PAGE No.
DATE

OOPS

1. Data Hiding
2. Abstraction
3. Encapsulation
4. Lightly encaplulates class
5. IS - A Relationship
6. Has - A Relationship
7. Method signature
8. Overloading *
9. Overriding *
10. static control flow *
11. Instance control flow *
12. Constructors
13. Coupling
14. Cohesion
15. type - casting

} Module 1

18 | Jan | 25 . module 1

PAGE No.
DATE

DATA - HIDING

Outside person, can't access our internal data directly or our internal data should not go out directly this OOP feature is nothing but DATA hiding.

After validation or identification outside person can access our internal data.

eg → ① After providing proper username & password we can able to access our gmail inbox information.

② even though we are valid customer of the bank we can able to access our account information & we can't access other's account info.

By declaring data member (variable) as private we can achieve data hiding.

eg →
```
public class Account
    {
        private double balance;
        :
        :
        public double getbalance()
```

```
    {
      //validation
      return balance;
      }
    }
```

The main advantage is Security.

NOTE → It is highly recommended to declare data member (variable) as private.

ABSTRACTION

Hiding internal implementation and just highlight the set of services we are offering. is the concept of Abstraction.

eg → through Bank ATM GUI screen bank people are highlighting the set of services what they are offering without highlighting internal implementation.

Advantages →

The main advantages are —

① We can achieve security because we are not highlighting our internal implementation.

(2) Without affecting outside person we can able to perform any type of changes in our internal system & hence enhancement will become easy.

(3) It improves maintainability of the application.

(4) It improves easiness to use our system.

NOTE →

By using interfaces & abstract classes we can implement abstraction.

ENCAPSULAION

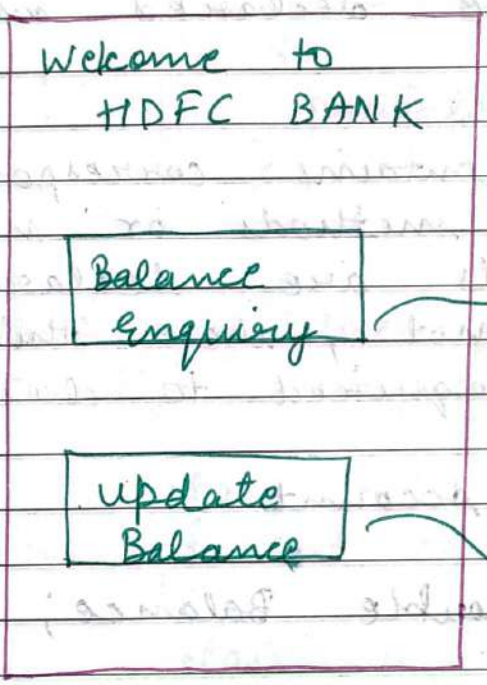
The process of binding data & corresponding methods into a single unit is called Encapsulation.

eg → class Student
{
data member
+
methods (behaviour)
}


(capsule)

If any component follows data hiding & abstraction such type of component is said to be encapsulated componend.

Encapsulation = DATA HIDING + ABSTRACTION




```
public class Account
{
    private double Balance;

public double getBalance()
    {
        // validation
        return Balance;
    }

public double setBalance()
    {
        // validation
        this.Balance = Balance;
    }
}
```

The main advantages of Encapsulation -

① We can achieve security.

② enhancement will become easy.

③ It improves maintainability of application.

⟹ The main disadvantage of encapsulation is it increase length of code & slows down execution.

tightly & Encapsulated & class

A class is said to be tightly encapsulated if & only if each & every variable declared as private.

whether class contains corresponding getter or setter methods or not & whether these methods are declared as public or not, these things we are not required to check.

eg →
```
public class Account
{
    private double Balance;

    public double getBalance()
    {
        return Balance;
    }
}
```

- Which of the following classes are tightly encapsulated ?

✓
```
class A {
  private int x = 10;
}
```

✗
```
class B extends A;
{  int y = 10;
}
```

✓
```
class C extends A
  {
    private int z = 70;
  }
```

- Which of the following classes are tightly encapsulated ?

✗
```
class A
{ int x = 10;
}
```

✗
```
class B extends A
{
  private int y = 20;
}
```

✗
```
class C extends B
{
  private int z = 30;
}
```

If parent class is not tightly encapsulated then, no child class is tightly encapsulated.

Module - 2

Is - a Relationship

* inheritance (also known)
* extends ← can implement using
* reusability (advantage)

⇒

```
class P
{
  public void m1()
    {
      Sopln("Parent");
    }
}
  class C extends P
{
   public void m2()
     {
       Sopln ("child");
     }
}

class Test
 {
   psvm (String [] args)
   {
① P p = new P();
   p.m1(); ✓
   p.m2(); ✗ → CE: cannot find symbol
               symbol: method m2()
               location class P
```

② C c = new C();
c.m1();
c.m2();

* ③ P p1 = new C();
p1.m1();
p1.m2(); → CE

④ C c1 = new P(); → CE: incompatible types
found : P
required : C

Conclusions →

① Whatever methods parent has by default available to the child & hence by child reference we can call both parent & child class methods.

② Whatever methods child has by default not available to the parent & hence by parent reference we can't call child specific methods.

③ Parent reference can be used to hold child object but by using that reference we can't call child specific methods but we can call methods present in parent class.

(4) Parent reference can be used to hold child object but child reference cannot be used to hold parent object.

example → without Inheritance

```
class VLoan
{
  300
  methods
}
```

```
class HLoan
{
  300 methods
}
```

```
class PLoan
{
  300
  methods
}
```

90$ hrs
900 methods

with Inheritance

```
class Loan
{
  250 common
  methods
}
```

```
class V Loan
extends Loan
{
  50 methods
}
```

```
class H Loan
extends Loan
{
  50 methods
}
```

```
class PLoan
extends Loan
{
  50 methods
}
```

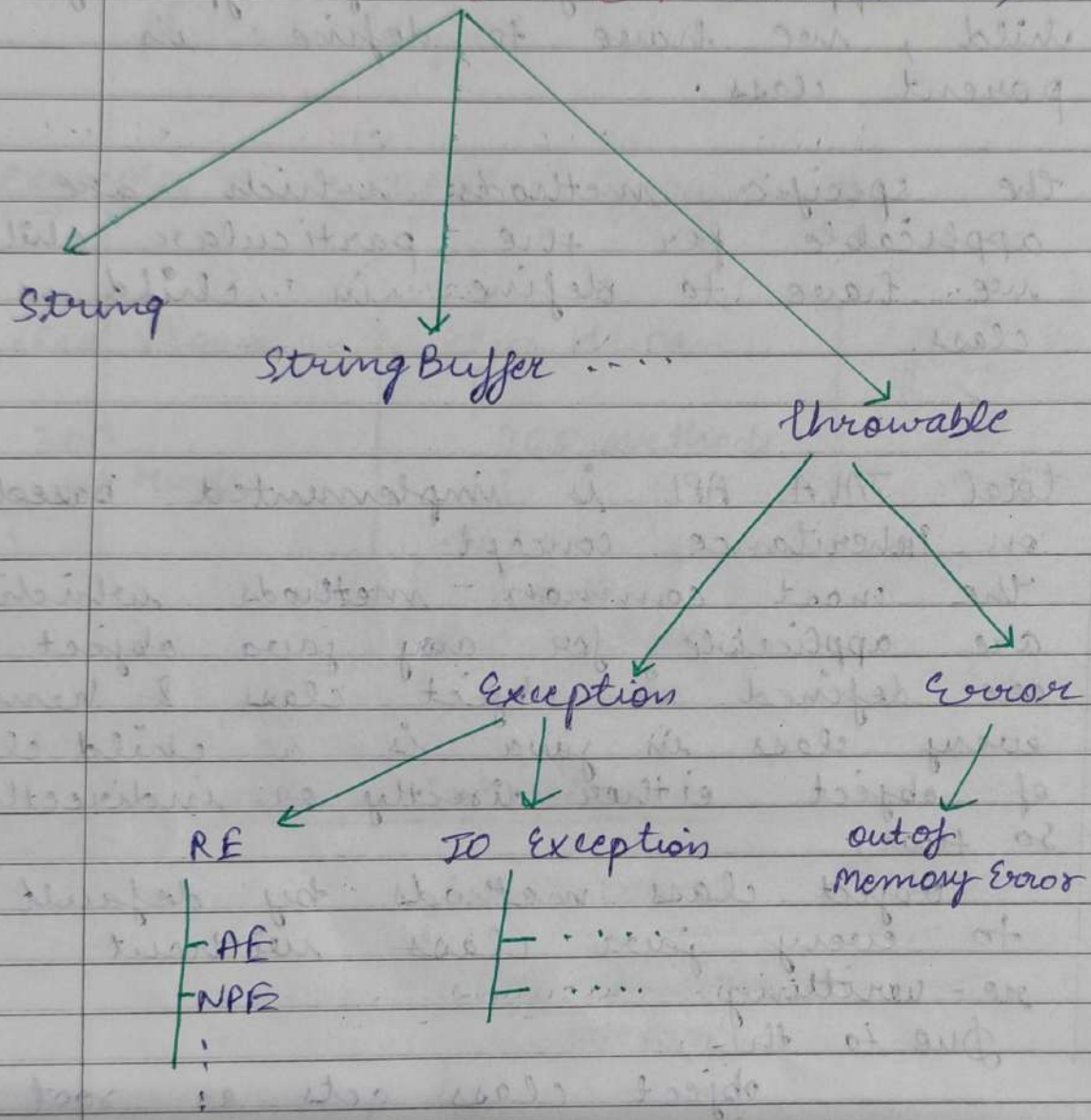
40 hrs
400 methods

NOTE →
the most common methods which are applicable for any type of child, we have to define in parent class.

the specific methods which are applicable for the particular child we have to define in child class.

→ total JAVA API is implemented based on Inheritance concept.
→ the most common methods which are applicable for any java object are defined in object class & hence every class in java is a child class of object either directly or indirectly.
so that,
object class methods by default to every java class without re-writting.
Due to this,
object class acts as root for all java classes.

→ Throwable class defines the most common method which are required for every exception & error classes & hence this class acts root for java exception hierarchy.


OBJECT (11 methods)
String
StringBuffer ....
Throwable
Exception
Error
RE
IO Exception
out of Memory Error...
AE
NPE

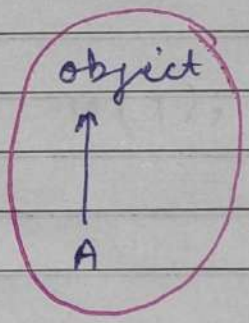
Multiple Inheritance

A java class can't extend more than one class at a time.
Hence,
JAVA won't provide support for multiple inheritance in classes.

eg → class A extends B, C
{
✗ → CE
}

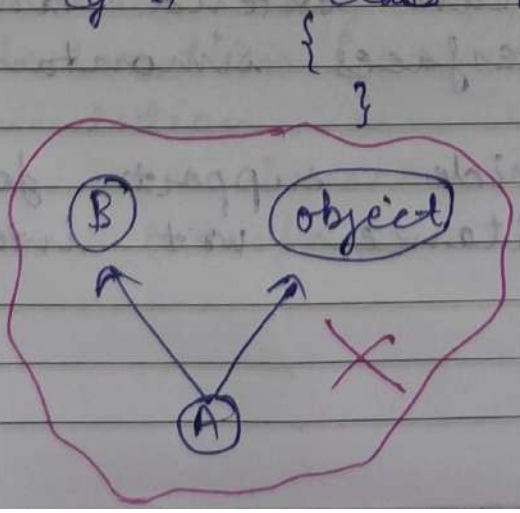
NOTE → If our class doesn't extends any other class then only our class is direct child class of object.

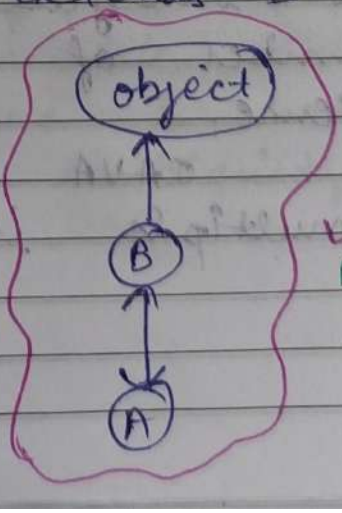
eg → class A
{
}



If our class extends any other class then our class is indirect child class of object.

eg → class A extends B
{
}



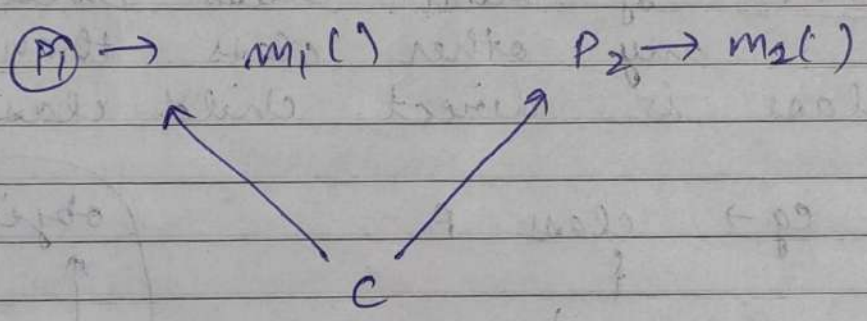


Multi-level inheritance

⇒ Either directly / indirectly java won't provide support for inheritance wrt classes.

Ques → Why java won't provide support for Multiple Inheritance?

Soln → there may be a chance of Ambiguity problem.
Hence,
JAVA won't provide support for multiple inheritance.



Ambiguity Problem

But interface can extend any number of interfaces simontaneously
Hence,
JAVA provide supports for multiple inheritance wrt interfaces.

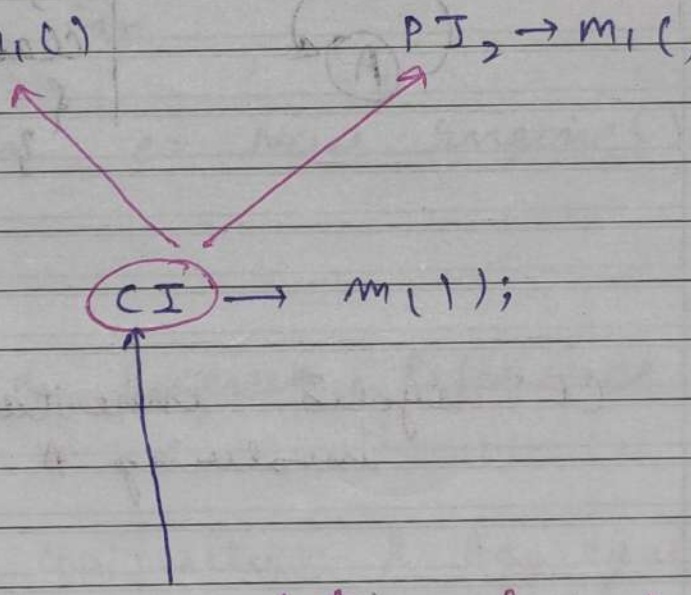
eg → interface A { }

interface B { }

interface C extends A, B { }

Ques → why Ambiguity problem won't be there in interfaces?

Ans →



Even though, multiple methods declaration are available but implementation is unique & hence there is no chance of ambiguity problem in interfaces.

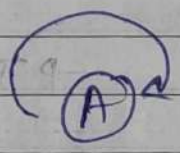
NOTE →

Strictly speaking,

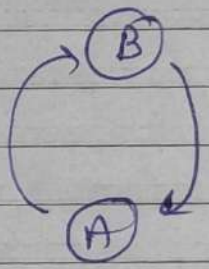
Through interfaces we won't get any inheritance.

Cyclic Inheritance

```
class A extends A
{
}
```




```
class A extends B
{
}
class B extends A
{
}
```



CE: cyclic inheritance involving A

cyclic inheritance is not allowed in java.

Ofc, it is not required.

Has - a Relationship

→ Has - a Relationship is also known as composition / Aggregation

→ There is no specific keyword that implements has - a relation but most of times we are depending on new keyword.

→ The main advantage of has a relationship is code reusability.

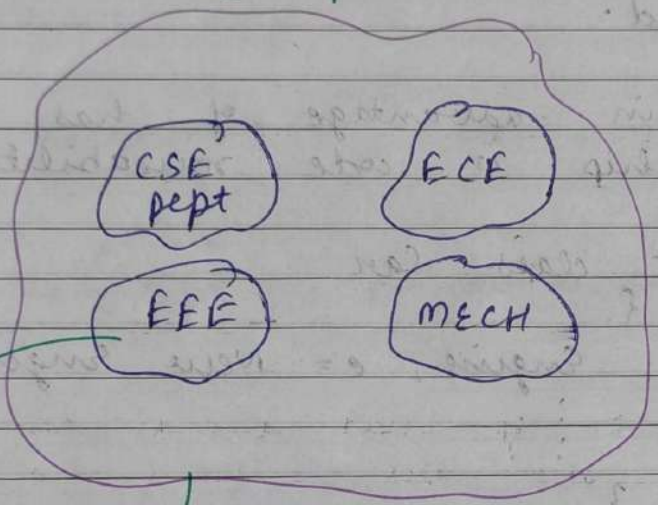
Example -
```
class Car
{
    Engine e = New Engine();
    :
    :
}
```
Car Has - a Engine Reference.

Difference b/w composition & Aggregation

Without existing container object if there is no chance of existing contained objects, then container & contained objects are strongly associated & this strong association is nothing but composition.

Eg → University consist of several Dept. without existing university there is no chance of existing dept, hence university & Dept are strongly associated & this strong association nothing but composition.

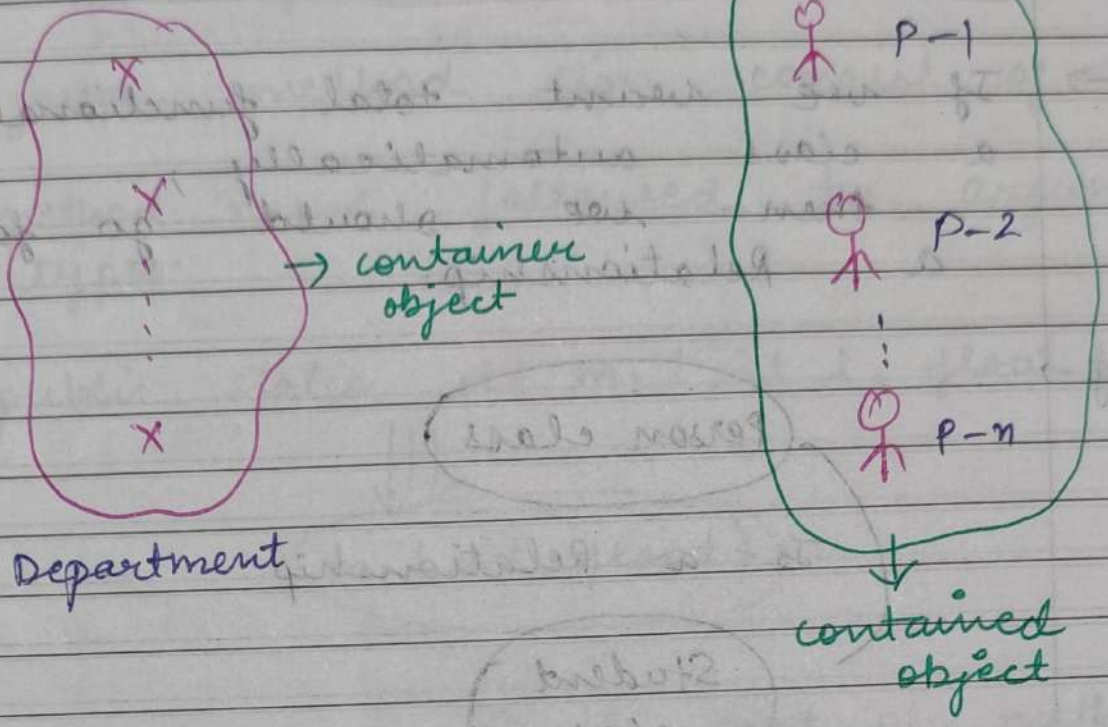
Composition



Aggregation

Without existing container object if there is a chance of existing contained object then container & contained objects are weakly associated & this week association is called Aggregation.

Eg →



Department consist of several professors without existing Dept there may be a chance of existing professors hence Dept & professors are weakly associated

NOTE → In composition, objects → strongly associated

In aggregation → objects → weakly associated

→ In composition, container object holds directly contained objects

In aggregation, container object holds references of contained objects.

NOTE →

→ In overloading, method resolution always takes care by compiler based on reference type.

→ In overloading, runtime object won't play any role.

Overriding

Whatever methods parent has, by default available to child through inheritance.

If child class not satisfied with parent class implementation then child is allowed to redefine the method based on its requirement.

this process is called Overriding.

the parent class method which is overridden is called overridden method & child class method which is overridding is called overriding method.

eg →

```
class P
{
    public void property()
    {
        sopln("Cash + Land + Gold");
    }
```

```
public void marry()
{
    Sopln("Subha");
}
```

overridden method

```
class C extends P
{
    public void marry()
    {
        Sopln("Mzn");
    }
}
```

overriding method ←

```
class Test
{
    psvm...
{
```

① P p = new P();
p.marry(); Parent method

② C c = new C();
c.marry(); Child method

③ P p1 = new C();
p1.marry(); Child method.

```
}
```

In overriding, method resolution always takes care by JVM based on runtime object and hence overriding is also considered as → runtime polymorphism

→ dynamic polymorphism

→ late Binding

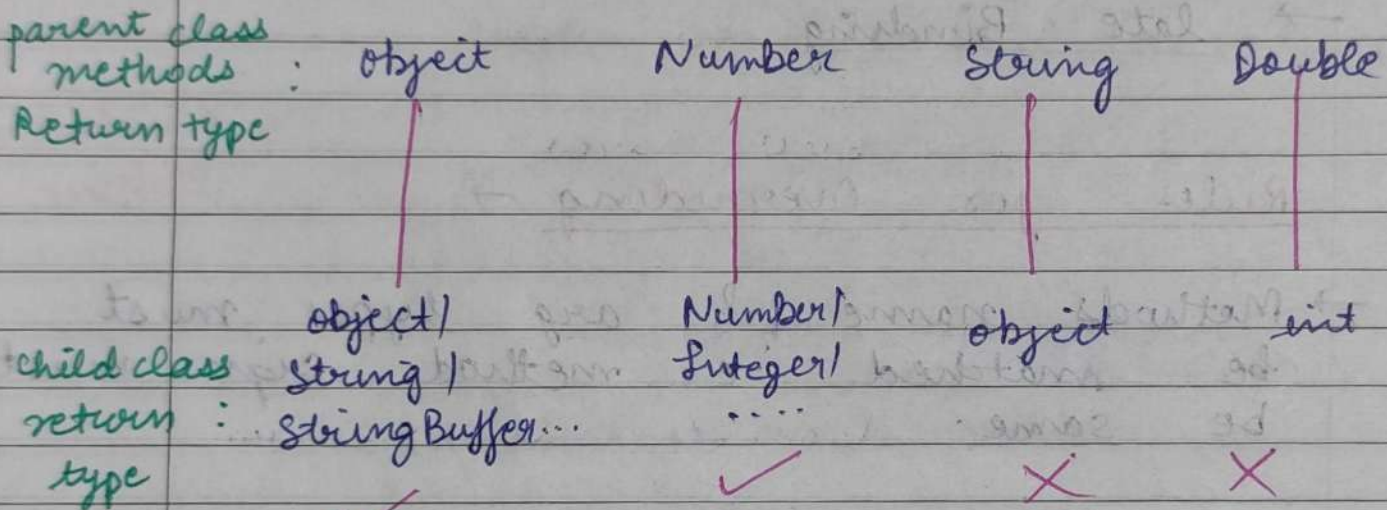
Rules for Overriding →

→ Methods name & arg types must be matched i.e method signs must be same.

→ Untill 1.4 versions,
Returns type must be same.
But from 1.5 V,
co-varient return type are also allowed.
ACT, child class method return type need not be same as parent method return type
→ its child type also allowed.

eg → class P
{
public Object m1()
{
return null;
}
}

```
class C extends P
{
    public String m1()
    {
        return null;
    }
}
```



→ co-varient return type concept applicable only for object types not for primitive types.

→ Parent class private methods not available to the child & hence overriding concept not applicable for private methods.

→ Based on our requirement, we can define exactly same private method in child class & it is

valid but not overriding.

eg →
```
class P
{
    private void m1()
    {
    }
}
class C extends P
{
    private void m1()
    {
    }
}
```

It is valid but not overriding.

→ We can't override parent class final methods in child classes if we are trying to override we will get CE.

eg →
```
class P
{
    public final void m1()
    {
    }
}
class C extends P
{
    public void m1()
    {
    }
}
```

→ CE: m1() in C cannot override m1() in P, overridden method is final.

P C	Non-Static Non-Static	Static Non	Non static	Static Static
	888	888	888	888
	999	999	999	999
	888	888	888	888

Difference btw

Overloading & Overriding

Properties	Overloading	Overriding
Method Names	must be same	must be same
Arg Types	must be differed (atleast order)	must be same (including order)
Method Sign	must be different	must be same
Return Types	No Restrictions	until 1.4v, must be same from 1.5 v, co-varient return types - allowed
private, static, final method	can be overloaded	cannot be overrid
Access modifier	No Restriction	the scope of access modifiers cannot be reduced but we can increase

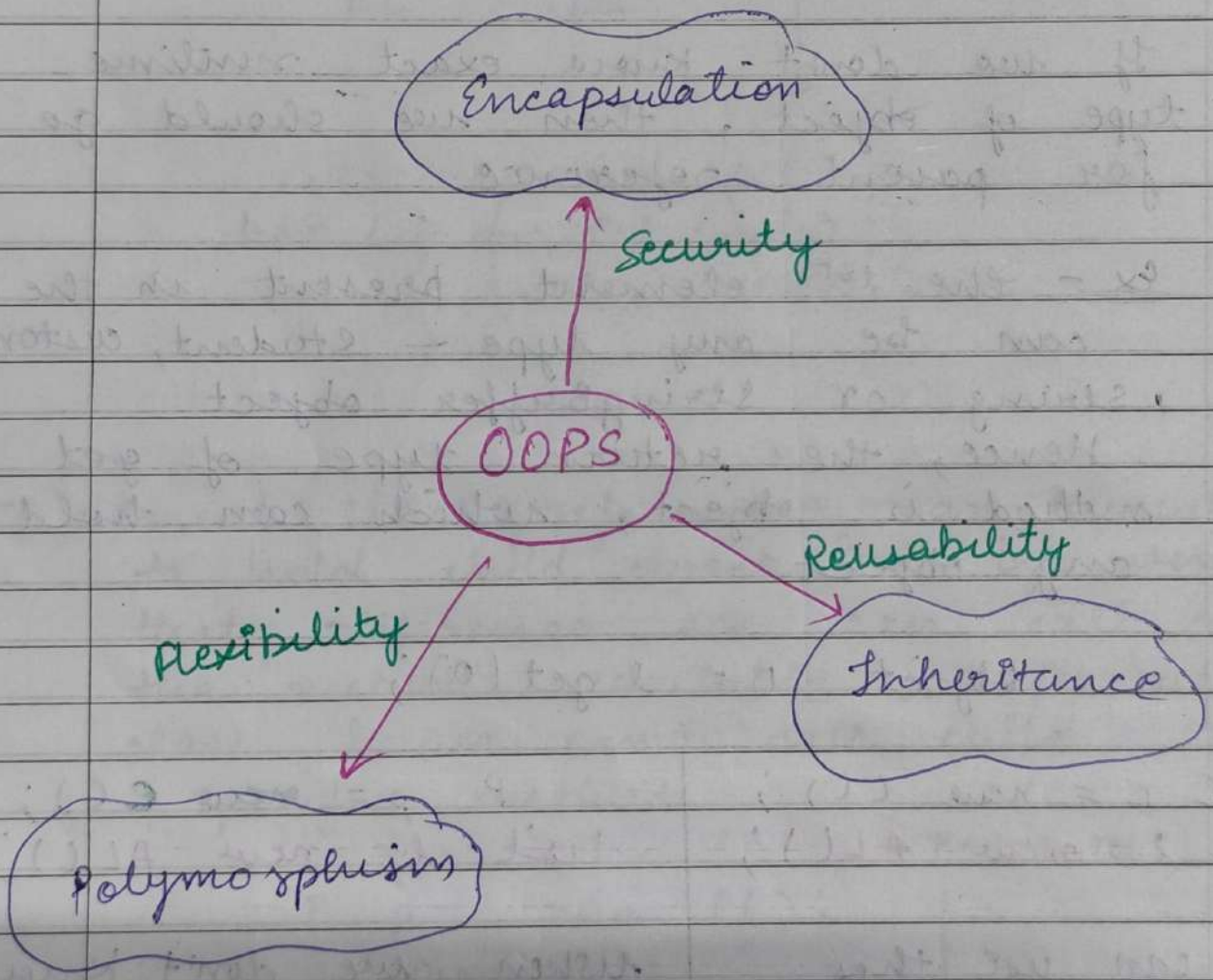
Throws clause	No Restrictions	If child class method throws any checked exception compulsary parent class method should throw the same checked exception or its parent. No Restriction — for unchecked methods
Method Resolution	Always taken care by compiler based on reference types	JVM will take care based on Runtime object.
Also called	compile-time poly. Static polymorphism Early Binding	Runtime poly. Dynamic Poly. Late Binding

NOTE → In overloading, we have to check only method names (same) & arg-type (different).
We are not required to check remaining properties.
But in Overriding, everything we have to check.

we can use child ref, to hold any particular child class object.
(Dis)

can hold any child object.
(Adv)

3 Pillars of OOPS



Polymorphism

Static poly.
or
compile-time poly.
or
Early Binding

- Overloading
- Method Hiding

Dynamic poly.
or
Runtime poly
or
Late Binding

- Overriding

Coupling

The degree of dependices btw the components is called coupling.

→ If dependices is more — then it is considered as tightly coupling (worst)

→ If dependencies are less — ~~less~~ losely coupling

eg →

```
class A
{
  static int i = B.j;
}

class B
{
  static int j = C.k;
}

class C
{
  static int k = D.m1();
}

class D
{
  public static int m1()
  {
    return 10;
  }
}
```

tightly coupling

The above components are said to be tightly coupled with each other bcoz — dependices btw component is more.

tightly coupling is not a good programming bcoz it has several disadvantages —

→ Without affecting remaining components we can't modify any component & hence enhancement will become difficult.

→ It suppress reusability.

→ It reduces maintainability of the application.

Hence,

we have to maintain dependencies between the components as less as possible i.e losely coupling is a good programming practice.

Cohesion

for every component, a clear well-defined functionality is defined, then that component is said to be follow high cohesion.

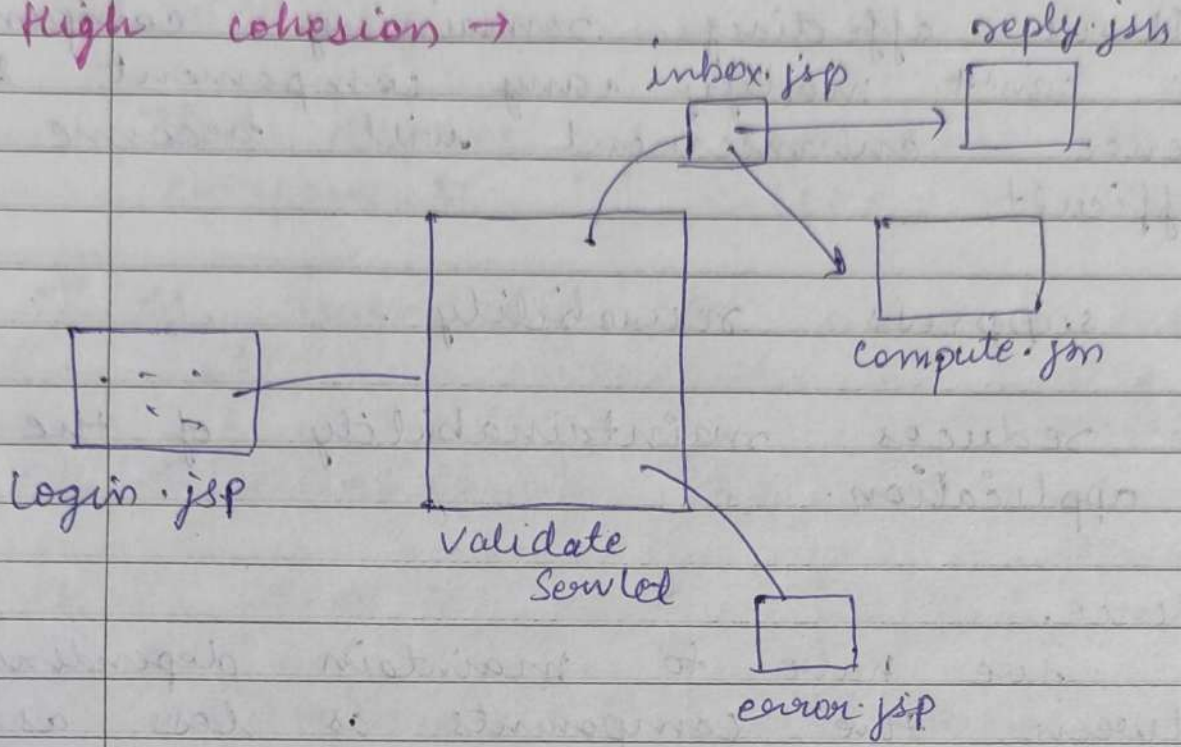
Total Servlet

Low cohesion →

10-lakh lines of code

LoginPage Display
Validation
inbox page
reply page
error page
.
.
.
.
.

High cohesion →



→ High cohesion is always a good programming practice bcoz it has several advantages.

(1) Without affecting remaining components we can modify any component. hence, enhancement will become easy.

(2) It promotes reusability of the code. (wherever validation is requise we can use same Validate servlet)

(3) It improves maintainability of the application.

Static block

Static blocks will be executed at the time of class loading & hence at the time of class loading if we want to perform any activity we have to define that inside a static block.

→ At the time of java class loading the corresponding native libraries should be loaded, hence we have to define this activity inside static block.

eg →

①
```
class Test
{
    static
    {
        System.loadLibrary("Native lib path");
    }
}
```

② After loading every DB driver class we have to register driver class with driver manager but inside DB driver class there is a static block to perform this activity & we are not responsible to register explicitly.

```
class DBDriver
{
   static
    {
      Register this Driver with
          DriverManager
    }
}
```

→ Within a class, we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

Que → Without writing main method is it possible to print some statements to the console?

→ Yes, by using static block.

eg →
```
class Test
{
   static
    {
      Sop|n ("Helo...");
      System.exit(0);
    }
}
```

o/p → Helo....

```
class DBDriver
{
    static
    {
        Register this Driver with
            DriverManager
    }
}
```

→ Within a class, we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

Que → Without writing main method is it possible to print some statements to the console?

→ Yes,
by using static block.

eg →
```
class Test
{
    static
    {
        Sopln("Hello...");
        System.exit(0);
    }
}
```

o/p → Hello...

Constructors

→ Once we creates an object, compulsory we should perform initialization then only the object is in position to respond properly.

→ Whenever we are creating an object some piece of the code will be executed to automatically to perform initialization of the object. this piece of code is nothing but constructor.

Hence,
The main purpose the constructor is to perform initialization of an object.

eg →
```
class Student
{
    String name;
    int rollno;
    Student(String name, int rollno)
    {
        this.name = name;
        this.rollno = rollno;
    }
    p.s.v.m
    {
        student S1 = new Studen("AB", 2);
        Student S2 = " " ("BC", 3);
    }
}
```

constructor ←

NOTE → the main purpose of constructor is to perform initialisation ~~is to~~ but not to create object.

(#) Difference btw constructor & Instance block

→ the main purpose of constructor is to perform initialization of an object.

→ But other than initization if we want to perform any activity for every object creation, then we should go for instance block (like updating 1 entry in DB for every obj creation or incrementing count value for every object creation etc)

→ Both constructor & instance block, have their own different purpose & replacing one concept with another concept may not work always.

→ Both cons & IB, will be executed for every object creation but IB first followed by constructor.

Overloaded Constructors

Within a class we can declare multiple constructors & all these constructors having same name but different arg-types, hence over all these constructor are considered as overloaded constructors.

Hence, overloading concept applicable for constructors.

eg →

```
class Test
{
  Test()
  { sopln("no-arg");
  }
   Test(10)
  { sopln("int-arg");
   }
   Test(10.5)
    { sopln("double-arg)
    }
}
psvm
{
Test t1 = new Test();  → no-arg
Test t2 = new Test(10); → int-arg
Test t3 = new Test(10.5); → double-arg
Test t4 = new Test(10d) → double-arg
}
```

(10) Every no-arg cons. is default cons ✗

(11) Default cons is always no-arg constructor. ✓

(#) Singleton classes

for any java class we are allowed to create only one object such type of class is called singleton class.

eg → Runtime
BusinessDelegate
ServiceLocater.

Advantages -

If several people have same requirement then it is not recommended to create separate object for every requirement.

We have to create only 1 object & we can reuse for every single requirement.
so that performance & memory utilization will be improved.

this is the central idea of singleton classes.