstraction وانہ الی ہیستخد ال function او العنصر فی التطبيق
مایکونش لیہ آی علاقہ بال function دی بتشتغل ازای وده غیر ال abstract class

Abstraction is an important part of object oriented programming. It means that only the required information is visible to the user and the rest of the information is hidden.

Abstraction is an important part of object oriented programming. It means that only the required information is visible to the user and the rest of the information is hidden.

True

False

Calculator Project

Requirements

هنعمل الة حاسبه

لو اليوزر خلي القسمه تكون علي صفر خليه انت تكون علي 1

```
1  using System;
2  class clsCalculator
74
75  internal class Program
76  {
77      static void Main(string[] args)
78      {
79
80          clsCalculator Calculator1= new clsCalculator();
81
82
83
84          Calculator1.Clear();
85
86          Calculator1.Add(10);
87          Calculator1.PrintResult();
88
89          Calculator1.Add(100);
90          Calculator1.PrintResult();
91
92          Calculator1.Subtract(20);
93          Calculator1.PrintResult();
94
95          Calculator1.Divide(0);
```



```

Calculator1.PrintResult();

Calculator1.Divide(2);
Calculator1.PrintResult();

Calculator1.Multiply(3);
Calculator1.PrintResult();

Calculator1.Clear();
Calculator1.PrintResult();

Console.ReadLine();

```

C:\Users\Acer\Dropbox\Personal\Advices\2021\16 OOP C#\Projects\

```

Result After Adding 10 is : 10
Result After Adding 100 is : 110
Result After Subtracting 20 is : 90
Result After Dividing 0 is : 90
Result After Dividing 2 is : 45
Result After Multiplying 3 is : 135
Result After Clear 0 is : 0

```

Solution

هنا يقولك انه حقق مبدأ ال abstraction لانه ماوجعش دماغ اليوزر بتفاصيل ماتخصصهوش
وحقق مبدأ ال encapsulation بانه مفيش أي method مرميه في الشارع

```

using System;
class clsCalculator
{
    private float _Result = 0;
    private float _LastNumber = 0;
    private string _LastOperation = "Clear";

    private bool _IsZero(float Number)
    {
        return (Number == 0);
    }

    public void Add(float Number)
    {
        _LastNumber = Number;
        _LastOperation = "Adding";
    }
}

```

```

        _Result += Number;
    }

    public void Subtract(float Number)
    {
        _LastNumber = Number;
        _LastOperation = "Subtracting";

        _Result -= Number;
    }

    public bool Divide(float Number)
    {
        bool Succeeded = true;
        _LastOperation = "Dividing";

        if (_IsZero(Number))
        {
            _LastNumber = Number;
            _Result /= 1;
            Succeeded = false;
        }
        else
        {
            _LastNumber = Number;
            _Result /= Number;
        }

        return Succeeded;
    }

    public void Multiply(float Number)
    {
        _LastNumber = Number;
        _LastOperation = "Multiplying";
        _Result *= Number;
    }

    public float GetFinalResults()
    {
        return _Result;
    }

    public void Clear()
    {
        _LastNumber = 0;
        _LastOperation = "Clear";
        _Result = 0;
    }

    public void PrintResult()
    {
        Console.WriteLine( "Result After {0} {1} is : {2}", _LastOperation , _LastNumber,
        _Result );
    }
};

internal class Program
{
    static void Main(string[] args)
    {
        clsCalculator Calculator1= new clsCalculator();

        Calculator1.Clear();

        Calculator1.Add(10);
    }
}

```

```
Calculator1.PrintResult();

Calculator1.Add(100);
Calculator1.PrintResult();

Calculator1.Subtract(20);
Calculator1.PrintResult();

Calculator1.Divide(0);
Calculator1.PrintResult();

Calculator1.Divide(2);
Calculator1.PrintResult();

Calculator1.Multiply(3);
Calculator1.PrintResult();

Calculator1.Clear();
Calculator1.PrintResult();

Console.ReadLine();

    }
}
```

Constructor and Destructor

Constructor

ال constructor هيا method بنفس اسم الكلاس بيتتم استدعائها لما تيجي تعمل object من الكلاس

C# Constructor

In C#, a constructor is similar to a method that is invoked when an object of the class is created.

However, unlike methods, a constructor:

- has the same name as that of the class
- does not have any return type

Create a C# constructor

Here's how we create a constructor in C#

```
class clsPerson{

    // constructor
    clsPerson() {
```

```
//code  
}  
  
}
```

Here, clsPerson() is a constructor. It has the same name as its class.

Call a constructor

Once we create a constructor, we can call it using the **new** keyword. For example,

```
new clsPerson();
```

In C#, a constructor is called when we try to create an object of a class. For example,

```
clsPerson Person1 = new clsPerson();
```

Here, we are calling the clsPerson() constructor to create an object Person1.

Types of Constructors

There are the following types of constructors:

- Parameterless Constructor
- Parameterized Constructor
- Default Constructor

We will explain everything in the next lessons :-)

Parameter-less Constructor

ال parameter less constructor اللي هوا ال constructor اللي انت تعمله ويكون من غير parameter وتقدر تكتب أي كود فيه

```
using ConsoleApp2;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
  
namespace ConsoleApp2  
{  
  
    class clsPerson {  
        public int Id { get; set; }  
        public string Name { get; set; }  
    }  
}
```

```

        public int Age { get; set; }

        public clsPerson() {
            Id = -1;
            Name = "Empty";
            Age = 0;
        }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            clsPerson Person1 = new clsPerson();

            Console.WriteLine("ID= {0}", Person1.Id);
            Console.WriteLine("Name= {0}", Person1.Name);
            Console.WriteLine("Age= {0}", Person1.Age);

            Console.ReadKey();
        }
    }
}

```

Parameterless Constructor

When we create a constructor without parameters, it is known as a parameterless constructor. For example,

```

using System;

class clsPerson
{

    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }

    public clsPerson()
    {
        Id = -1;
        Name = "Empty";
        Age= 0;
    }
}

internal class Program
{
    static void Main(string[] args)

```

```

{
    clsPerson Person1 = new clsPerson();

    Console.WriteLine("ID:= {0}", Person1.Id);
    Console.WriteLine("Name:= {0}", Person1.Name);
    Console.WriteLine("Age:= {0}", Person1.Age);
    Console.ReadKey();
}
}

```

In the above example, we have created a constructor named clsPerson().

```
new clsPerson();
```

We can call a constructor by adding a **new** keyword to the constructor name.

Parameterized Constructor

Constructor بتحتط فيه parameters عشان تدخل البيانات اللي علي مزاج اليوزر او تطلب منه بيانات معينه اثناء انشاء ال object

```

using ConsoleApp2;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    class clsPerson {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }

        public clsPerson(int Id, string Name, int Age) {
            this.Id = Id;
            this.Name = Name;
            this.Age = Age;
        }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            clsPerson Person1 = new clsPerson(0, "sdfs", 54);

            Console.WriteLine("ID= {0}", Person1.Id);
            Console.WriteLine("Name= {0}", Person1.Name);
            Console.WriteLine("Age= {0}", Person1.Age);

            Console.ReadKey();
        }
    }
}

```

```
}  
}
```

Parameterized Constructor

In C#, a constructor can also accept parameters. It is called a parameterized constructor. For example,

```
using System;  
  
namespace Constructor {  
  
    class Car {  
  
        string brand;  
        int price;  
  
        // parameterized constructor  
        Car(string theBrand, int thePrice) {  
  
            brand = theBrand;  
            price = thePrice;  
        }  
  
        static void Main(string[] args) {  
  
            // call parameterized constructor  
            Car car1 = new Car("Bugatti", 50000);  
  
            Console.WriteLine("Brand: " + car1.brand);  
            Console.WriteLine("Price: " + car1.price);  
            Console.ReadLine();  
  
        }  
    }  
}
```

Output

```
Brand: Bugatti
```

Price: 50000

In the above example, we have created a constructor named Car(). The constructor takes two parameters: theBrand and thePrice.

Notice the statement,

```
Car car1 = new Car("Bugatti", 50000);
```

Here, we are passing the two values to the constructor.

The values passed to the constructor are called arguments. We must pass the same number and type of values as parameters.

Default Constructor

ال default constructor ده اللي ال compiler بيعمله

Default Constructor

If we have not defined a constructor in our class, then the C# will automatically create a default constructor with an empty code and no parameters. For example,

```
using System;
class clsPerson
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}

internal class Program
{
    static void Main(string[] args)
    {
        clsPerson Person1 = new clsPerson();
        Console.WriteLine("ID:= {0}", Person1.Id);
        Console.WriteLine("Name:= {0}", Person1.Name);
        Console.WriteLine("Age:= {0}", Person1.Age);
    }
}
```



```
        Console.ReadKey();

    }
}
```

Output

```
ID:= 0
Name:=
Age:= 0
```

In the above example, we have not created any constructor in the `clsPerson` class. However, while creating an object, we are calling the constructor.

```
clsPerson Person1 = new clsPerson();
```

Here, C# automatically creates a default constructor. The default constructor initializes any uninitialized variable with the default value.

Hence, we get **0** as the value of the numbers and empty string for strings.

Note: In the default constructor, all the numeric fields are initialized to 0, whereas string and object are initialized as null.

Private Constructor vs Static Class

ال private constructor بي عمل نفس اللي بيعمله ال static class وهو انه يمنع أي حد انه ياخذ object من الكلاس

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApp2
{
    class Settings
    {
        private Settings() { }
        public static int DayNumber
        {
            get { return DateTime.Today.Day; }
        }

        public static string DayName
```

```

    {
        get { return DateTime.Today.DayOfWeek.ToString(); }
    }

    public static string ProjectPath
    {
        set;
        get;
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(Settings.DayNumber);
        Console.WriteLine(Settings.DayName);

        Settings.ProjectPath = @"C:\MyProjects\";
        Console.WriteLine(Settings.ProjectPath);

        Console.ReadKey();
    }
}

```

Private Constructor

We can create a private constructor using the `private` access specifier. This is known as a private constructor in C#.

Once the constructor is declared private, **we cannot create objects of the class in other classes.**

Example 1: Private Constructor

```

using System;

class Settings
{
    public static int DayNumber
    {
        get
        {
            return DateTime.Today.Day;
        }
    }

    public static string DayName
    {

```

```

        get
        {
            return DateTime.Today.DayOfWeek.ToString();
        }
    }

    public static string ProjectPath
    {
        get;
        set;
    }

    //this is a private constructor to prevent creating object from this class
    private Settings()
    {

    }
}

class Program
{
    static void Main()
    {

        // You cannot create an object here because class has private constructor
        // Settings Obj1 = new Settings();

        //
        // Read the static properties.
        //
        Console.WriteLine(Settings.DayNumber);
        Console.WriteLine(Settings.DayName);

        //
        // Change the value of the static bool property.
        //
        Settings.ProjectPath = @"C:\MyProjects\";
        Console.WriteLine(Settings.ProjectPath);
        Console.ReadKey();
    }
}

```

```
}  
}
```

In the above example, we have created a private constructor `Settings()`. Since private members are not accessed outside of the class, when we try to create an object of `Settings`

```
// when you try to create an object of this class  
Settings Obj1 = new Settings();
```

we get an error

Note: If a constructor is private, we cannot create objects of the class. Hence, all fields and methods of the class should be declared static, so that they can be accessed using the class name.

Static Class

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, **you cannot use the new operator to create a variable of the class type.**

Example 1: Static Class instead of Private Constructor

```
using System;  
  
static class Settings  
{  
    public static int DayNumber  
    {  
        get  
        {  
            return DateTime.Today.Day;  
        }  
    }  
  
    public static string DayName  
    {  
        get  
        {  
            return DateTime.Today.DayOfWeek.ToString();  
        }  
    }  
}
```

```

    }

    public static string ProjectPath
    {
        get;
        set;
    }
}

class Program
{
    static void Main()
    {
        // You cannot create an object here because class is static
        // Settings Obj1 = new Settings();

        //
        // Read the static properties.
        //
        Console.WriteLine(Settings.DayNumber);
        Console.WriteLine(Settings.DayName);

        //
        // Change the value of the static bool property.
        //
        Settings.ProjectPath = @"C:\MyProjects\";
        Console.WriteLine(Settings.ProjectPath);
        Console.ReadKey();
    }
}

```

Multiple Constructors using overloading

ال constructor overloading زي ه زي ال function overloading انك تعمل اكثر من constructor لنفس الكلاس ب parameters مختلفه

```

using System;

class clsPerson
{

```

```

public int Id { get; set; }
public string Name { get; set; }
public int Age { get; set; }

public clsPerson()
{
    this.Id = -1;
    this.Name = "Empty";
    this.Age = 0;
}

public clsPerson(int Id, string Name, short Age)
{
    this.Id = Id;
    this.Name = Name;
    this.Age = Age;
}
}

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Calling Parameterless Construcor");
        clsPerson Person1 = new clsPerson();
        Console.WriteLine("ID:= {0}", Person1.Id);
        Console.WriteLine("Name:= {0}", Person1.Name);
        Console.WriteLine("Age:= {0}", Person1.Age);

        Console.WriteLine("\n\nCalling Parametarized Construcor");
        clsPerson Person2 = new clsPerson(10, "Mohammed Abu-Hadhoud", 45);
        Console.WriteLine("ID:= {0}", Person2.Id);
        Console.WriteLine("Name:= {0}", Person2.Name);
        Console.WriteLine("Age:= {0}", Person2.Age);

        Console.ReadKey();
    }
}

```

Multiple Constructors

In C#, you can have multiple constructors in the class using overloading For example,

```

using System;

class clsPerson
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }

    public clsPerson()
    {
        this.Id = -1;
        this.Name = "Empty";
        this.Age = 0;
    }
}

```

```

    }

    public clsPerson(int Id, string Name, short Age)
    {
        this.Id = Id;
        this.Name = Name;
        this.Age = Age;
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Calling Parameterless Construcor");
        clsPerson Person1 = new clsPerson();
        Console.WriteLine("ID:= {0}", Person1.Id);
        Console.WriteLine("Name:= {0}", Person1.Name);
        Console.WriteLine("Age:= {0}", Person1.Age);

        Console.WriteLine("\n\nCalling Parametarized Construcor");
        clsPerson Person2 = new clsPerson(10, "Mohammed Abu-Hadhoud", 45);
        Console.WriteLine("ID:= {0}", Person2.Id);
        Console.WriteLine("Name:= {0}", Person2.Name);
        Console.WriteLine("Age:= {0}", Person2.Age);

        Console.ReadKey();
    }
}

```

Static Constructor

ال static constructor بيتنادي مره واحده بس وماينفعش تحطه parameter وماينفعش تستخدمه غير مع كلاس نوعه static وماينفعش الكلاس يكون فيه اكثر من constructor

```

using System;
static class Settings
{
    public static int DayNumber
    {
        get
        {
            return DateTime.Today.Day;
        }
    }
}

```

```

    }

    public static string DayName
    {
        get
        {
            return DateTime.Today.DayOfWeek.ToString();
        }
    }

    public static string ProjectPath
    {
        get;
        set;
    }
    //this is a static constructor will be called once during the program
    static Settings()
    {
        ProjectPath = @"C:\MyProjects\";
    }
}

class Program
{
    static void Main()
    {
        // You cannot create an object here because class is static
        // Settings Obj1 = new Settings();

        //
        // Read the static properties.
        //
        Console.WriteLine(Settings.DayNumber);
        Console.WriteLine(Settings.DayName);
        //
        // Change the value of the static bool property.
        //

        Console.WriteLine(Settings.ProjectPath);

        Console.ReadKey();
    }
}

```

C# Static Constructor

In C#, we can also make our constructor static. We use the `static` keyword to create a static constructor. For example,

```

using System;

static class Settings
{
    public static int DayNumber
    {
        get
        {

```



```

        return DateTime.Today.Day;
    }
}

public static string DayName
{
    get
    {
        return DateTime.Today.DayOfWeek.ToString();
    }
}

public static string ProjectPath
{
    get;
    set;
}

//this is a static constructor will be called once during the program
static Settings()
{
    ProjectPath = @"C:\MyProjects\";
}
}

class Program
{
    static void Main()
    {
        // You cannot create an object here because class is static
        // Settings Obj1 = new Settings();

        //
        // Read the static properties.
        //
        Console.WriteLine(Settings.DayNumber);
        Console.WriteLine(Settings.DayName);
        //
    }
}

```

```

        // Change the value of the static bool property.
        //

        Console.WriteLine(Settings.ProjectPath);

        Console.ReadKey();

    }
}

```

In the above example, we have created a static constructor.

We cannot call a static constructor directly. However, the static constructor gets called automatically.

The static constructor is called only once during the execution of the program. That's why when we call the constructor again, only the regular constructor is called.

Note: We can have only one static constructor in a class. **It cannot have any parameters or access modifiers.**

Destructor

ال destructor ده اللي بيتم استدعائه لما بيتدمر ال object لما تخلص شغل في ال object

```

using System;

class clsPerson
{
    public clsPerson()
    {
        Console.WriteLine("Constructor called.");
    }

    // destructor
    ~clsPerson()
    {
        Console.WriteLine("Destructor called.");
    }

    public static void Main(string[] args)
    {
        //creates object of Person
        clsPerson p1 = new clsPerson();
        Console.ReadKey();
    }
}

```

C# Destructor

In C#, destructor (finalizer) is used to destroy objects of class when the scope of an object ends. It has the same name as the class and starts with a tilde `~`. For example,

```
class Test {  
    ...  
    //destructor  
    ~Test() {  
        ...  
    }  
}
```

Here, `~Test()` is the destructor.

Example: Working of C# Destructor

```
using System;  
  
class clsPerson  
{  
  
    public clsPerson()  
    {  
        Console.WriteLine("Constructor called.");  
    }  
  
    // destructor  
    ~clsPerson()  
    {  
        Console.WriteLine("Destructor called.");  
    }  
  
    public static void Main(string[] args)  
    {  
        //creates object of Person  
        clsPerson p1 = new clsPerson();  
        Console.ReadKey();  
    }  
}
```

```
}  
}  
}
```

In the above example, we have created a destructor `~clsPerson` inside the `clsPerson` class.

When we create an object of the `clsPerson` class, the constructor is called. After the scope of the object ends, object p1 is no longer needed. So, the destructor is called implicitly which destroys object p1.

Features of Destructors

There are some important features of the C# destructor. They are as follows:

- We can only have one destructor in a class.
- A destructor cannot have access modifiers, parameters, or return types.
- A destructor is called implicitly by the Garbage collector of the .NET Framework.
- We cannot overload or inherit destructors.
- We cannot define destructors in structs.

Very Important: Real life Application of Using Static and Constructors.

هنا بيقولك انه لازم لما حد يعمل object يحط فيه بيانات مايسيبهوش فاضي
وانت تقدر تجبره انه يدخل داتا لما يعمل object باستخدام ال constructor
بص كده عالكود ده
انا فيه أجبرت اليوزر انه يدخل الداتا بايده عن طريق ال constructor

```
using System;  
  
class clsPerson  
{  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public int Age { get; set; }  
  
    public string UserName { get; set; }  
    public string PassWord { get; set; }  
  
    public clsPerson(int Id, string Name, short Age)  
    {  
        this.Id = Id;  
        this.Name = Name;  
    }  
}
```

```

        this.Age = Age;
    }

}

internal class Program
{
    static void Main(string[] args)
    {
        clsPerson Person1=new clsPerson(10,"gfh",455);

        Console.ReadKey();
    }
}

```

طيب افرض انا عندي داتا بيز مثلا فيها بيانات الأشخاص وعائز اللي هيستخدم الكلاس مايحطش الداتا بايداه لا انا عايزه يجيبلي ال id ياما يجيبلي اليوزر والباسوورد بتوع الشخص ده وانا في الكلاس بتاعي هدور عليه لو لقيتاه في الداتا بيز هرجعله ال object اللي هوا عاوزه ولو مالقيتهوش ارجعله null

في الحاله بتاعتنا دي ان هعمل method نوعها static عشان اخلي اليوزر يستخدمها من غير object وبترجع object من نفس نوع الكلاس ال function دي وظيفتها انها تدور عالداتا وترجع ال object

طيب اليوزر هيستخدم ال function دي ازاي ؟

قالك انه اليوزر هيكتب نفس الكود اللي بيكتبه عشان يعمل object جديد من أي كلاس بس بدل مايكتب new وبعدها اسم الكلاس هيستدعي ال function اللي احنا عملناها

بص كده عالكود اللي جاي(في ال find احنا عاملين أي كود وخلص كأننا جيبنا فعلا جيبنا الداتا من الداتا بيز لحد مانشوف بعد كده هنستعمل الداتا بيز ازاي)

```

using System;

class clsPerson
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }

    public string UserName { get; set; }
    public string PassWord { get; set; }

    public clsPerson(int Id, string Name, short Age)
    {
        this.Id = Id;
        this.Name = Name;
        this.Age = Age;
    }

    public static clsPerson Find(int ID) {
        if (ID == 10) {

```

```

        return new clsPerson(10, "readf", 21);
    }
    else{ return null; }
}

public static clsPerson Find(string UserName, string Password) {
    if (UserName == "User1" && Password == "p1234") {
        return new clsPerson(10, "readf", 21);
    }
    else {
        return null;
    }
}

}

internal class Program
{
    static void Main(string[] args)
    {
        clsPerson Person1 = new clsPerson(10, "gfh", 455);

        Console.WriteLine("\n\nFinding Person2 By ID");
        clsPerson Person2 = clsPerson.Find(10);
        if (Person2 != null)
        {
            Console.WriteLine("id ={0}", Person2.Id);
            Console.WriteLine("Name={0}", Person2.Name);
            Console.WriteLine("Age={0}", Person2.Age);
        }
        else {
            Console.WriteLine("Couldn't find the person by the given ID");
        }

        Console.WriteLine("\n\nFinding Person3 By UserName and PassWord");
        clsPerson Person3 = clsPerson.Find("dsd", "dsdsd");
        if (Person3 != null)
        {
            Console.WriteLine("id ={0}", Person3.Id);
            Console.WriteLine("Name={0}", Person3.Name);
            Console.WriteLine("Age={0}", Person3.Age);
        }
        else
        {
            Console.WriteLine("Couldn't find the person by the given ID");
        }

        Console.ReadKey();
    }
}

```

Third Principle/Concept of OOP: Inheritance

الوراثه شرحناها قبل كده في ال C++ وهيا عبارته عن انه كلاس بيورث جميع خصائص كلاس تاني وطريقته زيها زي ال C++ بتكتب نقطتين : وبعدهم اسم الكلاس اللي هتورث منه والحاجات اللي بتورثها بتكون public او protected بس والعلاقه بين ال subclass وال superclass هيا اسمها is يعني ال employee is a person

اما تيجي تعرف الكلاس الاب تعمله public لانك لو ماعملت هوش هيبقي private

```

using System;

public class clsPerson
{
    //properties
    public int ID { get; set; }
    public string FirstName { get; set; }
}

```

```

public string LastName { get; set; }
public string Title { get; set; }

//read only property
public string FullName
{
    //Get is use for Reading field
    get
    {
        return FirstName + ' ' + LastName;
    }
}
}

public class clsEmployee : clsPerson
{

    public float Salary { get; set; }
    public string DepartmentName { get; set; }

    public void IncreaseSalaryBy(float Amount)
    {
        Salary += Amount;
    }

}

internal class Program
{
    static void Main(string[] args)
    {

        //Create an object of Employee
        clsEmployee Employee1 = new clsEmployee();

        //the following inherited from base class person
        Employee1.ID = 10;
        Employee1.Title = "Mr.";
        Employee1.FirstName = "Mohammed";
        Employee1.LastName = "Abu-Hadhoud";

        //the following are from derived class Employee
        Employee1.DepartmentName = "IT";
        Employee1.Salary = 5000;

        Console.WriteLine("Accessing Object 1 (Employee1):\n");
        Console.WriteLine("ID := {0}", Employee1.ID);
        Console.WriteLine("Title := {0}", Employee1.Title);
        Console.WriteLine("Full Name := {0}", Employee1.FullName);
        Console.WriteLine("Department Name := {0}", Employee1.DepartmentName);
        Console.WriteLine("Salary := {0}", Employee1.Salary);

        Employee1.IncreaseSalaryBy(100);
        Console.WriteLine("Salary after increase := {0}", Employee1.Salary);
        Console.ReadKey();
    }
}

```

C# Inheritance

In C#, inheritance allows us to create a new class from an existing class. It is a key feature of Object-Oriented Programming (OOP).

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.

The class from which a new class is created is known as the base class (parent or superclass or base class). And, the new class is called derived class (child or subclass or derived class).

The derived class inherits the fields and methods of the base class. This helps with the code reusability in C#.

How to perform inheritance in C#?

In C#, we use the `:` symbol to perform inheritance. For example,

```
class clsPerson{
    // fields and methods
}

// Employee Class Inherits Person
class clsEmployee: Person
{
    // fields and methods of Person are inherited no need to rewrite them
    // fields and methods of Employee
}
```

Here, we are inheriting the derived class Employee from the base class Person. The Employee class can now access the fields and methods of Person class.

Example: C# Inheritance

```
using System;

public class clsPerson
{
    //properties
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Title { get; set; }

    //read only property
    public string FullName
    {
        //Get is use for Reading field
        get
        {
            return FirstName + ' ' + LastName;
        }
    }
}

public class clsEmployee : clsPerson
{
    public float Salary { get; set; }
    public string DepartmentName { get; set; }

    public void IncreaseSalaryBy(float Amount)
    {
        Salary += Amount;
    }
}
```

```

}

internal class Program
{
    static void Main(string[] args)
    {

        //Create an object of Employee
        clsEmployee Employee1 = new clsEmployee();

        //the following inherited from base class person
        Employee1.ID = 10;
        Employee1.Title = "Mr.";
        Employee1.FirstName = "Mohammed";
        Employee1.LastName = "Abu-Hadhoud";

        //the following are from derived class Employee
        Employee1.DepartmentName = "IT";
        Employee1.Salary = 5000;

        Console.WriteLine("Accessing Object 1 (Employee1):\n");
        Console.WriteLine("ID := {0}", Employee1.ID);
        Console.WriteLine("Title := {0}", Employee1.Title);
        Console.WriteLine("Full Name := {0}" , Employee1.FullName);
        Console.WriteLine("Department Name := {0}", Employee1.DepartmentName);
        Console.WriteLine("Salary := {0}", Employee1.Salary);

        Employee1.IncreaseSalaryBy(100);
        Console.WriteLine("Salary after increase := {0}", Employee1.Salary);
        Console.ReadKey();
    }
}

```

Output

```
Accessing Object 1 (Employee1):
```

```
ID := 10
Title := Mr.
Full Name := Mohammed Abu-Hadhoud
Department Name := IT
Salary := 5000
Salary after increase := 5100
```

In the above example, we have derived a subclass Employee from the superclass Person. Notice the statements,

```
Employee1.ID = 10;
Employee1.Title = "Mr.";
Employee1.FirstName = "Mohammed";
Employee1.LastName = "Abu-Hadhoud";
```

Here, all properties and methods came from Person Class via inheritance.

Also, we can access them all inside the employee class.

Inheritance

Super Class/ Base Class

Sub Class/ Derived Class

Object

Class Person

```
public class clsPerson
{
    //properties
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Title { get; set; }

    //read only property
    public string FullName
    {
        //Get is use for Reading field
        get
        {
            return FirstName + ' ' + LastName;
        }
    }
}
```

Class Employee

```
public class clsEmployee : clsPerson
{
    public float Salary { get; set; }
    public string DepartmentName { get; set; }

    public void IncreaseSalaryBy(float Amount)
    {
        Salary += Amount;
    }
}
```

Employee1

- ID
- FirstName
- LastName
- Title
- FullName
- Salary
- DepartmentName
- IncreaseSalaryBy()

Relationship her: "Is A"
Employee is a Person

is-a relationship

In C#, inheritance is an is-a relationship. We use inheritance only if there is an is-a relationship between two classes. For example,

- **Employee** is a **Person**

We can derive **Employee** from **Person** class.

What can you inherit?

you can only inherit the public and protected members, private members are not inherited.

الواجب

Inheritance: Inheritance is one in which a new class is created that inherits the properties of the already exist class. It supports the concept of code reusability and reduces the length of the code in object-oriented programming.

True

False

The class that inherits properties from another class is called Subclass or Derived Class

True

False

The class whose properties are inherited by a subclass is called Base Class or Superclass.

True

False

Derived Class and Sub Class and Child Classes are the same.

True

False

Base Class and Super Class and Parent Class are the same.

True

False

You can inherit only public and protected members, private members are not inherited.

True

False

Relationship between derived class and super class is call "Is-A" because derived class is super class.

True

False

Inheritance Constructor

لو عندك constructor في ال base class وعازي تعمل وراثته يبقي لازم تعمل constructor في ال derived class وتحط فيه كل ال parameters المطلوبة منك واي parameter ثاني انت محتاجها في الكلاس بتاعك وبعد كده تمرر القيم لل base class عن طريق انك تكتب نقطتين : وبعدهم كلمه base وتفتح قوسين تحط فيهم كل المتغيرات اللي عاوز تمرر هاله زي ال C++

```
using System;

public class clsPerson
{
    //properties
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Title { get; set; }

    //constructor
    public clsPerson(int ID, string FirstName, string LastName, string Title) {
        this.ID = ID;
        this.FirstName = FirstName;
        this.LastName = LastName;
        this.Title = Title;
    }

    //read only property
    public string FullName
    {
        //Get is use for Reading field
        get
        {
            return FirstName + ' ' + LastName;
        }
    }
}

public class clsEmployee : clsPerson
{
    public float Salary { get; set; }
    public string DepartmentName { get; set; }

    public clsEmployee(int ID, string FirstName, string LastName, string Title, int Salary, string DeptName)
        : base(ID, FirstName, LastName, Title)
    {
        this.Salary = Salary;
        this.DepartmentName = DepartmentName;
    }

    public void IncreaseSalaryBy(float Amount)
    {
        Salary += Amount;
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        //Create an object of Employee
    }
}
```

```

clsEmployee Employee1 = new clsEmployee(10, "Mohammed", "Abu-Hadhoud", "Mr.", 5000,
"IT");

Console.WriteLine("Accessing Object 1 (Employee1):\n");
Console.WriteLine("ID := {0}", Employee1.ID);
Console.WriteLine("Title := {0}", Employee1.Title);
Console.WriteLine("Full Name := {0}", Employee1.FullName);
Console.WriteLine("Department Name := {0}", Employee1.DepartmentName);
Console.WriteLine("Salary := {0}", Employee1.Salary);

Employee1.IncreaseSalaryBy(100);
Console.WriteLine("Salary after increase := {0}", Employee1.Salary);
Console.ReadKey();
}
}

```

Upcasting and Downcasting

ال upcasting وال down casting شرحناهم قبل كده في ال c++ وال upcasting انك بتحول ال object اللي من النوع employee ل object من النوع person

في الحاله دي أي داتا زياده هتخسر ها

وال down casting انك تحول ال person ل employee وفي الحاله دي هتلاقي انه فيه متغيرات ملهاش قيم وده ممكن يعملك مشكله في ال runtime

من فوايد ال upcasting انه لو ال object حجمه كبير وانت عاوز داتا معينه هوا وارثها تقدر تصغر ال object عشان يكون اسرع معاك

بيقولك عشان تتجنب مشكلة ال run time في ال down casting بتعمل object من الاب وتقوله بيساوي الكلاس الابن وبعدين تعمل object من الكلاس الابن وتخزن فيه الكلاس الاب وبعدين ترجع تاني تحوله للاب

```

using System;

public class clsPerson
{
    public string Name { get; set; }
    public int Age { get; set; }

    public void Greet() { Console.WriteLine($"hi, my name is {Name} and I'm {Age} year old"); }
}

public class clsEmployee : clsPerson {
    public string Company { get; set; }
    public decimal Salary { get; set; }
    public void Work() { Console.WriteLine($"i WORK at {Company} and earn {Salary:C} per year."); }
}

internal class Program
{
    static void Main(string[] args)

```

```

{
    clsEmployee employee = new clsEmployee {Name="John",Age=30,Company="Acme
Inc.",Salary=10 };
    clsPerson person = employee;
    person.Greet();

    clsPerson person1 = new clsEmployee {Name="jane",Age=20,Company="sdsd",Salary=20 };
    clsEmployee employee2 = (clsEmployee)person1;
    employee2.Work();

    Console.ReadKey();
}
}

```

UpCasting and DownCasting

In C#, upcasting and downcasting refer to converting an object reference to a base class or derived class reference, respectively.

Upcasting is a safe operation because a derived class is always a specialization of the base class, but downcasting can be dangerous because a base class may not have all the members of a derived class. Here is an example to illustrate upcasting and downcasting:

Example:

```

using System;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public void Greet()
    {
        Console.WriteLine($"Hi, my name is {Name} and I am {Age} years old.");
    }
}

public class Employee : Person

```



```

{
    public string Company { get; set; }
    public decimal Salary { get; set; }

    public void Work()
    {
        Console.WriteLine($"I work at {Company} and earn {Salary:C} per year.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Upcasting
        Employee employee = new Employee { Name = "John", Age = 30, Company = "Acme Inc.",
Salary = 50000 };
        Person person = employee;
        person.Greet(); // Output: "Hi, my name is John and I am 30 years old."

        // Downcasting
        Person person2 = new Employee { Name = "Jane", Age = 25, Company = "XYZ Corp.", Salary
= 60000 };
        Employee employee2 = (Employee)person2;
        employee2.Work(); // Output: "I work at XYZ Corp. and earn $60,000.00 per year."

        // Invalid downcasting - throws InvalidCastException at runtime
        // Person person3 = new Person { Name = "Bob", Age = 40 };
        // Employee employee3 = (Employee)person3; // Runtime exception: InvalidCastException

        Console.ReadKey();
    }
}

```

In this example, we have a `Person` class and an `Employee` class that inherits from `Person`.

The `Person` class has a `Name` and `Age` property, as well as a `Greet` method that prints a greeting to

the console. The `Employee` class has an additional `Company` and `Salary` property, as well as a `Work` method that prints information about the employee's job to the console.

In the `Main` method, we first create a new `Employee` object and assign it to a variable of type `Person`, which is an example of upcasting. We then call the `Greet` method on the `Person` variable, which outputs "Hi, my name is John and I am 30 years old." This is possible because the `Employee` class inherits from `Person`, so it can be safely upcast to `Person`.

Next, we create a new `Employee` object and assign it to a `Person` variable, which is another example of upcasting. We then downcast the `Person` variable to an `Employee` variable using an explicit cast with the `(Employee)` syntax. We can then call the `Work` method on the `Employee` variable, which outputs "I work at XYZ Corp. and earn \$60,000.00 per year." This is possible because the `Person` variable actually refers to an `Employee` object, which has the `Work` method.

Finally, we attempt to downcast a `Person` object to an `Employee` object, which is an example of invalid downcasting because the `Person` object is not actually an `Employee` object. This will throw an `InvalidCastException` at runtime.

Note:

- Up Casting is converting derived object to it's base object.
- Down Casting is Converting Base object to Derived object
- Upcasting is a safe operation because a derived class is always a specialization of the base class
- Downcasting can be dangerous because a base class may not have all the members of a derived class.

الواجب

Up Casting is converting derived object to it's base object.

True

False

Down Casting is Converting Base object to Derived object.

True

False

Upcasting is a safe operation because a derived class is always a specialization of the base class.

True

False

Downcasting can be dangerous because a base class may not have all the members of a derived class.

True

False

Method Overriding in C# Inheritance + Base Keyword

ال overriding انه عندك function في ال base class عاوز تغيرها بما يلانم ال subclass
عشان تقدر تعمل ال overriding بتيجي عند ال function في ال base class وتكتب جنبها
virtual

عشان تستدعي ال function اللي في ال base class وتوصلها من ال subclass بتعمل
override ليها وتكتب base وبعدها اسم ال function

```
using System;

public class clsA {
    public virtual void Print() {
        Console.WriteLine("Hi, I'm The print methhod from the base class A");
    }
}

public class clsB : clsA {
    public override void Print()
    {

        Console.WriteLine("Hi, I'm The print methhod from the base class B");
        base.Print();
    }
}
```

```

    }
}

internal class Program
{
    static void Main(string[] args)
    {
        clsB objB = new clsB();
        objB.Print();

        Console.ReadKey();
    }
}

```

فيه طريقه ثانيه عن طريق كلمة **new** بس دي اسمها **shadowing** وبتفرق عن ال **overriding**

Method Overriding in C# Inheritance

If the same method is present in both the base class and the derived class, the method in the derived class overrides the method in the base class. This is called method overriding in C#. For example,

```

using System;

public class clsA
{
    public virtual void Print()
    {
        Console.WriteLine("Hi, I'm the print method from the base class A");
    }
}

public class clsB : clsA
{
    public override void Print()
    {
        Console.WriteLine("Hi, I'm the print method from the derived class B");
        base.Print();
    }
}

internal class Program
{

```

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

    //Create an object of Employee
    clsB ObjB= new clsB();

    ObjB.Print();

    Console.ReadKey();
}
}
```

Output

```
Hi, I'm the print method from the derived class B
Hi, I'm the print method from the base class A
```

In the above example, the print() method is present in both the base class and derived class.

When we call print() using the B object,

```
ObjB.Print();
```

the Print inside B is called. This is because the Print method inside B overrides the Print method inside A.

Notice, we have used **virtual** and override with methods of the base class and derived class respectively. Here,

- **virtual** - allows the method to be overridden by the derived class
- **override** - indicates the method is overriding the method from the base class

base Keyword in C# Inheritance

In the previous example, we saw that the method in the derived class overrides the method in the base class.

However, what if we want to call the method of the base class as well?

In that case, we use the **base** keyword to call the method of the base class from the derived class.

```
public override void Print()

{
    Console.WriteLine("Hi, I'm the print method from the derived class B");
    base.Print();
}
```

base keyword is used to call the Print method in the base class.

```
base.Print();
```

Method Hiding in C#

ال hiding او ال shadowing هوا انك بدل ماتكتب كلمة **override** بتكتب **new** طب ايه الفرق؟

احنا دلوقتى عندنا **base class** و **derived class** وفيه **method** معمولها **overriding** لو جيت اخدت **object** من ال **base class** واستدعيت ال **method** هينفذ الكود اللي في ال **base** ولو جيت اخدت **object** من ال **subclass** واستدعيت ال **method** هينفذ الكود اللي في ال **subclass**

لحد هنا ال **overriding** بتتساوي مع ال **hiding**

لكن لو جيت عملت **casting** من ال **derived** لل **base** واستدعيت ال **method** هنا هيجصل الفرق

ف في ال **overriding** هيشغل الكود اللي في ال **derived** لان ال **object** اصله كان **subclass**

انما في ال **hiding** هيشغل الكود اللي في ال **base**

```
using System;
```

```
public class MyBaseClass {
```

```

public virtual void MyMethod() {
    Console.WriteLine("Base class implementation");
}

public virtual void MyOtherMethod()
{
    Console.WriteLine("Base class implementation of MyOtherMethod");
}
}

public class MyDerivedClass : MyBaseClass {
    public override void MyMethod()
    {

        Console.WriteLine("Derived class implementation using override");

    }

    public new void MyOtherMethod() {
        Console.WriteLine("Derived class implementation of MyOtherMethod using new");
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        MyBaseClass myBaseObj = new MyBaseClass();
        Console.WriteLine("\nBase Object:\n");
        myBaseObj.MyMethod();
        myBaseObj.MyOtherMethod();

        MyDerivedClass myDerivedObj= new MyDerivedClass();
        Console.WriteLine("\nDerived Object:\n");
        myDerivedObj.MyMethod();
        myDerivedObj.MyOtherMethod();

        MyBaseClass myDerivedObjAsBase = myDerivedObj;
        Console.WriteLine("\nAfter casting:\n");
        myDerivedObjAsBase.MyMethod();
        myDerivedObjAsBase.MyOtherMethod();


        Console.ReadKey();
    }
}

```

Method Hiding in C#

As we already know about polymorphism and method overriding in C#. C# also provides a concept to hide the methods of the base class from derived class, this concept is known as Method Hiding. It is also known as **Method Shadowing**. In method hiding, you can hide the implementation of the methods of a base class from the derived class using the *new* keyword. Or in other words, in method hiding, you can redefine the method of the base class in the derived class by using the *new* keyword.

```
using System;

public class MyBaseClass
{
    public virtual void MyMethod()
    {
        Console.WriteLine("Base class implementation");
    }

    public virtual void MyOtherMethod()
    {
        Console.WriteLine("Base class implementation of MyOtherMethod");
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void MyMethod()
    {
        Console.WriteLine("Derived class implementation using override");
    }

    public new void MyOtherMethod()
    {
        Console.WriteLine("Derived class implementation of MyOtherMethod using new");
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyBaseClass myBaseObj = new MyBaseClass();
        Console.WriteLine("\nBase Object:\n");
        myBaseObj.MyMethod(); // Output: "Base class implementation"
    }
}
```



```

myBaseObj.MyOtherMethod(); // Output: "Base class implementation of MyOtherMethod"

MyDerivedClass myDerivedObj = new MyDerivedClass();
Console.WriteLine("\nDerived Object:\n");
myDerivedObj.MyMethod(); // Output: "Derived class implementation using override"
myDerivedObj.MyOtherMethod(); // Output: "Derived class implementation of MyOtherMethod
using new"

MyBaseClass myDerivedObjAsBase = myDerivedObj;
Console.WriteLine("\nAfter Casting:\n");
myDerivedObjAsBase.MyMethod(); // Output: "Derived class implementation using override"
myDerivedObjAsBase.MyOtherMethod(); // Output: "Base class implementation of
MyOtherMethod"

Console.ReadKey();
}
}

```

In the `Main` method, we create an instance of the base class and call its `MyMethod` and `MyOtherMethod` methods, which output "Base class implementation" and "Base class implementation of `MyOtherMethod`", respectively.

We then create an instance of the derived class and call its `MyMethod` and `MyOtherMethod` methods, which output "Derived class implementation using override" and "Derived class implementation of `MyOtherMethod` using new", respectively.

We also cast the derived class instance to the base class type and call its `MyMethod` and `MyOtherMethod` methods, which output "Derived class implementation using override" and "Base class implementation of `MyOtherMethod`", respectively. This is because we have overridden `MyMethod` in the derived class but only hidden `MyOtherMethod`, so calling it on an instance of the base class type will invoke the implementation in the base class.

Types Of Inheritance

أنواع ال inheritance شرحناها في ال c++ وقولنا انها بتدعم جميع الأنواع واللي هما:-

1- single inheritance :- كلاس بيورث من كلاس

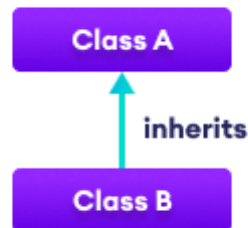
- 2 multilevel inheritance :- كلاس بيورث من كلاس بيورث من كلاس
 - 3 hierarchal inheritance :- مجموعه بيورثوا من كلاس واحد
 - 4 multiple inheritance :- كلاس واحد بيورث من مجموعه
 - 5 hybrid inheritance :- كلاس بيورث من مجموعه والمجموعه بيورثوا من كلاس
- ال c# بتدعم اول 3 أنواع بس

Types of inheritance

There are the following types of inheritance:

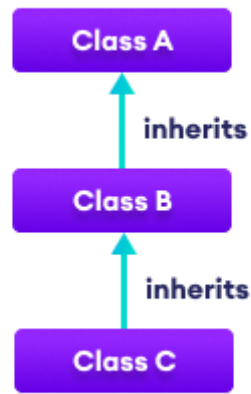
1. Single Inheritance

In single inheritance, a single derived class inherits from a single base class.



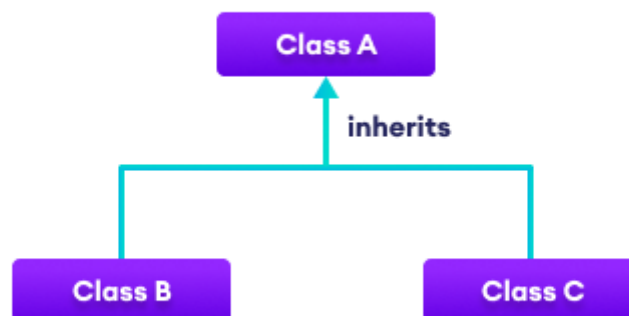
2. Multilevel Inheritance

In multilevel inheritance, a derived class inherits from a base and then the same derived class acts as a base class for another class.



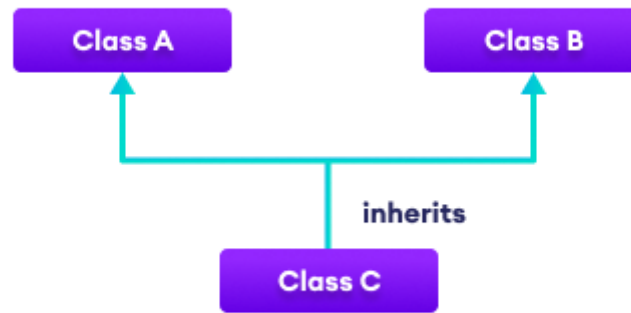
3. Hierarchical Inheritance

In hierarchical inheritance, multiple derived classes inherit from a single base class.



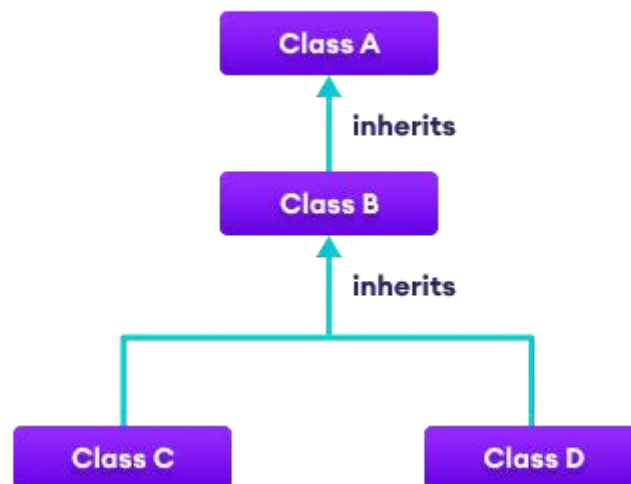
4. Multiple Inheritance

In multiple inheritance, a single derived class inherits from multiple base classes. **C# doesn't support multiple inheritance.** However, we can achieve multiple inheritance through interfaces.



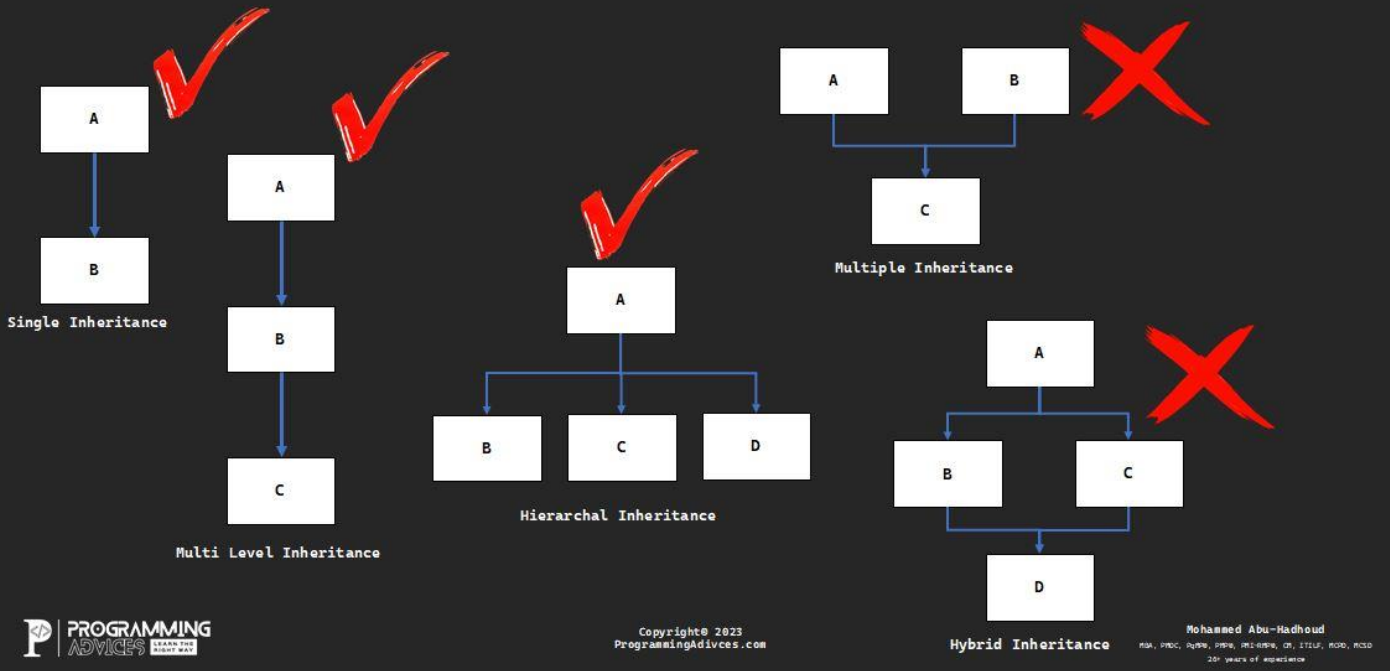
5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. The combination of multilevel and hierarchical inheritance is an example of Hybrid inheritance.



Types supported by C#

Inheritance Types



Note: it does not support hybrid inheritance that contains multiple inheritance.

Multi-Level Inheritance

ال multi level inheritance دكتور بيورث من موظف والموظف بيورث من شخص

```
using System;

public class Person {
    public string Name { get; set; }
    public int Age { get; set; }

    public void Introduce() {
        Console.WriteLine($"Hi, my name is {Name} and I'm {Age} years old");
    }
}

public class Employee : Person {
    public int EmployeeID { get; set; }
    public decimal Salary { get; set; }

    public void Work() {
        Console.WriteLine($"Employee with ID {EmployeeID} and Salary {Salary:C} is working.");
    }
}

public class Doctor : Employee {
    public string Speciality { get; set; }

    public void Heal() {
        Console.WriteLine($"Doctor {Name} with ID {EmployeeID} , Salary {Salary:C}, and SPeciality {Speciality}");
    }
}
```

```

internal class Program
{
    static void Main(string[] args)
    {
        Doctor doctor=new Doctor();
        doctor.Name = "dasdas";
        doctor.Age = 32;
        doctor.EmployeeID = 1;
        doctor.Salary = 50;
        doctor.Speciality = "eqwe";
        doctor.Introduce();
        doctor.Work();
        doctor.Heal();

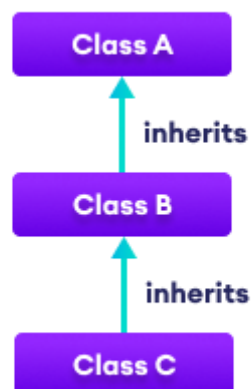
        Console.ReadKey();
    }
}

```

Multi Level Inheritance

Multilevel Inheritance

In multilevel inheritance, a derived class inherits from a base and then the same derived class acts as a base class for another class.



Example:

```

using System;

public class Person
{
    public string Name { get; set; }
}

```

```

    public int Age { get; set; }

    public void Introduce()
    {
        Console.WriteLine($"Hi, my name is {Name} and I'm {Age} years old.");
    }
}

public class Employee : Person
{
    public int EmployeeId { get; set; }
    public decimal Salary { get; set; }

    public void Work()
    {
        Console.WriteLine($"Employee with ID {EmployeeId} and salary {Salary:C} is working.");
    }
}

public class Doctor : Employee
{
    public string Specialty { get; set; }

    public void Heal()
    {
        Console.WriteLine($"Doctor {Name} with ID {EmployeeId}, salary {Salary:C}, and
specialty {Specialty} is healing a patient.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Doctor doctor = new Doctor();
        doctor.Name = "John";
        doctor.Age = 35;
    }
}

```

```

        doctor.EmployeeId = 123;
        doctor.Salary = 100000.00M;
        doctor.Specialty = "Cardiology";
        doctor.Introduce(); // Output: "Hi, my name is John and I'm 35 years old."
        doctor.Work(); // Output: "Employee with ID 123 and salary $100,000 is working."
        doctor.Heal(); // Output: "Doctor John with ID 123, salary $100,000, and specialty
Cardiology is healing a patient."

        Console.ReadKey();
    }
}

```

In this example, we have a Person class that has Name and Age properties, as well as an Introduce method. The Employee class inherits from Person and has an additional EmployeeId and Salary property and a Work method. The Doctor class inherits from Employee and has an additional Specialty property and a Heal method.

In the Main method, we create a new Doctor object and set its properties. Since Doctor inherits from Employee, which in turn inherits from Person, it has access to all of the properties and methods defined in those classes.

Hierarchal Inheritance

ال hierarchical inheritance انه مجموعه بيورثوا من واحد

```

using System;

public class Person {
    public string Name { get; set; }
    public int Age { get; set; }

    public void Introduce() {
        Console.WriteLine($"Hi, my name is {Name} and I'm {Age} years old");
    }
}

public class Employee : Person {
    public int EmployeeID { get; set; }
    public decimal Salary { get; set; }

    public void Work() {
        Console.WriteLine($"Employee with ID {EmployeeID} and Salary {Salary:C} is working.");
    }
}

public class User : Person {
    public string UserName { get; set; }
    public string Password { get; set; }
}

```



```

    public void Info() {
        Console.WriteLine($"User: {UserName} And PassWord: {PassWord}");
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        Employee Employee1 = new Employee();
        Employee1.Name = "dasdas";
        Employee1.Age = 32;
        Employee1.EmployeeID = 1;
        Employee1.Salary = 50;
        Console.WriteLine("\nEmployee:");
        Employee1.Introduce();
        Employee1.Work();

        User User1 = new User();
        User1.Name = "dasdas";
        User1.Age = 54;
        User1.UserName = "DFGDFGDFG";
        User1.PassWord = "DSADASD";
        Console.WriteLine("\nUser:");
        User1.Introduce();
        User1.Info();

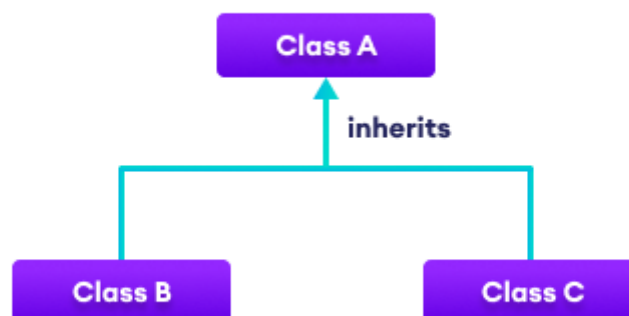
        Console.ReadKey();
    }
}

```

Hierarchical Inheritance

Hierarchical Inheritance

In hierarchical inheritance, multiple derived classes inherit from a single base class.



Example:

```
using System;
```

```
public class Person
```

```
{  
    public string Name { get; set; }  
    public int Age { get; set; }  
  
    public void Introduce()  
    {  
        Console.WriteLine($"Hi, my name is {Name} and I'm {Age} years old.");  
    }  
}
```

```
public class Employee : Person
```

```
{  
    public int EmployeeId { get; set; }  
    public decimal Salary { get; set; }  
  
    public void Work()  
    {  
        Console.WriteLine($"Employee with ID {EmployeeId} and salary {Salary:C} is working.");  
    }  
}
```

```
public class User : Person
```

```
{  
    public string Username { get; set; }  
    public string Password { get; set; }  
    public int Permission { get; set; }  
  
    public void Info()  
    {
```

```

        Console.WriteLine($"User: {Username} and Password {Password} .");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee Employee1 = new Employee();
        Employee1.Name = "John";
        Employee1.Age = 35;
        Employee1.EmployeeId = 123;
        Employee1.Salary = 100000.00M;
        Console.WriteLine("\nEmployee:");
        Employee1.Introduce(); // Output: "Hi, my name is John and I'm 35 years old."
        Employee1.Work(); // Output: "Employee with ID 123 and salary $100,000.00 is working."

        User User1 = new User();
        User1.Name = "Ali";
        User1.Age = 45;
        User1.Username = "User1";
        User1.Password = "1234";

        Console.WriteLine("\nUser:");
        User1.Introduce(); // Output: "Hi, my name is John and I'm 35 years old."
        User1.Info(); //Output: "User: User1 and Password 1234 ."

        Console.ReadKey();
    }
}

```

In this example, we have a Person class that has Name and Age properties, as well as an Introduce method.

The Employee class inherits from Person and has an additional EmployeeId and Salary property and a Work method.

The User class inherits from Person and has an additional Username, Password, Permissions properties and a Info method.

Both Employee and User Inherit from Person Class.

Abstract Class & Methods

ال abstract class هوا كلاس ماينفع تاخد object منه لازم تعمل كلاس يورث منه عشان تقدر تستخدمه

وبتقدر تحط فيه نوعين من ال methods اول واحده وهيا ال method العاديه والنوع الثاني اسمه abstract method ودي مابتحطش فيها اكواد الاكواد او ال implementation بتتعمل عن طريق ال overriding

```
using System;

public abstract class Person{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public abstract void Introduce();

    public void SayGoodbye() {
        Console.WriteLine("Goodbye!");
    }
}

public class Employee : Person
{
    public int EmployeeId { get; set; }
    public override void Introduce()
    {
        Console.WriteLine($"Hi, my name is {FirstName} {LastName}, and My employee ID is {EmployeeId}");
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();
        employee.FirstName = "kajfds";
        employee.LastName = "kajfds";
        employee.EmployeeId = 123;
        employee.Introduce();
        Console.ReadKey();
    }
}
```

Abstract Class

In C#, we cannot create objects of an abstract class. We use the **abstract** keyword to create an abstract class.

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).

An abstract class can have both abstract and regular methods:

Example:

```
using System;

public abstract class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public abstract void Introduce();

    public void SayGoodbye()
    {
        Console.WriteLine("Goodbye!");
    }
}

public class Employee : Person
{
    public int EmployeeId { get; set; }

    public override void Introduce()
    {
        Console.WriteLine($"Hi, my name is {FirstName} {LastName}, and my employee ID is {EmployeeId}.");
    }
}

public class Program
{

```

```
public static void Main()
{
    //You cannot create an object of an abstract class, you can only inherit it.
    // Person Person1= new Person();

    Employee employee = new Employee();
    employee.FirstName = "Mohammed";
    employee.LastName = "Abu-Hadhoud";
    employee.EmployeeId = 123;
    employee.Introduce(); // Output: "Hi, my name is John Doe, and my employee ID is 123."
    employee.SayGoodbye(); // Output: "Goodbye!"

    Console.ReadKey();
}
}
```

In this example, the abstract class `Person` has two properties `FirstName` and `LastName`, an abstract method `Introduce()`, and a non-abstract method `SayGoodbye()`. The `Introduce()` method is marked as `abstract`, which means it does not have an implementation in the `Person` class and must be implemented by any derived class that inherits from `Person`. The `SayGoodbye()` method is not marked as `abstract`, which means it has an implementation in the `Person` class and can be inherited by derived classes.

The `Employee` class is derived from `Person` and provides an implementation for the `Introduce()` method. It also has an additional property `EmployeeId`. We can create instances of the `Employee` class and call its methods, including the inherited `SayGoodbye()` method

الواجب

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

True

False

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).

True

False

What is Interface ? and Why?

ال interface هوا عبارته عن عقد زي ماقولنا قبل كده وانك بتجبر اليوزر انه يعمل implementation بتاع ال function

وهوا زيه زي ال abstract class بس من غير ال functions كلها من غير implementation طريقته انك بتكتب كلمة interface

```
using System;

public interface IPerson{
    string FirstName { get; set; }
    string LastName { get; set; }

    void Introduce();

    void Print();

    string To_String();
}

public abstract class Person : IPerson
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public abstract void Introduce();

    public void Print() {
        Console.WriteLine("Hi I'm the print method");
    }

    public string To_String() {
        return "Hi this is the to string";
    }

    public void SayGoodbye() {
        Console.WriteLine("GoodBye!");
    }

    public void sendEmail() {
        Console.WriteLine("Email Sent!");
    }
}

internal class Program
{
    static void Main(string[] args)
```

```
{  
  
    Console.ReadKey();  
  
}
```

Interface

In C#, an interface is similar to abstract class. However, unlike abstract classes, all methods of an interface are fully abstract (method without body).

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

We use the **interface** keyword to create an interface. For example,

```
public interface IPerson  
{  
    string FirstName { get; set; }  
    string LastName { get; set; }  
    void Introduce();  
    void Print();  
    string To_String();  
}
```

Here,

- IPerson is the name of the interface.
- By convention, interface starts with I so that we can identify it just by seeing its name.
- We cannot use access modifiers inside an interface.
- All members of an interface are public by default.
- An interface doesn't allow fields.
- Like **abstract classes**, interfaces **cannot** be used to create objects.
- Interface methods do not have a body - the body is provided by the "implement" class.
- Interfaces can contain properties and methods, but not fields/variables

- Interface members are by default **abstract** and **public**
- An interface cannot contain a constructor (as it cannot be used to create objects)

Implementing an Interface

We cannot create objects of an interface. To use an interface, other classes must implement it. Same as in [C# Inheritance](#), we use **:** symbol to implement an interface. For example,

```
using System;

public interface IPerson
{
    string FirstName { get; set; }
    string LastName { get; set; }

    void Introduce();
    void Print();

    string To_String();
}

public abstract class Person : IPerson
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public abstract void Introduce();

    public void SayGoodbye()
    {
        Console.WriteLine("Goodbye!");
    }

    public void Print()
```

```

    {
        Console.WriteLine("Hi I'm the print method");
    }

    public string To_String()
    {
        return "Hi this is the complete string....";
    }

    public void SedEmail()
    {
        Console.WriteLine("Email Sent :-");
    }
}

public class Employee : Person
{
    public int EmployeeId { get; set; }

    public override void Introduce()
    {
        Console.WriteLine($"Hi, my name is {FirstName} {LastName}, and my employee ID is {EmployeeId}.");
    }
}

public class Program
{
    public static void Main()
    {
        //You cannot create an object of an Interface, you can only Implement it.
        // IPerson Person1 = new IPerson();

        Employee employee = new Employee();
        employee.FirstName = "Mohammed";
        employee.LastName = "Abu-Hadhoud";
        employee.EmployeeId = 123;
    }
}

```

```
employee.Introduce(); // Output: "Hi, my name is John Doe, and my employee ID is 123."
employee.SayGoodbye(); // Output: "Goodbye!"
employee.Print();
employee.SendEmail();
Console.ReadKey();

}

}
```

Note: We must provide the implementation of all the methods of interface inside the class that implements it.

الواجب

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

True

False

We must provide the implementation of all the methods of interface inside the class that implements it.

True

False

Like abstract classes, interfaces cannot be used to create objects

True

False

Interface methods do not have a body - the body is provided by the "implement" class

True

False

Interfaces can contain properties and methods, but not fields/variables

True

False

Interface members are by default abstract and public

True

False

An interface cannot contain a constructor (as it cannot be used to create objects)

True

False

Implementing Multiple Interfaces

في ال c# ماتقدرش تعمل multiple inheritance انه كلاس يورث من اكثر من كلاس
بس بتقدر تعمل فيها multiple implementation يعني يورث من اكثر من interface ممكن انما
يورث من اكثر من كلاس عادي لا ماینفعش

```
using System;
```

```
public interface IPerson{  
    string FirstName { get; set; }  
    string LastName { get; set; }  
}
```

```

    void Introduce();

    void Print();

    string To_String();
}

public interface ICommunicate {
    void CallPhone();
    void SendEmail(string Title, string Body);
    void SendSMS(string Title, string Body);
    void SendFax(string Title, string Body);
}

public abstract class Person : IPerson, ICommunicate
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public abstract void Introduce();

    public void Print() {
        Console.WriteLine("Hi I'm the print method");
    }

    public string To_String() {
        return "Hi this is the to string";
    }

    public void SayGoodbye() {
        Console.WriteLine("GoodBye!");
    }

    public void CallPhone() {
        Console.WriteLine("Phone Called!");
    }
    public void SendEmail(string Title, string Body) {
        Console.WriteLine("Email Sent!");
    }
    public void SendSMS(string Title, string Body) {
        Console.WriteLine("SMS Sent!");
    }
    public void SendFax(string Title, string Body) {
        Console.WriteLine("Fax Sent!");
    }
}

public class Employee : Person
{
    public int EmployeeId { get; set; }

    public override void Introduce()
    {
        Console.WriteLine($"Hi, my name is {FirstName} {LastName}, and my employee ID is {EmployeeId}.");
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee();
        employee.FirstName = "kajfds";
        employee.LastName = "kajfds";
        employee.EmployeeId = 123;
    }
}

```

```
        employee.Introduce();
        employee.SayGoodbye();
        employee.Print();
        employee.SendEmail("HI", " BODY");
        employee.CallPhone();
        employee.SendFax("HI", " BODY");
        employee.SendSMS("HI", " BODY");
        Console.ReadKey();
    }
}
```

Implementing Multiple Interfaces

Unlike inheritance, a class can implement multiple interfaces. To implement multiple interfaces, separate them with a comma (see example below).

```
using System;

public interface IPerson
{
    string FirstName { get; set; }
    string LastName { get; set; }

    void Introduce();

    void Print();

    string To_String();
}

public interface ICommunicate
{
    void CallPhone();

    void SendEmail(string Title, string Body);

    void SendSMS(string Title, string Body);

    void SendFax(string Title, string Body);
}
```

```
public abstract class Person : IPerson, ICommunicate

{

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public abstract void Introduce();

    public void SayGoodbye()
    {
        Console.WriteLine("Goodbye!");
    }

    public void Print()
    {
        Console.WriteLine("Hi I'm the print method");
    }

    public string To_String()
    {
        return "Hi this is the complete string...";
    }

    public void CallPhone()
    {
        Console.WriteLine("Calling Phone... :-)");
    }

    public void SendEmail(string Title, string Body)
    {
        Console.WriteLine("Email Sent :-)");
    }

    public void SendSMS(string Title, string Body)
    {
        Console.WriteLine("SMS Sent :-)");
    }
}
```

```

    }

    public void SendFax(string Title, string Body)
    {
        Console.WriteLine("Fax Sent :-");
    }
}

public class Employee : Person
{
    public int EmployeeId { get; set; }

    public override void Introduce()
    {
        Console.WriteLine($"Hi, my name is {FirstName} {LastName}, and my employee ID is {EmployeeId}.");
    }
}

public class Program
{
    public static void Main()
    {
        //You cannot create an object of an Interface, you can only Implement it.
        // IPerson Person1 = new IPerson();

        Employee employee = new Employee();
        employee.FirstName = "Mohammed";
        employee.LastName = "Abu-Hadhoud";
        employee.EmployeeId = 123;
        employee.Introduce(); // Output: "Hi, my name is John Doe, and my employee ID is 123."
        employee.SayGoodbye(); // Output: "Goodbye!"
        employee.Print();
        employee.CallPhone();
        employee.SendEmail("hi", "Body");
        employee.SendSMS("hi", "Body");
        employee.SendFax("hi", "Body");
    }
}

```



```
Console.ReadKey();
```

```
}
```

```
}
```

الواجب

To implement multiple interfaces, separate them with a comma.

True

False

C# Nested Class

ال nested class هو كلاس جوا كلاس والأتنين مالهومش علاقه ببعض يعني ماتقدرش تستدعي متغير موجود في الكلاس الخارجي وانت جوه الكلاس الداخلي من غير object

```
using System;

public class OuterClass {
    private int OuterVariable;

    public OuterClass(int outerVariable) {
        this.OuterVariable = outerVariable;
    }

    public void OuterMethod() {
        Console.WriteLine("Outer Method called.");
    }

    public class InnerClass {
        private int InnerVariable;

        public InnerClass(int innerVariable) {
            this.InnerVariable = innerVariable;
        }

        public void InnerMethod() {
            Console.WriteLine("Inner Method with inner variable = "+InnerVariable);
        }

        public void AccessOuterVariable(OuterClass outer) {
            Console.WriteLine("Accessing outer variable from inner class: "+outer.OuterVariable);
        }
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        OuterClass outer1=new OuterClass(42);
    }
}
```

```
OuterClass.InnerClass inner1=new OuterClass.InnerClass(100);

outer1.OuterMethod();
inner1.InnerMethod();
inner1.AccessOuterVariable(outer1);

Console.ReadKey();

    }
}
```

C# Nested Class

In C#, we can define a class within another class. It is known as a nested class. For example,

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Here, we have created the class `InnerClass` inside the class `OuterClass`. The `InnerClass` is called the nested class.

Example:

```
using System;

public class OuterClass
{
    private int outerVariable;

    public OuterClass(int outerVariable)
    {
        this.outerVariable = outerVariable;
    }

    public void OuterMethod()
    {
        Console.WriteLine("Outer method called.");
    }

    public class InnerClass
```

```

{
    private int innerVariable;

    public InnerClass(int innerVariable)
    {
        this.innerVariable = innerVariable;
    }

    public void InnerMethod()
    {
        Console.WriteLine("Inner method called with innerVariable = " + innerVariable);
    }

    public void AccessOuterVariable(OuterClass outer)
    {
        Console.WriteLine("Accessing outerVariable from inner class: " +
outer.outerVariable);
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // create an instance of OuterClass
        OuterClass outer1 = new OuterClass(42);

        // create an instance of InnerClass
        OuterClass.InnerClass inner1 = new OuterClass.InnerClass(100);

        // call methods on the instances
        outer1.OuterMethod(); // prints "Outer method called."
        inner1.InnerMethod(); // prints "Inner method called with innerVariable = 100"
        inner1.AccessOuterVariable(outer1); // prints "Accessing outerVariable from inner
class: 42"
        Console.ReadKey();
    }
}

```

```
}  
}
```

In this example, InnerClass is defined inside OuterClass. It has its own private innerVariable field and a method called InnerMethod that prints the value of that variable. It also has a method called AccessOuterVariable that takes an instance of OuterClass as a parameter and prints the value of the outerVariable field.

Composition

فكرته انك بتاخذ object من كلاس جوه كلاس تاني
خلي بالك انه ال inner class ده كلاس جوه كلاس انما هنا عندك object جوه كلاس

```
using System;  
  
class clsA  
{  
    public int x;  
    public int y;  
  
    public void Method1()  
    {  
        Console.WriteLine("Method1 of class A is called");  
    }  
  
    public void Method2()  
    {  
        Console.WriteLine("Method2 of class A is called");  
        Console.WriteLine("Now i will call method1 of class B...");  
  
        //defining an object of another class inside this class is called composition.  
        clsB ObjectB1 = new clsB();  
        ObjectB1.Method1();  
    }  
}  
  
class clsB  
{  
    public void Method1()  
    {  
        Console.WriteLine("Method1 of class B is called");  
    }  
}  
  
internal class Program  
{  
    static void Main(string[] args)  
    {  
  
        //Create object from class  
        clsA ObjectA1 = new clsA();  
        ObjectA1.Method1();  
        ObjectA1.Method2();  
  
        Console.ReadKey();  
    }  
}
```

```
}  
}
```

C# Composition

Composition is a design pattern in object-oriented programming where a class is composed of other objects, and those objects are usually created and managed by the class itself.

In simple words, you can create an object of another class from inside your class.

Example:

```
using System;  
  
class clsA  
{  
    public int x;  
    public int y;  
  
    public void Method1()  
    {  
        Console.WriteLine("Method1 of class A is called");  
    }  
  
    public void Method2()  
    {  
        Console.WriteLine("Method2 of class A is called");  
        Console.WriteLine("Now i will call method1 of class B...");  
  
        //defining an object of another class inside this class is called composition.  
        clsB ObjectB1= new clsB();  
        ObjectB1.Method1();  
    }  
}  
  
class clsB  
{  
    public void Method1()  
    {  
        Console.WriteLine("Method1 of class B is called");  
    }  
}
```

```

    }
}

internal class Program
{
    static void Main(string[] args)
    {

        //Create object from class
        clsA ObjectA1 = new clsA();
        ObjectA1.Method1();
        ObjectA1.Method2();

        Console.ReadKey();
    }
}

```

Sealed Class

عایز امنع انه حد یورث من الكلاس ده یدوب بتکتاب کلمه sealed

```

using System;

sealed class clsA
{

}

// trying to inherit sealed class
// Error Code
class clsB : clsA
{

}

class Program
{
    static void Main(string[] args)
    {

        // create an object of B class
        clsB B1 = new clsB();

        Console.ReadKey();
    }
}

```

Sealed Class

In C#, when we don't want a class to be inherited by another class, we can declare the class as a **sealed class**.

Why Sealed Class?

We use sealed classes to prevent inheritance. As we cannot inherit from a sealed class, the methods in the sealed class cannot be manipulated from other classes.

It helps to prevent security issues. For example,

```
using System;

sealed class clsA
{

}

// trying to inherit sealed class
// Error Code
class clsB : clsA
{

}

class Program
{
    static void Main(string[] args)
    {

        // create an object of B class
        clsB B1 = new clsB();

        Console.ReadKey();
    }
}
```

As class A cannot be inherited, class B cannot override and manipulate the methods of class A.

In the above example, we have created a sealed class A. Here, we are trying to derive B class from the A class.

Since a sealed class cannot be inherited, the program generates the following error:

```
error CS0509: 'B': cannot derive from sealed type 'A1'
```

Sealed Method

ده لو مش عايز حد يعمل override لل method بعد مرحلة معينه من الوراثة ال method تفضل تعمل override عادي لاي كلاس بيورث من الاب انما الاحفاد مايقدروش يعملوا override

```
using System;

public class Person
{
    public virtual void Greet()
    {
        Console.WriteLine("The person says hello.");
    }
}

public class Employee : Person
{
    public sealed override void Greet()
    {
        Console.WriteLine("The employee greets you.");
    }
}

public class Manager : Employee
{
    //This will produce a compile-time error because the Greet method in Employee is
    //sealed and cannot be overridden.
    //public override void Greet()
    //{
    //    Console.WriteLine("The manager greets you warmly.");
    //}
}

public class Program
{
    public static void Main(string[] args)
    {
        Person person = new Person();
        person.Greet(); // outputs "The person says hello."

        Employee employee = new Employee();
        employee.Greet(); // outputs "The employee greets you."

        Manager manager = new Manager();
    }
}
```



```
manager.Greet(); // outputs "The employee greets you."

Console.ReadKey();

}

}
```

Sealed Method

During method overriding, if we don't want an overridden method to be further overridden by another class, we can declare it as a **sealed method**.

We use a **sealed** keyword with an overridden method to create a sealed method. For example,

```
using System;

public class Person
{
    public virtual void Greet()
    {
        Console.WriteLine("The person says hello.");
    }
}

public class Employee : Person
{
    public sealed override void Greet()
    {
        Console.WriteLine("The employee greets you.");
    }
}

public class Manager : Employee
{
    //This will produce a compile-time error because the Greet method in Employee is
    //sealed and cannot be overridden.
    //public override void Greet()
    //{
    //    Console.WriteLine("The manager greets you warmly.");
    //}
```

```

}

public class Program
{
    public static void Main(string[] args)
    {
        Person person = new Person();
        person.Greet(); // outputs "The person says hello."

        Employee employee = new Employee();
        employee.Greet(); // outputs "The employee greets you."

        Manager manager = new Manager();
        manager.Greet(); // outputs "The employee greets you."

        Console.ReadKey();
    }
}

```

In this example, we have a `Person` base class with a virtual method `Greet`, which can be overridden by derived classes. We then define an `Employee` class that inherits from `Person` and overrides `Greet` with the `sealed` modifier. This means that any class that derives from `Employee` will not be able to override the `Greet` method further.

Finally, we define a `Manager` class that inherits from `Employee` and attempts to override `Greet`, but this will produce a compile-time error because the `Greet` method in `Employee` is sealed and cannot be overridden.

In the `Main` method, we create instances of `Person`, `Employee`, and `Manager` and call their `Greet` methods. The `Person` object outputs "The person says hello.", the `Employee` object outputs "The employee greets you.", and attempting to call the `Greet` method on the `Manager` object will call the inherited `Greet` method from `Employee`.

C# Partial Class

ال `partial class` سيتم استخدامه لما يكون عندك كلاس الكود فيه كبير جدا وعمايز تقسمه علي عدة ملفات بتكتب كلمة `partial` في كل الأجزاء

```

using System;

public partial class MyClass {
    public void Method1() {
        Console.WriteLine("Method1 is called ");
    }
}

public partial class MyClass
{
    public void Method2()
    {
        Console.WriteLine("Method2 is called ");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        MyClass obj=new MyClass();
        obj.Method1();
        obj.Method2();

        Console.ReadKey();
    }
}

```

C# Partial Class

There are many situations when you might need to split a class definition, such as when working on a large scale projects, multiple developers and programmers might need to work on the same class at the same time. In this case we can use a feature called **Partial Class**.

Introduction to Partial Class

While programming in C# (or OOP), we can split the definition of a class over two or more source files. The source files contains a section of the definition of class, and all parts are combined when the application is compiled. For splitting a class definition, we need to use the **partial** keyword.

Example 1:

Here is file named as MyClass1.cs with the same partial class MyClass which has only the method Method1.

```
// File MyClass1.cs
```

```
using System;

public partial class MyClass
{
    public void Method1()
    {
        Console.WriteLine("Method 1 is called.");
    }
}
```

Here is another file named as MyClass2.cs with the same partial class MyClass which has only the method Method2.

```
// File MyClass2.cs
using System;

public partial class MyClass
{
    public void Method2()
    {
        Console.WriteLine("Method 2 is called.");
    }
}
```

Here now we can see the main method of the project:

```
// File: Program.cs
using System;

class Program
{
    static void Main()
    {
        //the code of MyClass is seperated in 2 files class1.cs and class2.cs
        MyClass obj = new MyClass();
    }
}
```

```
obj.Method1();  
obj.Method2();  
  
Console.ReadKey();  
  
}  
}
```

In this example, the class `MyClass` is split into two files (`MyClass1.cs` and `MyClass2.cs`) using the `partial` keyword. The `Main` method in the `Program.cs` file creates an instance of `MyClass` and calls both `Method1` and `Method2`. Although the class definition is split across multiple files, the compiler will combine the two parts into a single class definition, so the code behaves as if `MyClass` was defined in a single file.

Places where `partial` class can be used:

1. While working on a larger projects with more than one developer, it helps the developers to work on the same class simultaneously.
2. Codes can be added or modified to the class without re-creating source files which are automatically generated by the IDE (i.e. Visual Studio).

Things to Remember about Partial Class

The `partial` keyword specify that other parts of the class can be defined in the namespace. It is mandatory to use the `partial` keyword if we are trying to make a class partial. All the parts of the class should be in the same namespace and available at compile time to form the final type. All the parts must have same access modifier i.e. `private`, `public`, or so on.

- If any part is declared abstract, then the whole type is considered abstract.
- If any part is declared sealed, then the whole type is considered sealed.
- If any part declares a base type, then the whole type inherits that class.
- Any class member declared in a partial definition are available to all other parts.
- All parts of a partial class should be in the same namespace.

Introduction to Partial Methods

ال partial method ماتقدرش تستخدمها غير جوا ال partial class ماتقدرش تعمل جوا كلاس عادي وكل جزء بيكون في partial class غير الثاني لو ماعملتهاش implementation مش هيضرب منك وهيشتغل عادي ومن استخداماتها هوا انك تعمل code generator ومن فوايدها ان الكود بيفضل نضيف

```
using System;

public partial class Person
{
    public int Age { get; set; }

    partial void PrintAge();

    public void Birthday() {
        Age++;
        PrintAge();
    }
}

public partial class Person
{
    partial void PrintAge()
    {
        Console.WriteLine("Current age: {0}", Age);
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Person person1=new Person();
        person1.Age = 25;
        person1.Birthday();
        Console.ReadKey();
    }
}
```

Introduction to Partial Methods

A partial class may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls are removed at compile time.

Example 2:

```
// File: Person.cs
public partial class Person
{
    public int Age { get; set; }

    partial void PrintAge();

    public void Birthday()
    {
        Age++;
        PrintAge();
    }
}

// File: PersonPrinting.cs
public partial class Person
{
    partial void PrintAge()
    {
        Console.WriteLine("Current age: {0}", Age);
    }
}

// File: Program.cs
class Program
{
    static void Main()
    {
        //the code of Person Class is seperated in 2 files Person1.cs and PersonPrinting.cs
        Person person1 = new Person();
        person1.Age = 25;
        person1.Birthday(); // Output: "Current age: 26"
    }
}
```

In this example, the `Person` class declares a partial method `PrintAge`, which prints the current age of the person. The `Birthday` method of the `Person` class calls the `PrintAge` partial method after incrementing the person's age. If the `PrintAge` is implemented it will be called, if not implemented the compiler will ignore it.

The `PersonPrinting.cs` file provides an implementation of the partial method, which writes the current age to the console.

When the `Main` method in `Program.cs` creates an instance of `Person` and calls its `Birthday` method

Things to remember about Partial Method

- `partial` keyword.
- return type `void`.
- implicitly `private`.
- and cannot be `virtual`.

Partial methods are a feature in C# that allow you to declare a method in one part of a partial class, but provide its implementation in another part of the same class. Partial methods are optional, and you can use them when you want to allow other parts of your code to optionally provide an implementation for a specific method.

Here are some scenarios where you might use partial methods:

1. **Code generation:** When you are generating code using a tool or framework, you can use partial methods to generate the method signature in one file and provide its implementation in another file. This allows you to separate the generated code from the manually written code and makes it easier to maintain.
2. **Performance optimization:** You can use partial methods to write code that can be optimized by different compilers or environments. For example, you can write a partial method that uses platform-specific code to achieve better performance on a particular platform.

3. Framework design: You can use partial methods to provide a hook for external developers to customize the behavior of your framework. For example, you might provide a partial method that is called at a specific point in your framework's execution and allow external developers to provide their own implementation of the method.
4. Code organization: You can use partial methods to organize your code by splitting a large method into smaller parts, each with its own file. This can make it easier to navigate and understand your codebase.

It's important to note that partial methods can only be defined in a partial class or partial struct, and their return type must be void. Also, partial methods cannot be accessed outside of the partial class or struct where they are defined, so they can't be used to implement a public API.

4th Principle/Concept in OOP - Polymorphism

ال polymorphism زي ماقولنا قبل كده معناه تعدد الاشكال وهو ليه أنواع :-

- 1- Compile time polymorphism :- اللي هوا ال overloading
- 2- Runtime polymorphism :- اللي هوا ال overriding وال shadowing
- 3- Inheritance برضه من اشكال ال polymorphism

4th Principle/Concept Of OOP -Polymorphism

Polymorphism in C# refers to the ability of an object to take on multiple forms, i.e., objects of different types can be treated as objects of a common base type.

Polymorphism in C# refers to the ability of an object to take on multiple form.

C# supports two types of polymorphism:

1. compile-time polymorphism (also known as method **overloading**) : Method overloading allows multiple methods to have the same name, but with different parameters. The compiler selects the appropriate method to call based on the number, types, and order of the parameters.
2. Runtime polymorphism (also known as method **overriding**): Method overriding allows a subclass to provide a specific implementation of a method that is already

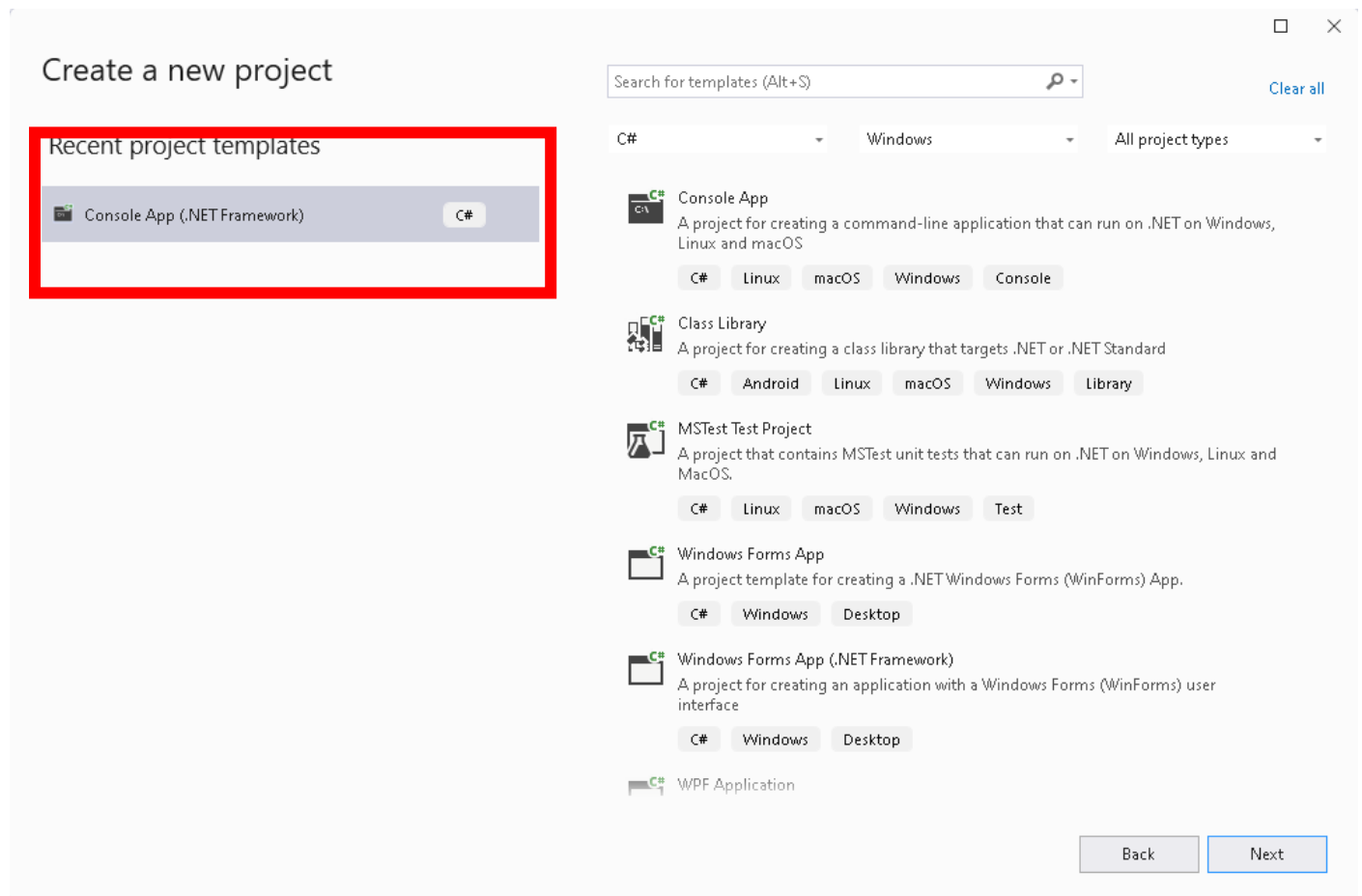
provided by its parent class. The method in the subclass must have the same signature (name, return type, and parameters) as the method in the parent class.

inheritance is also a form of polymorphism known as "subtyping" or "subtype polymorphism".

.NET Class Library

فيه حاجه اسمها class library ودي عباره عن انك بتحط الكود في ملف منفصل تقدر تستخدمه في أي نوع من التطبيقات وحتى ينفع لو بتعمل لعبه بال C#

تعالى نعمل console application



Configure your new project

Console App (.NET Framework)

C#

Windows

Console

Project name

TestMyFirstClassLibraryProject

Location

E:\projects\source\repos

Solution name ⓘ

TestMyFirstClassLibraryProject

☐ Place solution and project in the same directory

Framework

.NET Framework 3.5

Project will be created in "E:\projects\source\repos\TestMyFirstClassLibraryProject\TestMyFirstClassLibraryProject\"

Back

Create

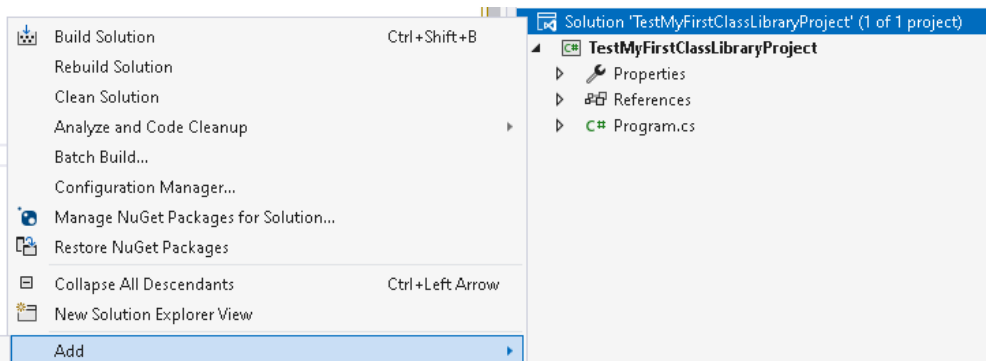
وبعدين نعمل جملة طباعه عاديه

```
using System;
```


```
internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("hi");
    }
}
```

دلوقتي عاوزين نعمل المكتبة


من القايمه نعمل كليك يمين علي ال solution ونختار add وبعدين new project





وبعدین نختار CLASS LIBRARY .NETFRAMEWORK


Search for templates (Alt+S)  [Clear all](#)


C# Windows Library

**Class Library**
A project for creating a class library that targets .NET or .NET Standard
C# Android Linux macOS Windows Library

**WPF Class Library**
A project for creating a class library that targets a .NET WPF Application
C# Windows Desktop Library

**WPF Custom Control Library**
A project for creating a custom control library for .NET WPF Applications
C# Windows Desktop Library

**WPF User Control Library**
A project for creating a user control library for .NET WPF Applications
C# Windows Desktop Library

**Class Library (.NET Framework)**
A project for creating a C# class library (.dll)
C# Windows Library

Configure your new project

Class Library (.NET Framework) C# Windows Library

Project name
MyFirstClassLibrary

Location
E:\projects\source\repos\TestMyFirstClassLibraryProject ...

Framework
.NET Framework 3.5

Project will be created in "E:\projects\source\repos\TestMyFirstClassLibraryProject\MyFirstClassLibrary\"

Back Create

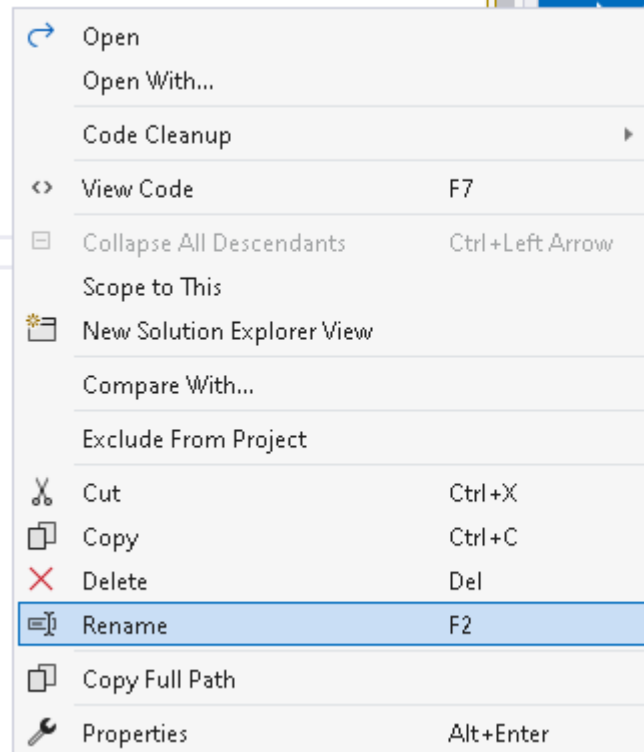
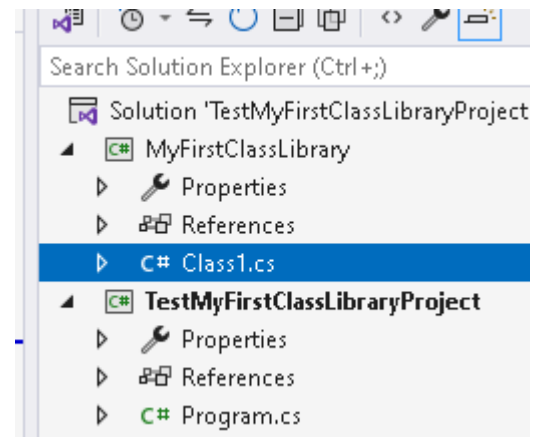
```
using System;

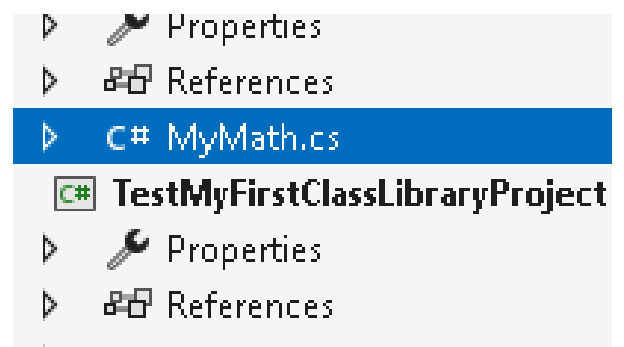
namespace MyFirstClassLibrary
{
    public class MyMath
    {
        public int Sum(int x, int y) { return x + y; }

        public int Sum(int x, int y, int z) { return x + y + z; }
    }
}
```

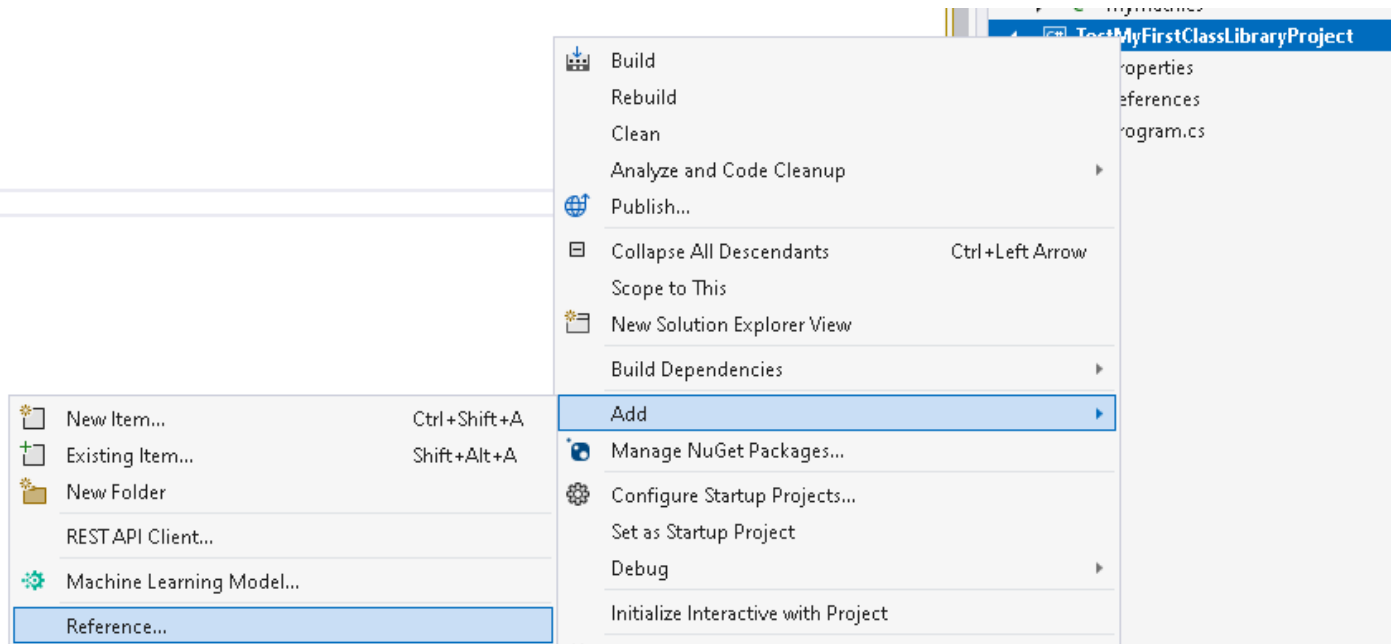
وبعدين بتعمل start

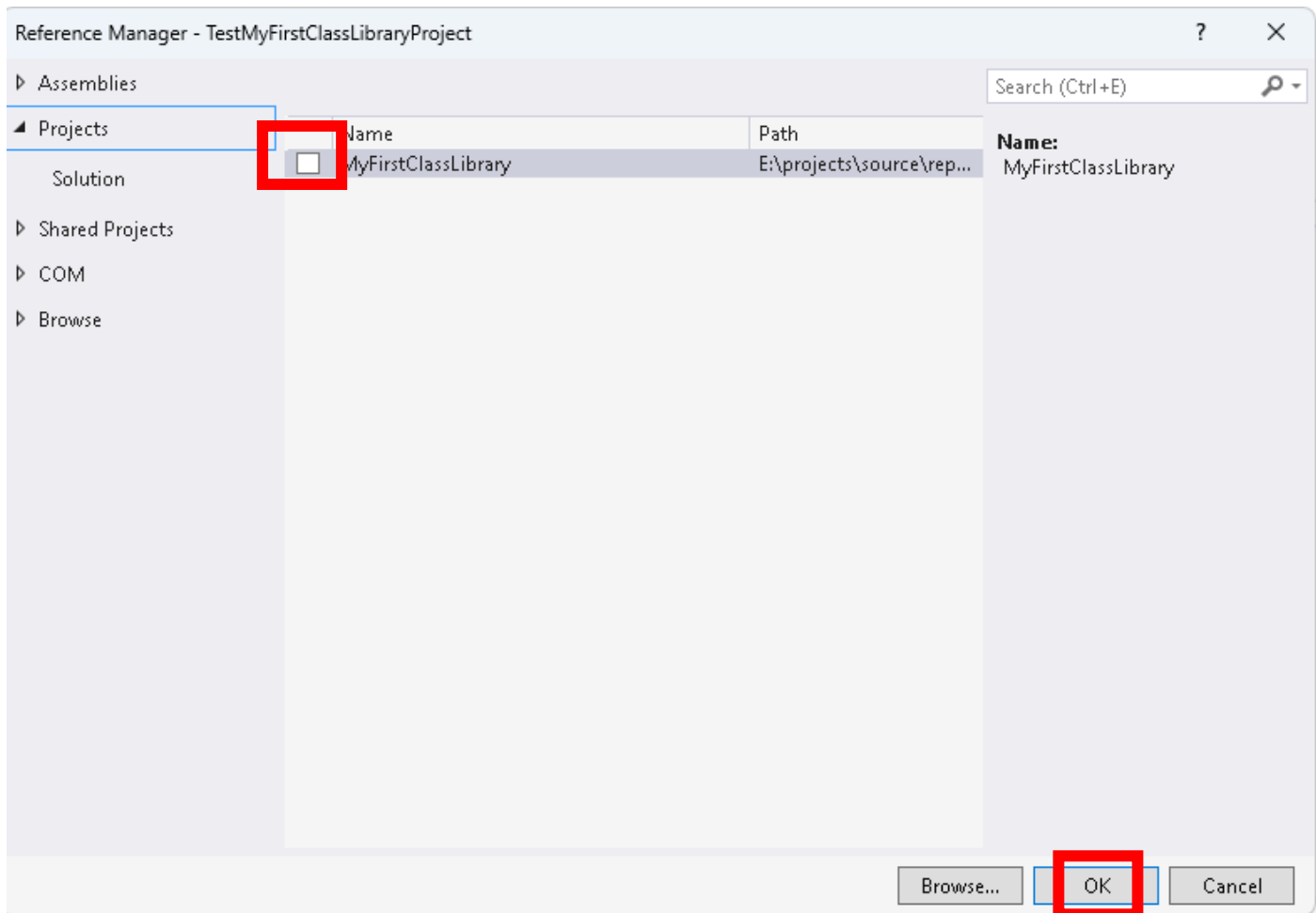
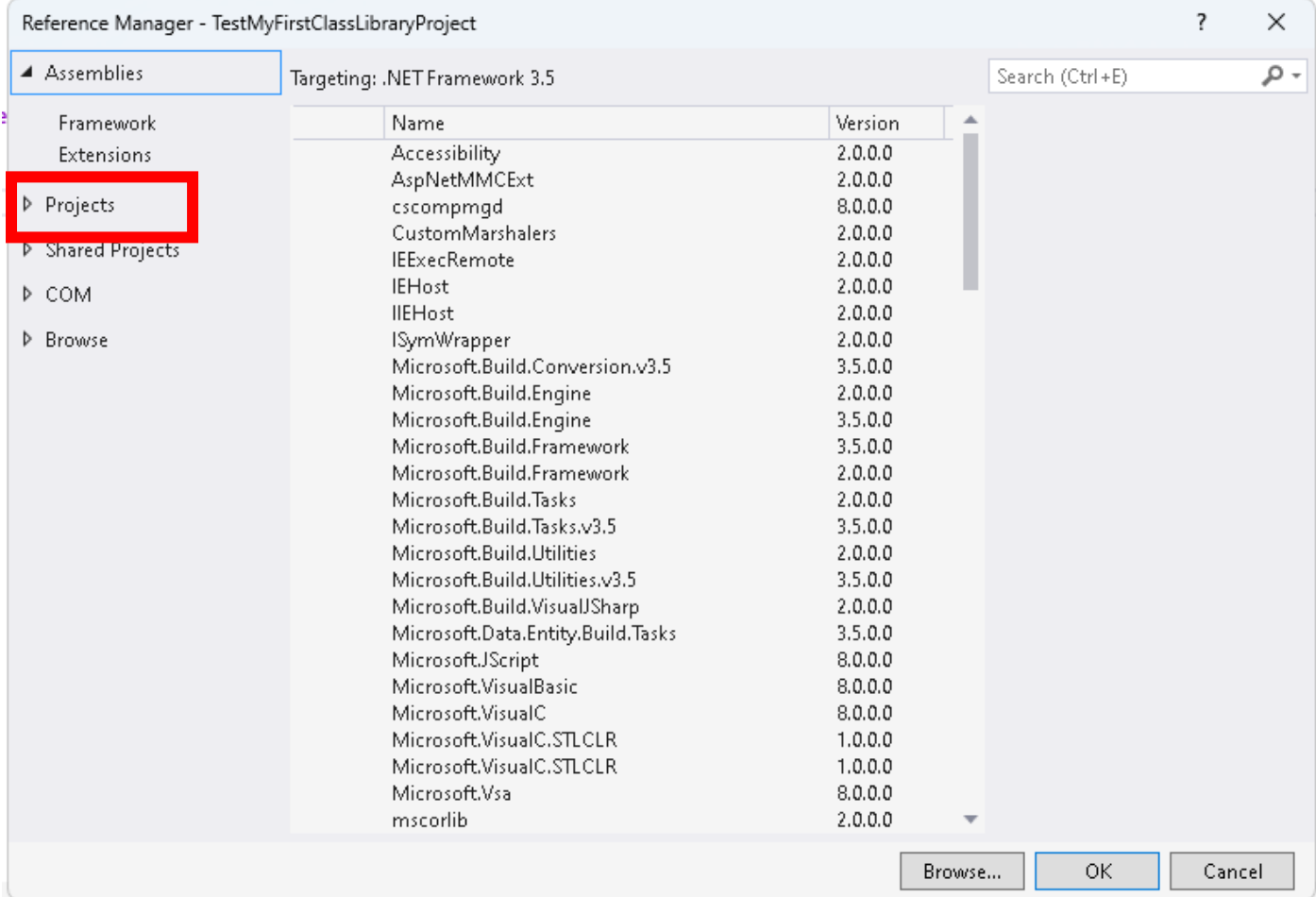
ال name space ده اللي هتستدعي منه الكلاس



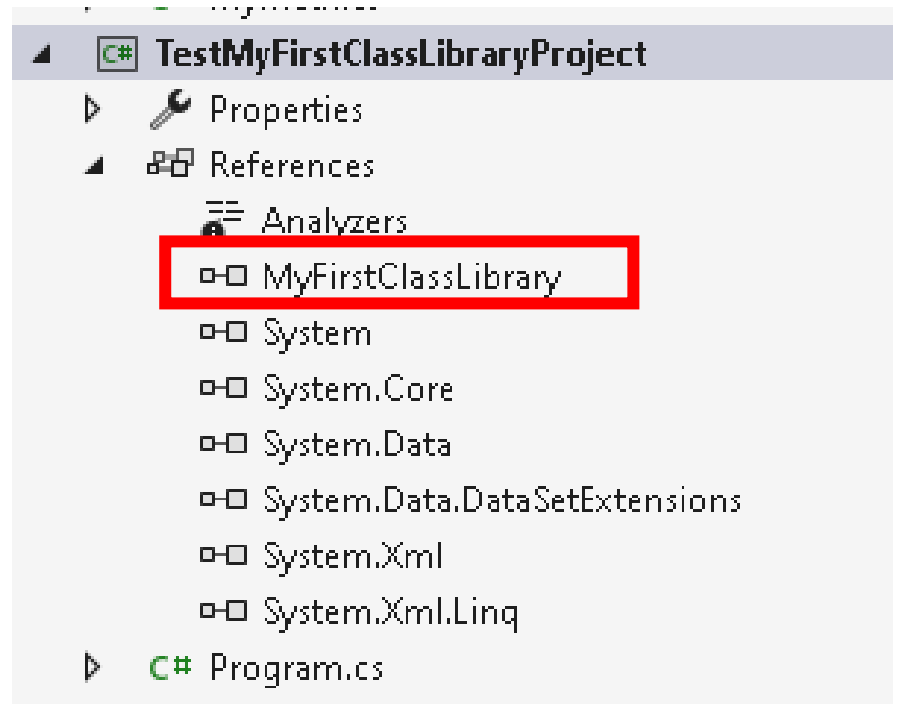


دلوقتي عشان نشغل المكتبة دي بنروح عمل كليك يمين علي ال project اللي هنستخدمها فيه ونختار
add reference





Name	Path	N.
<input checked="" type="checkbox"/> MyFirstClassLibrary	E:\projects\source\rep...	N.



كده ربطنا المكتبة بالمشروع

تعالى بقى نستدعي المكتبة

```
using System;
using MyFirstClassLibrary;

namespace TestMyFirstClassLibraryProject
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("hi");
            MyMath myMath = new MyMath();

            Console.WriteLine(myMath.Sum(10, 10));
            Console.ReadKey();
        }
    }
}
```



هنا بيقولك انه المكتبة اللي عملتها دي بتطلع بصيغة dll تقدر تعملها reference في أي مشروع انت علوزه


Name




Create a new project


Recent project templates


-  Class Library (.NET Framework) C#
-  Console App (.NET Framework) C#


Search for templates (Alt+S)  [Clear all](#)


C# Windows Library


 **Class Library**
A project for creating a class library that targets .NET or .NET Standard
C# Android Linux macOS Windows Library

 **WPF Class Library**
A project for creating a class library that targets a .NET WPF Application
C# Windows Desktop Library

 **WPF Custom Control Library**
A project for creating a custom control library for .NET WPF Applications
C# Windows Desktop Library

 **WPF User Control Library**
A project for creating a user control library for .NET WPF Applications
C# Windows Desktop Library

 **Class Library (.NET Framework)**
A project for creating a C# class library (.dll)
C# Windows Library

 **WPF Custom Control Library (.NET Framework)**
Windows Presentation Foundation custom control library
C# XAML Windows Desktop Library

Back Next

Configure your new project

Console App (.NET Framework) C# Windows Console

Project name

Location

 ...

Solution name ⓘ

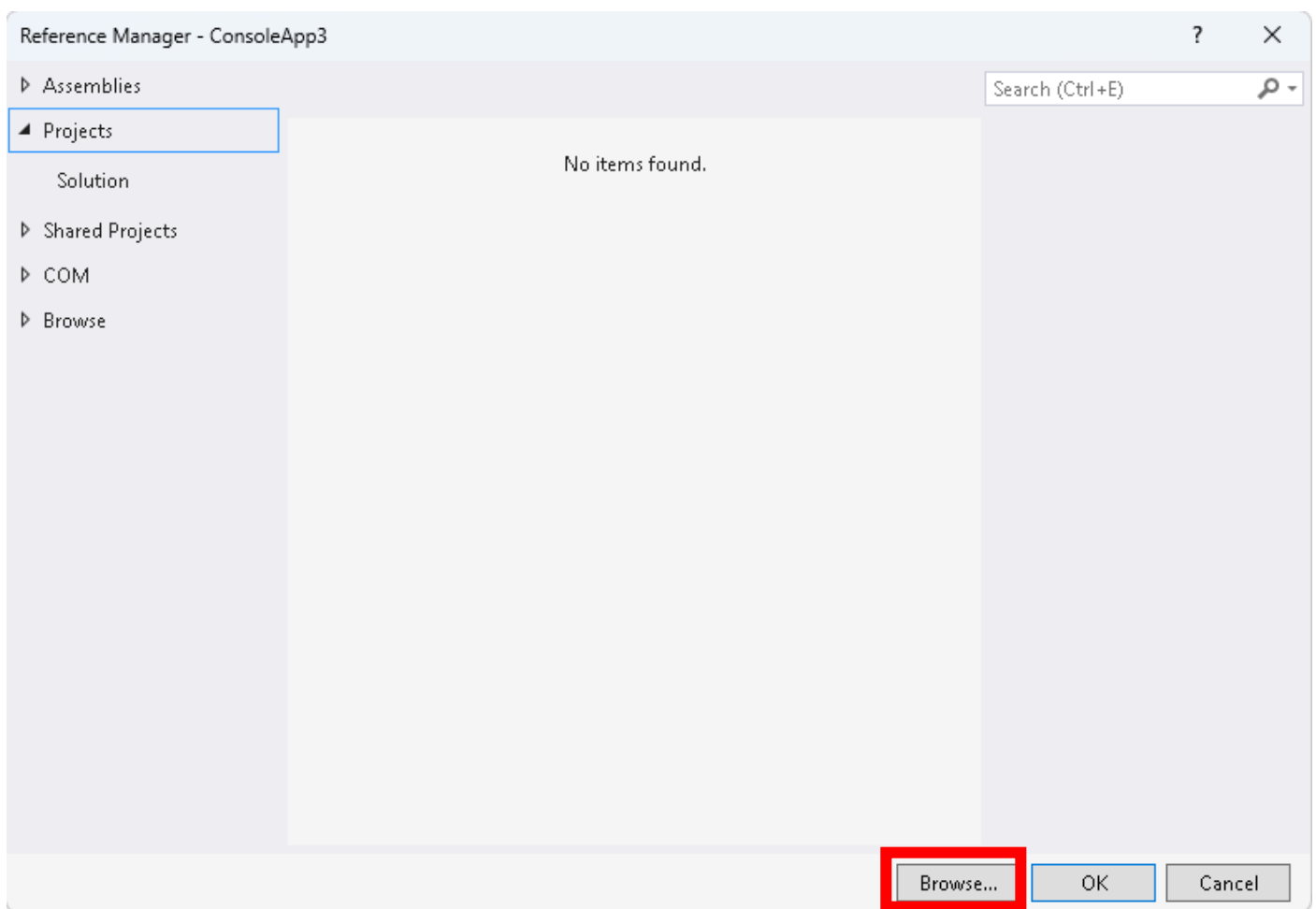
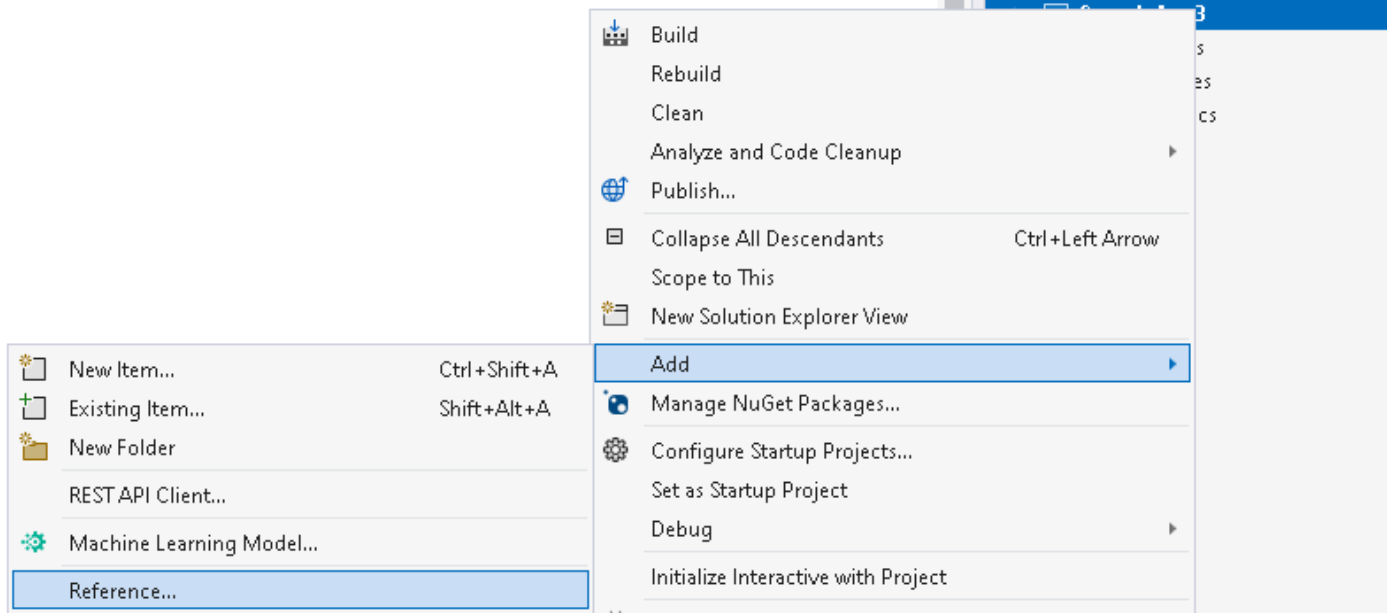
☐ Place solution and project in the same directory

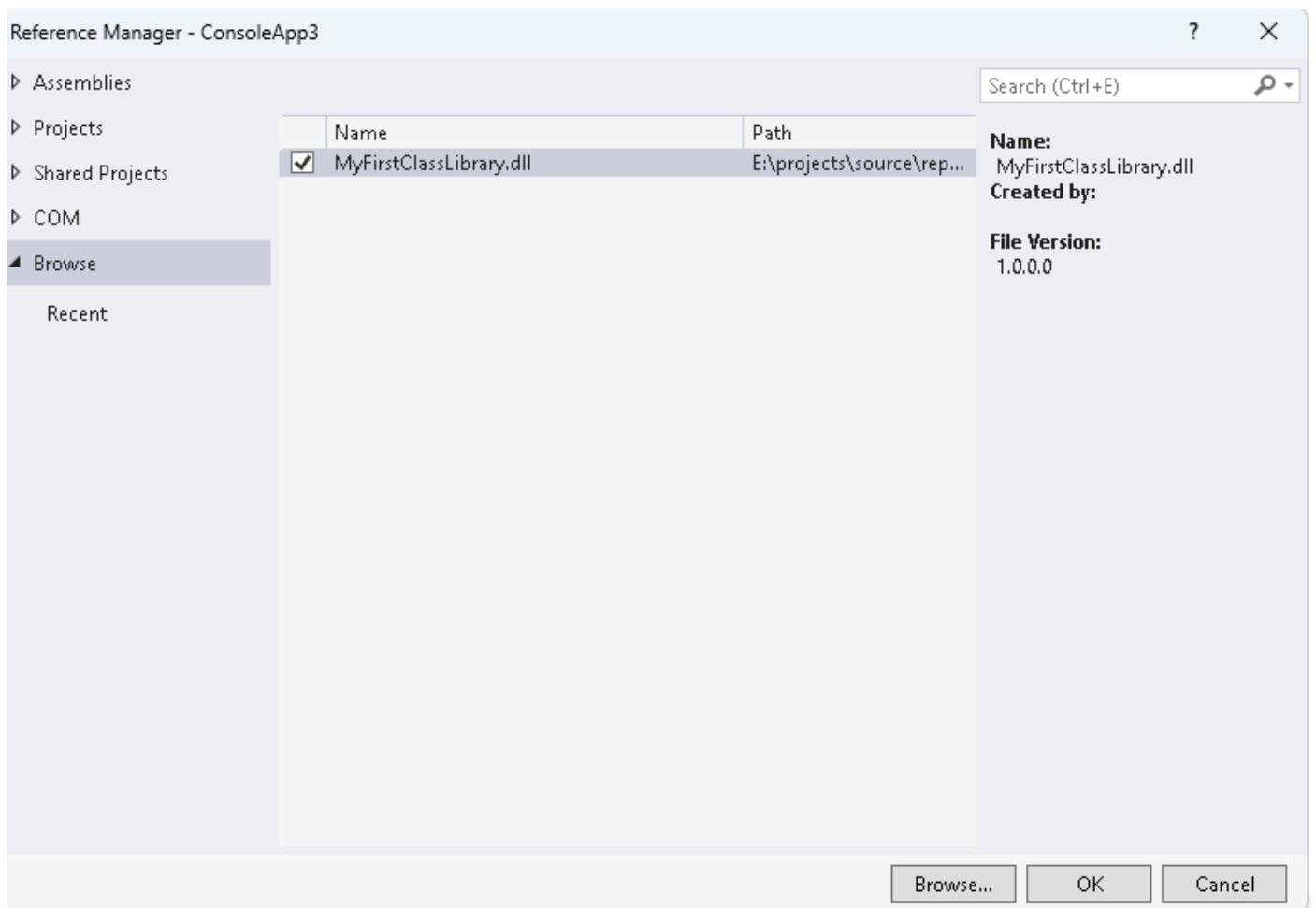
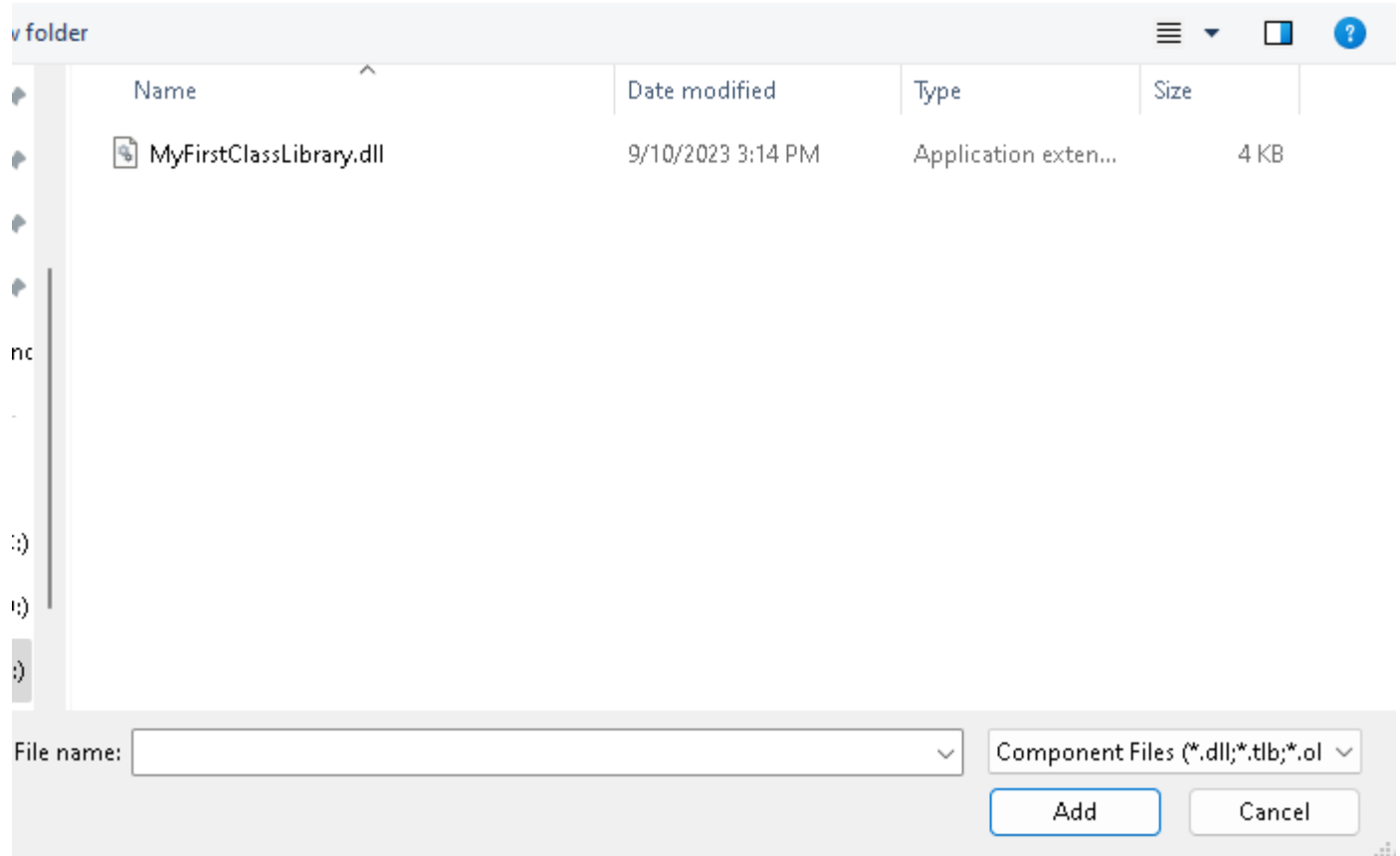
Framework

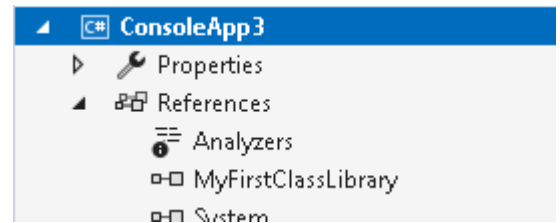
Project will be created in "E:\projects\source\repos\ConsoleApp3\ConsoleApp3\"

Back

Create







```
using System;
using MyFirstClassLibrary;

namespace ConsoleApp3
{
    internal class Program
    {
        static void Main(string[] args)
        {
            MyMath m=new MyMath();
            Console.WriteLine(m.Sum(10,20));
            Console.ReadKey();
        }
    }
}
```

ال DLL هو اختصار DYNAMIC LINK LIBRARY لما بتيجي تعمل build لل
class library بيجمعك الاكواد كلها في assembly

The .NET Class Library

The .NET Class Library is a collection of reusable classes, interfaces, and types that are provided by Microsoft as part of the .NET Framework or .NET Core. These classes provide a wide range of functionality that developers can use to build applications for various platforms such as desktop, web, mobile, and more.

The .NET Class Library is organized into namespaces, each containing related classes and types. These namespaces cover a wide range of topics such as file I/O, networking, data access, cryptography, and more.

Developers can use the classes provided in the .NET Class Library to build applications faster and more efficiently because they don't have to write code from scratch to perform common tasks. Instead, they can use the pre-built classes and types to add the required functionality to their applications.

Additionally, developers can create their own classes and types and include them in the .NET Class Library, making them available to other developers for reuse. This allows for code sharing and collaboration, reducing development time and increasing code quality.

Overall, the .NET Class Library is a valuable resource for developers building applications using the .NET Framework or .NET Core.

Internal Access Modifier

هنا بيفكر بال access modifier اللي نوعه internal اللي كان مع ال public وال private وال protected

وفكرة ال internal هوا انه بيخلي العنصر متاح في ملف ال dll فقط اللي هوا لما بيتعمله build بيتحول لملف assembly وان ال assembly بيكون عبارته عن مجموعه من ال classes وال interfaces وداتا زي المكتبة بالظبط

internal access modifier

When we declare a type or type member as **internal**, it can be accessed only within the same assembly (Same DLL).

An assembly is a collection of types (classes, interfaces, etc) and resources (data). They are built to work together and form a logical unit of functionality.

That's why when we run an assembly all classes and interfaces inside the assembly run together.

Note: Internal in C# is equivalent to friend in C++.

Internal in C# is equivalent to friend in C++.

True

False

Class vs Struct

هنا بيقولك انه ال structure هوا عبارته عن value type يعني زي ال int انما الكلاس هوا reference type

لو عندك struct اسمه x وواحد تاني اسمه y تقدر تقول $x=y$ وبكده هيتم نسخ القيم

انما لو عملت كده وكان ال x وال y كانوا objects من كلاس معين فانت كده خلّيت ال x يشاور عال y من غير ماينسخ القيم وبالتالي لما تيجي تغير قيمه معينه في كلاس منهم هيتغير في الاثنين

Difference between class and struct in C#

In C# classes and structs look similar. However, there are some differences between them.

A class is a reference type whereas a struct is a value type. For example,

```
using System;
namespace CsharpStruct {

    // defining class
    class Employee {
        public string name;

    }

    class Program {
        static void Main(string[] args) {

            Employee emp1 = new Employee();
            emp1.name = "John";

            // assign emp1 to emp2
            Employee emp2 = emp1;
            emp2.name = "Mohammed";
            Console.WriteLine("Employee1 name: " + emp1.name);

            Console.ReadLine();
        }
    }
}
```

Output

```
Employee1 name: Mohammed
```

In the above example, we have assigned the value of emp1 to emp2. The emp2 object refers to the same object as emp1. So, an update in emp2 updates the value of emp1 automatically.

This is why a class is a **reference type**.

Contrary to classes, when we assign one struct variable to another, the value of the struct gets copied to the assigned variable. So updating one struct variable doesn't affect the other. For example,

```
using System;
namespace CsharpStruct {

    // defining struct
    struct Employee {
        public string name;
    }

    class Program {
        static void Main(string[] args) {

            Employee emp1 = new Employee();
            emp1.name = "Mohammed";

            // assign emp1 to emp2
            Employee emp2 = emp1;
            emp2.name = "Ali";
            Console.WriteLine("Employee1 name: " + emp1.name);

            Console.ReadLine();
        }
    }
}
```

Output

```
Employee1 name: Mohammed
```

When we assign the value of emp1 to emp2, a new value emp2 is created. Here, the value of emp1 is copied to emp2. So, change in emp2 does not affect emp1.

This is why struct is a **value type**.

Moreover, [inheritance](#) is not possible in the structs whereas it is an important feature of the C# classes.

In C#, both classes and structures are used to define custom data types that can contain fields, properties, methods, and events. However, there are some differences between them. Here are some of the main differences between classes and structures in C#:

1. **Syntax:** Classes are defined using the "class" keyword, followed by the class name and the class body, which contains the class members. Structures are defined using the "struct" keyword, followed by the struct name and the struct body, which also contains the struct members.
2. **Inheritance:** Classes can be inherited by other classes to create a hierarchy of related classes, whereas structures cannot be inherited or derived from other structures.
3. **Default constructor:** Classes have a default constructor that is automatically provided by the compiler if a constructor is not explicitly defined. Structures, on the other hand, do not have a default constructor and require all fields to be initialized explicitly.
4. **Reference type vs Value type:** Classes are reference types, which means that when an instance of a class is created, a reference to that instance is returned. Structures are value types, which means that when an instance of a structure is created, the value of the instance is returned.
5. **Performance:** Structures are generally faster than classes for small, simple types, as they are stored on the stack rather than the heap. This means that accessing and manipulating a structure's fields can be faster than accessing and manipulating a class's fields.
6. **Memory management:** Since structures are value types, they are allocated on the stack, which is a limited resource, while classes are allocated on the heap, which is a larger, more flexible memory pool. This means that using too many structures or large structures can quickly consume the available stack memory, causing a stack overflow error.

In summary, classes and structures are both used to define custom data types in C#, but they have some differences in syntax, inheritance, default constructors, reference types vs value types,

performance, and memory management. The choice between using a class or a structure depends on the specific needs of the application and the type of data being represented.

C# Enums (Enum is a special class)

هنا يقولك انه ال enum ده عبارته عن كلاس خاص مكون من مجموعه من المتغيرات اللي نوعها constant وده بيخلي الداتا اللي فيه ثابتة ولا يمكن تغييرها

C# Enums

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

To create an **enum**, use the **enum** keyword (instead of class or interface), and separate the enum items with a comma:

Example

```
enum Level
{
    Low,
    Medium,
    High
}
```

You can access **enum** items with the **dot** syntax:

```
Level myVar = Level.Medium;
Console.WriteLine(myVar);
```

Enum is short for "enumerations", which means "specifically listed".

Enum inside a Class

You can also have an **enum** inside a class:

Example

```
class Program
```

```
{  
    enum Level  
    {  
        Low,  
        Medium,  
        High  
    }  
    static void Main(string[] args)  
    {  
        Level myVar = Level.Medium;  
        Console.WriteLine(myVar);  
    }  
}
```

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

True

False

An enum is a special "class" .

True

False

Course Complete

تم بحمد الله الانتهاء من هذا الكورس

نلتقاكم على خير في الكورس القادم بإذن الله تعالى

The End