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, Accemblies
- 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:0 INTRO OOPs

#1) MODULAR V/S OOP Programming

* Modular Programing

- modular programming is collection of function
- if software is developed just for single function (inventory) it can be build using modular or Procedural Programming

* Application Development Methodologies

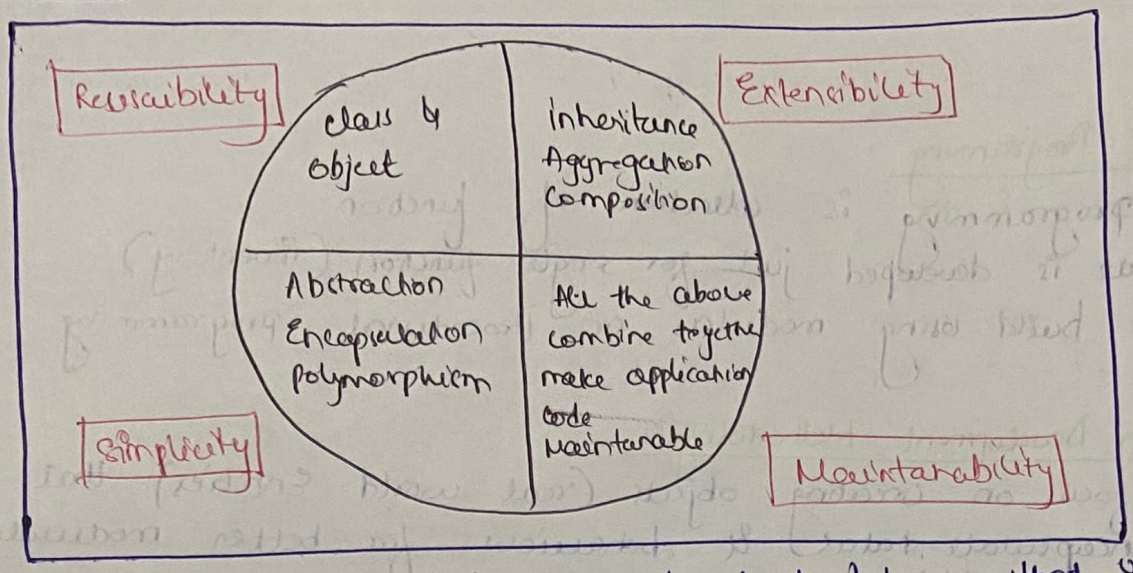
1) OOP: focus on creating objects (real world Entities) that encapsulate data & behaviour for better modularity & reusability
2) Structured programming: Emphasizes a clear, linear flow of control using sequence, selection & iteration constructs
3) Procedural Programing: also called as Modular Programing it is collection of function

* disadvantages of Modular Programing

1) Reusability: same code must be written multiple times to extend feature of a fun.
2) Extensibility: not possible to extend features of function, you have to write that fun again with new features
3) Simplicity: As Extensibility & Reusibility are impossible it is very difficult to manage & maintain application code
4) Maintainability: As we don't have Reusibility, Extensibility & Simplicity, it becomes difficult to manage & maintain application code

- To over come these disadvantages we have Concept of OOP or Methodology as OOP

* OOPs



- OOP is a software development Methodology that uses design Principal to create Application
- OOP is collection of objects which contains functions & data related to those objects
- you need to think Everything as an object

* Why Choose OOP

- High level view : OOP allows developers to view the system holistically by defining object, their data & behaviour
- Better Representation : Objects are Encapsulated with data & functions, making it easier to model the real world structure & behaviour of the organization
- Scalability & Reusiblity : Suitable for developing large complex application By representing each Entity as real world object like person

* disadvantages of OOP

1) tightly coupled : poorly design classes can lead to tight coupling, making the code difficult to modify
2) OOPs featur like polymorphism & dynamic methods may cause slower execution compared to similar procedural programs
3) Code Bloat : small simple programs can become unnecessarily large and verbose with OOP

#2 class & object

* class

- class is simply a user defined data type that represents both state & behavior
- state represents the properties
- behaviour represents the action that objects can perform
- class is a blue-print / template that ~~describes~~ describes details of an object
- Types of classes :
  1) Abstract class : Contains both Astract & implemented methods
  2) Concrete class : Contains only implemented method.
  3) Sealed class : that class cannot be inhirited
  4) Partial class : class can have defination in diff files
  5) Static class : Cannot do inhiritance & cannot be instanciated. & only contains static method & static constructor (which is private by default)

* object

- object is an instance of class that brings class to life
- object in simple words is a variable of data type as class
- objects behaviour is the activities (function) it can perform
- always you need to use new keyword to create object

* object lifecycle

1) object creation : using new keyword, allocated memory in heap
2) object Usage : object performs operation as defined in class
3) Garbage collection : C# use automatic Garbage Collection to manage memory
   - once an object is no longer needed or accesible, it becomes eligible for garbage collection
4) finalization : if class has Distructor it can be called before object is destroyed (in Destructor)

(48) Constructors

* Constructors

- Constructor is a special method present under class responsible for initializing the variable (with default values) of the class
  → // static Properties / data members even constructor is require for making objects
- name of constructor is same as class,
- Constructor does not return any value
- Constructors can be assigned access specifiers, based on which it decides till where you can make object of that class
- Each & Every class requires the constructor. if we want to create the instance of that class
- if you don't create any constructor, then compiler has its ~~one~~ default constructor (implicit constructor gets defined in that class by compiler)
- We don't need to (explicitly) call the constructor, Constructors are automatically called (invoked) when an object is created using new keyword
- When we create object of class, Constructor is implicitly called
- The compiler will create default constructor if you don't create any constructor. The default constructor of compiler look like this

  implicit = automatic

- eg:

```
class Test
{
    int i;
    String s;
    public Test()
    {
        i=0;
        s=null
    }
}
```

// default or implicit constructors are public

- Constructor is responsible for initializing the variables of class / data members

// Constructor defined by Compiler
// Implicit Constructor or default Constructor which initializes the data members with default values

- if you don't assign access modifyer to ^ Constructor, by default its ~~public~~ private. You can assign any access modifier

  explicit / user defined

- if you define any constructor then default or implicit constructor is not there
- Constructor is called for each object
- Calling Constructor is always explicit, when you use new and name of class that is nothing but calling of Constructor ∴ Constructor calling is Explicit

- if you don't make constructor, compiler makes it & is called default Constructor
- if constructor is not there (no implicit or default constructor also) then we cannot create instance or object bcoz whenever we are going to create instance we are explicitly calling the constructor using new keyword
- So compulsory there must be constructor (at least default constructor which is made automatically) for creating instance

> by default constructor made by programmer are private. Private constructor restricts the class to be instantiated from outside the class if it does not have any public constructor

- syntax :

```
class_name (parameters)
{
    //body
}
```

```
class_name obj = new class_name ();
                          //pass parameter if needed by constructor
```

↳ this is not class name, its constructor, we are calling constructor

- Implicit Constructor (also called default Constructor is by default public) is always public
- but constructors defined by programers are by default private
- Constructors are called automatically while creation of object (partially true statement)
- Constructor are Explicitly called by writing new & class name
- Constructors are property of obj ∴ they are non static by default, but you can also create static constructors
- when both parent & child have non parametric Constructor (default Constructor) then when you create obj of child class at that time child constructor is called & child constructor calleds parent Constructor automatically & then Execute body of child constructor
  ∴ parent Constructor gets Executed first
- all this happens automatically, no need to use Base keyword

* Types of Constructor

1) Default or Parameter less : you can Explicitly create it, or else compiler will implicity create it
- The default constructor is not automatically provided when you create any parametrized constructor :. always create default constructor if you create parametrized constructor

2) Parametrized : you can create 'n' no. of parametrized constructor and can do Constructor Overloading
- you need to make Constructor as public if you are making obj in another class beoz by default its private

3) Copy Constructor : C# does not have default copy constructor like the way we have in C++.
- Copy Constructor should be defined by programmer manually he needs to copy data by taking the class type object & assining the value to called object

- use case of private constructor is that, it will restrict you from creating objects

- if your constructor is private & you don't have any public constructor in your class then you can't inherit that class
- if child class is the inner class of ~~private~~ parent class & parent class has private constructor, then inheritance is possible in child class beoz of it is inner class

```
namespace Proj1
{
    public class class3
    {
        public int x;
        public class_3 () // default Constructor
        public class3 (int z) // parametrized Constructor
        {
            this.x = z;
        }
        public class3 (class3 obj) // copy Constructor
        {
            this.x = obj.x;
        }
        static void Main ()
        {
            class3 obj1 = new class3 (); // default Constructor called
            class3 obj2 = new class3 (10); // parametrized called
            class3 obj3 = new class3 (obj2); // copy called
            Cwl (obj1.x); // output : 0
            Cwl (obj2.x);          : 10
            CwL (obj3.x);          : 10
        }
    }
}
```

- you can make obj if you have only private constructor, then obj can be made only in that class, out of that class you can't make obj beoz you can't access the ~~private~~ private Constructor from outside the class
- if you have both private & public constructor then from outside the class you can make objects but it will just execute your public constructor

4) Static Constructor: When you define a Constructor Explicitly Static it is called static or class constructor

- By Default Constructors are non static or instance Constructor
- Static Constructor are implicitly (by default) private and you cannot apply access modifier on it
- Static Constructors are automatically called by runtime & Cannot be invoked manually
- as the program starts execution from Main() fun before executing Main() fun instructions, the Static Constructor is invoked & executed without even creating the obj of that class.
- Static constructor are responsible in initializing static variables of class & these constructors are first to execute
- there is only one static Constructor per class, it must not have parameters, it cannot be overloaded & it runs only once
- Static Constructors are only called when a class is accessed for the first time
- if no static members are accessed or obj is not created then static constructor is not called, you have to access the class for invoking static constructor in any way
- Syntax: Static class_name()

```
{
    // body
    // assign values to static data members
    // no parameters can be taken
}
```

- you can overload private constructors, but you need to create obj in the same class and pass parameters to private Constructors
- you can use private constructor to implement a singleton design pattern

- A static class can only have a static constructor, & it is implicitly (auto) added by compiler if not explicitly defined.
- A non static class can have both Constructors static & non static
- there can be only one static Constructor, which is used to initialize static data

eg:
```
public class A
{
    public static int A;
    static classA
    {
        A=10;
    }
}
class Program{
    static void Main()
    {
        // no static constructor is invoked bcoz you have not access classA
    }
}
```

- static constructor ~~are~~ is executed first in class & after that when we create obj then non-static constructor is called
- Static constructor executes one & only one time of a class

only non-static class contains an implicit non-static constructor

(11) Distructor

5) Private Constructor : private constructor restricts object creation from outside the class
- object can be created within the class only
- for creating obj from outside the class you need both public & private constructor
- A class with only private constructor cannot be inherited
- A inner (nested) class can be inherit the parent class even if it has only private constructor
- private constructor can do overloading, you need to create obj in the class only for this

* Singleton design Pattern
- Singleton design pattern ensures that only one instance of a particular class is going to be created & then provide simple global access to that instance for entire application
- singleton pattern में किसी भी class का सिर्फ एक ही instance हम use करते हैं & उसे instance बनाने से रोकते हैं
- using private constructor & sealed class we can make singleton design pattern
- singleton pattern में class से inherit नहीं करना चाहिए, उसका सिर्फ एक ही object होना चाहिए and वो object publically accessible होना चाहिए
- we use sealed keyword to Ristrict inheritance & private constructor to ristrict instantiation (object making)

* Deep copy : it copys the whole object (values) to nao memory location
- C# copy constructor does deep copy
- it is time consumining proces beoz new memory location is given

* shallow copy : it only copyes the Refrence..
- both the object point same memory location, so changing data using one object will reflect the change in other objects data also
- it is faster beoz only refrences are copied

• Sealed class : it cannot be inherited, even inner class within a sealed outer class cannot inherit the outer sealed class

```
Sealed class outer {
    public class inner : outer { // X error
    }
}
```

#14 Distructors

* Distructors

- Distructors are also called finalizers, are used to perform final clean up when class instance is being collected by GC
- Distructor cannot have any modifiers, parameter & does not return anything (void) & is always called implicitly (automatic) when obj of class is destroyed
- Distructor method is called automatically by GC when the obj of class is destroyed. Distructor is only one in one class.

* When obj of class is Destroyed

1) End of pgm Execution: All obj at the End of program are destroyed by GC
2) Implicit GC: ~~The programmer~~ occurs automatically when memory is full, identifying & destroying unused objects by GC
3) Explicit GC: The programmer can manually invoke GC.Collect() method to destroy unused obj by calling GC

- the destructor method is implicitly (automatic) called by GC & we cannot predict this, when it calls distructor & hence we cannot see the print statements written in destructor
- you can make Explicit call to GC, so that GC will call destructor in middle of Execution using GC.Collect() method
- for each object destructor is called if you run GC.Collect() method & that obj is ~~read~~ ready for GC to get destry by GC, for this make that obj as null

eg:
```
namespace DestructorEg
{
    class Dest
    {
        public Dest();
        ~Dest()
        {
            string type = GetType.Name;
            Cwl("Obj is destroyed");
        }
    }
    class Program
    {
        static void Main()
        {
            Dest obj1 = new Dest();
            Dest obj2 = new Dest();
            obj1 = null;
            obj2 = null;
            GC.Collect();
        }
    }
}
```

making obj eligible for GC

// calling GC. now GC will force to come & see whatever obj are not being used & are null should get destroyed, so it will call destructor & then destroy them

- Destructor is executed by GC before the obj are collected by GC
- Destructor Execution is controlled by GC

* Imp points

- destructors (or finalizers) cannot be defined in structs, you cannot call destructor, only GC can control the Execution of destructor, you can only req GC to come & clean the memory using GC.Collect method & becoz of which destructor will run
- Destructor cannot take parameters, cannot be Overloaded
- Destructor cannot be called Explicitly, They are invoked automatically by GC, At most we can req GC to execute Destructor by GC.Collect()

> • Read about, Destructor, IDisposal, GC, diff Between finalize & dispose from .dotNetTutorials Imp

- Destructor don't have any access modifiers
- Destructor is internally (in IL code) translated to finalize() try & finally block
- if you have destructor in parent class & you created obj of child class & called GC.Collect() then all destructor will get Execute first child class, then above parent classes destructor

> • there is no default destructor if you don't write any like constructor, GC has control to released managed resources

* When to Use destructor

- When we work with unmanaged resources (those resource which are not managed by CLR are called unmanaged resource) like windows, files, and network connection, DB connection, opening local disk files etc, then we should use a destructor to free the memory for those unmanaged resource.
- Destructor gets converted to finalize during IL code generation
  ∴ finalize, finalizer & destructor are same

> • you need to define destructor only if your class contains unmanged resources, to clean them you need to write clean up code

* IDisposable Interface

- To Release a Resource Explicitly we need to inherit (implement) the IDisposable interface. (in your class inherit the IDisposable interface)
- In IDisposable interface there is Dispose() method, you need to implement that method
- you need to write the actual code to free up the ~~resi~~ managed & unmanaged resource inside this dispose() method
- GC.SuppressFinalize(); this method will instruct the GC to ignore the destructor or finalizer. ∴ finalizer or destructor will not run

> • overriding a finalizer means, Explicitly defining a destructor

* Dispose Pattern

- disposedValue : variable that detects & handle multiple Dispose() call & prevents redundent clean ups
  - if Dispose() is called more then once, then second called does noth
- Dispose (bool disposing) : Actual clean up logic is written here
  - if disposing = true : then clean both manag & unmanged Resources
  - if we call dispose method from code then, we need to suppress the destructor call beoz dispose will clean both manage & unmanaged resources :. to speed up don't call destructor :. use suppressfinalize() method
  - if disposing = false : then clean only unmanaged resource
  - calling Dispose(true) Ensures full clean up, while Dispose(false) is called by destructor

- ~Destructor/finalizer : acts as backup for clean up if Dispose() method isn't called explicitly, then GC will call destructor & destructor will clean all manged resources & then call dispose(false) to clean the unmanaged resources
  - finalizer should only clean unmanaged resources beoz managed resources are taken care by GC & CLR

- Dispose() : This method is public & manually called in code
  - This method will further call the dispose(true) method for cleaning both managed & unmanaged resources
  - This method after calling dispose(true) method, it will call GC.SuppressFinalize(), so that GC will not run destructor beoz we have already cleaned managed & unmanaged resources through dispose(true) call :. GC.SuppressFinalize will increase the performance

- Use Using block with this disposal() method beoz it will automatically call the dispose() method at the End of Using block Even if there is Exception in the using block. Using block will not catch exceptions automatically. if Exception occurs inside the using block block, it will propogate unless you handle it Explicitly with a try catch
- & if you don't handle the exception then program will terminate

eg:
```
namespace DisposeDemo
{
    public class Resource : IDisposable
    {
        private bool disposeIterator = false;

        protected virtual void Dispose(bool disposing)
        {
            if (!disposeIterator)
            {
                if (disposing)
                {
                    // clean up code
                    for managed resource bcoz destructor is
                    not going to be called
                }
                //clean up code for unmanaged resource
                CWL("Unmanged resource clean");
                disposeIterator = true;
            }
            else
            {
                CWL("Resource already disposed");
            }
        }

        public void Dispose()          // this method is invoked by using block
        {
            CWL("dispose() method called");
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        ~Resource() // if using block or programmer does not call the
                    dispose method, then GC will automatically call the destructor
        {
            CWL("this is destructor : only clean unmanged resource");
            Dispose(false); // destructor will call Dispose() & will only
                            clean the unmanged resource
                            bcoz managed are clean by GC
        }
    }
```

// we will override the Dispose method ∴ use virtual. we will also overload it

```
class program
{
    static void Main()
    {
        cwl("using block will autocall the Dispose()");
        using (Resource obj = new Resource())   → make obj of Resource in using
        {
            cwl("obj will be using unmanged resource");
            obj.DBConnection()
            // obj has used some files & DB Connection
            // obj has open the file
            // now we need to release the resouse ∴ we
            // need to call the dispose method
            // but as we are in using block, it ~~will~~ will
              automatically call the dispose() method even
              if there are Exception raised in using blocks
        } // dispose is automatically called by using block

        cwl("using block has ended");
    }
}
```

- destructor is called by garbage collector · befor obj is reclaimed since the GC already cleans up managed resources automatically, ∴ the finalizer is only needed for unmanaged resources that GC doesn't handle ∴ only unmanged resources are cleaned ~~by~~ in destructor

} → // closing of namespace

- using block will Ensure automatic clean up by calling Dispose() automatically & using is Exception Safety
- Even if Exception occurs in using block the Dispose() method still runs

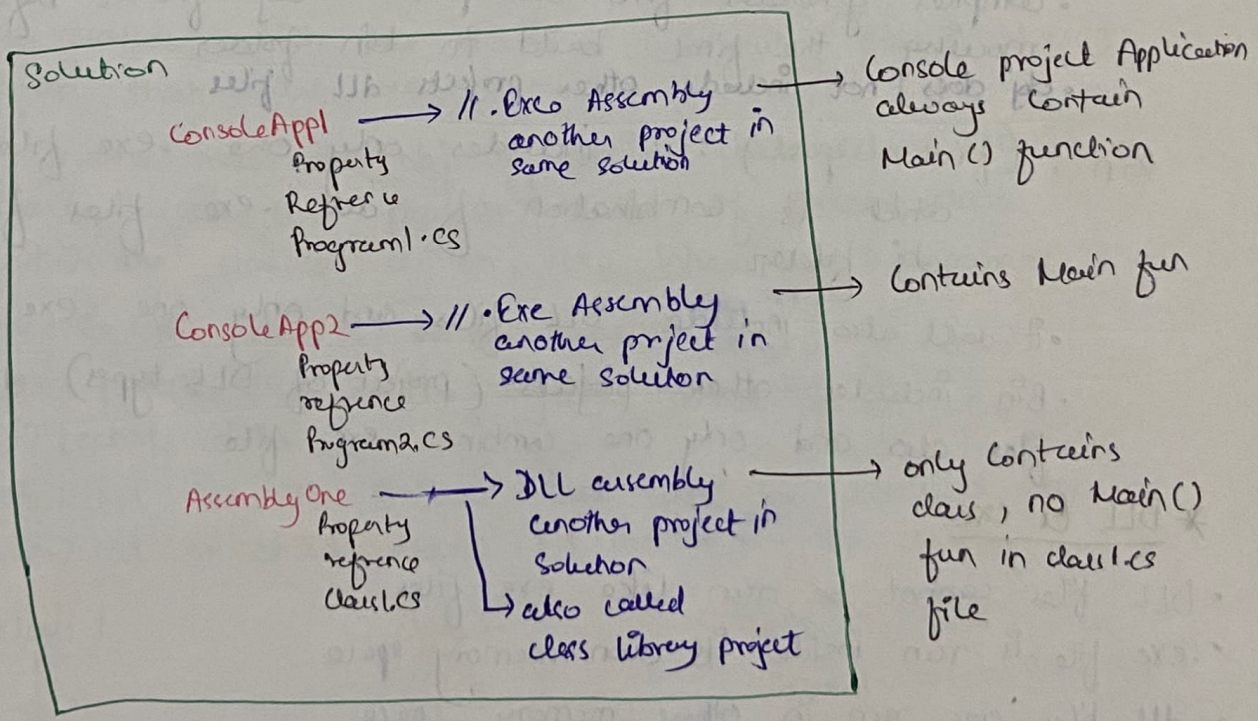
finalizer / Destructor	Dispose()
• slow and we cannot determine when the clean up will happen	• fast & immidiate memory & resource clean up
• Automatically called by GC	• manually called by programmer
• for releasing unmanged resource	• for both managed & unmanged resource
• no control & non deterministic clean up	• We have control & deterministic clean up
• GC.Collect() is Expensive then Dispose() ∴ don't force GC to call destructor using GC.Collect(), ∴ use dispose() method	• use GC.SuppressFinalize to avoid redundant finalizer call
• finalize should only be used as a backup mechanism in case dispose() is not called	• immidiatly releases the resources

CH : 01 ABSTRACTION

#11 ACCESS MODIFIERS

* Types of Access Modifiers
  - Private
  - Public
  - Protected
  - Internal
  - Protected Internal
  - Private Protected (C# version 7.2 onwards)

* default
  - The members (data variables, function, constructor) by default have private as access Modifiers & you can apply any of the 6 modifiers on data members of class, struct, interfaces, delegates etc.
  - The type (classes, struct, Enum, interfaces & delegates) can have only 2 (public & internal) as Access Modifiers. By default it is internal



* Assemblies In .Net

- Assembly is the IL code along with meta data
- Assemblies is the building block of deployment.
- Assembly is smallest unit of deployment, meaning you can copy it & move to another computer
- CLR reads & Executes this assembly
- Types : 1) DLL : dynamic link library, it is Used as reusable code that other program uses
  - DLL does not contain Main() program
  - class library when build creates .dll files
  2) EXE : executable it is A standalone application that runs on its own
  - Exe assembly contains Main() function
  - Console app projects when build creates .exe files
  - Exe is Executable file :. when you double click it, it will run

* Bin & Obj folder

obj : it includes the intermediate Build files
- there are lot of small .exe files of all main functions
- it includes
- it includes temporary files created during build process and also includes partially compiled assemblies
- compiler generates first out put in obj folder before moving the final build to bin folder
- obj does not include other projects dll files

Bin : (final output) : it includes only one .exe file which is combination of all .exe files from obj folder
- it will also include DLL files and only one .exe file
- Bin includes other projects (project of DLL types) dll files also and only one combine .exe file

* DLL & exe

- DLL files cannot be run like exe files
- .exe file is run inside its own memory space
- .dll files are run in others memory space
- you need to add refrence of .dll (build file) in Console project, so when you run .exe file of that project then both files are run in .exe application only
- class library project will Generate a DLL file

* Scope of Access Specifiers

1) **private** : private data members (functions, variables, constructors) can be accessed within the class only that class

- outside the class even the object of that class cannot access them
- Even if you inherit the class, you cannot access these data members or data functions
- when a child class inherits from a parent class, it inherits all the fields including private members, but private members remain inaccessible in the child class beoz they are private

~~2) protected~~

- Conclusion, private data members & private functions are inherited but child class cannot access them
- child class inherits the presence of private data, but does not inherit their visibility
- child class can access the private data, if the parent class has a public or protected method which ~~rete~~ returns that private data, then child class can call the public/protected function of its parent class and can access the value of private data field

2) **protected** : protected data Members or Member function (constructor, function) are available within that class, as well as all the classes which are derived (inheriting parent class) from that class, but not outside the class

- it means that the obj of derived class also can directly access the protected data of parent class within the classes which are inheriting the parent
- outside the class (in class which is not derived from parent class) the objects cannot access the protected data
- protected data is private but can be accessed by derived or child class, but not outside the class

eg: class 1

↓

class 2 : class 1

class 2 obj can access protected inside class 2

class 3:

obj of class 1 & obj of class 2 cannot access protected data in class 3

✓ with the class
✓ Derived class in Same Project
✗ Non Derived class in same Project
✓ Derived class in other project
✗ Non Derived class in other project

3) Internal : when data members / member functions (Constructor function) are marked internal, then they are available anywhere within that particular project (namespace or Assembly)
- Outside that project or outside in a diff project or in different DLL then members functions are not ~~are~~ available to access

within the class ✓
Derived class in same Assembly/Project ✓
Non Derived class in same Assembly/Project ✓
Derived class in diff Assembly ✗
Non Derived class in diff Assembly ✗

4) Protected Internal (PI) : Combination of Protected & Internal
- Internal : allows Everywhere in same project
- Protected : same class, derived class in other as well as same project
- ∴ protected Internal is accessible Everywhere within same project as well as all derivide classes of other project
- Restricted only in the non derived class of diff project

~~5) Private Protected~~

within the class ✓
Derived class in same Assembly ✓
Derived classes in other Assembly ✓
Non Derived classes in same Assembly ✓
Non Derived classes in other Assembly ✗

5) Private Protected : Private Protected members are Accessible only within the class & within the derived classes of same project/Assembly
- PP cannot be Accessed from another assembly

within the class ✓
Derived class in same Assembly ✓
Derived classes in Other Assembly ✗
Non derived classes in same Assembly ✗
Non derived classes in other Assembly ✗

	within the class	Derived class (same Assembly)	non derived class (same Assembly)	Derived class (other Assembly)	non derived class (other Assembly)
Private	✓	✗	✗	✗	✗
private protected	✓	✓	✗	✗	✗
internal	✓	✓	✓	✗	✗
Protected	✓	✓	✗	✓	✗
Protected - internal	✓	✓	✓	✓	✗
public	✓	✓	✓	✓	✓

#2 ABSTRACTION

* Abstraction

- Abstraction is a process of hiding the implementation details of an object & showing only the necessary features
- In simple words, user should only know what the object functionalities do, rather then how they do it
- Abstraction can be implemented using :
  1) Abstract class
  2) Abstract methods (~~virtual function~~)
  3) Overriding the methods (Virtual function)
  4) Interfaces

Abstract class can have protected constructor, so that the class which inherits the Abstract class can call that protected constructor using Base keyword

- Abstract classes cannot be instantiated bcoz if compiler will allow to make obj of Abstract class then we can call the Abstract method using obj, which CLR cannot execute at Runtime bcoz it doesn't have any implementation

* Abstract class

- Abstract class is a class which cannot be instantiated and compulsory these classes are ment to be inherited
- it may contain Abstract method (without implementation) and concrete methods (with implementation)
- Abstract classes only supports single inheritance
- Syntax :

```
abstract class class_name
{
abstract method → public abstract void MakeSound() ;
                  public void othermethods()
                  { // definition }
} → implemented method
```

- abstract class can contain implemented method also
- Abstract method must be overriden
- Abstract method cannot be declared in non abstract class
- to override a implemented method, you need to make it as virtual

- Abstract class cannot create objects

* Abstract Methods

- you can create Refrence of Abstract class pointing child class object

- Any class can contain Abstract methods, which must be Implemented (define) in the child class and parent class must be abstract
- ~~Any~~ methods cannot be decalare Abstract in a non abstract class, it will give Error
- ∴ the methods which are abstract must be in a class which is abstract
- without abstract class, you cannot have abstract methods
- Abstract methods must be implemented in derived class, otherwise it will give Error
- Method without body is called Abstract Method

* Virtual functions

- There is no pure virtual fun concept in C#, like we have in C++
- for achiving abstraction, you can make the class fun as virtual
- The virtual fun can have implementation but still can be overriden in derride class
- This method avoids forcing inheritance (unlike abstraction classes, which ~~to~~ must do inheritance)
- When you want default behaviour in base class but allow customization in derived class use virtual fun
- no compulsion on overriding the method
- Syntax:

```
class parent
{
    public virtual void fun_name()
    {
        // defination // you can override this virtual fun
                      but not compusory to override the fun
    }
}
```

* Using Interface

- An interface provides 100% abstraction
- Interface only has method signature (no implementation)
- multiple inheritance supported, A class can inherit multiple Interfaces
- all interface name starts with capital I
- By default all data members & functions are Abstract & public and non static

X • you can also define the methods in Interface only in 8+ version

X • you can also override the interface methods if they are define in interface only in 8+ version

X • The methods which are not define must be define in derived class only in 8+V

✓ • By default all methods in interface are public & Abstract

- We cannot create obj of interface, but we can create a refrence of an interface and assign it to a obj of class that implements that interface
- Syntax:

```
interface Iname
{
    // fun without defination
    // no implementation of function
}
```

* Example of Abstraction

1) ATM machine : functionalities : Insert ATM
Enter PIN
Withdraw Money

Hidden : How card is validated
How transaction is processed
How data is stored in DB

eg: social Media Login : you only need to provide the user_id & pass & automatically authentication is done in backend you don't know how

* Encapsulation & Abstraction

Encapsulation	Abstraction
• it is all about data hiding • we can protect data from being accessed from outside	• all about hiding implementation details • using Abstraction we are exposing only the services, so that user can consume them

- parent (even Abstract class can create refrence) class refrence, even if created using child class instance, cannot call child class methods. but overriden methods in child class can be called using parent class refrence becoz child has taken permission from the parent to re-implement the methods. So parent refrence can call them

CH: 02 Encapsulation

#1) Encapsulation

* Encapsulation

- The process of binding/bundling/Grouping up the Data Members & the Member functions into a single unit (classes, struct, interface etc) & classes is called Encapsulation
- Encapsulation Ensures that the Data Members & the function Members cannot be access directly from other units (classes, interface etc)
- Encapsulation = Abstraction + data hiding
- Encapsulation is noth but hiding data
- Real world eg: your school bag which contains items (data items) like pen, book etc, so bag is bundling or Grouping all data into a single unit called bag
- Use Case: Encapsulation ristricts other classes from accessing their data
- Data hiding is a process in which we hide internal data (private data) from outside world, purpose is to protect data
- Encapsulation is like putting something (data Members) inside a protective box so that ~~certain~~ only certain people (objects) can access it
- Encapsulation Ensures data security & controlled Access
- Encapsulation can be acheived by:
  1) Using classes & obj
  2) using Access Modifiers
  3) using getter & setter
  4) Using const & readonly keywords

* Using getters & setters

- getters: are function which are used to get the value of private data member & these functions are public
- setter: are function used to set the value of that private data member & these functions are public

eg: class parent
{
    private string _name; // private data
    public string Name ⟶ // this Name is property not function
    {
        keywords ⟵ [get { return _name; } set { _name = value; }] // getter & setter
        }
}
⟶ this is a keyword

eg: public int Age { get ; set ; } ⟶ // this will automatically generate private backing field

↓ it is same as writting this

```
private int age;
public int Age ⟶ // this is property not fun
{
    get { return age; }
    set { age = value ; }
}
```

- Using getter and setter you can do data validation & security by restricting direct access to variables
- Advantages of Data Hiding/Encapsulation:

1) Data protection
2) Data hiding
3) security
4) flexibility

CH: 03 INHERITANCE

#1) INHERITANCE

* Inheritance

- Inheritance is a mechanism in which parent class properties are consumed by child class
- it promotes code reusibility & DRY principal
- inheritance will Established a relation btw 2 class as parent & child, the child class will inherit all properties of parent private properties also are inherited but child class cannot directly access the private properties of its parent
- it will be see as the child class owns the parent class properties & child class can access them as those properties are define in child class
- Even obj of child class access the properties of parent class directly if they are public & internal
- C# supports only single inheritance (a class can inherit only one class)
- for multiple inheritance we have Interfaces
- A derived class can override base class methods using virtual & override keywords
- you can use base keyword to access the parents methods like to call parents constructor you can call base() function
- Syntax

```
class class_child : parent_class_name
{
    //body of child class
}
```

- use sealedd keyword to avoid or to restrict inheritance

* 6 Rules

Rule 1) parent class constructor must be accessible to child class, otherwise the inheritance will not be possible, because when you create obj of child class, first it goes and calls the parent class constructor implicitly, it happens so that parent class variables will be initialized & child class obj can consume them
∴ make constructor of parent class as public

Rule 2: In inheritance child class can access the member of parent class but parent class cannot access members of child class

Rule 3: Refrence Rule

class A = P

class B: class A = q

- p is refrence of class A, but it is pointing to an instance of class B
- Using P you can only access the methods of class A
- P cannot access methods of class B even if it is refrencing or pointing to obj of class B

eg:
```
class Parent
{
    Public void Method1()
    {
        Cwl("Method1 printed");
    }
}
class child : parent
{
    Public void Method2()
    {
        Cwl("child Method");
    }
}
class Program
{
    Static void Main()
    {
        Parent obj = new child();
        obj.Method1()
        Obj.Method2() // gives Error beaz obj is of class Parent ∴ can't access the child class methods
    }
}
```

~~even if obj is of type child~~

obj cannot access the methods of child class

- if the methods are overriden using virtual keyword then only it will access child class methods

This is called Coversicance

```
child obj = new Parent(); // this cannot happen beaz Parent class can't call Constructor of child class
```

you can't do this

Rule 4 : Every class we define is derived from object class
- object is parent for all the classes, so it has some functions inherited like Equals, GetHashCode, GetType, and ToString

eg:
```
class Program
{
    static void Main()
    {
        string str1 = "Hello";
        string str2 = "Hello";
        string str3 = new string("Hello");

        CWL(str1.Equals(str2)) // output True
                    └→ // Equals() fun checks the values of obj
        CWL(object.ReferenceEquals(str1, str3)); //output : false
                    └→ This fun checks the address they are pointing
        CWL(str1.GetHashCode()) // output : Hash Code of "Hello"
                    └→ // gives you hashcode
        CWL(str1.GetType()) // output : system.string
                    └→ // gives type of that obj
    }
```

- if constructor is parameter less then child class constructor automatically calles Parent class constructor
- if Parameterized parent constructor then use base() keyword

Rule 5 : if your parent class has parameterized constructor, then in child class constructor you need to pass all the parameters to parent class
- you need to take the parameters input from child class & then pass the variables using Base keyword

Syntax :
```
public child(int num) : base(num)
       └→ constructor of child class
```
└→ Base will call parent class constructor & send num to parent constructor

Rule 6: using classes you can only have

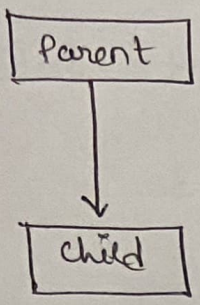
1) single inheritance
Parent → child

2) Multilevel inheritance
Grand Parent → Parent → child

3) Hirarchical inheritance
Parent → child1, child2

* Types of Inheritance

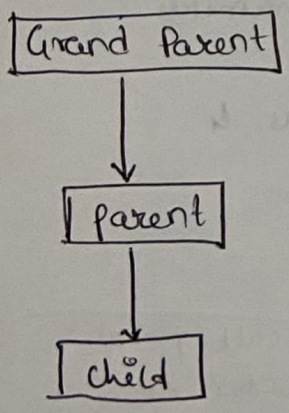
1) Single Inheritance : single child class inherits from single parent class




```
eg: class Parent()
    {
        public void show()
        {
            cwL ("This is Parent class");
        }
    }
    class child : Parent
    {
        Public void display()
        {
            show()
            cwL (" child class ");
        }
    }
    class Program
    {
        static void Main()
        {
            child obj = new child();
            obj.show();
            obj.display();
        }
    }
```

2) Multilevel Inheritance : A child class inherits from parent class. parent class inherits from grand parent class

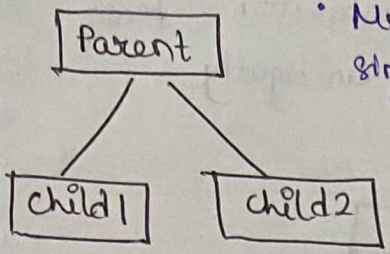
- So child class will have all properties of parent as well as grandparent class




```
eg: class Gparent
    {
        public void GP()
    }
    class parent : Gparent
    {
        Public void Parent()
    }
    class child : parent
    {
        static void Main()
        {
            child obj = new child();
            obj.GP();
            obj.Parent();
            obj.child();
    }   }
```

3) Hierarchical Inheritance : Multiple childs are produced from single parent class

- Multiple child classes inherit from same ~~parent~~ single parent

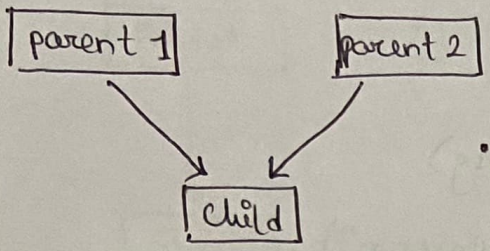


eg:
```
class parent
{
    public parent(string msg)
    {
        cwL(msg "of constructor");
    }
    public parent() { }
}
class child1 :parent
{
    public child1() { }
    public child1(string msg, string msg2): base(msg2)
    {
        cwL(msg + "from child1 class");
    }
}
class child2 : parent
{
    public child2(string m) : base(m) { }
    static void Main()
    {
        child1 obj1 = new child1("msg1", "msg2");
        obj1.parentclass-methods();

        child2 obj2 = new child2("msg to parent");
    }
}
```

you need to use base() keyword & pass all parameters to the parent class from child class constructor

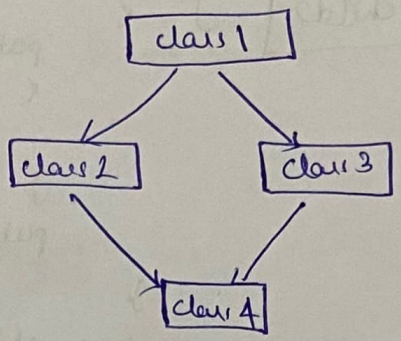
- first child class constructor will get call but, this child constructor will call the parent constructor ∴ parent constructor will get Execute first & after that Execution of child constructor takes place

4) Multiple inheritance (not supported by classes) : A single child class inherits multiple parent classes



- C# does not allow multiple inheritance becoz of diamond problem or ambiguity issues

- diamond problem :
  - so if multiple inheritance is allowed child class 4 will inheright same methods of class 1 from both both classes class 2 & class 3



- This will create data duplicacy & Ambiguity
- ∴ multiple Inheritance can be Achieved by Interfaces

* Interfaces

- Interface is a contract that defines a set of members that a class or struct must implement
- Interface do not contain implementation details, only declaration this allows for loose ~~coo~~ coupling & supports multiple inheritance
- Properties :
  1) No implementation, only decalaration of Methods
  2) By default interfaces are public, abstract & non static
  3) you cannot decalare static members in interface but after C# 8.0+ we have static Methods (with implementations), static properties, static fields, static Abstract Members
  4) interface cannot have instance data members
  5) interface cannot contain Data fields, Constructor, destructor
  6) Interface can contain implementation of fun from C# 8.0+
  7) By default all members are public, non static & abstract but after 8.0+ we can also decalare, private & internal member functions

8) Interface can inherits from other interfaces, one or more Interface can be inherited using multiple inheritance

9) Interface cannot be initiated, object cannot be made

* Interface can Contain :

1) Abstract Methods
2) Properties
3) Indexes
4) Events

eg:
```
public Interface IExample
{
    static int x = 10;
    Static void show()
    { cwl(x);
    }
}
class Program
{
    static void Main()
    {
        IExample.show()
    }
}
```
↳ static method called

* Interface cannot Contain :

1) Non abstract function (after 8.0 version they can include non abstract fun also)
2) Data fields (eg int x = 10) (not allowed) ✗
3) Constructors & Destructors
4) static fields (eg static void add()) (not allowed)

// static fields are allowed after 8.0 version & you can access them static field using interface_name.that field

* Rules of interfaces

Rule 1 : default scope of function members of interface is public, abstract & non static

- non static means, they (fun which are non static) must be accessed or implemented by object of class which inherit the interface
- As every member fun is Abstract, you must implement that fun in the class which inherits the interface

Rule 2 : We cannot declare fields/variables, Constructor, destructor in Interface int x = 20 // gives Error before 8.0 & even in 8+V it gives error // you cannot have non static data fields or members even in 8+ version also

Rule 3 : Interfaces can inherit more then one interfaces or another interfaces

Rule 4 : Every member fun must be implemented in its child class but after 8+ version you can Even define method body Even in Interfaces. So interface can contain non Abstract fun also

- So if a method has implementation in interface, then it is not necessary to define them in child class. ✓
- but you can override the implemented method as well as not implemented method without using override keyword
- only static fields are allowed after 8.0 version, but non static field, not allowed

Rule 5 : We cannot create instance of ~~Variables~~ Interfaces

eg: static int x = 10; ⟶ allowed in 8+V
int xy = 100; ⟶ Not allowed even in 8+V

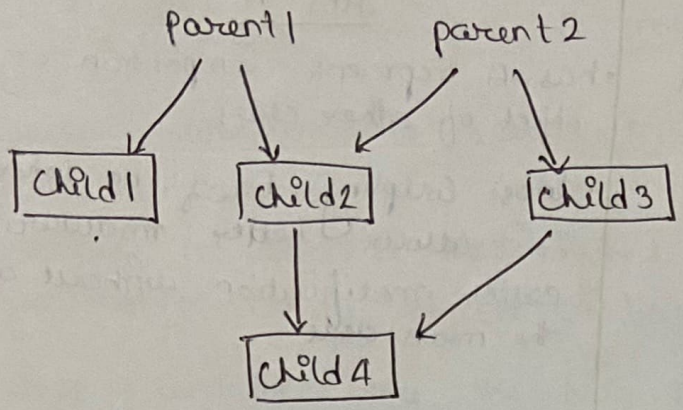
eg:
```
public interface IFirst
{
    void show(); // no defination is allowed of show in interface
}

public interface ISecond
{
    void display()   // by default public, non static & abstract

not allowed  int x = 10;   // not allowed Even in 8+V

    static int xyz = 100;   // not allowed allow only in 8+V

    static void add()   // not allowed allow only in 8+V
    {
        return 10;
    }
}

public interface IThird : IFirst, ISecond   // multiple inheritance
{
    void add(int x, int y);

not allowed  int add(int x, int y)   // not allowed to implement body of fun allow in 8+ version
    {
        return x+y;
    }
}

class program : IThird, ISecond, IFirst
{
    void show()
    {
        Cwl("show of IFirst");
    }
    void display()
    {
        Cwl("display of ISecond");
    }
    void add(int x, int y)
    {
        Cwl("x & y" + (x+y));
    }
    static void Main()
    {
        program obj = new Program();
        obj.show();
        obj.display();
        obj.add(10,20);
    }
}
```

NOTE
Even if you inherit all 3 interface, you won't get Error, Even though IThird has IFirst & ISecond inherited

// These 3 fun must be overloaded, otherwise you will get Error

8+V
- private methods, private properties, but no fields (data fields) allowed
- interface cannot have instance data fields
- Can create static constructor but no instance constructor
- private data fields ~~are~~ are not allowed
- private properties are allowed read only
- Interface do not support protected members

5) Hybrid inheritance : It is combination of any 2 types or more types of inheritance

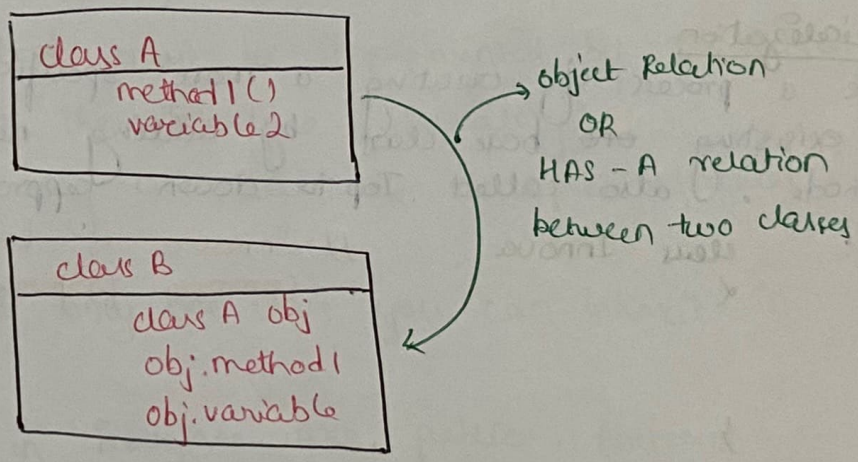


- you can implement this only using interfaces & one class

#2 Is a – has a Relation

* has-a : when you create obj of a class A and access the data of class A in class B through object of class A then this Relation b/w class A & class B is called "has-A" Relation

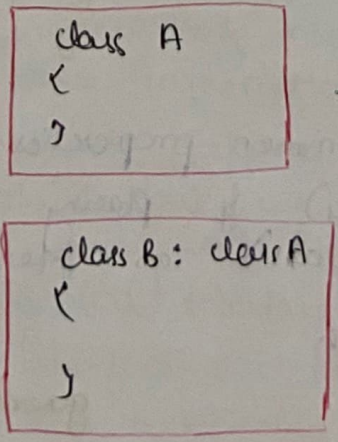
- Composition : when you declare a variable of one class inside another class it is called composition or you can say HAS-A relationship



* Is-a : If you inherit classA in class B then directly you can access these data members of class A into class B bcoz of inheritance, this is called IS-a Relation b/w classes

eg: class Cuboid { };
class Rectangle : cuboid
{
}

- Rectangle is a cuboid
∴ inheritance is a Relation



* Diff b/w is A & has A

IS-A (inheritance)

- Is-A represents inheritance b/w classes
- Strong coupling: beoz of inheritance, the derived class is tightly coupled with base class. changes in the Base (parent) class can affect all derived class
- Used when class needs to extend another class functionalities
- eg: BMW is A car

```
class BMW : Car
{
}
```

HAS-A (object)

- has-A represents composition or object of other class
- loose coupling, beoz no inheritance :. it allows better modularity & easier modification without affecting the main class
- used when class requires another class work but not its extention

eg: Employe HAS A address

```
class Details
{
    Employee obj = new Employee()
    obj: address
}
```

* Specialization

- it is a process of creating a more specific subclass from an exisisting ~~one~~ base class by adding new properties or methods. also called Top to down approach
- eg:

```
class Innova                    // Innova is base class with common properties
{

}

class Forturer : Innova         // forturer is specialized Innova by adding luxury features
{

}

// This demonstrates specialization beoz forturer is an advance version of Innova
```

* Generalization

- it is process of Extracting common properties & behaviours from multiple classes (child classes) & placing them under one single base class (Parent class) or interface
- also called Bottom up approach
- eg: BMW, Innova, Audi . general class = Car
- process of creating common base class from multiple similar classes

#3 Interfaces

* Interface

- Interface is a fully unimplemented class
- in simple words interface is nothing but a pure abstract class
- interface are used to achive multiple inheritance & full abstraction beacuse it cannot have method body
- interface is also a user defined data type like classes but you cannot create objects or variables of interface
- class : Contains only the Non-Abstract Methods (methods with Body). also called Concrete Class
- Abstract class : Contains both Non Abstract Methods (Methods with Body) and Abstract Methods (Methods without Body)
- Interface : Contains only Abstract Methods (Methods without Body)
- Every Abstract Method must be implemented by the child class which inherits the Interface (Mandatory)
- A class can inherit only one class & multiple interfaces at the same time
- ~~Multiple~~ interfaces & only one class you can inherit in any child class
- by default Methods in Interface are public, Abstract
- you cannot decalare fields, variables Constructor & destructor for interfaces
- no need to use override keyword ~~while~~ while implementing the methods in child class of interface
- We cannot create instance of interface beacuse the methods in them doesn't have implementation
- you can create interface Refrence, the refrence is going to hold child class instance. but we cannot invoked pure child class methods using the Refrence, but can invoked the decalare methods in Interface with that refrence
- When the methods of interface are implemented in child class using the interface name, then it is called Explicit interface implementation
- eg: void ITestInterface.Sub (int a, int b) { //body } → // Explicit interface implementation

* imp Points

- Interface cannot contain:
  1) Non Abstract functions
  2) Data fields
     eg: int a
         int b=10;
  3) Constructor & destructor
  4) Method body
     eg: int sum(int a, int b)
         {
             return a+b;
         }

- Interface Can Contain only
  1) Abstract Method
  2) Properties
  3) Indexes
  4) Events

* Eg: Banking Account

- syntax:

```
interface IinterfaceName
{
    //only method decalaration
    void Display();
    void sum();
}
```

- By default interface are having Internal as access modifier
- you can only apply public & Internal on classes, interfaces, structures
- syntax:

```
public interface Iname
{
    //void Draw();
}
```

- you can apply private, protected and all access specifies only on inner classes but not on outer class

* Inner Interfaces

- Interface can contain: inner interfaces
- Interface cannot contain: inner classes, or classes, or nested classes
- Interface only contains Abstract Methods & inner interfaces
- Interface can not contain: inner classes
  inner structure
  nested classes
  normal classes

eg:
```
public interface IOuter
{
    void show();
    interface IIner
    {
        void Display();
    }
}
class Outer : IOuter
{
    public void show()
    {
        Cwl("Implementing outer interface");
    }
}
class Both : IOuter, IOuter.IIner
{
    public void show()
    {
        Cwl("outer interface in Both class");
    }
    public void Display()
    {
        Cwl("Display implemented in Both class");
    }
}
class Program
{
    static void Main()
    {
        IOuter obj1 = new Outer();
        obj1.show()

        IOuter.IInex obj2 = new Both();
        obj2.show();
        obj2.Display();
    }
}
```

// created obj refrence of Interface which points to class object & by the refrence also you can call the implemented methods

eg: Real Life Eg of Interface/Abstraction, Banking Account

```
namespace Bank
{
    public interface IBankAcc
    {
        bool DepositeAmount (decimal amt)
        bool WithdrawAmt (decimal amt)
        decimal checkBal ();
    }
    public SavingAcc : IBankAcc
    {
        Private decimal Bal = 0;
        Private readonly decimal WithdrawLimit = 1000; // non static & you cannot modify the value now being it readonly
        Private decimal TodayWithdraw = 0;
        public bool DepositeAmount (decimal Amt)
        {
            Bal = Bal + Amt;
            cwl ("your total AC Bal" + Bal);
            return true;
        }
        public
```

* Multiple Inheritance Using Interface

- if a class does not wish to provide implentation for the method it inherits from interface, then that class has to be marked as Abstract and also that method in that class needs to be decalare as Abstract
- we don't face diamond problem in interface because interface provides the method to child class for implementation, but not for consumption.
- Consumption creates ambiguity problem, not implementation
- As child will define the method, Even if there are 2 interfaces with same methods, the child will implement them & when the call is made the method implemented by child is call ∴ no Ambiguity, so when class implements those method, both interfaces are happy beoz they got implementation for their methods
- you can also implement the same name methods of 2 diff interfaces seperatly by using "Explicit interface implementation"
- eg:

```
public interface Ione
{
    void Tes();
    void show();
}
Public Interface Itwo
{
    void Tes();
    void show();
}
class Program
{
    static void Main()
    {
        Testing obj = new Testing();
        obj.Test();
        // obj.show(); // creates Ambiguity ∴ can't do this
        Ione obj2 = obj;
        obj2.show();
    }
}
```

you can't call show method using class obj

refrencing pointer to class obj

// you can call the show method specifying the interface, by making refrence to that interface

```
class Testing : Ione, Itwo
{
    public void Tes()
    {
        cwl("Comon to both Ione & Itwo");
    }
    public
    void Ione.Show()
    {
        cwl("Ione show called");
    }
    void Itwo.Show()
    {
        cwl("Itwo show called");
    }
}
```

using Interface name. method you can implement them seperatly

- you can also typecast the obj of that class to interface type & call the show method

eg:

```
((Itwo)obj).show();
```

↳ type caste the class variable to Interface & then call the method

~~* ABSTRD~~

- Implementing Each interface method seperately & type casting them, so that we can call them

eg:
```
interface Car
{
    void Drive();
}
interface Bus
{
    void Drive();
}
class Demo : Bus, Car
{
    void Bus.Drive()
    {
        Cwl("Bus is Driving");
    }
    void Car.Drive()
    {
        Cwl("Car is Driving");
    }
}
class Program
{
    static void Main()
    {
        Demo obj = new Demo();
        ((Car)obj).Drive();
        ((Bus)obj).Drive();
```

type casting

```
        OR//
        Car ref1 = obj
        ref1.Drive();
```

// reference of interface pointing to the class object can call & also call that interfaces implemented method

```
        OR//
        Bus ref2 = obj
        ref2.Drive();
    }
}
```

* Interface V/S Abstract class

feature	Abstract class	Interface
implementation	Can have both Abstract & Concreat (implemented) Methods	• Only method decalaration (C++ allows implementation also)
data fields	• Can have fields, Constructor Destructor	• Cannot have fields, Constructor & destructor
Multiple inheritance	• only single inheritance	• Support all
Access Modifyers	• members can have any access specifyer	• all have only Abstract By default
Use Case	• when some Methods need default implementations • when some fun needs to be shared amoung derived classes • when we need Constructors & data fields • when you want partial implementation in Methods • when we want some implementation that will be same for all derived classes then its better to go for an abstract class	• when all methods should be implemented in derived class • when we need to Define a Contract without implementations • when you want Multiple inheritance • Even you can ~~use~~ inherit Interfaces in Abstracet classes • with interface you can move your implementation to any class that implements the interface
Couplying	More tightly Coupled	Loosely Coupled

- If your class is not implenting the inherited Method from interface, then in that class, ~~the~~ in that ~~that~~ Method defination you can throw a Exception as NotImplementedException()

CH: 04 Polymorphism

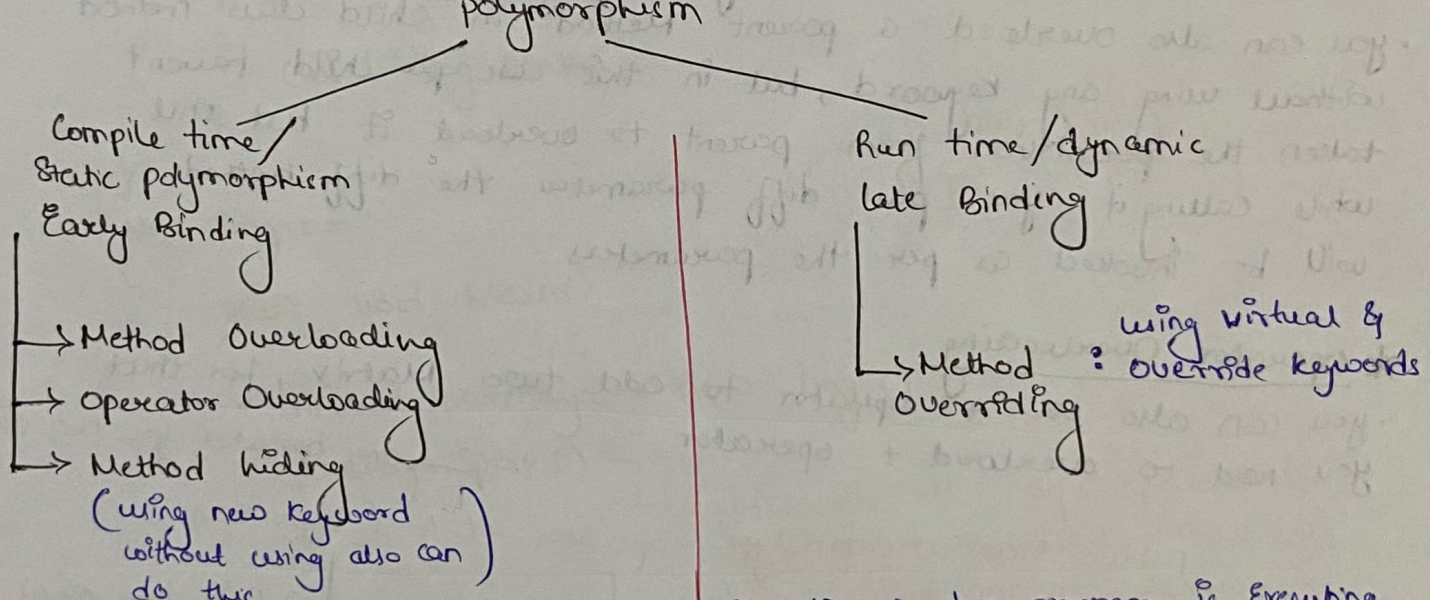
(1) INTRO

* Polymorphism

- polymorphism means many forms
- polymorphism is a ability to take more then one form, in simply words one thing that can take different forms as per the input types
- using polymorphism single Entities can behave differenty in different class, ~~&~~ Entities (~~&~~ Methods) can take many forms
- eg: yourself : Employee : in office
  student : in clg
  son : in home
  friend : for you collegues

* Types of polymorphism

- polymorphism can be achived using following ways:



- Compile time: source code converted to Machine Code that time is called compile time
- In this, the behaviour or which fun is called is ~~decide~~ decided during compilation (Converting source code to IL or ML code) only becoz the call has diff parameters, so the signature of fun is diff. So the compiler binds the method calls with method defination during compile time only
- binding is basically link blw fun call & fun defination

- Run time: when program is Executing it is called Run time (That time is called Run time)
- In case of Method overriding, we have multiple methods with same signature, Parent & child have same method implemented, so in this case we will be able to known at runtime (at Execution time) which method is called

Compile Time

- **Early Binding**: Since the Binding (link blw fun call & fun defination) is done at compilation (converting sc to MC) only ∴ it is called Early Binding. beoz the call & defination of fun is binded at Compilation
- also called static polymorphism

Run time

- **Late Binding**: the behaviour or which method to call is decided at Run time beoz all funs have same signature. Therefore CLR binds the method call with method body at run time by invoks the relevent method at runtime when method is called ∴ This Binding is late Binding
- also called Dynamic polymorphism

* Method Overloading

- We should overload Method beoz for eg:
- We have Add method which adds two integers & returns it, but now we want same method for floating no. ∴ we can't use diff names for creating Add fun for float no. , so we will overload the method by defining diff parameters
- you can also overload a parent Method in child class method without using any keyword, but in this case ~~per~~ child haven't taken the permission from parent to overload it but still while calling if you pass diff parameter the diff methods will be invoked as per the parameters

* Operator Overloading

- you can also use + operator to add two Matrix, for that you need to overload + operator
- syntax:

```
public static return_type operator +/-/= (type t)
{
    // statements
}
```

operator: this is keyword

+/-/=: any operator you can use

(type t): This type must be a class

* Method Overriding

- for this you need to use virtual & override keyword
- virtual : it gives permission to child class, even if there is implementation in base class the child class can re-implement that method using override keyword
- if you create a refrence of parent class & point to the child class obj, then you can access the overriden fun by using refrence of parent class beoz that functions are not pure fun of child class, the overriden defination will be call from refrence variable. This happens beoz you use virtual keyword. if you don't use virtual then this won't happen
- eg:

```
class Animal
{
    public virtual void Speek()
    {
        CWL("Animal make sound");
    }
}
class Dog : Animal
{
    Public Eat();
    public override void speek()
    {
        CWL("Dog Barks");
    }
}
class Program
{
    static void Main()
    {
        Animal parshaya = new Dog()        // Animal class refrence pointing to Dog class object
        parshaya.Speek()                   // This will call Dog's speek method beoz you have used virtual
                                           // This is called runtime polymorphism
        parashaya.Eat()                    // The parent refrence cannot call pure methods of Dog class, Even if it points to the Dog instance
                                           // can't call Eat method using refrence
    }
}
```

* Virtual Methods V/s Abstract Method

feature	Virtual	Abstract
Implementation	can have a default implementation	• Cannot have default implementation
Overriding	Can be overridden, but not compulsory	• Must be overriden in child class

* Method Hiding or shadowing

- Both Method Hiding & Method overriding are same only, but in Method overriding we take permission from parent to override the method by using virtual & override keyword
- But in case of Method Hiding, we don't use virtual & even don't take permission also, just we use new keyword & re-implement the parent class method
- you can also reimplement the fun without new keyword, but then you will get warning
- if you create Refrence of parent class pointing to child then also you can't call the the child overriden methods from the Refrence beoz, you haven't taken permission from parent to re-implement it, ∴ if you call Method of child through parent refrence then parent methods will be called beoz you haven't used virtual keyword

- eg:

```
class Parent
{
    public void show()
    {
        cw("Parent show");
    }
}
class child : Parent
{
    public new void show()
    {
        cw("child show");
    }
}
```

```
class Program
{
    static void Main()
    {
        Parent obj = new child();   // calls parent show not child show
        obj.show();

        child obj2 = new child();
        obj2.show();                // calls child show
    }
}
```

CH: 05 MORE ON OOPs

(41) 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 ~~assem~~ assembly 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