OOPs Concepts in Java

1. Data Hiding
2. Abstraction
3. Encapsulation
4. Tightly encapsulated class
5. Is-A Relationship } -inheritance
6. Has-A Relationship } -inheritance
7. Method Signature
8. Overloading } under polymorphism concepts
9. Overriding } under polymorphism concepts
10. Static Control flow
11. Instance Control flow
12. Constructor → postmasters of any thing
13. Coupling } Design, framework, support loosly coupling
14. Cohesion } Design, framework, support loosly coupling
15. Type-Casting

module-1 → Security

Data Hiding :- Outside person Contact us our internal data directly but our internal data should not go out directly. This oop feature is nothing but data hiding. after validation and authentication out side person can access our internal data. e.g.

Eg- 1. After providing proper user name and password. we can able to access our gmail inbox information.

2. Even though we are valid customer of bank we can able to access our account information but we can't access other's a/c information

Eg-
```
public class Account
{
    private double balance;
    :
    public double getbalance()
    {
        //validation
        return balance;
    }
}
```

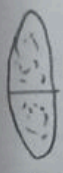
→ By declaring data member (variable only) as private we can achive datahiding.
→ The main advantage of data hiding is Security.
→ Good programmer is recommended modifier for variable is private
→ Note + It is highly recommended to declare data member (variable) as private.

2) Abstraction :
Hide internal implementation and just highlight set of services what we are offering is the concept of abstraction.
→ Through bank ATM GUI screen bank people are highlighting the set of services what they are offering without highlighting internal implementation
→ outside person doesn't aware our internal implimentation so, we can get security.

The main advantages of abstraction are —
1) we can achive security because we are not highlighting our internal implementation.
2) without effecting outside person we can able to perform any type of changes in our internal system. and inhencement will become easy.
3) It improves maintainablity of the application
4) It improves easyness to use our system.
5) By using interface and abstract class we can implement abstraction

3) Encapsulation : The process of binding data and corresponding method into a single unit is nothing but Encapsulation.

Eg - class Student {
data member
+
methods (behaviour)
}



Capsule

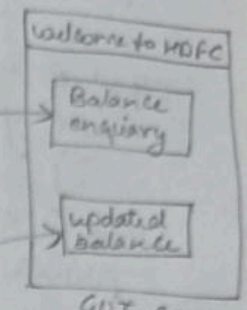
* If any
such
compo
eg - pu

4) T

eg

* If any component follows data hiding and abstraction such type of component is said to be encapsulation component.

```
public class Account
{
    private double balance;
    public double getbalance()
    {
        // Validation
        return balance;
    }
    public void setbalance(dible balance)
    {
        // Validation
        this.balance = balance
    }
}
```



4) Tightly Encapsulated class :-

A class is said to be tightly encapsulated if and only if each and every variable declared as private. weather class contain corresponding getter and setter methods are not. and weather these methods are declared as public or not these things we are not require to check.

Eg:
```
public class account {
private double balance;
public double getBalance()
    {
      return balance;
    }
}
```

Transaction by ATM debit
1. User id and pwd
2. OTP
3. Transaction pwd
4. backside of atm card 4 digit no.

Note :- It increase slowdown execution over all system

Q. Which of the following class are tightly encapsulated

→ class A {
  private int x=10;
  }
  class B extends A {
  int y=20;
  }
  class C extends A {
  private int z=30;
  }

→ class A {
  int x=10;
  }
  class B extends A {
  private int y=20;
  }
  class C extends B {
  private int z=30;
  }

Note :- If the Parent class is not tightly encapsulated then no child class is tightly encapsulated.

5) Is-A Relationship (Inheritance) :-

→ It is also known as inheritence
→ The main advantage is code reusability.
→ By using extends keyword we can implement is-A relationship

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

public static void main(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()

④ C c1 = new P(); → CE: incompatible type
Found: P
required: C

}
}

Conclusions:

② what ever methods parent has by dyault available to the child and hence on child refrence we can call both parent and child class methods.

① what ever methods child has by dyault not available to the parent and hence on the parent refrence we can't call child specific methods.

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

④ Parent refrence can be used to hold child object. But child refrence can't be used to hold parent object.

without inheritence :-

class Vloan {	class Hloan {	class Ploan {
300 methods	300 methods	300 methods
}	}	}
		900 methods 90 hours.

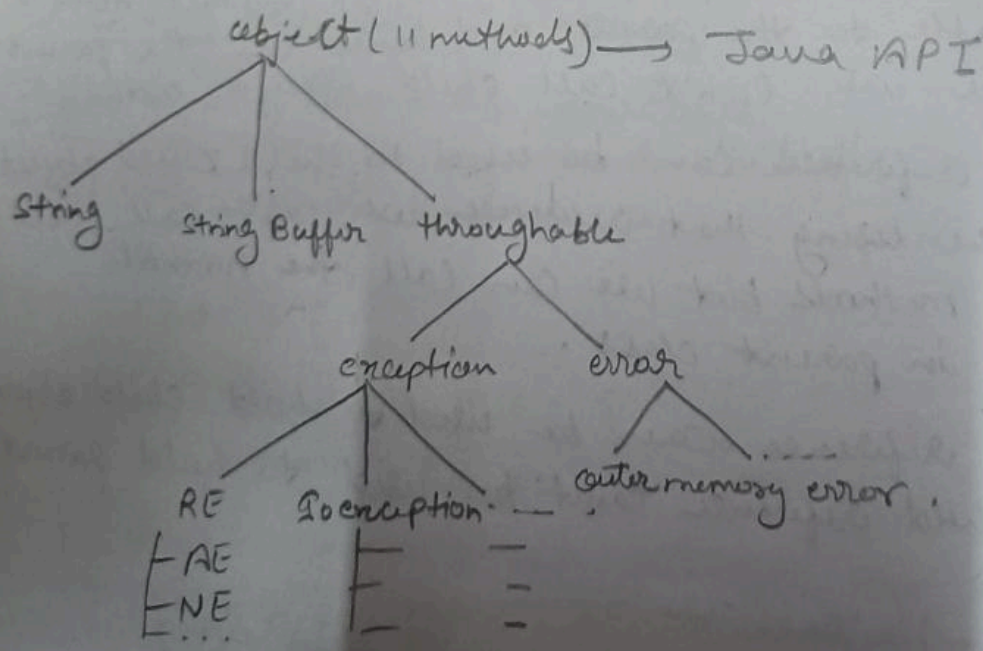
With inheritance :—

```
class Loan {
    250 common methods
}

class Vloan extends Loan
{
  30 specific methods
}
class Hloan extends Loan
{
  50 specific methods
}
class Ploan extends Loan
{
  50 specific methods
}
```

400 methods
40 hr

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 a 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. and hence every class in Java is a child class of object either directly or indirectly so that object class methods by default available to every java class without re-writing due to this object class acts as root for all java class.

→ Throwable class defines most common methods which are required for every exception and error class hence this class acts as root for exception hierarchy.

⑥ multiple inheritance :- A java class can't extend more than one class at a time. Hence java wouldn't provide support for multiple inheritance.

eg :- class A extends B,C { } (CE)

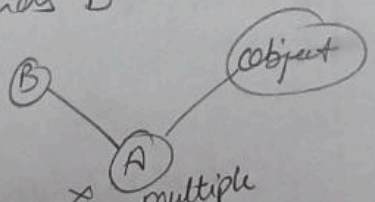
×

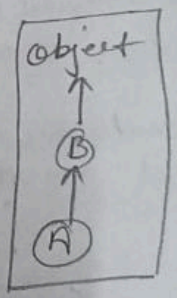
Note :- if our class doesn't extend any other class then only our class is direct child class of object.

* class A { }

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

→ class A extends B { }

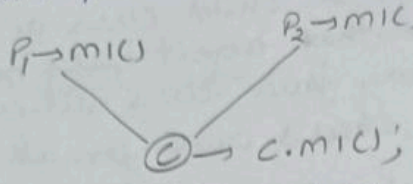




multilevel inheritance

Note :- Either directly or indirectly java wouldn't provide support for inheritance with respect to classes.

Q: why java won't provide support for multiple inheritance
There may be chance of ambiguity problem. Hence Java won't provide support for multiple inheritance.

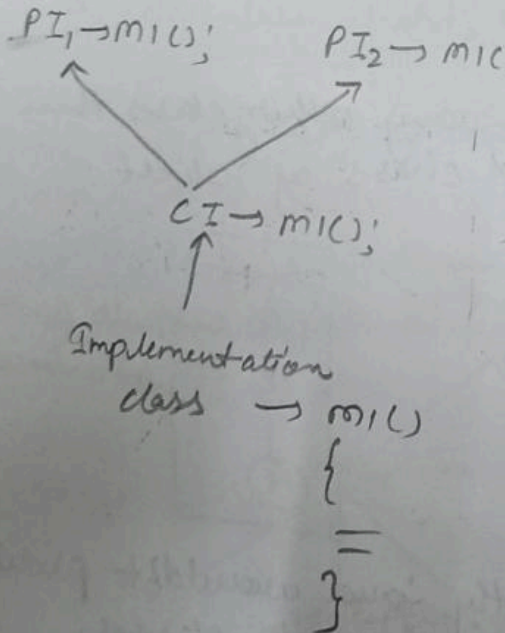


C extends P_1 and P_2 and P_1 Contain m1() & P_2 Contain m1(). one the child object if called m1() may be chance of ambiguity problem.

→ But interface can extend any no. of interfaces simultaneously hence java provide support for multiple inheritance with respect to interfaces.

```
interface A        interface B        interface C extends A,B
{                  {                  {
}                  }                  }
```

Q: why ambiguity problem won't be there in interface?



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

Note :- strictly speaking through we won't get any inheri -tence.

Cyclic inheritence :-

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

CE:- Cyclic inheritence involving A.

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

Note :- Cyclic inheritence is not allowed in java ofcourse its not required.

Has-A Relationship

→ Has-A relationship is also known as composition and aggregation.

→ There is No specific keyword to implement has-A Relation But most of the time we are depending on new keyword.

→ The main advantage of Has-A Relationship is reuse -bility of the code

```
Eg :- class Car
{
    Engine e = new engine();
}
```

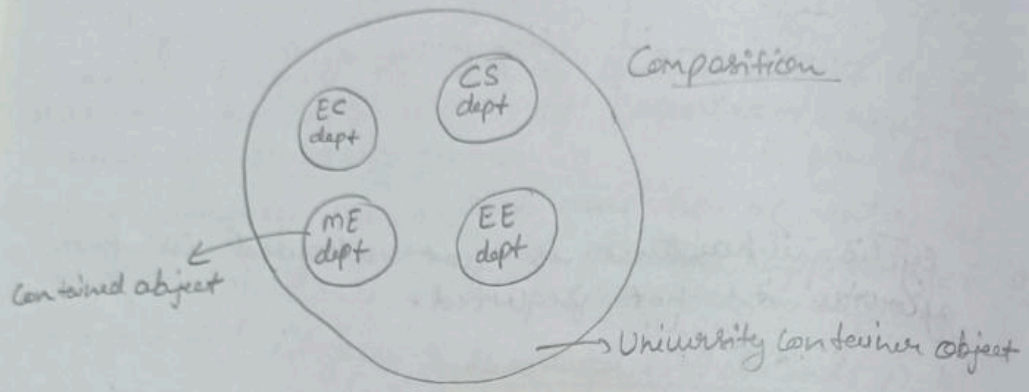
```
class Engine
{
// Engine specific
  functionality
}
```

Car has-A engine reference

* most impotend interview question d/w composition and aggregation —

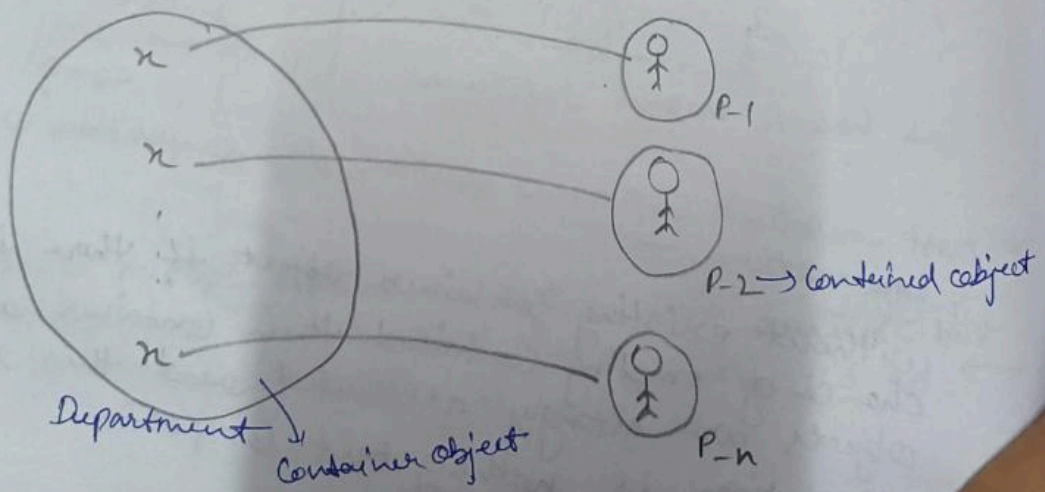
→ without existing container object if there is no chance of existing contained then container and contained objects are strongly associated and this strong association is nothing but composition

1) University consists of several departments without existing university there is no chance of existing department. Hence University and department strongly associated and this strong association is nothing but Composition.



Aggregation :- without existing container object if there is chance of existing contained object then container and contained object are weakly associated and this weak association is nothing but aggregation.

Eg: 1) Department consists of several persons without existing dept. there may be chance of existing prophusors objects. Hence dept. and prophusor object are weakly associated and this weak association is nothing but aggregation.



Note :- 1. In Compo
as in ag
2. In comp
contained
object cl

* If we
matically

eg -

Perso
Clas
Is
Stud
Jav

* we
shoul

T

In ja
follow

Eg

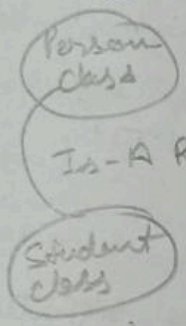
Note :- 1. In Composition object are strongly associated where as in aggregation object are weakly associated.

2. In composition container object holds directly contained object where as in aggregation container object holds just refrences of contained object.

Is-A Vs Has-A

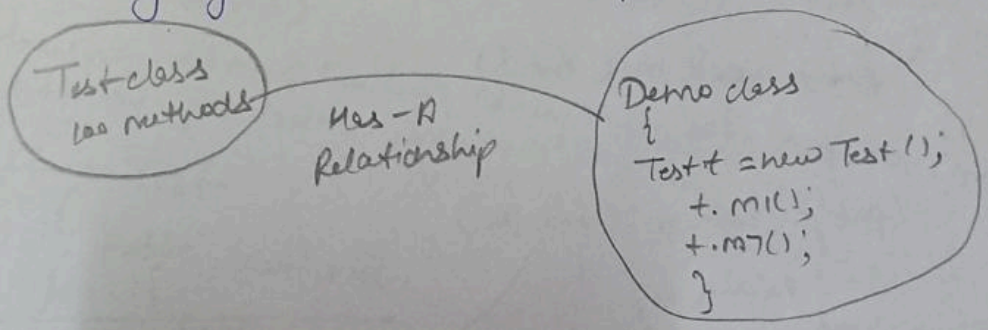
* If we want totaly functionality of a class automaticaly then we should go for Is-A relationship.

eg-



Complete functionality eg person class is required for student class.

* we want part of the functionality then we should go for Has-A relationship.



(7) Method - Signature

In java method signature consists of method name followed by argument types.

Eg- public static int m1(int i, float f)

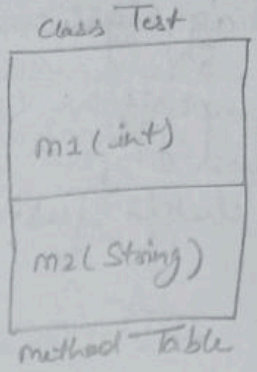
↓

m1(int, float)

Note :- Return type is not part of method signature in Java.

* Compiler will use method signature to resolve method calls.

```
class Test
{
    public void m1(int i)
    {
    }
    public void m2(String s)
    {
    }
}
```




```
Test t = new Test();
t.m1(10);
t.m2("Ram");
t.m3(10.5);
```

CE: Cannot found Symbol
Symbol : method m3(double)
location : class Test.

* with in a class two methods with same signature not allow.

```
class Test
{
    public void m1(int i)
    {
    }
    public int m1(int n)
    {
        return 10;
    }
}
```

```
Test t = new Test();
t.m1(10);
```

CE = m1(int) is already defined in Test.

8) Overloading — Two methods are said to be overloaded if and only if both methods having same name but different argument types is the concept of method overloading.

→ In C language methods overloading concept is not available hence we can't declare multiple methods with same name but different argument types if there change in argument complexity we should go for new method name with increase complexity of programming —

```
abs(int i) → abs(10);
labs(long l) → labs(10l);
fabs(float f) → fabs(10.5f);
```

→ But in java we can declare multiple methods with same name but different argument types such type of methods are called overloaded methods.

```
abs(int i)
abs(long l)      } overloaded methods
abs(float f)
```

Having overloading concept in java reduces complexity by programming.

Eg:

```
class Test
{
    public void m1()
    {
        Sopln(no-arg);
    }
    public void m1(int i)
    {
        Sopln(int-arg);
    }
    public void m1(double d)
    {
        Sopln(double args);
    }
}
```

(overloading method)

```
P S V main(String[] args)
{
    Test t = new Test();
    * m1(); no.args
    * m1(10); int args
    * m1(10.5); double args
}
}
```

* In overloading method resolution always taken care by compiler based on reference type hence overloading is also considered as compile time polymorphism and static

Polymorphism and early binding.

Case 1: Automatic promotion in overloading —
while resolving overloaded methods if exact matched method is available then won't get any compile time error immideatly first it will promote argument to the next level and check weather matched method is available or not.

if matched method is available then it will be considered and if matched method is not available then compiler promotes argument once again to the next level. this process will be continued untill all possible promotion still if the matched is not available then we will get compile time error.

The following are all possible promotion in over loading.

byte → short → int → long → float → double
char → int ↗

This process is called automatic promotion in overloading.

```
class Test
{
    public void m1(int i)
    {
        Sopln("int-arg");
    }
    public void m1(float f)
    {
        Sopln("float-arg");
    }
}
```

overloaded method

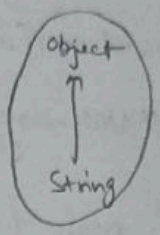
```
P s v main(String[] args)
{
    Test t = new Test();
    t.m1(10);   → int-arg
    t.m1(10.5f); float-arg
    t.m1('a');  int-arg
    t.m1(10l);  float-arg
    t.m1(10.5); → CE:
}
}
```

Can't find symbol
symbol: method m1(double)
location: class Test.

Case 2:

```
class Test
{
    public void m1(String s)
    {
        Sopln("String version");
    }
    public void m1(Object o)
    {
        Sopln("object version");
    }
}
```

(overloaded method)




```
P S V main(String[] args)
{
    Test t = new Test();
    t.m1(new Object());  → object version
    t.m1("durga");  → String version
    t.m1("null");  → String version
}
}
```

→ Collector example for permission

Note: while resolving overloaded methods compiler will always gives a prisidence for child ~~class~~ type argument then compared with parent type argument.

Case 3:

```
class Test
{
    public void m1(String s)
    {
        Sopln("String version");
    }
    public void m1(StringBuffer sb)
    {
        Sopln("StringBuffer version");
    }
}
```

(overloaded method)

```
P S V main(String[] args)
{
    Test t = new Test();
    t.m1("durga");  String version
    t.m1(new StringBuffer("durga"));
      StringBuffer version
    t.m1(null);
```

↓

CE: reference to m1() is ambiguous.

Case 4:

```
class Test
{
    public void m1(int i, float f)
    {
        Sopln("int-float version");
    }
    public void m1(float f, int i)
    {
        Sopln("float-int version");
    }
}
```

```
P S V m(String[] args)
{
    Test t = new Test();
    t.m1(10,10.5f);  → int float version
    t.m1(10.5f,10);  float-int
    t.m1(10,10);  → CE: Reference to m1() is ambiguous.
    t.m1(10.5f,10.5f);
}
}
```

CE: Cannot find symbol
Symbol: method m1(float, float)
location: class Test

Case 5:

```
class Test
{
  public void m1(int x)
  {
    Sopln("General method");
  }
  public void m1(int... x)
  {
    Sopln("var-arg method");
  }
  p s v main(String[] args)
  {
    Test t = new Test();
    t.m1();      var-arg method
    t.m1(10,20); var-arg method
    t.m1(10);    General method
  }
}
```

switch(x)
{
case 1:
case 2:
.
.
default:
}

→ in general var-arg method got least priority that is if no other method matched then only var-arg method will get a chance it is exactly same as default case inside switch.

Case 6:

```
class Animal
{
}
class monkey extends Animal
{
}
class Test
{
  public void m1(Animal a)
  {
    Sopln("Animal version");
  }
  public void m1(monkey m)
  {
    Sopln("monkey version");
  }
```

```
P
{
    public static void main(String[] args)
    {
        Test t = new Test();
        Animal a = new Animal();
        t.m1(a);        Animal version.
        monkey m = new monkey();
        t.m1(m);        monkey version.
        Animal a1 = new monkey();
        t.m1(a1);       Animal version
    }
}
```

Note :- In overloading method resolution always takes care by compiler based on reference type. so in overloading run time object won't play any role.

(9) Overriding method :- what ever method parent has by default available to the child through inheritance if child class not satisfied with parent class implementation then child is allow to redefine that method based on its requirment. this process is called overriding.

The parent class method which is overridden is called Overridden method and child class method which is overriding is called Overriding method.

Eg :
```
class P
{
    public void property()
    {
        Sopln("Cash + land + Gold");
    }
    public void marry()
    {
        Sopln("Subba Lakshmi");
    }
}
class C extends P {
    public void marry()
    {
        Sopln("Isha/Tara");
    }
}
```

Overridden method

Overriding

Overriding method

```
class Test {
  P s v main(String[] args)
  {
    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 / late binding polymorphism.

→ Rules for overriding :-

In overriding method names and argument types must be matched that is method signature must be same.

→ In overriding return type must be same but this rule applicable until 1.4 version only from 1.5 onwards co-variant return type also allowed.

Eg-

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

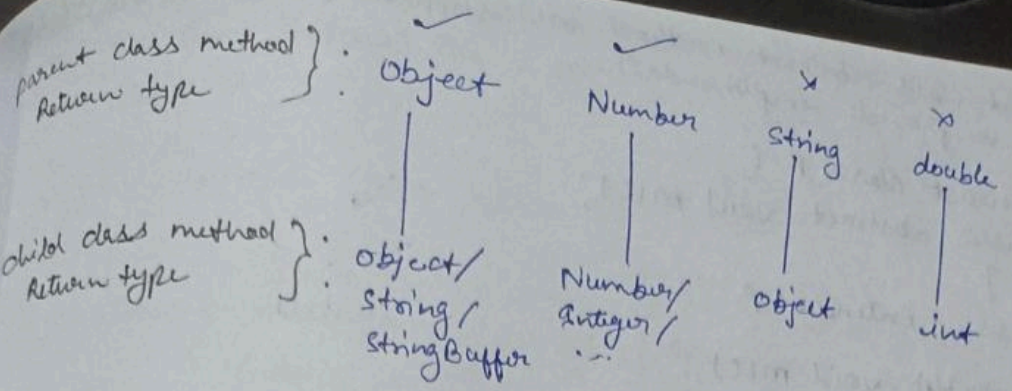
1.4 version

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

1.5 version

→ According to this child class method return type need not be same as parent method return type. Its child type also allowed.

→ If it is invalid in 1.4 version but from 1.5 version onwards it is valid.



Note: Co-variant return type concept applicable only for object type but not for primitive types.

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

→ Based on our requirement exactly same private method in child class it is valid but not overring.

It is valid but not overriding

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

* → We can't override parent class final method in child classes if we are trying to override we will get compile time error.

```
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.

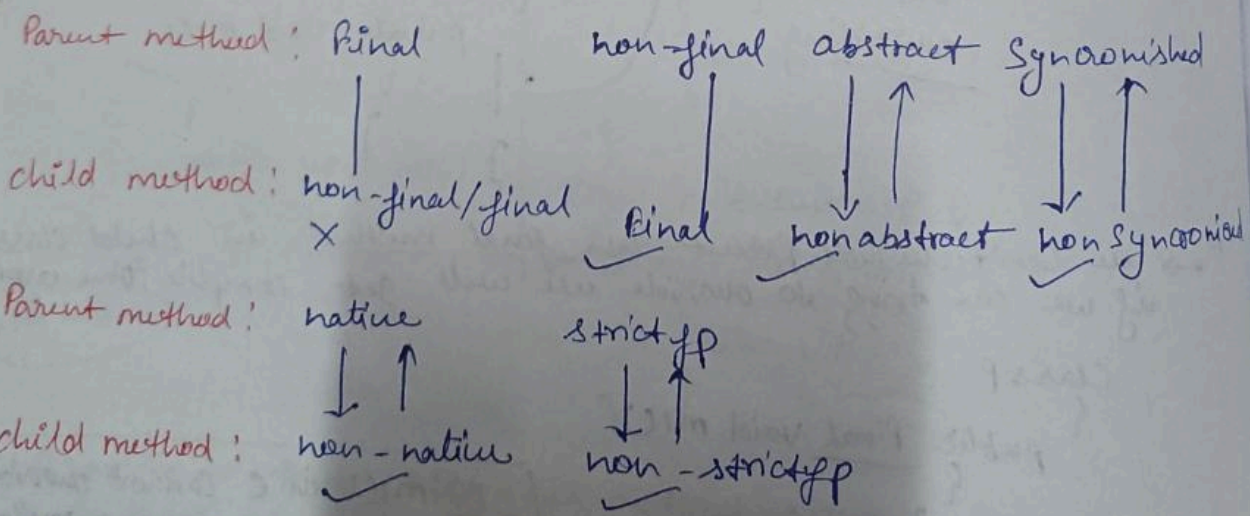
→ Parent class abstract method we should override in child class to provide implimentation

Eg-
```
abstract class P {
  public abstract void m1();
}
class C extends P
{
  public void m1()
  {
  }
}
```

→ we can override non-abstract method as abstract

eg-
```
class P {
    public void m1()
    {
    }
}
abstract class C extends P {
    public abstract void m1();
}
```

The main advantage of this approch is we can stop the availability of parent method implementation to the next level child classes.



In overriding the following modifiers won't keep any restriction
Synchronised, native, strictfp.

→ while overriding we can't reduce scope of access modifier but we can increase the scope.

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

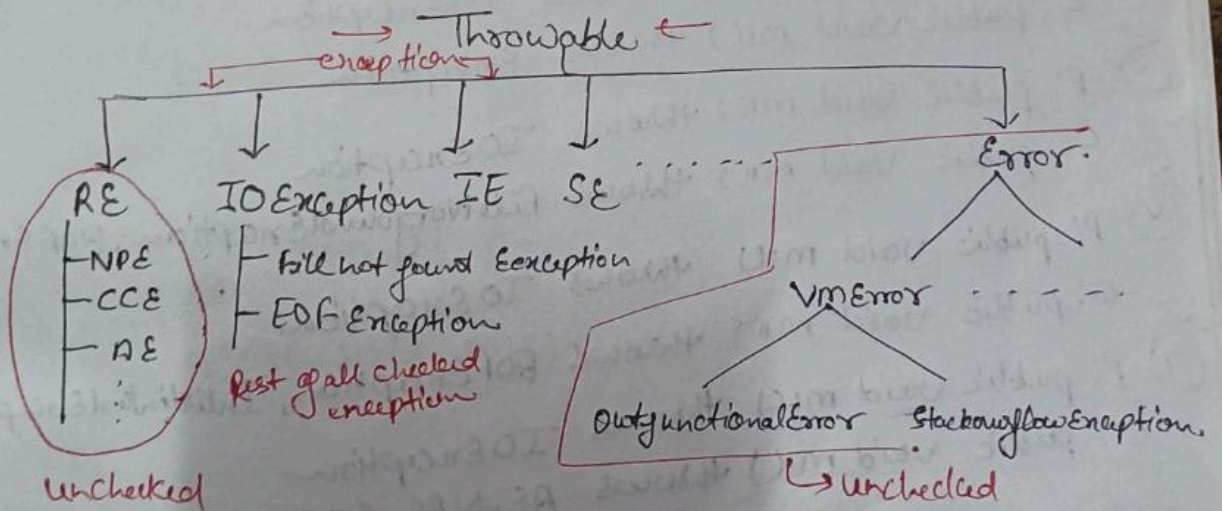
CE: m1() in C cannot override m1() in P, attempting to assign weaker access privileges, was public.

public [private < default < protected < public]

parent class method : public | protected | default | private

child class method : public | protected / public | default / protected / public

Overriding method ~~not~~ concept not applicable for private method.



Rule - If child class method throws any checked exception compulsory parent class method should throw the same exception or its parent exception. otherwise we will get compile time error but there are no restriction for unchecked exceptions.

① class P {
public void m1() throws IOException
{ }
}
class C extends P {
public void m1() throws EOFException, InterruptedException
{ }
}

CE: m1() in C cannot override m1() in P, overridden method does not throw java.lang.InterruptedException

① P: public void m1() throws Exception
C: public void m1()

② P: public void m1()
C: public void m1() throws Exception

③ P: public void m1() throws Exception
C: public void m1() throws IOException

④ P: public void m1() throws IOException
C: public void m1() throws Exception

⑤ P: public void m1() throws IOException
C: public void m1() throws FileNotFoundException, EOFException

⑥ P: public void m1() throws IOException
C: public void m1() throws EOFException, InterruptedException

⑦ P: public void m1() throws IOException
C: public void m1() throws AE, NPE, CCE

Eg- class P {

```
    public void m1() throws IOException.
        {
        }
    }
class C extends P
  {
  public void m1() throws Exception ✗ not parent exception
                          IOException ✓ same exception or
                          EOFException ✓ child exception
P p = new C();
try {
    p.m1();
    }
catch (IOException e)
    {
    }
```

(our overriding should not affect outside person)

Overriding with respect to static method :-

i) we can't override a static method as non-static otherwise we will get compile time error.

eg: class P

```
    {public static void m1()
      {
      }
Class C extends P
  {
    public void m1()
      {
      }
  }
```

CE: m1() is C can not override m1() in P; overridden method is static

→ class P {
public void m1()
{
}
}
class C extends P
{
public static void m1()
{
}
}

CE: m1() in C can not override m1() in P; overriding method is static.

→ If both are static method (Parent, child) we won't get compile time error, its seems overriding concept applicable for static method only. But it is not overriding it is method hiding.

Eg- class P {
public static void m1()
{
}
}
class C extends P {
public static void m1()
{
}
}

It is method hiding but not overriding.

Method Hiding :-

class P
{
public static void m1()
{
System.out.println("parent");
}
}
class C extends P
{
public static void m1()
{ System.out.println("child"); }
}

method hiding but not overriding.

```
class Test {
    public static void main(String[] args)
    {
        P p = new P();
        p.m1();

        C c = new C();
        c.m1();
        P p1 = new P();
        p1.m1();
```

All rules of method hiding are exactly same as over-riding except the following differences.

method hiding	overriding
① Both static (P,C method)	① Non-static (P,C method)
② Compiler reference (method resolution)	② JVM based on runtime object
③ C.T. Polymorphis, static, early binding	③ R.T, Dynamic, late binding.
① Both parent and child class method should be static.	① Both parent and child class method should be non-static.
② Compiler is responsible for method resolution based on reference type.	② JVM is always responsible for method resolution based on runtime object.
③ It is also known as Compile time polymorphis, static or early binding.	③ It is also known as run time polymorphis, Dynamic, late binding.

→ If both parent and child class methods are non-static then it will become overriding, in this case output is - Parent
child
child

→ in overriding old copy will replace with new copy and old copy gone. But in hiding old copy just hide with new copy. (it show both copy).

Overriding with Var-arg methods :-

→ we can override var-arg method with another var-arg method only if we are trying to override with normal method then it will become overloading but not overriding -

```
class P
{
  public void m1(int... x)
  {
    Sopln("Parent");
  }
}
class C extends P
{
  public void m1(int x) / (int... x)
  {
    Sopln("child");
  }
}
class Test
{
  P s v main(String[] args)
  {
    P p = new P();
    p.m1(); → Parent
    C c = new C();
    c.m1(); → child.
    P p1 = new C();
    p1.m1(10); → ~~child~~ Parent / → ~~Parent~~ child
```

It is overloading but not overriding

Overriding with Respect to Variable :-

Variable resolution always take care by Compiler based on reference type, irrespective of weather variable is static or non-static. (Overriding concept applicable for methods but not for variables).

```
eg - class P {
        int x = 888;
     }
     class C extends P
     {
       int x = 999;
     }
     class Test {
       P s v main ( String[] args)
       {
         P p = new P();
         Sopln (P.x); → 888
         C c = new C();
         Sopln (C.x); → 999
         P p1 = new C();
         Sopln (p1.x); → 888
       }
     }
```

P → non-static C → non-static	P → static C → non-static	P → non-static C → static	P → static C → static
888	888	888	888
999	999	999	999
888	888	888	888

Difference b/w overloading and overriding —

Properties	Overloading	Overriding
(1) method Names	must be same →	must be same
(2) Argument Name	must be different [at least order] →	must be same [including order]
(3) method Signature	must be different →	must be same
(4) Return Types	No Restriction →	must be same Ubill (1.4 version) from 1.5 version Co-variant return type allowed.
(5) Private, static, final method	Can be overloaded →	Can't be overridden
(6) Access modifiers	No Restriction →	The scope of Access modifier can not be reduced but we can increase.
(7) throws Clause	No Restriction →	if child class method throws any checked exception Compulsory Parent class method should throw the same checked exception or it's parent But no restriction for unchecked.
(8) method Resolution	Always takes care by Compiler based on reference type →	Always take care by JVM based on runtime object.
(9) It is also known as	C.T.P / early binding / S.P	R.T.P / D.P / late binding.

Note :- In overloading we have to check only method names (must be same) and argument types (must be different) we are not require to check Remaining like return type access modifiers etc,

But in overriding everything we have to checked like method name, arg. type, return type, access modifier, throws keyword.

throws Cl

Public

Consider th

In the cl take -

(1) public

(2) publi

(3) publ

(4) publi

(5) publ

Polyme

Eg - m di

a

d

a

Eg -

Overriding

throws class etc.

Consider the following method in parent class. → Public void m1(int z) throws IOException → Parent
In the child class which of the following method we can take -

(1) public void m1(int i) → overridden ✓
(2) public static int m1(long l) → overloading ✓
(3) public static void m1(int i) → overriding ✗
(4) public void m1(int i) throws Exception → overriding ✗
(5) public static abstract void (double d); → CE ✗
↳ illegal combination of modifiers.

Polymorphism :- One name but multiple form is the concept of polymorphism.

Eg - method name is the same but we can apply different type of argument this concept is overloading.

abs(int)
abs(long)
abs(float)

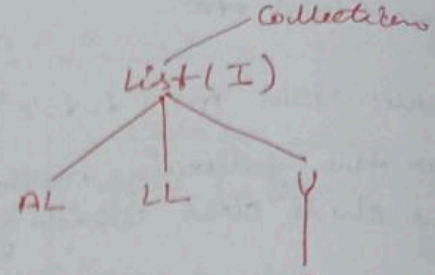
Eg - method signature but in parent class one type of implimentation and then the child class another type of implimentation (overriding).

```
Class P {
    marry()
    {
      Soply("Sudha Ammi");
    }
}
Class C extends P {
    marry()
    { Soply("Isha/Gtara/Ami");
    }
}
```

overriding

Eg.

```
List l = new AL();
         new LL();
         new Stack();
         new Vector();
```

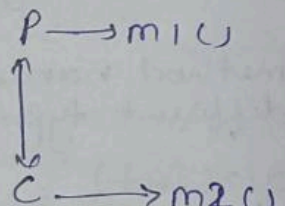


Usage of parent reference to hold child object is the concept of polymorphism.

* Parent class reference can be used to hold child class object. But by using that reference we can call only the method available in parent class. and we can't call child specific method.

```
P p = new C();
p.m1();
p.m2();  ×
```

↳ CE: cannot find symbol
symbol: method m2()
location: class P.



* But By using child reference we can call both parent & child class method.

```
C c = new C();
c.m1();
c.m2();
```

→ when we should go for parent reference to hold child object :-

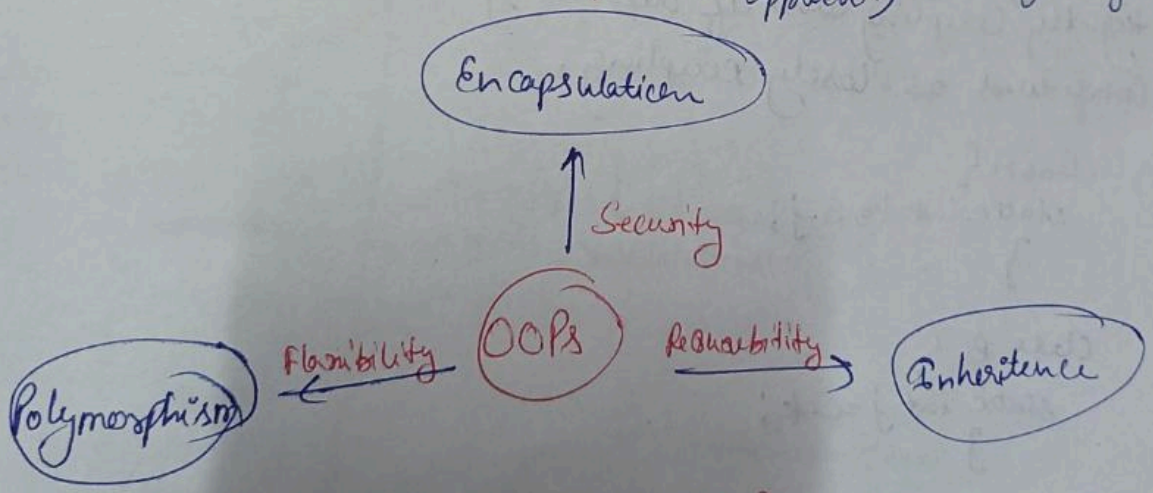
if we don't know a exact runtime type of object then we should go for parent reference
For. Eg. the first element present in array list can be any type it may be student object, customer, string object, stringbuffer

Hence the return type of get method is object which can hold any object. ~~object~~

object o = d.get(0)

#

C c = new C();	P p = new C();
Eg. AL d = new AL();	Eg. List d = new AL();
① we can use this approch if we know exact runtime type of object	① we can use this approch if we don't know runtime type of object.
② By using child reference we can call both parent class & child class methods. this is the advantage of this approch.	② By using ~~this~~ Parent reference we can call only methods available in parent class. and we can't call child specific methods. (this is the disadvantage of this approch).
③ we can use child reference to hold only particular child class object (this is the disadvantage of this approch).	③ we can use parent reference to hold any child class object this is the advantage of this approch.)



3 Pillars of OOPs.

Polymorphism

- Static polymorphism
  or
  Compile time polymorphism
  or
  Early binding
  - overloading
  - method hiding
- Dynamic polymorphism
  or
  Runtime Polymorphism
  or
  Late binding
  - Overriding.

Beautyfull defination of Polymorphism :-

A boy starts LOVE with the word FRIENDSHIP, but Girl ends LOVE with the same word friendship. Word is the same but attitude is diffrent. This beautiful concept of OOPS is nothing but polymorphism....

Coupling

The degree of dependence b/w the Component is called coupling. If dependency is more then it is considered as tightly coupling and if dependency is less then it is considered as 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; } }
```

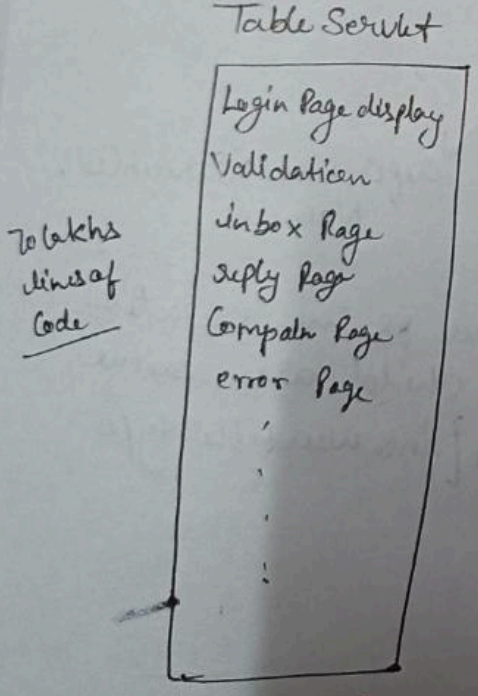
→ The above components are said to be tightly coupled with each other because dependency b/w the components is more. Tightly coupling is not a good programming practise bcz it has several serious disadvantages.

1. without effecting remaining components we can't modify any component and hence inhancement will dificult.
2. It suppresses reusebility.
3. It reduces maintainbility of the application.

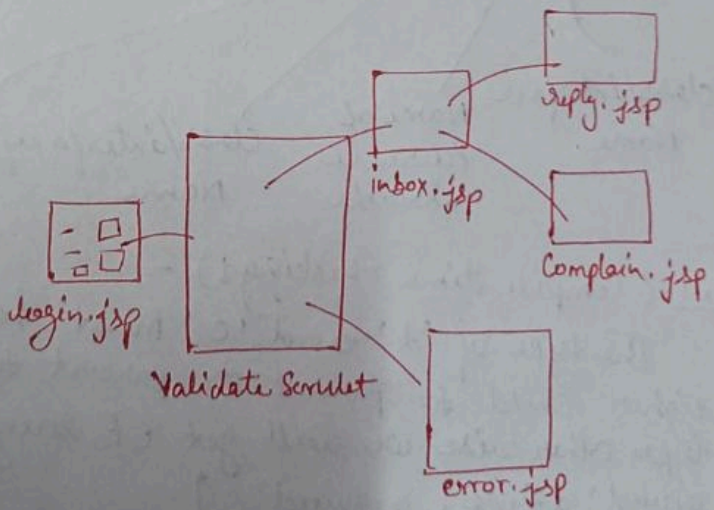
Hence we have to maintain dependency b/w the components as less as possible that is loosely coupling is good programming practice.

Cohesion

For every component a clear, well defined functionality is required/defined then that component is said to be follow high cohesion.



Low Cohesion



High-cohesion

High-Cohesion is always a good programming practise bcz it has the several advantages.

1. without effecting remaining component we can modify any component hence, enhancement will become easy.
2. It promotes reusebility of the code (wherever validation is required we can reuse the same validate method without re-writing.)
3. It improves maintainbility of the application.

Note - Loosly Coupling and high Cohesion are good programming practises.

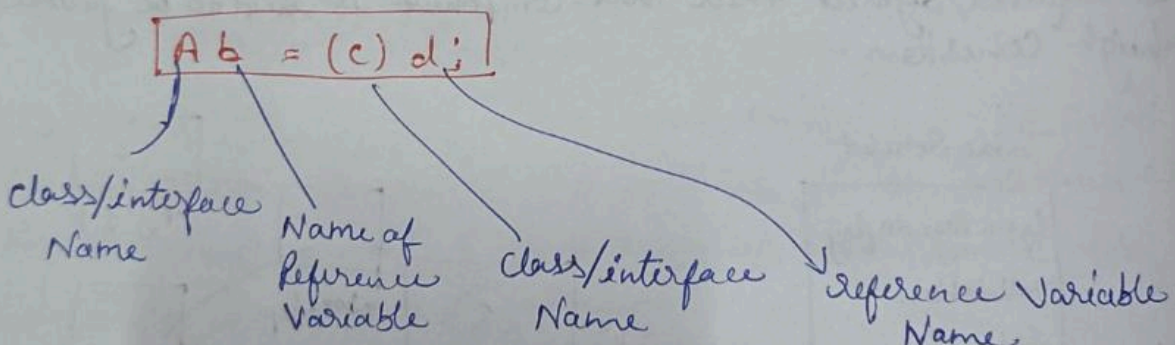
Object Type-Casting -

we can use parent reference to hold child object.

Eg. Object o = new String ("durga");

we can use interface reference to hold implemented class object.

Eg. Runable r = new Thread ();



Rule 1 (Compile time checking 1) -

The type of 'd' and 'c' must have same relation either child to parent or parent to child are same type other wise we will get CE saing [inconvertable type found: d type, required: C]

Eg. Object o = new String ("durga");
StringBuffer sb = (StringBuffer) o;

Eg- String s = new String ("durga");
StringBuffer sb = (StringBuffer) s;
↳ CE: inconvertable types
found: java.lang.String
required: java.lang.StringBuffer

Rule 2: (Compile time checking 2):-
'C' must be eigther same or derived type of A
Otherwise we will get Compile time error saying
incompatible type
found: C
required: A

Eg1. Object O = new String ("durga");
StringBuffer sb = (StringBuffer) O; } Valid.

Eg2. Object O = new String ("durga");
StringBuffer sb = (String) O;
CE: incompatible types
found: J.l.String.
required: J.l.S.B.

Rule 3: (Runtime checking)- Runtime object type of 'd' must be eigther same or derived type of C Otherwise we will get runtime exception saying classCastException.

Eg1. Object O = new String ("durga");
StringBuffer sb = (StringBuffer) O;

RE: ClassCastException: j.l.String cannot be cast to j.l.StringBuffer.

* Eg2. object O = new String("durga");
object O1 = (String)O;
} Valid.

(1)
(2)
(3)
(4)