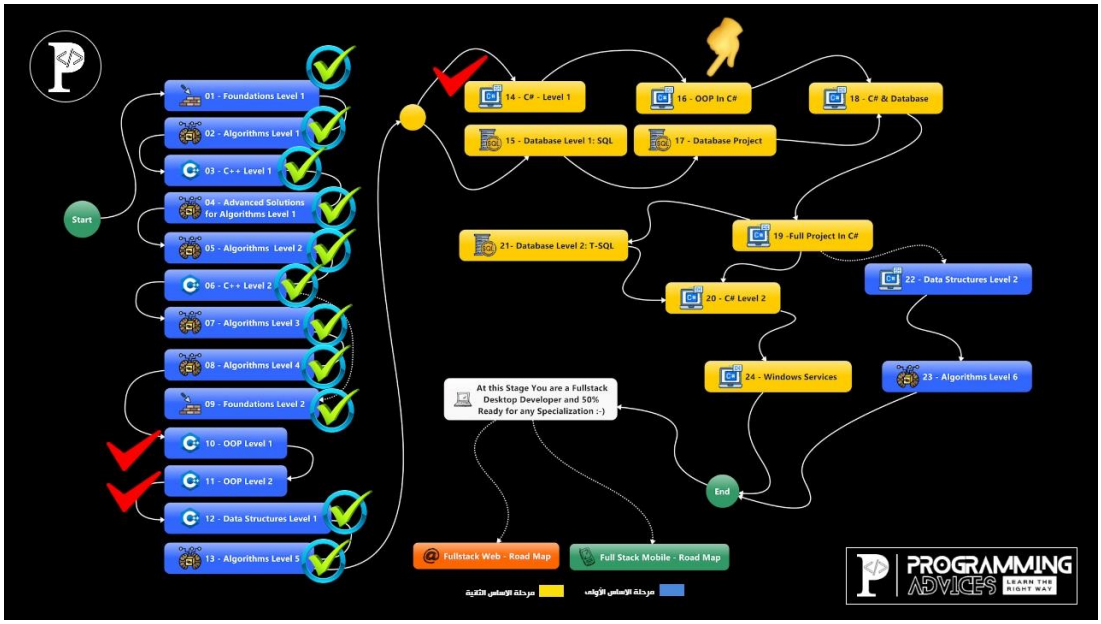
بسم الله الرحمن الرحيم

الفهرس

Important Introduction



مراجعة المفاهيم والمبادئ و Const بشكل عام للكورس 10 : OPP Level 1 C++
لأنها مشتركة بين جميع اللغات مع اختلافات و Feature بسيطة

والكورس 11 هو تطبيق للكورس 10

والكورس 14 لمعرفة Syntax ل C#

سيتم دراسة Class libraries و كيف تعمل Class project وكيفية استخدمها مع project آخر
بمعنى آخر تستخدمها في Consol / web / desktop / Mobile Application أو أي نوع آخر

Lesson 1 : (Revision))What is Object Oriented Programming? And Why ?

Function يعمل شغلة واحدة فقط

Function Programming (FP) هي مجموعة من Function و Procedure التي تبني البرنامج ، يتم استدعاءها وقت الحاجة إليها

الفرق بين Functional (FP) vs Object Oriented Programming (OOP) ؟

مثال لإنشاء نظام للجامعة University System

باستخدام FP : سيتم كتابة العديد والعديد من **Functions** لأنه مشروع كبير جدا ، فيتم تجزيئه الى **Small Function** ثم يتم تركيب Function بعضه على بعض مثل **Lego**

عند استخدام (**FP**) في المشاريع الكبيرة : يكون عندك آلاف Functions ستنسى بعض Function – ك إعادة كتابة Function موجود – + يكونوا غير مرتبين ، وعددهم هائل جدا فلا تستطيع تذكرهم ، طريقة التعامل ب FP في الأنظمة الكبيرة تكون شبه مستحيلة

المشكلة ليست في عدد Functions الهائل وإنما في طريقة تنظيمهم – ونظرتك لهم أو تعاملك معهم

ليست المشكلة في عدد Functions الهائل لطالما أنهم منظمين تستطيع الوصول إليهم بسهولة = Object Oriented Programming (OOP)

الفرق بين FP vs OOP : أن OOP تغير نظرتك للكود بشكل أقرب للواقع أو الحياة العملية أي تجعلك تفكر في تعاملك للكود كأنك تتعامل مع الحياة الواقعية ، تتعامل مع Functions عن طريق Object

باستخدام OOP : ماهي **الأشياء أو الكائنات Object** التي تريد برمجتها في الجامعة ؟
(طلاب ، كورس ، موظفين ، دكتور ، الأقسام أو الكليات ، التخصص) كل شيء من هذه الأشياء يسمى Object لذا **أنت تبرمج أشياء Object وليس Functions** + أنك تفكر في الكود من فوق الى تحت أو من الكليات الى الجزئيات وليس العكس
يتم توزيع آلاف أو مئات Functions تحت ما يناسبه – أي له علاقة – في Object مثال طلاب **Student Object == يسمى Class Student** يندرج تحته كل Members – كل شيء له علاقة ب Student – هي تشبه Structure مبدئيا لسهولة الشرح لكن بإمكانك إضافة Method بداخلها

من فوائد استخدام OOP

- أنك تتعامل مع class عن طريق **Object** يندرج تحت العديد من Functions
- تعطيك مفاهيم أو ميزات عديدة تجعلك تختصر كتابة الكود أو إعادة استخدامه
- تحكم في الوصول – أو عدمه – الى **Method** ل Developer == أمان للكود
- OOP تعطيك تحكم كامل في الكود وأمان أكثر وإعادة استخدام للكود

لدى OOP المزيد من المفاهيم والطرق التي تتيح لك التعامل مع Code الخاص بك بطريقة أسهل بكثير ، ولديك سيطرة أكبر على Code الخاصة بك

مصطلحات برمجية

Method يكون في class وهو Function & Procedure

Members هي Variables & Method

Paradigms إما OOP vs FP

Lesson 2 : Class & Object

OPP تصنف Functions الكثيرة تحت Class يجمعهم علاقة مثل الطلاب ، المعلمين ، الكليات ...

اسم Class جاء من Classification وهو التصنيف[1]

Class تعتبر Data Type و Data Structure

ولابد من إنشاء Object للاستفادة منها أو استخدامها ، للوصول الى Method

Class مثل المخطط للمنزل و Object مثل المنزل على أرض الواقع – فتشئ أكثر من منزل بمخطط واحد -

Object هو جزء أو حبة من Class

وطريقة إنشاء Class مثل الطريقة في C++ وتختلف في طريقة إنشاء Object

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

Answer	Question
Class is the blue-print of object Class is a Datatype Class is a Data-structure	**What is Class ?**
True	**You can have multiple objects from the same class**
True	**Any Function or Procedure inside class is called "Method"**
True	**Object is an Instance of class**
True	**C# is an Object Oriented Language**

[1] راجع الكورس 10 : Lesson 2: Classes and Objects

True	**Class members means any variable or function inside the class is called "Member"**
True	**Data Member is any variable inside the class that holds data**
True	**Function Member is any function or procedure inside a class**
True	**Class Members are Data Members and Function Members**

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
    {



    //Fileds
    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 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.`new`

```
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
    {

        //Fileds
        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 dont 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.

Lesson 3 : Objects In Memory

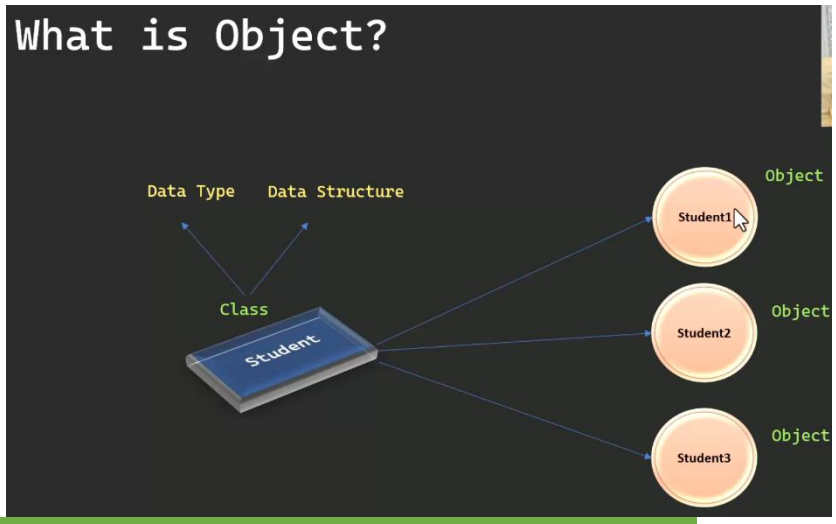
كيف يتم تمثيل Objects In Memory[2]

هل Members كلها يتم تمثيلها في Memory ؟
أم فقط Data Members ؟

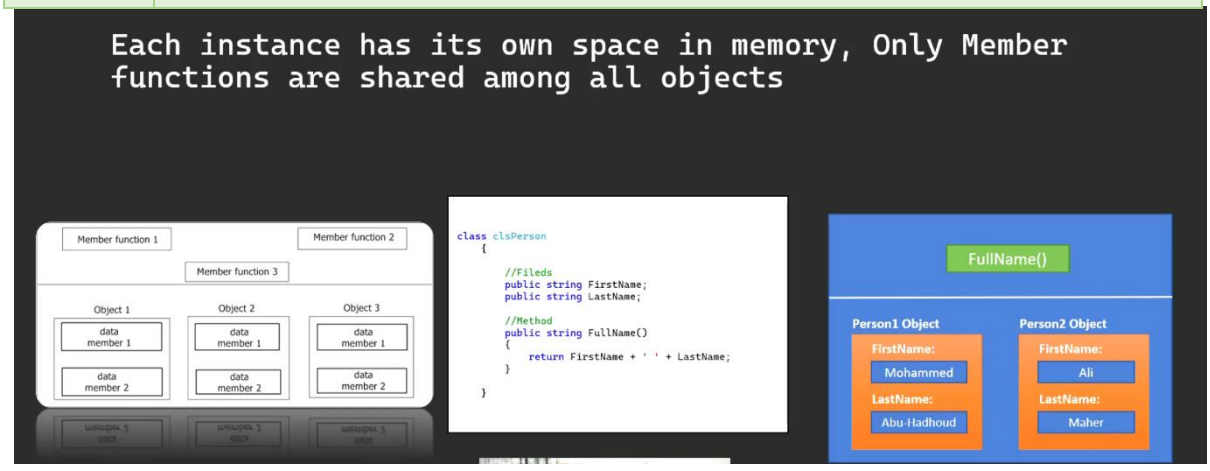
كل Objects له مساحة في Memory لتخزين Data Members – تحفظ البيانات –
وكل Functions / Methods التي في Class لها مكان واحد فقط في الذاكرة لكل Objects

```
 //Fields Or Property Or Member
// Data Members
public string FirstName;
public string LastName;


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



Answer	Question
True	**Function Members are shared to all objects in memory and has one memory space for them**
False	**Every Object has it's own space in memory that hold both Data / Function Members**
True	**.Every Object has it's own space in memory that holds only Data Members**



[2] راجع الكورس 10 : Lesson 4: Object In Memory

Lesson 4 : Access Modifiers

Access Modifiers[3] أنواع الصلاحيات للوصول الى Members في class وهي

١. **Private** : تستطيع الوصول الى Members من داخل Class فقط
٢. **Protected** : يكون داخل Class فقط + كل Classes التي ترث Inheritance هذه Class
٣. **Public** : أي أحد يستطيع الوصول إلى Members من أي مكان
٤. **Internal** : يمكن الوصول إليها فقط من داخل نفس المشروع وليس من مشرع آخر
   a. من داخل **assembly** سيدرس لاحقا –
   b. **ولا يوجد في C++** Internal وإنما يشبه friend في C++

Answer	Question
Private Protected Public Internal	**Which of the following is Access Specifiers/Modifiers**
No, You have to declare an object of the class first, and access all members and methods through the object not .class	**Can you access class members and methods directly**
ObjectName.FunctionName();	**How to access member function of class using Object**
No, only public members and methods can be accessed through the object, all private members and methods are for internal use inside the class	**If you have a private member or method in class, can you access this member or method through Object**
True	**Access modifiers (or access specifiers) are keywords in object-oriented languages that set**

[3] راجع الكورس 10 : Lesson 5 : Access Specifiers / Modifiers

	the accessibility of classes, methods, and other members
True	**Public Members can be accessed from inside and outside the class**
False	**Private Members can be accessed from outside the class through object**
False	**Private Members can be accessed by any class .inherits the current class**
True	**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**
Protected	**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**
False	**Protected Members can be accessed from outside class through objects**
True	**Protected Members can be accessed from inside class and from all classes inherits the current class**
True	**OOP is more secured because you can hide members from developers**
True	**Inside the class I can access everything including Public, Private , and Protected Members**
True	**When we declare a type or type member as internal, it can be accessed only within the same assembly**
True	**If we use internal within a single assembly, it works just like the public access modifier**
True	**internal in C# is equivalent to friend in c++**

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 ca
n 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 accessable and internal
            Console.WriteLine("All public members are accessable");
            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 me
mbers 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, lets focus on `public` and `private` and `protected`modifiers.

Lesson 5 : Static Members

عند إنشاء أكثر من **Object** لنفس **Class** : يتم حجز مساحة لكل **Members العادية** في الذاكرة –
فالتعديل على Object معين لا يؤثر على أي Object آخر –

أما عند تعريف Members بأنها Static / Sherd[4] فإنها تصبح مشتركة بين جميع Objects وتكون لها مساحة واحدة في الذاكرة على مستوى Class ، فأي تعديل عليها من أي Object يؤثر على كل Objects

في C# لا يسمح لك بالتعديل أو مناداة Static / Sherd من Object وإنما عن طريق Class بخلاف C++

```
clsA.x2 = 100;
```

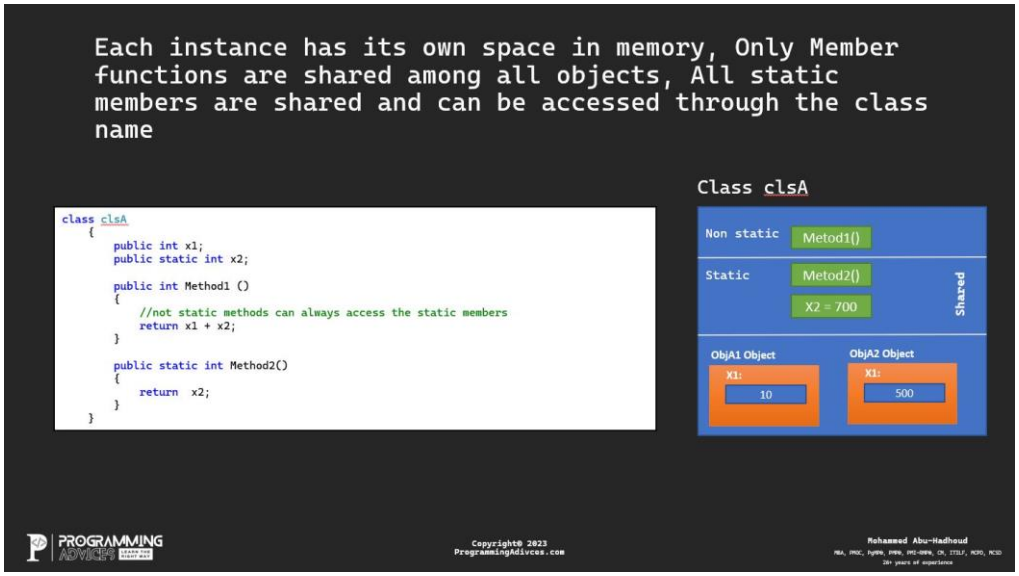
قاعدة مهمة في Static / Sherd
Static Method (Function) لا تعدل إلا على Static Member (Variable) فقط

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.

[4] راجع الكورس 10 : Lesson 16 & 17 : Static Members (Variable) & Methods (Functions)




```
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 memebers because the
re 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 التعامل معها في C# أسهل بكثير من C++

عند استخدام Private يفضل وضع شرطة سفلية قبل الاسم _Name

تعرف Function من نوع public بداخله Set & Get[5]

من فوائد استخدام Set : تخزين تاريخ التعديل أو تخزين القيمة القديمة في Data base

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
```

[5] راجع الكورس 10 : Lesson 6 : Properties Set and Get

```
        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.

Lesson 7 : ReadOnly Properties

ReadOnly Properties تجعل Developer لا يستطيع التعديل على القيمة وإنما قراءتها فقط

تعرف Function من نوع public بداخله Get [6] فقط

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
```

[6] راجع الكورس 10 : Lesson 7 : Read Only Property

```
        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.

Lesson 8 : Auto Implemented Properties

في الدروس السابقة عند استخدام Set & Get

- نعرف Member (variable) من نوع Private
- ونعرف Method (Function) من نوع public
  - بداخله Two Function وهما : Set & Get أو Get فقط

يوجد في C# اختصار Set & Get العادية – تعديل القيمة Set أو ارجاعها Get –

- **لا يتم تعريف Member (variable) من نوع Private وإنما فقط public**

```
//ID Property
public int ID
{
    get;
    set;
}
```

أما عند استخدام Set : لتخزين تاريخ التعديل أو تخزين القيمة القديمة في Data base مثلا فلابد من استخدام الطريقة الطويلة

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.

Lesson 9 : Static Properties & Static Class

Static Members Properties لا يختلف عن **Members Properties**

Static يتم الوصول الى Method عبر Class وليس Object

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.

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.

Lesson 10 :(Revision) First Principle or Concept of OOP- Encapsulation

OOP لها أكثر من مبدأ أو مفهوم منها : Encapsulation

الذي تعلمناه في الدروس السابقة أن : Class هو فئة يندرج تحته كل Function & Procedure or Variable التي لها علاقة بهذه الفئة وتسمى **Encapsulation**

Encapsulation : مأخوذة من كبسولة الدواء البلاستيكية التي هي Class التي تجمع كل Methods ذات العلاقة تحت سقف واحد ولها جزئين :

١. Methods وتسمى Member Method (Function)
٢. Variables وتسمى Data Members

ولا تستطيع الوصول إليهما من خارج Class إلا عن طريق Object

Encapsulation له علاقة في إخفاء البيانات التي هي **Property**

Answer	Question
True	**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**

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

Lesson 11 : (Revision) Second Principle or Concept of OOP - Abstraction

Abstraction هذا المفهوم مهم جدا جدا في البرمجة – ستتم دراسته نظريا –

مثال لتصوير صورة في هاتفك : تفتح التطبيق ثم تضغط على زر التصوير
هذا الزر عبارة عن Function اسمه Take Picture ، وفي هذا Object يوجد Functions كثيرة أخرى تم إخفاؤها عنك إما بسبب Security وليس هو السبب الرئيسي **وإنما أخفوها لأنك كمستخدم لا تهمك مثل الأشياء المعقدة التي لا تفيدك**

مثال آخر `String S1.size()` تظهر **Method** بعد **.** التي تحتاجها في `String`
يوجد في `String` Class تقريبا ضعف Method الظاهرة في object التي تستخدم داخل Class لكنها لا تفيدك أو لا تهمك كمستخدم

من أوائل من سيستخدم Class الخاص بك عن طريق Object هم المبرمجين لذلك تظهر لهم Method التي يحتاجونها فقط

Abstraction تظهر لك فقط Art tribute وهي **Property** & Method عناصر Class التي تهمك – المطور أو المستخدم – فقط

في OOP تفكر أيضا في تصميم البرنامج ومن سيستخدم هذه Class الخاصة بك ، وهدف المطور هو تسهيل حياته أو حياة الاخرين

Abstraction يختلف تماما عن Abstract Class – ستدرس لاحقا –

Answers	Question	**Quiz**
True	In simple terms, abstraction “displays” only the relevant attributes of objects and “hides” the unnecessary details	
Through Private Members Only	You Achieve Abstraction ?	

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.

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 Cancelling Last Operation 0 is: 45

Result After Clear 0 is: 0

```
using System;

// Calculator هذه Class يندرج تحتها كل Members التي تتعلق ب
class clsCalculator
{
    // Abstraction هذا هو مبدأ
    // object عن المستخدم في private Members إخفاء
    // Abstraction وتسمى Class هذه private لا يستطيع مناداتهم إلا من داخل
تحت هذا
    private float _Result = 0;
    private float _LastNumber = 0;
    private string _LastOperation = "Clear";

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

```
// Object هذه كل Method التي يستطيع المستخدم استخدامها في
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)
        {
        // Encapsulation هذا هو
        // object لا يتم الوصول إليها إلا عن طريق Members كل
```

```
        // Class من نوع  Object / Instance عبارة عن نسخة من Calculator1
        // Object إلا عن طريق Class لا تستطيع الوصول الى
        clsCalculator Calculator1 = new clsCalculator();

        // Public من نوع Class الموجودة في Method توصلك الى Object هذه
        //Abstraction التي تظهر للمستخدم هي التي يحتاجها أو التي تهمه
هذه Method فقط وهذه تسمى

        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();
    }
}
```

Lesson 14 : Constructor

Constructor & Destructor من أهم التي يجب إتقانها في OOP – ستتم دراستها بأساليب مميزة –

Constructor[7] هو Function داخل Class له نفس اسم Class
ويتم استدعاء Constructor عند استدعاء new Class – عند إنشاء Object –

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.

[7] راجع الكورس 10 : Lesson 13 : Constructors

Call a constructor

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

```
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:

- Parameter less Constructor
- Parameterized Constructor
- Default Constructor

We will explain everything in the next lessons :-)

Constructor هو Function داخل Class له نفس اسم Class
ويتم استدعاء Constructor عند استدعاء new Class – عند إنشاء Object –

أنواع Constructor

- **Parameter less Constructor بدون Parameterized**
- Parameterized Constructor
- Default Constructor

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; }

// سيتم دراسة private Constructor لاحقا
    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.

أنواع Constructor

- Parameterized بدون Parameter less Constructor
- **Parameterized Constructor** [8]
  - **لا تستطيع إنشاء Object من غير Parameterized**
- Default Constructor

Parameterized Constructor

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

```
using System;

namespace Constructor {
    class Car
    {

        public string brand;
        public int price;

        // parameterized constructor
      public  Car(string theBrand, int thePrice)
        {

            this.brand = theBrand;
            this.price = thePrice;
        }

    }
```

[8] راجع الكورس 10 : Lesson 21 : Parameterized Constructor of the Base Class#

```
    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.

أنواع Constructor

- Parameter less Constructor بدون Parameterized
- Parameterized Constructor
- **Default Constructor – لا يوجد Class من غير** Constructor -
  - **إذا لم تنشئ أي** Constructor **بداخل Class ف Compiler ينشئ** Constructor **بشكل تلقائي من غير** Parameterized **وفاضي – لا يوجد فيه أي Code –**

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.

يوجد طريقتان لمنع إنشاء Object من Class

- عند تعريف Constructor من نوع **Private** لا تستطيع إنشاء Object من Class
- وعند تعريف **Class** أنه **Static** فلن تستطيع إنشاء Object من Class

شرط مهم لابد من تعريف Members أنها Static

Private Constructor

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

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();
    }
}
```

overloading هو عدد من Function لهم نفس الاسم ولكن يختلفوا في عدد Parameter أو في Data type

وبما أن Constructors هو Function فتستطيع إنشاء أكثر من واحد (overloading)

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();
    }
}
```

Lesson 20 : Static Constructor

Static هو على مستوى Class وليس على مستوى Object

تستطيع إنشاء Static Constructor ينادى مرة واحدة في حياة البرنامج ولا يستقبل أي Parameter

أما Static العادي ينادى على كل Object

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 هو عكس Constructor

Constructor عند استدعاء Class – إنشاء Object –

Destructor يتم استدعاؤه عند الانتهاء من Object

عند الانتهاء من Object يتم حذف Object من الذاكرة عبر Garbage Collector فينادي Destructor بشكل تلقائي

Destructor طريقة إنشاءه = ~NameClass()

Destructor

- **لا يستقبل أي Parameter أو Return**
- **هناك Destructor واحد فقط في Class**

من فوائد Destructor حفظ Data في Data base

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, is the destructor.~Test()

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 inside the class.`~clsPersonclsPerson`

When we create an object of the 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.`clsPerson`

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.

Lesson 22 : Very Important: Real life Application of Using Static and Constructors

مشروع تطبيقي على إجبار المستخدم على إنشاء Object معبئ باستخدام Static & Constructors

قاعدة : لا تسمح ل Developer بأن ينشئ Object فارغ – لابد من استخدام Parameterized Constructor – لابد أن يكون معبئ Data سواء من .. Data base / file

```
using System;

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

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

        // لإرجاع Object معبئ
        // Class وهي static للوصول إليها على مستوى
        public static clsPerson Find(int Id)
        {
            // تستطيع مناداة Data base للتحقق من وجوده
            if (Id == 10)
                 return new clsPerson(10, "Saeed", 25);
            else
                return null;
        }
```

```
        public static clsPerson Find(string UserName , string Password )
        {
            // تستطيع مناداة Data base للتحقق من وجوده
            if (UserName == "Saeed" && Password == "p1234")
                 return new clsPerson(10, "Saeed", 25);
            else
                return null;
        }
    }
class Program
{
    static void Main()
    {
            // لإنشاء Object معبئ
            clsPerson person1 = new clsPerson(10, "Saeed", 22);

            Console.WriteLine("Finding person1 by Id");

            // لإرجاع Object معبئ
            // null أو new clsPerson لأن Find يرجعها new لم يتم استخدام
            clsPerson personId = clsPerson.Find(10);

            // إذا كانت null فالبرنامج لن يشتغل //
            //Console.WriteLine("ID : {0}", personId.Id);

            if (personId != null)
            {
                Console.WriteLine("ID : {0}" , personId.Id);
                Console.WriteLine( "Name : {1}" ,personId.Name);
                Console.WriteLine("Age : {2}" , personId .Age);
            }
           else
            {
                Console.WriteLine("Could Not find the Person by the givin ID");
            }
```

```
            Console.WriteLine("Finding person1 by UserName and Password");

            // لإرجاع Object معبئ
            // لم يتم استخدام new لأن Find يرجعها new clsPerson أو null
            clsPerson personUser = clsPerson.Find("Saeed" , "p1234");



            if (personUser != null)
            {
                Console.WriteLine("ID : {0}" , personUser.Id);
                Console.WriteLine( "Name : {1}" , personUser.Name);
                Console.WriteLine("Age : {2}" , personUser.Age);
            }
           else
            {
                Console.WriteLine("Could Not find the Person by the givin UserName
/ Password");
            }



        }
}
```

Lesson 23 : Third Principle/Concept of OOP: Inheritance

Inheritance[9] مهمة جدا لأنها تعيد استخدام Code – من أهم الدروس في البرمجة –

مثال : عندما ترث clsEmployee - Derived Class (child) - من clsPerson - Base Class (parent)
فإنها ترث كل Members / Methods التي في clsPerson من نوع public / protected

ثم عليك إضافة Members / Methods الخاصة في clsEmployee

وطريقة Inheritance في C# هي : clsEmployee : clsPerson

لابد من تعريف – clsPerson - Base Class (parent) من نوع public لترث clsEmployee - Derived Class (child) التي من نوع public

Answer	Question
True	**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	**The class that inherits properties from another class is called Subclass or Derived Class**
True	**The class whose properties are inherited by a subclass is called .Base Class or Superclass**
True	**Derived Class and Sub Class and Child Classes are the same**
True	**Base Class and Super Class and Parent Class are the same**
True	**You can inherit only public and protected members, private members are not inherited**
True	**Relationship between derived class and super class is call "Is-A" because derived class is super class**

[9] راجع الكورس 10 : Lesson 20 : Third Principle / Concept of OOP : Inheritance

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 Empoyee
        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
```

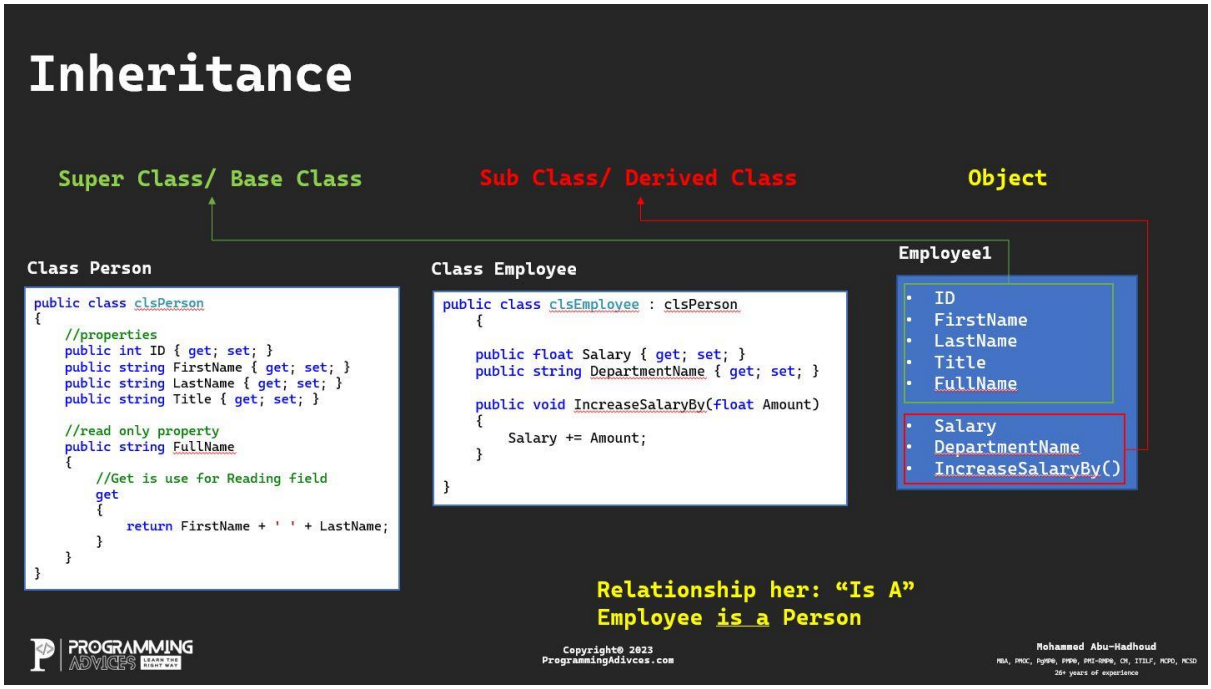
```
Title := Mr.
Full Name := Mohammed Abu-Hadhoud
Department Name := IT
Salary := 5000
Salary after increase := 5100
```

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.



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.

Lesson 24 : Inheritance Constructor

عندما ترث من **Base Class** وهي تتطلب Constructor Parameterized فتنشئ Derived Class فيه Parameterized الخاصة Base Class + Constructor Parameterized الخاصة Derived

ولابد أن يكون Constructor من نوع public

قاعدة : لا تجعل أي أحد ينشئ Object فارغ من Class (أي بدون أن تكون فيه بيانات)

```
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 clsPerson(int iD, string firstName, string lastName, string title)
        {
            this.ID = iD;
            this.FirstName = firstName;
            this.LastName = lastName;
            this.Title = title;
        }
}
public class clsEmployee : clsPerson
```

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


    public void IncreaseSalaryBy(float Amount)
    {
        Salary += Amount;
    }
        public clsEmployee(int iD, string firstName, string lastName, string title
,float salary, string departmentName) : base(iD, firstName, lastName, title)
        {
            Salary = salary;
            DepartmentName = departmentName;
        }
}
    internal class Program
    {
        static void Main(string[] args)
        {
            clsEmployee Employee1 = new clsEmployee(10 , "Mr." , "Mohammed", "Abu-
Hadhoud" ,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();
        }
    }
}
```

Lesson 25 : Upcasting and Downcasting

Upcasting[10] : تستطيع – تصغير أو تحويل أو إرجاع – **Derived Class** الى أصلها وهي Base Class
فالتحويل أو التصغير من الكبير أو الفرع الى الصغير أو الأساس آمن لأن لديه جميع Data
وهو يسرع البرنامج والأكثر استخداما

Downcasting : التحويل من الصغير أو الأساس Base Class الى الكبير أو الفرع **Derived Class**
لا يكون دائما آمنا لأن Data تكون ناقصة – يوجد طريقة آمنة للتحويل –

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

[10] راجع الكورس 10 : Lesson 27 : Up Casting vs Down Casting

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 = "Acm
e 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 Co
rp.", 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: InvalidCas
tException


        Console.ReadKey();
    }
}
```

In this example, we have a class and an class that inherits from . The class has a and property, as well as a method that prints a greeting to the console. The class has an additional and property, as well as a method that prints information about the

employee's job to the console.`PersonEmployeePersonPersonNameAgeGreetEmployeeCompanySalaryWork`

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

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

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

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.

Lesson 26 : Method Overriding in C# Inheritance + Base Keyword

عندما ترث من **Base Class** ترث معها كل **Methods** التي من نوع **public / protected** فيوجد بعض **Method** التي تريد استبدالها ب **Code** آخر بنفس اسم **Method** في **Derived Class**

عند عمل Object من Derived Class وتريد مناداة Method التي عُمل لها Overriding[11] فيتم مناداة Method التي في Derived Class وليس التي Base Class

عندما تتوقع أن تعمل Overriding ل **Method** معين في Base Class فلابد أن تعمل لها **virtual**

```
public  class clsA
{
  public virtual void  Print()
```

وتعمل في **Derived Class** : **override** لمناداتها بدل التي في **Base Class**

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

شرط مهم عند عمل Overriding لابد أن تكون Signature أي تتساوى

- **عدد Parameter**
- **ونوع Parameter**
- **واسم Function**

Keyword. **Base** ترجعك الى **Base Class** للوصول الى **Methods** من داخل **Derived Class**

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

[11] راجع الكورس 10 : Lesson 28 : Virtual Functions

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 Empoyee
        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();
```

Lesson 27 : Method Hiding in C# (Shadowing)

Method Overriding : يوجد في Base Class : **Method** وقد عملت **override** لهذه **Method** في Derived Class **يعني أنك قد ألغيت Method التي Base Class تماما واستبدلتها بالتي في Derived Class**

Method Hiding (Shadowing) هي نفس عمل Method Overriding لكن هناك فرق بينها **وهو أنها تخفي Method التي Base Class ولا تلغيها – هي موجودة ولكن تم إخفاؤها –**

والفرق بينهما هو عندما تعمل Upcasting ل

- ❖ Method Overriding لا يتم الرجوع الى Method التي في Base class **لأنها ألغيت**
- ❖ **Method Hiding (Shadowing)** ترجع الى **Method** التي في Base class **لأنها أخفيت**

عندما تتوقع أن تعمل Hiding (Shadowing)ل **Method** معين في Base Class فلابد أن تعمل لها **virtual**

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

وتعمل في **Derived Class** : **new** لمناداتها بدل التي في **Base Class**

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

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 MyOthe
rMethod"



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



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


        Console.ReadKey();
    }
}
```

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

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

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

أنواع Inheritance (هؤلاء 3 مستخدمين في أغلب لغات البرمجة ومنها C#) [12]

١. **Single Inheritance** مثال `clsB` يرث من `clsA`
٢. **Multi-Level Inheritance** مثال `clsC` يرث من `clsB` وهو يرث من `clsA`
٣. **Hierarchal Inheritance** مثال `clsD` و `clsC` و `clsB` الكل يرث من `clsA`

تستطيع دمجهم كلهم معا فيصبح **Hierarchal Inheritance**

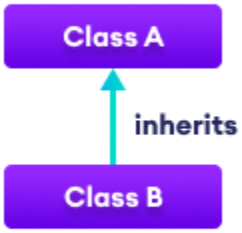
أنواع Inheritance الإضافية الموجودة فقط في C++ (استخدامه قد يسبب مشاكل في البرنامج)

☒ **Multiple Inheritance** مثال `clsC` يرث من – اثنان – `clsB` و `clsA`
☒ **Hybrid Inheritance** مثال `clsD` يرث من – اثنان – `clsC` و `clsB` وهما يرثان من `clsA`

Answers	Question	Quiz
(**Single** & **Multi-Level** & **Hierarchal**) (**Multiple** & **Hybrid**) **Inheritance**	What are types of Inheritance ?	
True	Multiple inheritance are not supported by modern languages such as JAVA and C#	

Types of inheritance

There are the following types of inheritance:



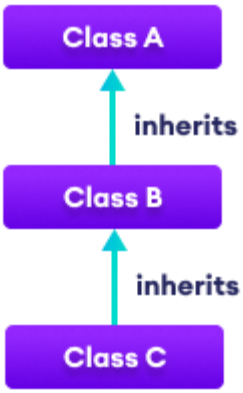
1. Single Inheritance

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

[12] من الكورس 10 : Lesson 26 : Inheritance Types

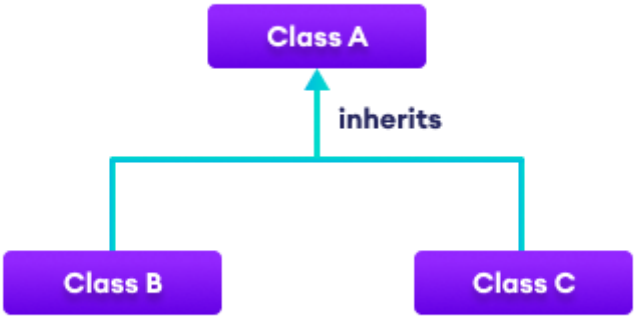
2. Multilevel Inheritance

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



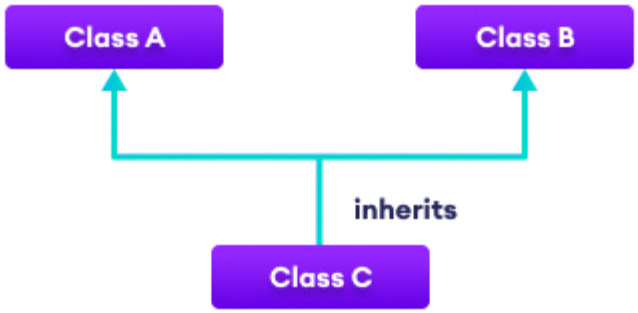
3. Hierarchical Inheritance

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



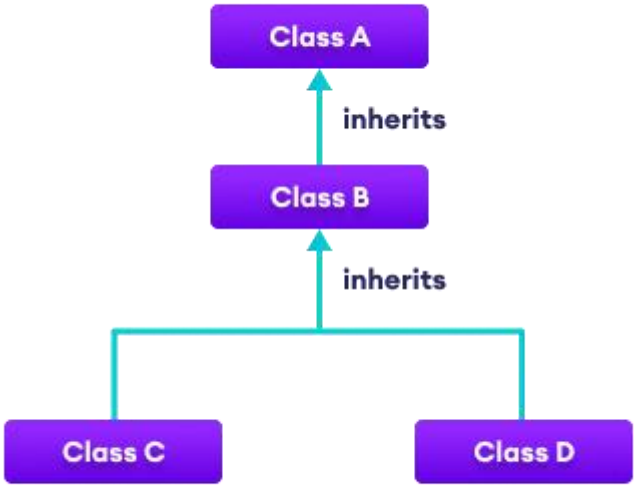
4. Multiple Inheritance

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.

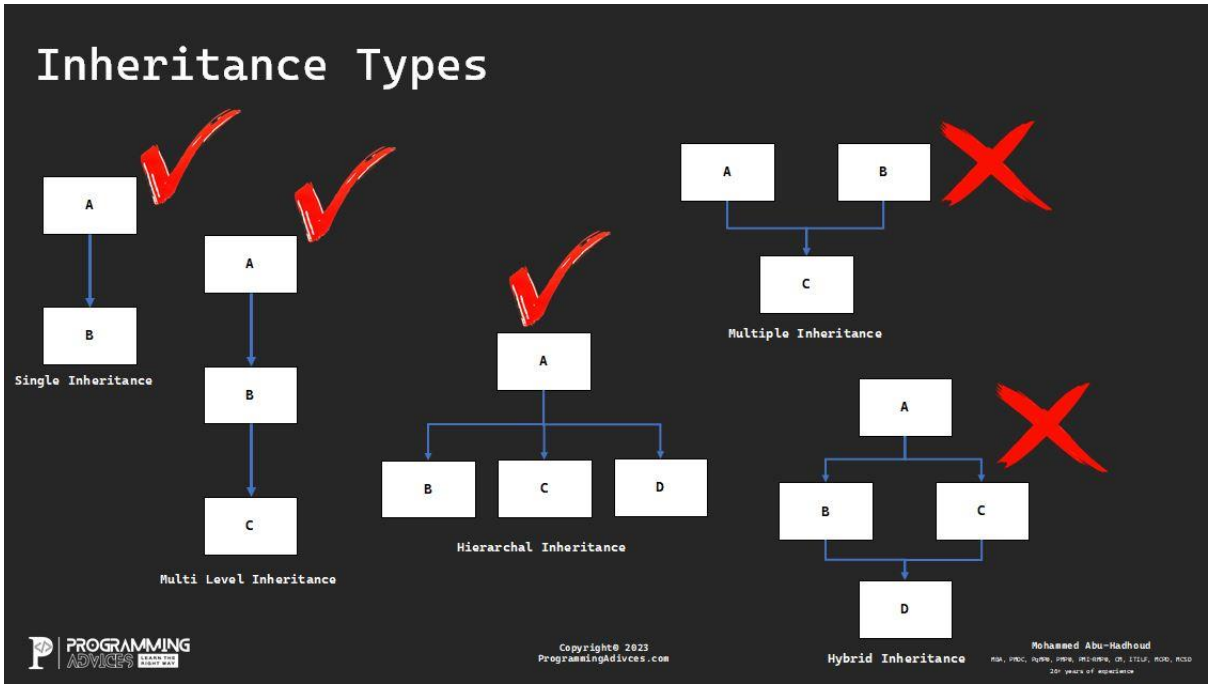


5. Hybrid Inheritance

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#



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

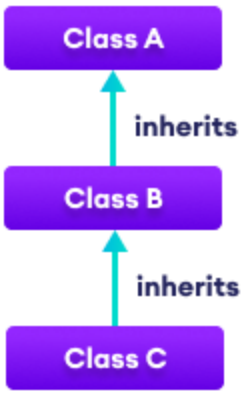
Multi-Level Inheritance مثال `clsC` يرث من `clsB` وهو يرث من `clsA`

`clsC` يوجد فيه كل Method / Member التي ورثها من `clsB` + التي ورثها من `clsA`

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 wor
king."
        doctor.Heal(); // Output: "Doctor John with ID 123, salary $100,000, and s
pecialty 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.

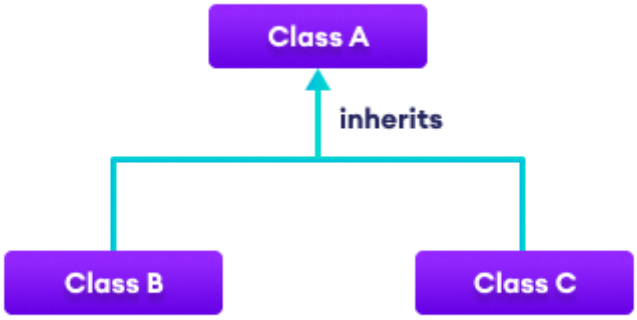
In the Main method, we create a new Doctor object and set its properties. Since Doctor inherits from Employee, which in turn inherits from Person, it has access to all of the properties and methods defined in those classes.

Hierarchal Inheritance مثال `clsD` و `clsC` و `clsB` الكل يرث من `clsA`

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 ol
d."
        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.

Lesson 31 : Abstract Class & Methods

Abstract Class هي مثل أي Class **لكن لا تستطيع عمل Object** منها **بل لابد أن inherited يرثها Class آخر لأنه لا يكون هناك فائدة من استخدام Object معها**

وتعمل implementation function بداخل Abstract Class ويمكن أن يكون بداخلها نوعين من Method

- Method العادية
- **Abstract Method يكون واجهة أو عنوان Function موجودة فقط – interface –**
  - فلابد أن تعمل له implementation بداخل Class التي ورثها

```
public abstract void Introduce();
```

وتعمل في **Derived Class** : **override** لمناداة Abstract Class التي في **Base Class**

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

Answer	Question
True	**Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class)**
True	**Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from)**

Abstract Class

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 i
t.
      //  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

Lesson 32 : What is Interface ? and Why

Interface[13] هو بمثابة عقد بينك وبين من يستخدم Class Interface

عندما يتم استخدامها – أو وراثتها – **فإن Compiler هو الذي يتحقق من استيفاء كل الشروط التي في Class Interface مثل**

- **التحقق من كتابة اسم Method == اسم Method الذي Class Interface**
- **والتحقق من نوعهم ... int , string , float**
- **التحقق من Parameter سواء العدد أو النوع**

إذا تريد إنشاء **Class Interface فاجعل بداية الاسم يبدأ ب I = INameInterface**

الفرق بين Class Interface و Abstract Class

- **Abstract Class** : قد يكون فيه implementation – كود للتنفيذ –
- **Class Interface : يكون واجهة أو عنوان Function موجودة فقط – interface –**
  - Members التي بداخلها هي إما public / abstract

Answer	Question
True	**An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies)**
True	**We must provide the implementation of all the methods of interface inside the class that implements it**
True	**Like abstract classes, interfaces cannot be used to create objects**
True	**Interface methods do not have a body - the body is provided by the "implement" class**
True	**Interfaces can contain properties and methods, but not fields/variables**
True	**Interface members are by default abstract and public**
True	**An interface cannot contain a constructor (as it cannot be used to create objects)**

[13] راجع الكورس 10 : Lesson 31 : Interfaces : Pure Virtual Functions and Abstract Classes

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

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

```
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 and `abstractpublic`
- 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.SedEmail();
        Console.ReadKey();
    }
}
```

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

☒ **Multiple Inheritance** مثال `clsC` يرث من – اثنان – `clsB` و `clsA` هذه ممنوعة أو لا يدعمها C#

- أما في Interfaces فتدعم **Multiple implementation**

```
public abstract class Person : IPerson, ICommunicate
```

Answer	Question
True	**To implement multiple interfaces, separate them with a comma**

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();



    }
}
```

Nested Class هي **Class بداخل Class** ، وكلاهما منفصلان عن بعض – مثل أي Classes في البرنامج

Nested Class فائدتها ترتيب Code , وتستخدم للضرورة

طريقة الوصول الى Class الداخلية لإنشاء Object منها

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

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 inside the class . The is called the nested class.`InnerClassOuterClassInnerClass`

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 = " + innerV
ariable);
        }


        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 fro
m inner class: 42"
        Console.ReadKey();



    }
}
```

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

Composition هو إنشاء Object – ل ClassB أخرى – بداخل ClassA

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 comp
osition.
```

```
            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();
        }
    }
```

لعدم السماح بأي تعديل على Class أو عمل امتداد Extend ل Function نستخدم
Sealed Class لتمنع أي أحد يرث inherited من هذه Class

```
sealed  class clsA
```

Inherited من فوائدها : إعادة استخدام Code وتطويره لمنع ذلك نستخدم Sealed Class

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 'Al'
```

Sealed Method لمنع عمل **override / Hiding** عند Class معين – بعد وراثتها –

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.

Partial Class : تدل على أن هذه Class هي جزء من Class الكبيرة – مقسمة على عدة أجزاء –

و Compiler يدمج هذه Partial Classes معا كأنها Class واحدة

من فوائدها أن Developers يشتغلون على Class واحدة بنفس الوقت – لكن مجزئة بينهم –

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 keyword.`partial`

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 is split into two files (and) using the keyword. The method in the file creates an instance of and calls both and . 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 was defined in a single file.`MyClass`MyClass1.cs`MyClass`MyClass2.cs`partial`Main`Program`.cs`MyClassMethod1`Method2`MyClass`

Places where class can be used:`partial`

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 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.`partial`

- 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.

Lesson 39 : Introduction to Partial Methods

هناك شرط مهم عند استخدام Partial Methods وهو أنه لابد أن تكون Partial Class وتستخدم داخل Partial Class أما Class العادية فلا تستخدم

هذه هي Partial Methods

```
partial void PrintAge();
```

وهذه هي implementation Partial Methods

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

أنت تعرف Heder ل Partial Methods وإذا عمل لها implementation يتم تنفيذها وإذا لم يعمل يشتغل البرنامج بدون مشاكل Error

من فوائد فصل Heder عن implementation أنه الكود يظل نظيف بدون أخطاء

مثال PrintAge يتم طباعتها على طابعة فكود الطباعة يختلف على حسب نظام التشغيل : Windows , OS ... فيكون هناك أكثر Partial Class لكل نظام التشغيل Partial منفصل

Partial Methods تستخدم في Code generator - ستدرس لاحقا –

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 PersonPr
inting.cs
        Person person1 = new Person();
        person1.Age = 25;
        person1.Birthday(); // Output: "Current age: 26"
    }
}
```

In this example, the class declares a partial method , which prints the current age of the person. The method of the class calls the 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.`PersonPrintAgeBirthdayPersonPrintAge`

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

When the method in creates an instance of and calls its method`MainProgram.csPersonBirthday`

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.

تعدد الأشكال Polymorphism[14]

١. **Compile time polymorphism** وهي باختصار **Overloading**
   a. مثل Functions لهم نفس الاسم ولكن يختلفوا في عدد parameters
٢. **Runtime polymorphism** وهو باختصار : **Overriding** و **Shadowing**
   a. إلغاء أو إخفاء Method التي Base Class
٣. **Inheritance** لأنه قد يتم وراثته من أكثر من Class – فهو متعدد الأشكال Polymorphism

مبادئ OOP باختصار

١. **Encapsulation** : جمع كل Methods ذات العلاقة معا تحت Class واحد والوصول إليها عبر . عن طريق Object
٢. **Abstraction** : إخفاء جميع التفاصيل غير الأساسية أو غير المهمة قدر الإمكان عن المستخدم
٣. **Inheritance** الوراثة
٤. **polymorphism**

[14] راجع الكورس 10 : Lesson 30: Fourth (Principle / Concept) of OOP : 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:

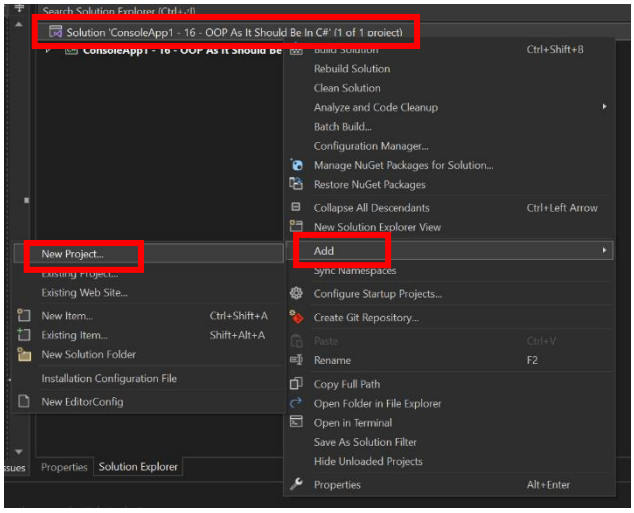
1. compile-time polymorphism (also known as method **overloading**) : Method overloading allows multiple methods to have the same name, but with different parameters. The compiler selects the appropriate method to call based on the number, types, and order of the parameters.
2. Runtime polymorphism (also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

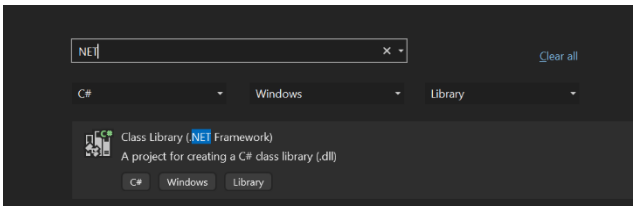
inheritance is also a form of polymorphism known as "subtyping" or "subtype polymorphism".

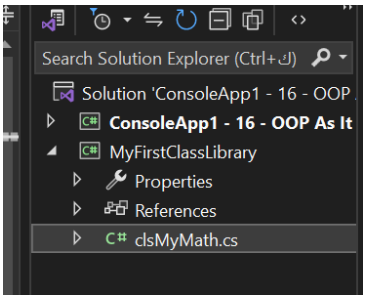
Lesson 41 : .NET Class Library

.NET Class Library.عند عمل مشروع ما لا يتم وضع كل Code في مكان واحد وإنما يتم وضعه في Classes و لذلك يتم وضعه Project منفصل اسمه Class Library **واستدعاءها مع أي مشروع** آخر سواء ... Desktop , Mobile , Web , Game

لإضافة Project – مثل Class Library – داخل Solution - بمعنى آخر أن Solution تستطيع فيه إضافة أكثر من Project تحت مشروع Solution واحد

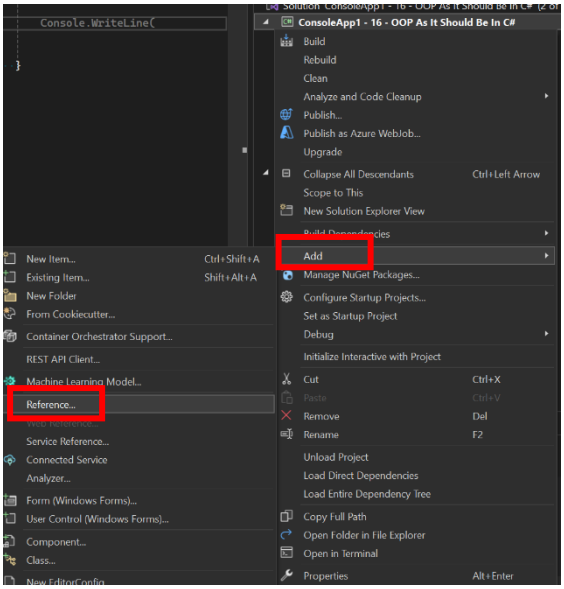




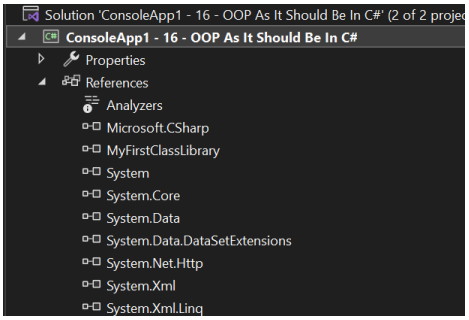


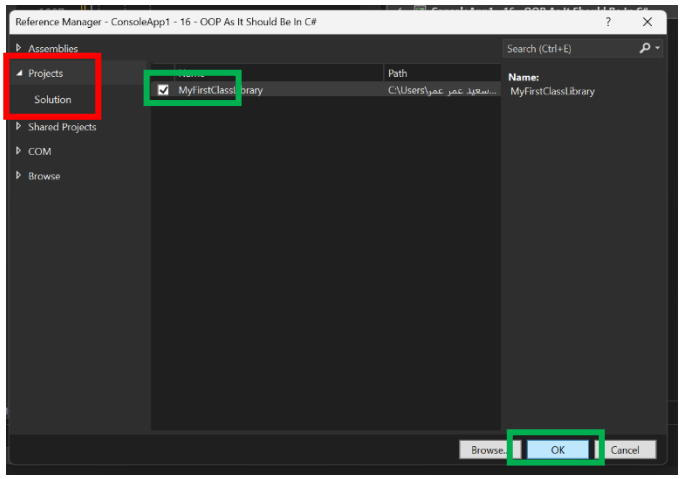
- نضغط على Solution بزر الفأرة الأيمن
- ثم Add ثم **New Project ...**
- ثم (.NET Farmwork) **Class Library**
- ثم اسم Class Library
- فيصبح عندك Two Project في داخل Solution
- فتكتب Code بداخل clsMyMath تحت MyFirstClassLibrary : Project

لربط Class Library ب Project البرنامج :

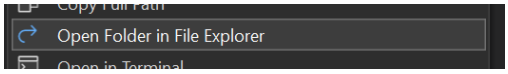


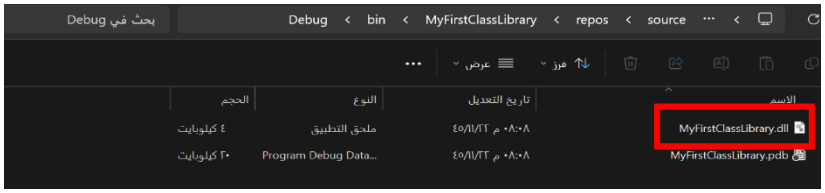
- نضغط على **Project** المراد لإضافته فيه بزر الفأرة الأيمن
- ثم Add ثم Reference
  ١. **Project** واختيار MyFirstClassLibrary
  a. إذا كانت من ضمن Solution
- ثم بعد إضافتها يتم استخدامها في داخل Project
  - using MyFirstClassLibrary;
  - لابد أن يتطابق الاسم مع namespace الذي في Class Library





- عند عمل Bild ل Class Library ينشئ ملف DLL وهو اختصار ل Dynamic Link Library وعن طريقه تستطيع استدعاءه من أي Project
  - **كيفية عمل Bild ل Class Library = Ctrl + Shift + B**
  - فيتم تجميع الكود في Assembly الذي منه DLL
- للوصول الى الملف DLL تضغط بزر الفأرة الأيمن
  - MyFirstClassLibrary\bin\Debug
  - تنسخ Path كامل

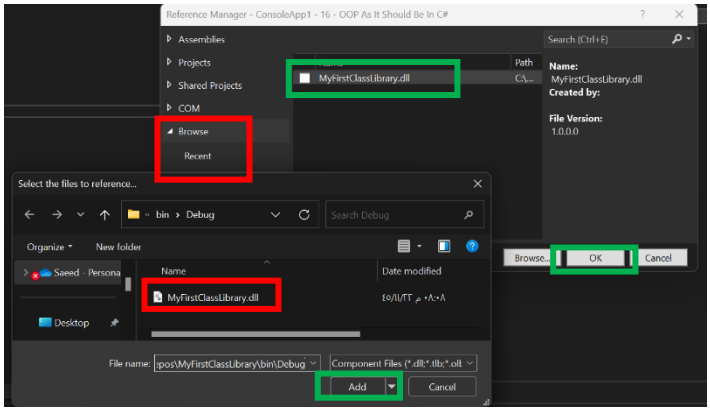




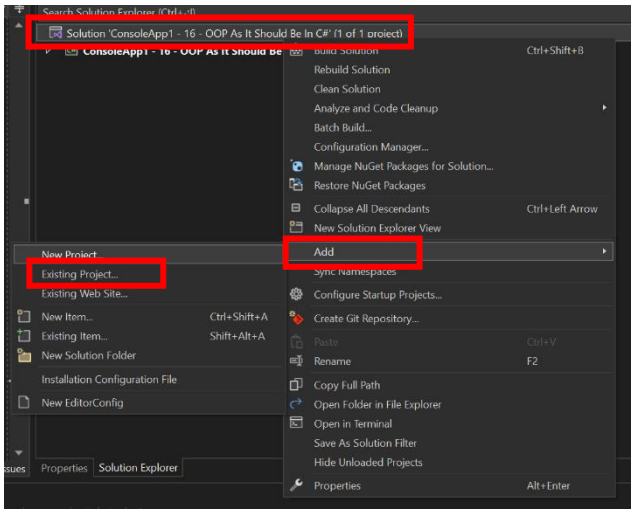
- **نضغط على Project** المراد لإضافته فيه بزر الفأرة الأيمن
- ثم Add ثم Reference

٢. **Browse** ولصق **Path** في File Name للوصول الى MyFirstClassLibrary

a. إذا لم تكن من ضمن Solution - أي مشروع آخر –

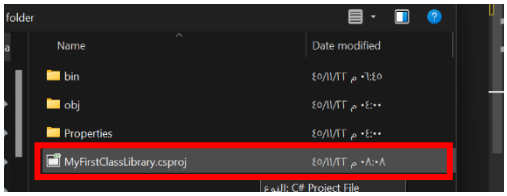


- ثم بعد إضافتها يتم استخدامها في داخل Project
  - using MyFirstClassLibrary;
  - لابد أن يتطابق الاسم مع namespace الذي في Class Library



للتعديل على Class Library – للاطلاع على الكود أو جعله Project -

- نضغط على Solution بزر الفأرة الأيمن
- ثم Add ثم **Existing Project ...**
- ثم الوصول الى Class Library المرادة
- ثم الضغط على اسم Class Library مثال لفتحها MyFirstClassLibrary.csproj



الملخص

- ❖ إنشاء (.NET Farmwork) **Console App**
- ❖ ثم نضيف Project – Class Library – داخل Solution
- ❖ ثم نضيف Classes تحت namespace
- ❖ ثم استدعاءها سواء عن طريق
  - **Project**
  - **Browse** عن طريق DLL

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** : يمكن الوصول إليها فقط من داخل نفس المشروع DLL وليس من مشرع آخر
  a. فيتم تجميع الكود في Assembly الذي منه DLL
  b. **ولا يوجد في C++** Internal وإنما يشبه friend في C++

Answer	Question
True	**Internal in C# is equivalent to friend in C++**

internal access modifier

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

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++.

Class vs Struct

Class

- الـ Class هو (**Reference Type**) يشير إلى مكان في الذاكرة.
- يتم تخزين الـ Class **(heap)**.
- يمكن أن يحتوي الـ Class على (Constructors) و (Destructors) ويدعم (Inheritance)

Struct

- الـ Struct هو **(Value Type)** يحتوي على البيانات نفسها.
- يتم تخزين الـ Struct في **(Stack)**.
- لا يمكن لـ Struct أن يرث من Struct آخر
- يجب استخدام الـ Struct عندما يكون الكائن صغيرًا وقصير العمر وغير قابل للتعديل ولا يتم تحويله بشكل متكرر.

في النهاية، يعتمد اختيار استخدام الـ Class أو الـ Struct على الحاجة والسياق. إذا كنت بحاجة إلى تمثيل كائن معقد، فالـ Class هو الخيار الأفضل، أما إذا كنت بحاجة إلى كائن بسيط وخفيف، فالـ Struct هو الخيار المناسب.

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

Lesson : C# Enums (Enum is a special class)

Enum in C# هي نوع خاص من class تمثل مجموعة القيم الثابتة constants – أي أنها غير قابلة للتعديل وإنما هي للقراءة فقط –

وطريقة الوصول العناصر هي : NameEnum.

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

C# Enums

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

To create an , use the keyword (instead of class or interface), and separate the enum items with a comma:enumenum

Example

```
enum Level
{
  Low,
  Medium,
  High
}
```

You can access items with the **dot** syntax:enum

```
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 inside a class:`enum`

Example

```
class Program
{
  enum Level
  {
    Low,
    Medium,
    High
  }
  static void Main(string[] args)
  {
    Level myVar = Level.Medium;
    Console.WriteLine(myVar);
  }
}
```

الفهرس

صفحة	العنوان
1	Important Introduction
2	Lesson 1 : (Revision))What is Object Oriented Programming? And Why ?
4	Lesson 2 : Class & Object
10	Lesson 3 : Objects In Memory
11	Lesson 4 : Access Modifiers
18	Lesson 5 : Static Members
21	Lesson 6 : Properties Set and Get
24	Lesson 7 : ReadOnly Properties
26	Lesson 8 : Auto Implemented Properties
28	Lesson 9 : Static Properties & Static Class
30	Lesson 10 :(Revision) First Principle or Concept of OOP- Encapsulation
31	Lesson 11 : (Revision) Second Principle or Concept of OOP - Abstraction
32	Lesson 12 & 13 : Calculator (Requirements & Solution)
36	Lesson 14 : Constructor
38	Lesson 15 : Parameter-less Constructor
40	Lesson 16 : Parameterized Constructor
42	Lesson 17 : Default Constructor

44	Lesson 18 : Private Constructor vs Static Class
48	Lesson 19 : Multiple Constructors using overloading
50	Lesson 20 : Static Constructor
52	Lesson 21 : Destructor
55	Lesson 22 : Very Important: Real life Application of Using Static and Constructors
58	Lesson 23 : Third Principle/Concept of OOP: Inheritance
64	Lesson 24 : Inheritance Constructor
66	Lesson 25 : Upcasting and Downcasting
70	Lesson 26 : Method Overriding in C# Inheritance + Base Keyword
74	Lesson 27 : Method Hiding in C# (Shadowing)
78	Lesson 28 : Types Of Inheritance
81	Lesson 29 : Multi-Level Inheritance
84	Lesson 30 : Hierarchal Inheritance
87	Lesson 31 : Abstract Class & Methods
90	Lesson 32 : What is Interface ? and Why
95	Lesson 33 : Implementing Multiple Interfaces
99	Lesson 34 : C# Nested Class
102	Lesson 35 : Composition
104	Lesson 36 : Sealed Class
106	Lesson 37 : Sealed Method

108	Lesson 38 : C# Partial Class
111	Lesson 39 : Introduction to Partial Methods
115	Lesson 40 : 4th Principle/Concept in OOP - Polymorphism
117	Lesson 41 : .NET Class Library
121	Lesson : Internal Access Modifier
122	Lesson : Class vs Struct
126	Lesson : C# Enums (Enum is a special class)