CH : 04 OOPs In C#

0 : INTRO TO OOPs
- modular v/s OOPs Programming
- class & objects, constructors & its Types
- Destructor, IDisposable Interface

1 : ABSTRACTION
- Access Modifieres, Assemblies
- Abstraction

2 : ENCAPSULATION
- Encapsulation

3 : INHERITANCE
- Inheritance, types, Is-a & has-a
- Interfaces, multiple inheritance
- Interface v/s Abstract class

4 : POLYMORPHISM
- Intro, types, method hiding

5 : MORE ON OOPs
- partial class & methods
- Sealed Partial class, Partial methods
- Sealed class & methods
- Extention Methods & static class
- is, as & type of
- Relationships

CH: 05 MORE ON OOPs

(11) PARTIAL CLASSES & METHODS

* Partial classes

- Partial class feature was added in C# 2.0, which allows us ~~to~~ to define classes on multiple files
- We can physically split the content of class into diff files but logically they are single unit
- A class in which code can be written in 2 or more files is called partial class
- it is also possible to split defination of struct, interface into 2 or more files like we can do in class
- you can create 2 class files as extension of .cs & write partial class in it, you cannot create ~~partial~~ partial classes into different projects/Assembly
- the class signature should be same in both files including Partial keyword
- decalare class using partial keyword ~~& must be public or internal~~
- All parts must be in the same namespace & same assemble (~~&~~ same project)
- Compiler combines these classes into a single class at Compile time
- partial can be also used for class, structs, & Interfaces
- partial classes cannot be spread over across multiple assemblies or project, they must be in same ~~assemg~~ asembly or project
- The Compiler merges the partial class defenition at Compile time
- Since each project compiles into different or seperate assembly (.dll or .exe), the compiler cannot merge partial class defination across different assemblies
- if you define partial class in diff projects or assemblies & Even if you add their refrences, then also it wont work
- partial class must be in same assembly (project) it can be public or internal, becoz Compiler merges them at time of compilation, & when you refrence another project, you are refrencing a already compiled source code (.dll or .exe), so the compiler now cannot merge them

eg: Solution
> Partial class Project
> Emp1.cs
> Emp2.cs
> Program.cs

Emp1.cs>

```
namespace Partial
{
    public partial class Emp
    {
        public string Name {get; set;}
        public int Age {get; set;}
    }
}
```

Emp2.cs>

```
namespace Partial
{
    public partial class Emp
    {
        public void display()
        {
            Console.WriteLine(Name + Age);
        }
    }
}
```

Program.cs

```
using System
namespace Partial
{
    class Program
    {
        static void Main()
        {
            Emp obj = new Emp();
            obj.Name = "Vishwajeet";
            obj.Age = 21;
            obj.display();
        }
    }
}
```

- // partial class should be in same namespace & same Assembly (project)

// Even though partial class definition splits into multiple files, after compilation (IL code) they will become a single class. you can see the IL code generated using ILDASM

- Use Case: in using Entity framework DB first Approach, Entity Framework will create the models (the classes) based on DB & it will create these models (classes) as partial classes. it will autogenerate the partial class, so that you can add some properties in it
2) in large projects you can use partial classes

* Rules for Partial class

1) must be in same Assembly (Project)
2) partial classes can be in same file also, but usually they should be in diff files But always inside same project
3) All partial parts must have same access modifiers, or else you will get Error
4) you can use the modifiers like abstract, sealed, or inherit the partial classes

1 * if any part (not all partial class parts) of partial class is decalared as abstract, then entire class (all parts are at the End a single class) will become abstract, means you cannot create objects of that class

eg:
```
public abstract partial class Emp
{
    public string Name {get; set;}
}
public partial class Emp // no need to use abstract keyword here but the class ~~should~~ should be public (same access modifier should be there)
{
    public void Display()
    {
        Cw (Name "is Employee");
    }
}
class Program
{
    static void Main()
    {
        Emp obj = new Emp() --> // you cannot create obj of Emp
    }
}
```

2*) Sealed Partial class

Rule: if any part of partial class is defined as sealed, then Entire class will become sealed, meaning you cannot inherit that class anywhere

eg:
```
public sealed partial class Emp
{
    public string Name;
}
public partial class Emp
{
 public void display ()
 {
    cout ("Name");
 }
}

class Program : Emp ⟶ // you cannot inherit the Emp class beoz it is sealedd
{


}
```

3*) Inheritance in Partial class

- if any part of the partial class inherits any parent class, then the Entire class inherits the parent class even if you don't write the parent class with (:) for inheritance in front of Each part
- Even any other class when ~~Each~~ inherits partial class then that child class can access any part of the partial class in that child class
  inheritance applies to the whole partial parts of that classes
- diff parts should not inherit diff parent classes, beoz classes cannot do multiple inheritance, But diff parts can inherit diff Interfaces for multiple inheritance

* Partial Methods

- Partial Methods can only be created in Partial classes
- you cannot mark a fun as partial in normal classes
- Even you can create partial methods in partial struct.
~~or interface which is partial~~
- you can't create partial methods in partial interfaces also, in &tv it is allowed & in interface they can have modifier as public also, but no other & modifier is allowed on partial functions
- partial methods must be private by default, no Access allowed
- partial methods cannot return any value, must be void
- partial ~~class~~ methods can be declared in one partial class & implentation can be in another partial class, but its optional to provide implentation to the partial method
- if you left partial method without implementation, then compiler will completly remove it
- in partial method, one class ~~contain~~ contains its decalaration & another partial class will contain its implementation, & if implentation is not provided then the methods & calls to that method are removed by Compiler
- partial method cannot have access modifiers, virtual, override, abstract, new, sealed or extern modifiers
- Partial method decalaration & implementation should not be at same time, means should be in diff parts of partial class
- Signature of partial method decalaration & implementation should be matched
- A partial method should not have multiple implentations & decalarations. Partial methods can take parameters & are always private

- eg:

```
public partial class Emp
{
    partial void display();
}
public partial class Emp
{
    partial void display()
    {
        Con.L("LL"); // only one implementation
    }
}
```

#12 SEALED CLASSES & SEALED METHODS

* Sealed class

- Sealed keyword will not allow that class to be inherited by any other class
- prevents inheritance
- Sealed class cannot be inherited by any other class

eg:
```
sealed class Emp
{

}
class student : Emp --> // Error, cannot inherit Emp class
{

}
```

- Sealed class is completly opposite of an abstract class
- Sealed class cannot contain any abstract Method
- it should be the bottom most class in the inheritance hierarchy
- A sealed class can ~~never~~ never be used as a base or parent class, but sealed class can inherit some other class
- Sealed class can be child (inherit some other parent class) but cannot be parent to some class
- keyword sealed can be used with Methods & properties of a class & that class may or may not be Sealed.
- you can create the instance from sealed class
- A sealed class can be Partial, Static, Public or internal also
- A sealed class cannot be Abstract
- We cannot use sealed for : 1) Interfaces
  2) Enums
  3) Struct
- only classes & Methods can be sealed.

* Sealed Methods

- Sealed Method is a Method in derived class that overrides a base class method & prevents that (sealed Method) from further overriding prosses in any subclasses
- Sealed Methods must be in normal classes only, beoz sealed classes cannot be inherited ∴ no need of Sealed Methods in sealed class
- Sealed Methods basically cannot be overriden
- Sealed Method must be a overeriden Method, override keyword is required
- Sealed Method can only exists in child / derived classes, not in a parent class
- A method written in parent class, if that method cannot be overriden under its child class, we call it as a Sealed Method, that means every method is a Sealed Method until its mark virtual
- if you mark a method as virtual, then that can be overriden in its child as we as grandChild classes. so to avoid or stop overriding in child class only you can use sealed, sealed that virtual Method
- sealed methods reduces polymorphism basically

modifiers that are not allowed with sealed methods are: abstract, virtual, new, static.

// sealed method must have override

// sealed method, the parent should declare that method as virtual or abstract, but sealed method can't be abstract or virtual

```
class GParent
{
    public virtual void show()        // This method must be virtual, or abstract, and overriden in child class.
    {
        Cwl (" show of GParent");
    }
}
class Parent : GParent
{
    public sealed override void show()        // sealed method must have override keyword & it must override its parent class method & avoid further overriding
    {
        Cwl (" show final");
    }
}
class Son : Parent
{
    public override void show() // Error
    {
    }
}
```

// sealed method can have all modifiers except private

// without a method decalare as virtual or abstract you can make use of sealed keyword

// private method is not inherited. sealed method is inherited but cannot be overriden, & sealed method can be called from sub classes

// sealed must have override keyword also

#13 Extention Methods & static class

* Extention Methods

- Extention method is a special type of static method that allows you to add new functionality to an ~~Exst~~ Exsisting class, struct, or interface without modifying the original type or creating a derived type
- you cannot create static struct or static interface, only static class can be there
- This is useful when you want to Extend the functionality of sealed class like string, int
- Extention Method must meet following requirment
  - it must ~~begin~~ be defined in a static class
  - it must be a static method
  - it must have first parameter prefixed with this, which specifies the type being extended
- Extention methods cannot access private or protected members of extented type class
- Cannot override Exsisting methods
- heavily used in LINQ
- if a method with same name exisits in the type class, then it (class method) takes precedence over Extention method
- you can also use inheritance for Extending the functionality of a class but, Extention Methods can do this without inheritance
- in case of inheritance we call the methods defined in the old & new classes by creating object of new class, but in Extention method, we call the new methods using object of old class
- Sometime inheritance is not possible (sealed class) in that case you can use Extention methods
- Syntax :

```
public static class newClassname
{
    public static ReturnType MethodName (this oldClass variable, other parameter)
    {
        //body
    }
}
```

* Imp points

- Extention methods must be static & must be inside static class
- Static class contains only static methods
- Extention method is defined under static class, which means, the extention should be created as static method where as once the method is bound with another class, the method changes into non-static
- This first parameter binds (this) the method into another type, beoz of ~~which~~ which we can call it as non static methods This is possible beoz the compiler re-write the method call as a regular static method call behind the sences
- first parameter of Extention Method is called binding parameter, which should be the name of class to which the method has to be bound along with prefix this
- Extention Method has only one Binding parameter, & should be always defined in first parameter only, & after that you can pass other normal parameters if any
- Extention Methods can have only public, internal & static modifiers, no Abstract, virtual or ovverride
- Extention Methods class Must be public or internal along with static only
- static class cannot be Abstract.
- Extention Method class (static class) cannot be inherited, that class cannot be sealed
- Extention Method class (static class) can be partial, but all partial parts must be static
- String class is sealed class :. we can Extend its functionality

- eg:

```
public static class StringExt
{
    public static int WordCount(this string input)
    {
        if(!string.IsNullOrEmpty(input))
        {
            string[] strArr = input.Split(' ');
            return strArr.Count();
        }
        else
            return 0;
    }
}
```

```
class Program
{ static void Main () {
    String myWord = " Welcome Lavdo " ;
    int wCount = myWord.WordCount ();
    CWL (" string : " + myWord);
    CWL (" string count " + wCount);
  }
}
```

// This string class is been Extented using Extention Method WordCount () method & we have used that method from obj of string class

Use Case : LINQ is the eg of Extention Methods that add query functionality to the Existing System.Collection.IEnumerable & System.Collections.Generic.IEnumerable<T> types

* Static Class

- how do we avoid OOPs principals in a class? using static keyword or static class
- Static classes are sealed, which means you cannot inherit a static class further
- static class can contain only static Members & you cannot create instance/object of static class. the static class Methods or Members can be called using class name & ~~dot~~ dot operator
- static class is used to hold 'utility methods, constants or extention methods. that do not require instance of the class
- Character sticks : Cannot be inherited & even cannot inherit some other class
  Can only contain static Members
  Can have static constructors only
  Can not be inherited bcoz it is implicitly sealed
- allow modifiers are : public, internal, partial & static
- Method inside static class can have, public, internal, private, static, async, Partial
- partial methods must be private & static inside static class & class also must be partial

eg: public static class MathHelper
{
  public static in square (int num) → // call this Method by MathHelper.square ();
  { return num * num; }
}

#4 is & as , typeof

* Type checking
- you can do type checking using is, as & typeof

* is : it is operator, checks if object is of specific type & returns true or false

```
eg: object obj = "Hello";
    if (obj is string)
    {
        // code
    }
```

```
eg: object obj = "Hello";
    if (obj is string str)
    {
        cwL(str.ToUpper());
    }
```

* as : operator (use for safe Type Casting), used for reference types & nullable types
- Returns null if conversion fails (instead of throwing Exception)
- eg:

```
object obj = "Hello";
string str = obj as string; // works, str = "Hello";
object num = 123
string xyz = num as string // * fails, xyz = null

cwL(str);
if (xyz == null) // true
{
    cwL("not converted xyz to string");
}
```

* typeOf : used to get type at compile time, Returns the Type of object of a class, struct or data type.
- only used with type names, not instances
- Cannot be used on variables, for that use GetType();
- eg:

```
Type a = typeof(int);
cwL(a); // System.Int32

Type b = typeof(string);
cwL(b); // System.String
```

* checked & unchecked

1) checked & unchecked keywords control how the runtime handles integer overflow during arithmetic operations & conversions

- Integer overflow occurs when you try to convert bigger data type into smaller data type eg into to byte bcoz byte can just store 255 numbers

- eg:
```
int max = int.MaxValue;
int res = max + 10; // overflow happens
Console.WriteLine(res); // incorrect value will be printed, any value
```

- Instead of throwing error C# wraps around the value by default

- checked: keyword Enable overflow checking, it throws Exception if overflow occurs

eg:
```
static void Main()
{
    int max = int.MaxValue;
    // int res
    try
    {
        int res = checked(max + 10); // throws Exception
        Console.WriteLine(res); // it won't get executed bcoz Exception happen
    }
    catch (Exception e)
    {
        Console.WriteLine("overflow detected " + e);
        // this will get printed bcoz exception is caught
    }
}
```

- un-checked: keyword disables overflow checking, allows arithmetic operations to ignor overflow silently

eg:
```
int max = int.MaxValue;
int res = unchecked(max + 10);
Console.WriteLine(res); // some random value will be printed
```

- you can also use checked & unchecked as blocks

#8 RELATIONSHIP

(*1) **Association**: a relationship b/w 2 classes, where 2 classes are related but can exist independently. There is no ownership b/w them

eg: Teacher & student are related but both can exisits independently

- Teacher & student have association
- A teacher can teach multiple students & student can have multiple teachers.
- But Both exists independently
- we pass student obj to Teacher class func (student s)

(*2) **Aggregation**: is a special form of association where one class (whole) contains objects of another class (part) but the lifetime of that part is not dependent on the whole

eg: A Department can have multiple Teachers, but teachers can exists independently of the department

- Department class aggregates (has) multiple teacher objects
- Teacher objects are not owned by department, they can exists independently
- if department is deleted, Teacher object still exists
- we create a teacher obj in department class as public & as a list of teachers

(*3) **Composiation**: is a strong form of aggregation, where the contained object (part) cannot exsist independently of the containes (whole

eg: A car has an Engine. if the car is ~~dest~~ destroyed the engine is also destroyed

- we will create Engine obj as private member in class Car
- The Engine object cannot Exist independently

(*4) **Dependency**: Dependency represents a situation where one class depends on another temporarily. The dependent class only uses another class but not store a refrence to it

- eg: A customer depends on a PaymentService to make a payment
- Customer depends on PaymentService for payment processing
- paymentService is created & used temporarily in MakePayment(), but Customer does not own it
- if paymentSeaviCe is removed or changed, customer needs to update the method accordingly