OPPS Concepts

① Data hiding
② Abstraction
③ Encapsulation
④ Tightly encapsulated class

} Module 1 (security)

⑤ IS-A Relationship (Inheritance)
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.

① Data hiding :- Out side person can't access our internal data directly (or) our internal data should not go out directly these OOP feature is a nothing but "data hiding".

after validation (or) authentication outside person can access our internal data.

Example :- ① after providing proper username & password we can able to access our gmail inbox information

Example ② : Even though we are valid customer of the bank we can able to access our account information and we can't access other's account information.

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

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

⇒ The main advantage of data hiding is "Security".
it is highly recomanded to declare data member(variable) as private.

Abstraction:-

Hiding internal implimentation and just hiblet the set of services what we are offering, is the the Concept of "Abstraction".

⇒ threw Bank ATm GUI screen bank people are weigulating the set of services what they are offering with out heiluting internal implementation

Main Advantages of Abstraction are

* ① Security
② Enchancement easily
③ Improve Easyness
④ Maintanability

⇒ we Can achieve security because we arenot heilating our internal implementation.

⇒ with out effecting outside person we can able to perform any type of changes in our internal system. and hence enchancement will become easy

⇒ it improves maintainability of the Application.
⇒ it improves easyness to use our system.

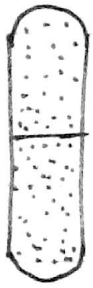
By using interfaces and Abstract Classes we can implement abstraction

Encapsulation :-

The process of binding data and Corresponding methods into a sigle unit is nothing but encapsulation.

Example :-

```
Class student
{
    data members
      +
    methods ( behaviour)
}
```



Capsule

Encapsulation = Data hiding + Abstraction

If any component follows data hiding and Abstraction Such type of component is said to be encapsulated Component.

Example

```
public class Account
{
    private double balance;
public double get balance()
  {
    //validation
    Return balance;
  }
public void set balance ( double balance)
  {
    // validation
      this. balance = balance;
  }
  :
}
```

⇒ The main advantage of encapsulation are we can achieve security.
Enchancement will become easy
it improves maintanability of the application

⇒ The main advantage of encapsulation is we can achieve security but the main disadvantage of encapsulation is it increases length of the code and slows down Execution.

Tightly Encapsulated Class :-

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

Ex:-

```
public class Account
{
private double balance;
    public double getbalance()
    {
     Return balance;
     }

class
```

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

Tightly encapsulated ✓

not ✗

Tightly ✓

which of the following classes are tightly encapsulated.

Ex:-

```
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 child classes also be not tightly encapsulated

(8) IS-A Relationship :- (Inheritance Concept)

→ IS-a Relationship also known as "Inheritance".

→ The main advantage of IS-A Relationship is Code Reusability.

→ By using Extends keyword we can implement IS-A Relationship.

Ex:

```
class Test
{
① P c v main (String[] args)
  {
    P P=new P(); ✓
    P.m1(); x
② C C=C=new C(); ✓
  C.m1(); ✓
  C.m2(); ✓
③ P P1=new C(); ✓
  P1.m1(); ✓
  P1.m2(); x
④ C C1=new P(); x
```

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

CE: incompatible type
found : P
required: C

Before this Example

↓ Stop

```
class P
    public void m1()
    {
     s o.p/n ("Parant")
    }
}
 class C extends P
{
    public void m2()
    {
       s.op/n ("child").
    }
    }
```

Conclusion:1

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

2) whatever methods child has by default not available to the parent and hense on the parent reference we can't call child specific methods

3) parent Reference can be used to hold child object but using that, Reference we can't call child speicific methods but we can call the methods present in parant class.

4) Parent Reference can be used to hold child object but child reference can not be used to hold parent object

with out inheritance

```
class vLoan
{ 300 method
}

class HLoan
{ 100 methods
}

class pLoan
{ 100 methods
}
```

with inheritance

```
class Loan
{
 250 methods
}

① class vLoan extents Loan
  {
    50 speitic methods
  }

② class HLoan extends Loan
   {50 speitic methods
   }

③ class pLoan extends Loan
  {
  50 speitic methods
  }
```

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

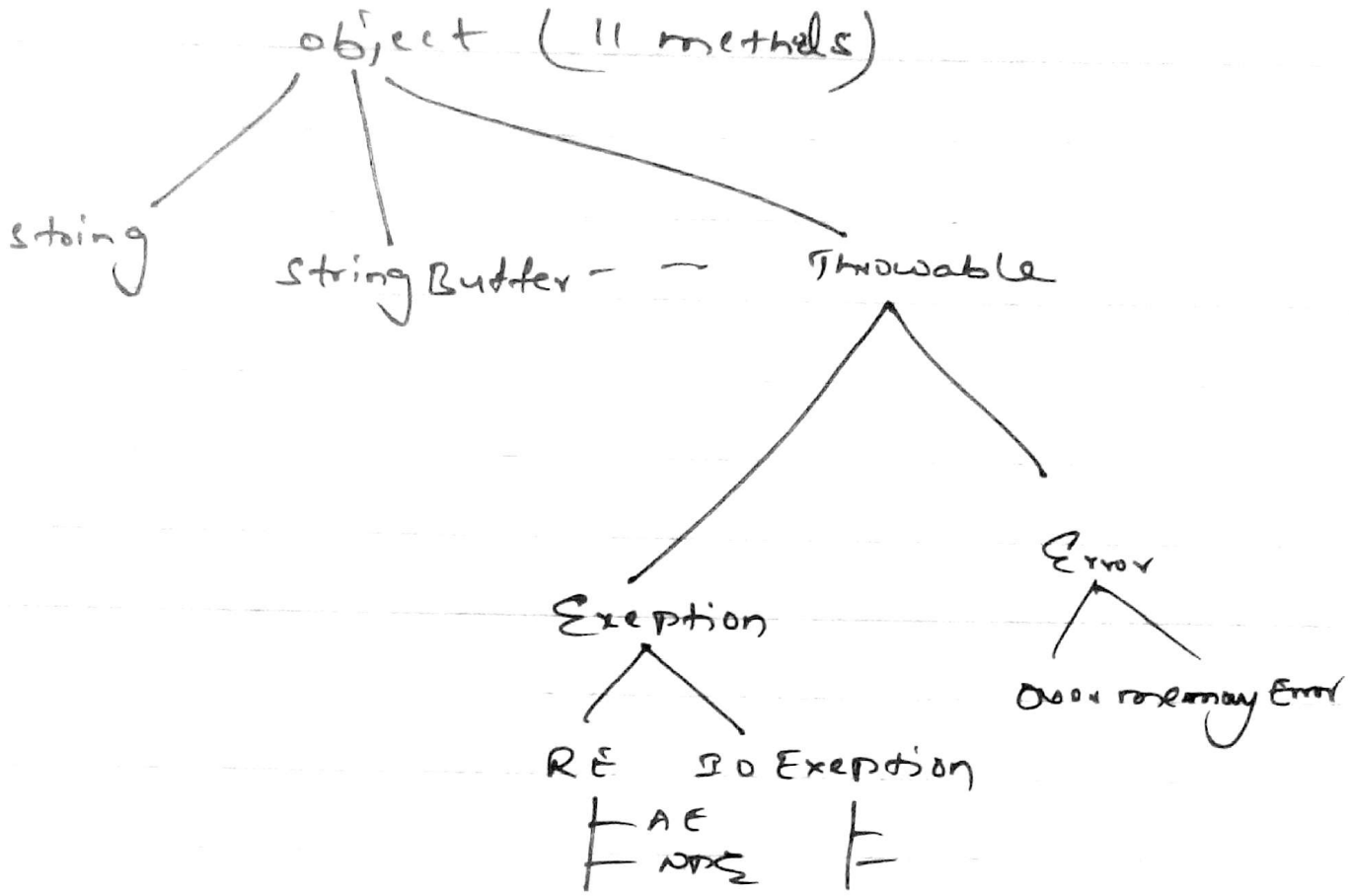
The specific methods which are applicable for ~~particular any type of child, we declare~~ a particular child we have to define in child class.

=) Total Java API is implemented based on inheritance concept

→ The most common methods which are applicable for for any Java object (or) defined in object class and hence every class in Java is the child class of object either directly (or) indirectly. So that object class methods by default available to the every Java class with out rewriting due to this object class acts as root for all Java classes.

→ Throwable class defines the most commons methods which are required for every exception and error classes. hence this class acts as root for Java Exception heirarcy.

∴ Look at the diagram for clear understand in next page.



Multiple inheritance :-

A Java class can't extend more than one class at time. hence Java can't provide support for multiple inheritance in classes.

Ex :- class A Extends B, C

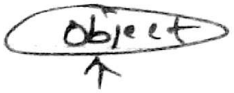
{

}

✗ CE

NOTE:- ① if our class doesn't extend any other class then only our class is direct child class of object.

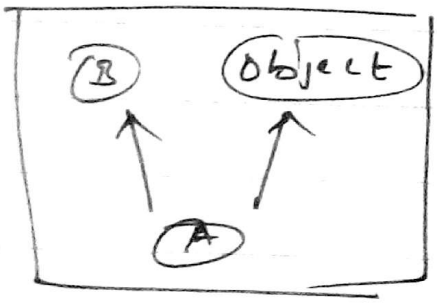
Ex:- Class A
{
}



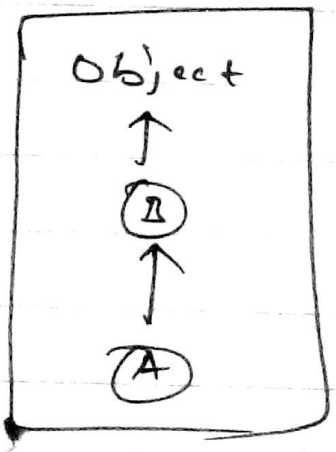
A is the child of object.

② if our class extends any other class then our class is indirect child class of object.

Ex:- class A extends B



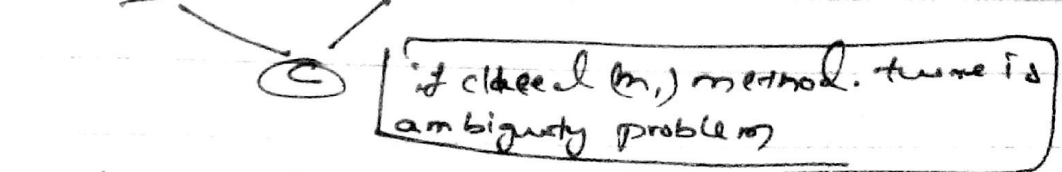
multiple inheritance



multilevel inheritance.

either directly (or) indirectly Java won't provide support for inheritance with respect to classes.

Why Java won't provide support for multiple inheritance?-



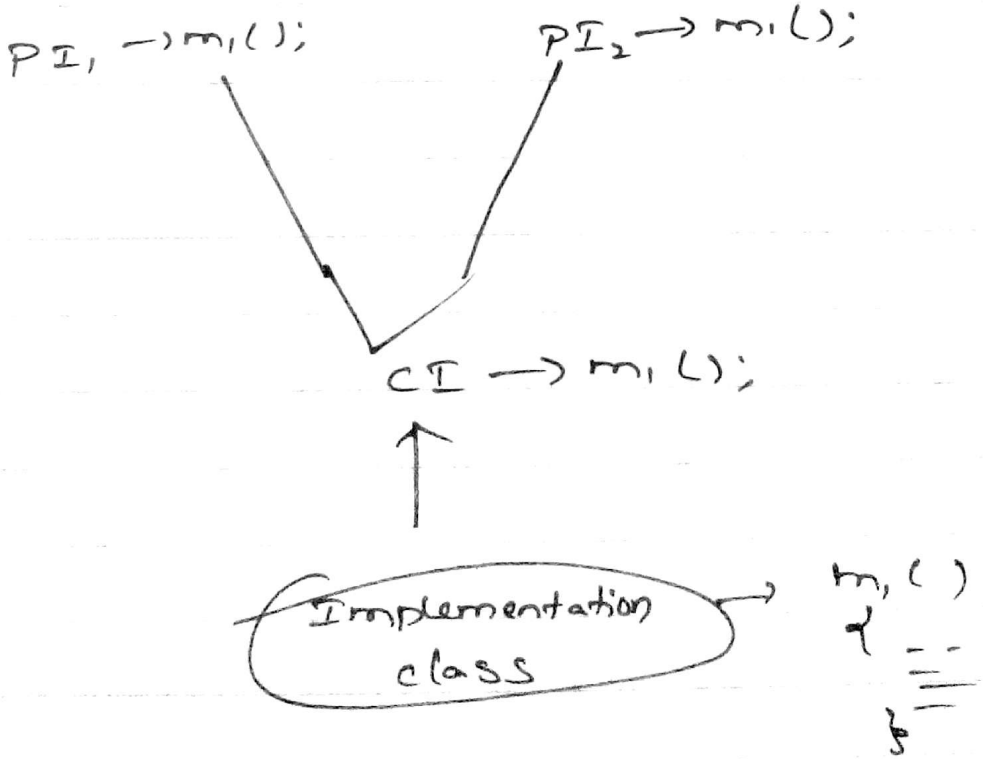
There may be a chance of ambiguity problem. hence Java won't provide support for multiple inheritance

But interface can extend any number of interfaces simultaniously. hence. Java provide support for multiple inheritance with respect to interfaces.

```
interface A          |  interface B
{                    |  {
}                    |  }
```

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

why ambiguity problem won't be there in interfaces :-



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

NOTE: Strictly speaking interfaces we can't get any inheritance.

Cyclic inheritance:- cyclic inheritance is not allowed in java. Ofcourse it is not required

Example:

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

CE:-

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

CE:- Cyclic inheritance involving a

HAS-A Relationship:-

① HAS-A Relationship it also known as composition (or) Aggregation.

② There is no specific keyword to implement has-a relation. but most of the times we are depending on "New" keyword

③ The main advantage of has a relationship is Reusability of the code.

Example:-

```
class Car
{
Engine e = new Engine()
}
```

Car has a Engine reference

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

* Difference b/w Composition & Aggregation:-

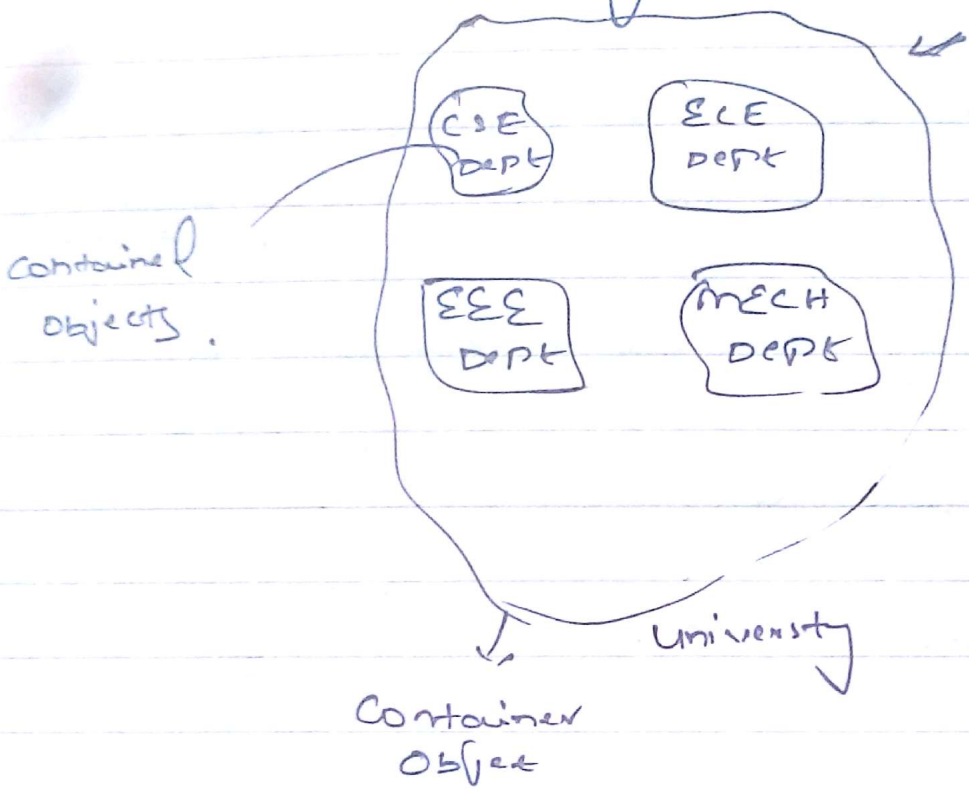
=> without existing Container object if there is no chance of existing Contained objects. Then Container and contained objects are Stongly associated and this strong association is nothing but composition.

Example

University Consist of Several departments with out existing University there is no chance of existing department. hence University and

department strongly associated. and this strong association it nothing but composition

Composition Example.



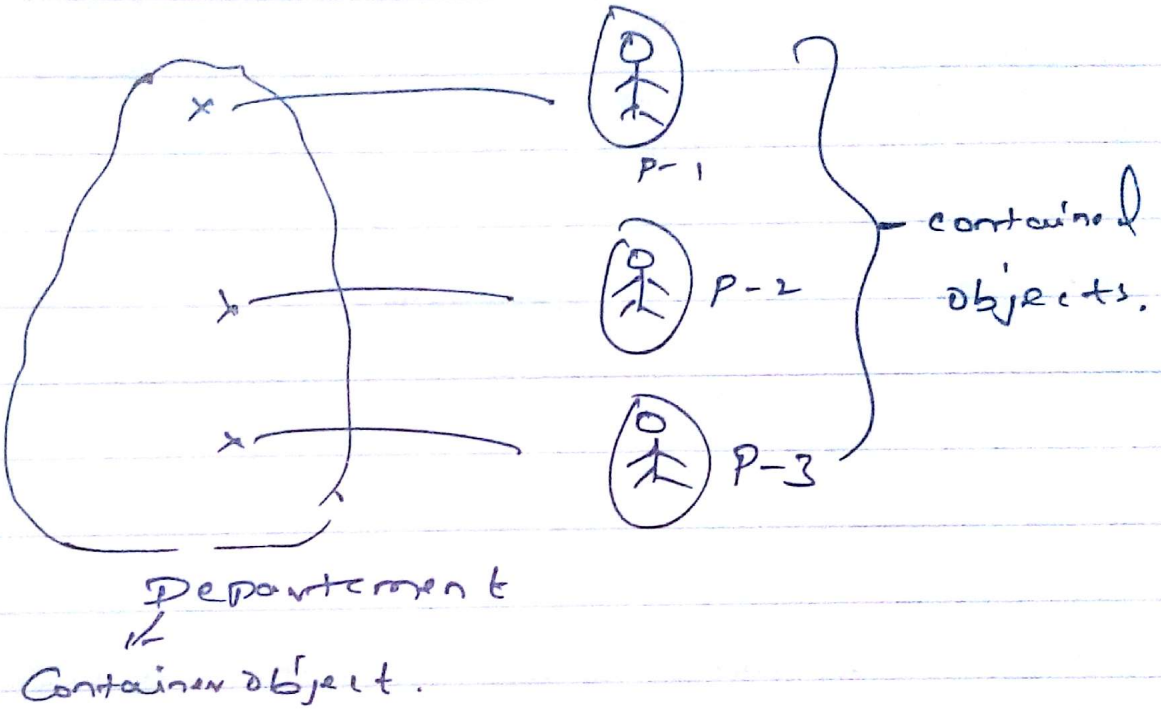
Aggregation:-

with out existing container Object if there is a chance of existing contained Object. Then container and contained objects are ~~stro~~ weakly associated and this weak association is nothing but aggregation.

Ex:-

department consist of several professers with out existing department There maybe a chance of existing professer objects hence department and professer objects are weakly

associated and this weak association is nothing but "aggregation".



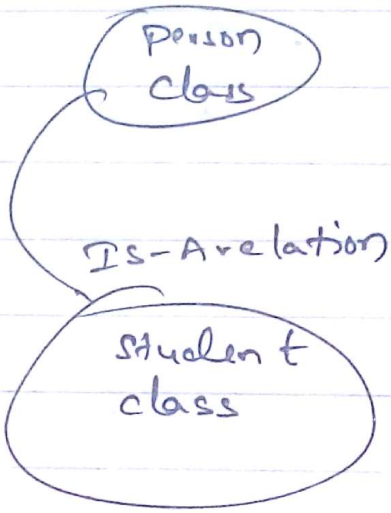
NOTE:-

1) In Composition objects are strongly associated where as in aggregation objects are weakly associated.
2) In Composition container object holds directly contained objects. Where as In aggregation container objects holds just references of contained objects.

IS-A V/S HAS-A

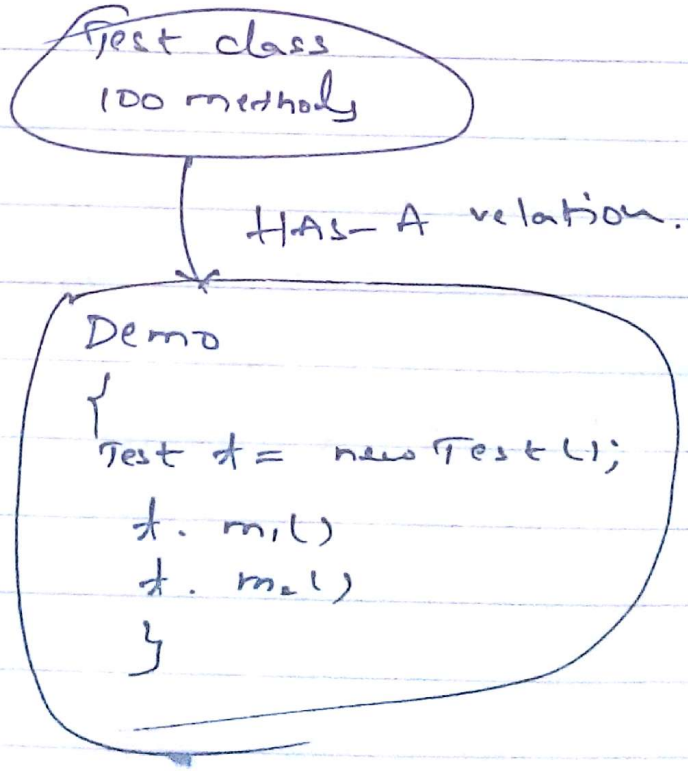
if we want Total functionality of a class automatically Then we should go for "Is-a" relationship.

Ex:-



if we want part of the functionality Then we should go for "HAS-A" relation ship.

Ex:-



Method signature:-

In Java method signature consist of method names followed by argument types.

Ex:-

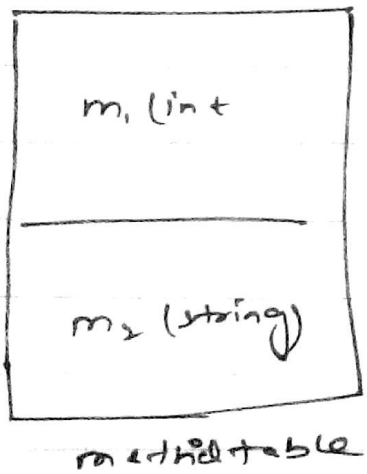
```
public static int m1(int i, float f)
```

$m_1(int, float)$

Return type is not part of method signature in java.

Compiler will use method signature to resolve method calls

Ex:-




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

```
Test t = new Test();
t.m1(10);  ✓
t.m2("durga");  ✓
t.m2(10 5);  ✗
```

CE: cannot find symbol
symbol: method m3(double)
Location: class Test

with in a class two methods with the same signature not allowed.

Ex:-

```
class Test
{
  public void m1(int i) ≡> m1(int)
  {
  }
  public int m1(int x); ≡> m1(int)
  {return 10;
  }
}
```

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

CE:- m1(int) is already defined in Test.

Overloading :- Two methods are said to be overloaded if and only if both methods having same name but different argument types.

In C Language method overloading concept is not available hence we can't declare multiple methods with same name, but different argument types. if there is change in argument type compulsory we should go for new method name. which increases complexity of programing.

C language :-

```
abs (int i) => abs (10);
labs (long l) => labs (10l);
fabs (float f) => fabs (10.0f);
```

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 of programing.

```
class Test
{
  public void m1()
      S.o.pln (no-arg);
  }
  public void m1(int i)
  {
  S.o.pln (int arg);
  }
  public void m1(double d)
  {
   S.o.pln (double arg)
  }
}
```

(overloaded methods)

```
p s v main (string[] args)
{
  Test t = new Test();
  t.m1(); no-args
  t.m1(10); int-args
  t.m1(10.5); double arg
}
}
```

* in overloading method resolution always takes care by compiler based on reference type, hence overloading is also considered as compile-time polymorphism (or) static polymorisplism (or) Early binding.

Case(i) Automatic promotion in Overloading:-

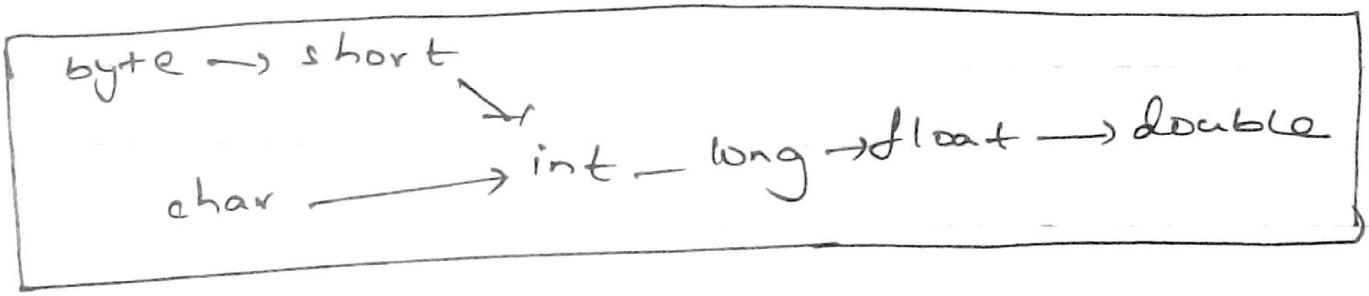
While resolving overloaded methods if exact matched method is not available then we won't get any compile time error immediatly, first it will promote argument to the next level and check whether matched method is available or not.

if matched method is available. Then it will be considered.

if the matched method is not available then compiler promotes arguments once again to the next level.

This process will be continued until all posible promotions still if the matched method is not available then we will get compile time error.

The following are all posible promotions in overloading



This process is called automatic promotion in Overloading.

ex: class Test

```
public void m1(int i)
{ s.o.println(int arg);
}
public void m1(float f)
{ s.o.println(float arg);
}
```

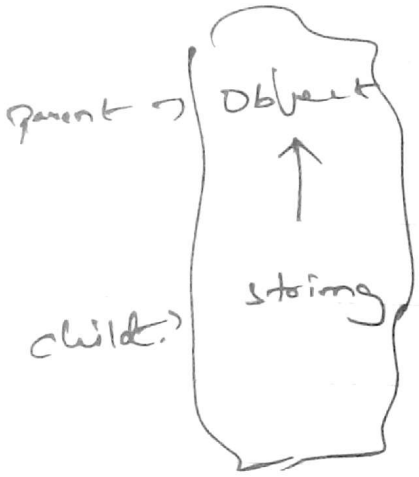
```
p.s.v. main(String[] arg)
{
Test t = new Test();
t.m1(10), int-arg
t.m1(10.5f), float-arg
t.m1('a'); int-arg
t.m1(10l); float-arg
t.m1(10.5) → CE:
```

Cant not find symbol
Symbol: method m1(double)
Location: class Test.

Case ii)

```
class Test
    public void m1(String s)
    { S.O.println("String version");
    }
    public void m1(Object o)
    { S.O.println("Object version");
    }
```

```
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.
}
}
```



NOTE: While resolving Overloaded methods Compiler will gives presidence for child ~~stat~~type argument than compared with parent type argument.

Ex Case iii)

```
class Test
    public void m1(String s)
    { S.O.P("String version");
    }
    P. V m1(StringBuffer sb)
    { S.O.println("StringBuffer version");
    }
```

```
Ps. 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 iv)

```
class Test {
    public void m1(int i, float f)
    {
        S.O.P.ln("int-float version");
    }
    public void m1(float f, int i)
    {
        S.O.P.ln("float-int version");
    }
}
```

```
P.S.V m(String[] args)
{
    Test t = new Test();
    t.m1(10, 0.5f); int float version
    t.m1(10.5f, 10); float int version
    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 (v)

```
class Test
public void m1(int x)
{
    S.O.P.ln("General method");
}
P.S.V m1(int...)
{
    S.O.P.ln("vararg method");
}
```

```
P.S.V main(String[] args)
{
    Test t = new Test();
    t.m1(); vararg method.
    t.m1(10,20); vararg method
    t.m1(10); General method.
}
}
```

In - General var-arg method will get least priority That is no - other method matched. Then only var-arg method will get the chance. it is Exactly same as default Case inside switch.

Case(vi) :-

```
class Animal
{
}
class monkey extends Animal
{
}
class Test
{
p.v. m1(Animal a)
  {
  s.o.pln("Animal version");
  }
p.v. m1(monkey m)
  {
  s.o.p.ln("mony version");
  }
}
```

(Overloaded methods)

```
p.s.v main(String[] args)
  Test t = new Test();
```

(1) Animal a = new Animal();
t.m1(a); Animal version

(2) Monkey m = new monkey();
t.m1(m); monkey version

(3) Animal a1 = new monkey();
t.m1(a), Animal version

```
}
}
```

In Overloading method resolution always takes care by Compiler based on reference type. in overloading Run time object won't play any role.

Overriding:- Whatever methods parent has by default available to the child thereon inheritance. if child class not satisfied with parent class implementation then child is allowed to redefined that method. based on its requirement. this process is called Overriding

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

Ex:-

```
class P
{
public void property()
{
S.O.pln ("cash + land + Gold");
}
public void marry()
{
S.OP. ln ("subbalaxmi");
}
}
class C extends P
{
public void marry()
{
S.O.pln ("Isha | ume | gtara");
}
}
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
}
}
```

Overridden

Overriding

Overriding method.

* in overriding method resolution always takes care by JVM based on runtime object. and hence overriding is also considered as Runtime polymorphism. (or) dynamic polymorphism (or) late binding.

rules for overriding :-

① in overriding method names and argument types must be matched i.e., method signatures must be same.

② In Overriding return types must be same. but this rule is applicable (until 1.4 version) only. from 1.5 version onwards we can take co-variant return types. According to this child class method return type need not be same as parent method return type. its child type also allowed.

Ex:-

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

It is invalid in 1.4 version But from 1.5 version it is valid

Parent class method return type	object	Number	String	double
child class method return type	object/String/ StringBuffer ✓	Number/ Integer ✓	object ✗	int ✗

Covariant return type concept applicable only for object types but not primitive types.

⇒ Parent class private methods not available to the child and 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

Ex:

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

it is valid but not Overriding

we cant override parent class final method in child classes.
if we are trying to override we will get compile time error

Ex:
```
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 methods we should override in child class to provide implementation.

Ex:
```
abstract class P
 { public abstract void m1();
}
 class C extends P
 {
   public void m1()
   {
   }
 }
```

* we can override non abstract method as abstract

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

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

In overriding the following modifiers wont keep any Restrictions.

① synchonized ② native ③ strictfp

Parent method	final	non-final	abstract	synchonized
child method	non final/ final ✗	final ✓	non-abstract ✓	non-synchonized ✓

native ↑↓ (machine code) non native ✓

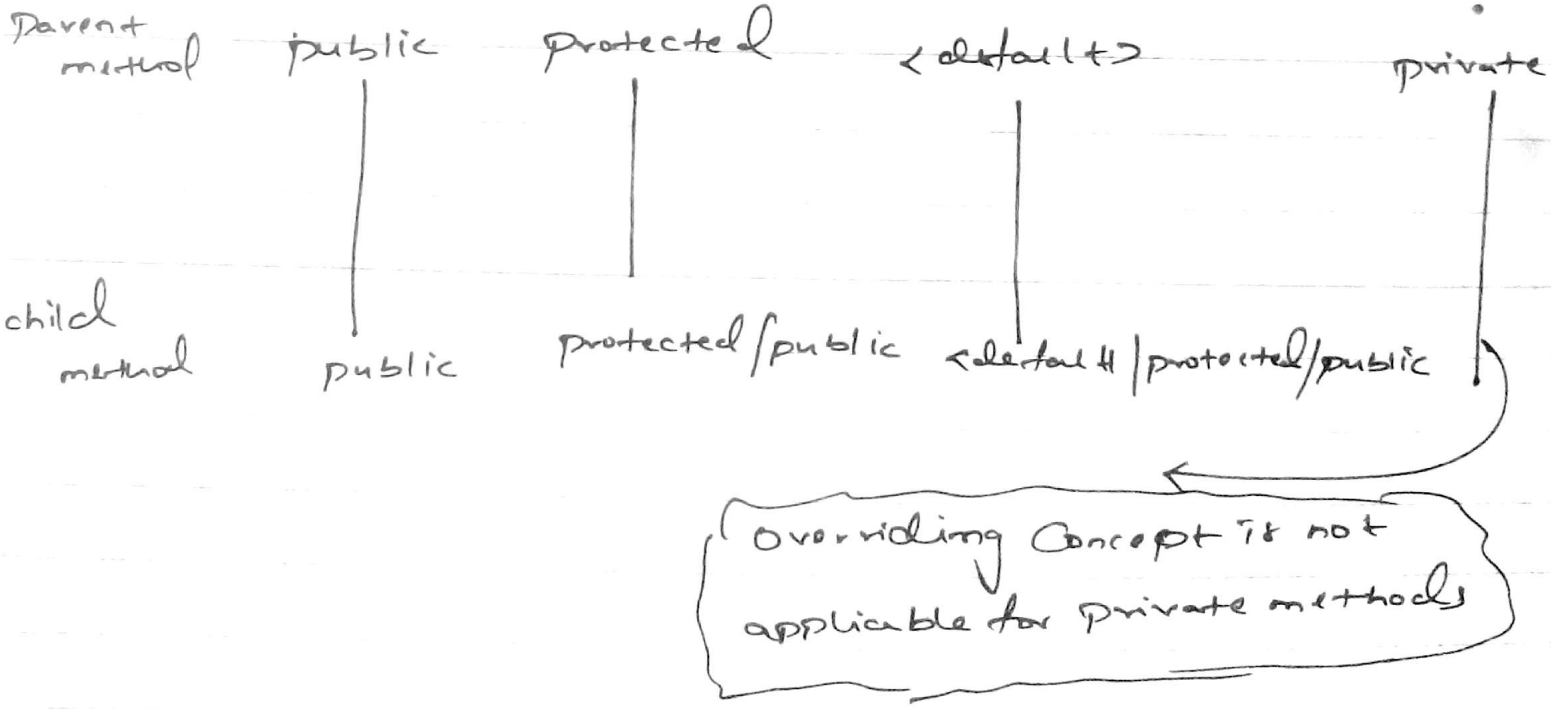
strictfp ↑↓ non-strictfp ✓

⇒ while overriding we can't "reduce" scope of access modifiers. but we can "increase" the scope.

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

CE: m1 in C cannot override m1 in P;
attempting to assign weaker access privileges was public

* private < default < protected < public



* if child class method throws any cheaked Exception compulsory parent class method should throw the same cheaked exption (or) its parent. otherwise coe will get Compile time error.

But there are no restrictions for Uncheaked Exceptions.

Ex:

```
import java.io.*;
class P
{ public void m1() throws IOException
{
}
};
class C extends P
{ public void m1() throws EOFException, InterruptedExcept
{
}
}
```

CE: m1 in C cannot override m1 in P; overridden method doesn't 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 IO Exception

C: public void m1() Throws ~~Exception~~ FileNotFoundE, EOF

⑥ P: public void m1() Throws IO Exption.
×

C: public void m1() Throws EOF Exception, InterruptedException

⑦ P: public void m1() Throws IO Exception
✓

C: public void m1() Throws AE, NPE, CCE. (class cast)

Overriding with respect to static methods

Case

① we can't override a static method as non static other wise we will get compile time Error.

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

CE: m1 in C cannot override m1 in P: Overridden method is static

case ②

Similarly we can not override a non-static method as static method.

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

CE: m1 in C cannot override m1() in P; overriding method is static

case 3 : if both parent and class methods are static then we can't get any Compile time Error. it seems Overriding Concept applicable for static methods. but it is not ridding and it is method hidding.

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

it is a method hiding but not Overriding

Method hiding:-

All rules of method hiding are Exactly same as Overriding. Except the following differences.

Method hiding	overriding
① both method (parent & child) classes should be static	① both parent & child class methods should be not static
② Compiler is responsible for method resolution based on reference type	② JVM is always responsible for method resolution based on Run time object
③ it is also known as compile time polymorphism (or) static polymorphism (or) early binding	③ it is also known as Run time polymorphism (or) dynamic polymorphism (or) late binding.

Ex:-

```
class P
{
public static void m1()
{
s.o.pln (parent);
}
}
class C extends P
{
public static void m1()
{
s.o.pln (child);
}
}
```

```
Class Test
{
psv main (Str)
{
P p = new P();
p m1(); ---> Parent
C c = new C();
c m1(); ---> child
P p1 = new C();
p1 m1(); ---> Parent
}
}
```

it method
iding but
not overriding

if both Parent and child class methods are non-static Then it will become Overriding in this case output

Ts

```
Parent
child
child
```

Overriding with respect to varargs methods :-

We can override varargs method with another varargs method only. If we are trying to override with normal method then it will become overloading. but not overriding.

Ex:

```
class P
{
    public void m1(int... x)
    {
        s.o.println("parent");
    }
}
class C extends P
{
    public void m1(int x)
    {
        s.o.println("child");
    }
}
```

it is overloading but not overriding

```
class Test
{
    psvm main(String[] args)
    {
        P p = new P();
        p.m1(10);   ---> parent
        C c = new C();
        c.m1(10);   ---> child
        P p1 = new C();
        p1.m1(10);  parent.
    }
}
```

In the above program if we replace child method with varargs method. Then it will become overriding. in this case the output is

o/p :- parent
child
~~parent~~
child

Overriding with respect to variables :-

variable resolution always takes care by Compiler based on reference type irrespective of whether the variable is static (or) non-static. (Overriding Concept applicable only for methods but not for variables.

```
class P
{
int x = 888;
}
class C extends P
{int 999;
}
```

```
class Test
{
P.S.V. main (String[] args)
{P.P = new P();
 S.O.Pln (P.x); ---> 888
 C. C = new C();
S.O.Pln (C.x); ---> 999
 P. P1 = new (C);
   S.O.Pln (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

Diff between Overloading & Overriding

property	Overloading	overriding
1) method names	must be same	must be same
2) argument types	must be different (at least order)	must be same (including order)
3) method signatures	must be different	must be same
4) Return types	no restrictions	must be same until 1.4 version but from 1.5 version onwards co-variant return types also allowed.
5) private static, final methods	Can be overloaded	Can not be overridden.

property	Overloading	Overriding
⑥ Access modifiers	No restriction	⇒ we can't reduce scope of Access modifier we but we can increase the scope
⑦ Throws Clause (key)	No Restriction	⇒ if child class method throws any checked exception compulsory parent class method should throw same checked exception or its parent. but no restrictions for unchecked exceptions.
⑧ method resolution	always takes care by compiler based on Reference type	⇒ always takes care by JVM based on Runtime Object.
⑨ it is also known as	compiled time polymorphism (or) static polymorphism. (or) Early binding.	⇒ Runtime polymorphism (or) dynamic polymorphism (or) late binding.

NOTE: in overloading we have to check only method names (must be same) & argument types (must be diff)

→ we are not required to check remaining like return type, access modifiers etc..

→ but in overriding everything we have to check. like method names, argument types, Return types access modifiers, throws keywords (Class) etc,...

Consider the following method in Parent class

```
public void m1(int x) throws IOException
```

① ⟶ in the child class which of the following methods we can take?

overriding ✗ ① public void m1(int i)

overloading ✓ ② public static int m1(long l)

overriding ✗ ③ public static void m1(int i)

overriding ✗ ④ public void m1(int i) throws Exception

✗ ⑤ public static abstract void m1(double d);

CE.

↓

illegal combination of modifiers.

polymorphism :- One name but multiple forms is the Concept of polymorphism.

Ex:- I) Method name is the Same but we Can apply for different types of arguments. (Overloading)

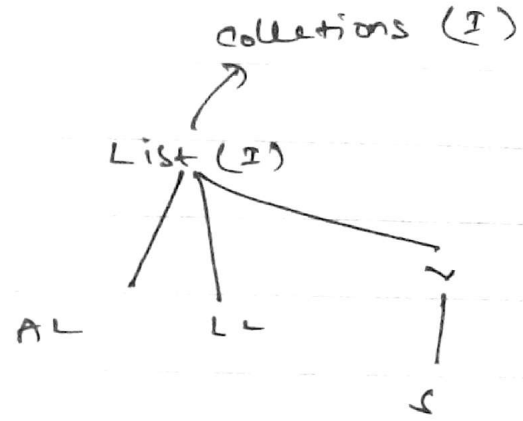
Ex:- abs (int)
abs (long) } overloading
abs (float)

Ex2: method signature is same but in Parent class one type implementation and in the child class another type of implementation. (Overriding)

Ex:
```
class P
{
  marry ( )
   { s.o.pln ("Subhalaxmi");
   }
 }
class c extends P
 { marry ( )
   { s.o.pln( "isha | 4me | utara");
   }
 }
```

Ex:- Usage of parent Reference to hold child Object is the Concept of polymorphism.

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



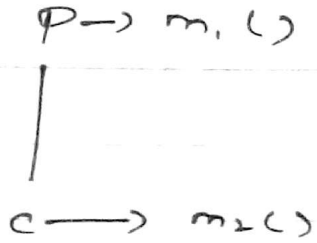
Parent class reference can be used to hold child object but by using that reference we can call only the methods available in parent class and we can't call child specific methods.

But by this

P P = new C();

P.m1(); ✓

P.m2(); ✗



CE: can not find symbol symbol: method m2()
location : class P

But by using child object reference we can call both Parent & child class methods. Ex:- C c = new C();

c.m1(); ✓

c.m2(); ✓

When we should go for Parent reference to hold child object:-

-) If we don't know Exact Runtime type of object Then we should go for Parent Reference.

=) For Example the first Element present in the array list can be any type. it may be Student object (or) Customer object (or) String object (or) StringBuffer object Hence the Return type of get method is object, which can hold any object.

```
Object o = l.get(0);
```

C c = new C();	P p = new C();
AL l = new AL();	List l = new AL();
① we can use this approach if we know Exact Runtime type of object.	② we can use this approach if we don't know Exact Runtime type of object.
② By using child Reference we can call both parent and child class methods (This adv of this approach)	② By using the 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 approach)
③ we can use child reference to hold only particular child class object. (This is the disadvantage of this approach)	③ we can use Parent reference to hold any child class object. (this is the adv of this approach)

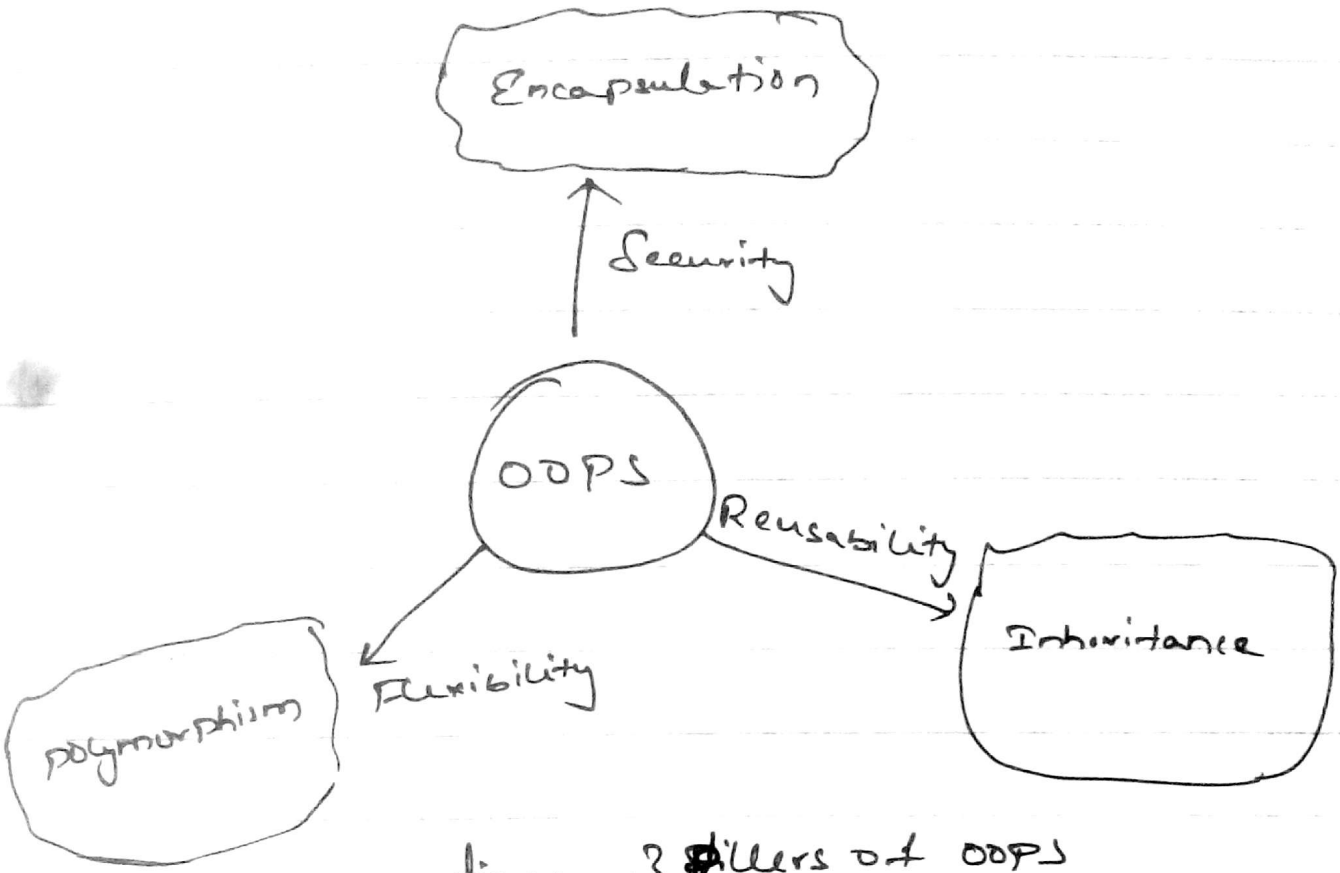


fig: 3 pillers of OOPS

polymorphism

Static polymorphism
(or)
Compile-time polymorphism
(or)
Early binding

Overloading
method hiding

Dynamic polymorphism
(or)
Run time polymorphism
(or)
late binding

Overriding

Beautiful definition of polymorphism!-

A Boy starts love with the word "FRIENDSHIP", but girl ends the love with the same word "FRIENDSHIP". The word is the same but attitude is different. This is the concept of polymorphism.

remaining topics:

Coupling:-

The degree of dependency b/w the components is called Coupling. if dependency is more then it is considered as "tightly coupling" & if dependency is less then it is considered as "loosely coupling.

Ex:-

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

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

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

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

Tightly coupled

⇒ The above components are said to be tightly coupled with each other because dependency b/w the components is more.

=> Tightly coupling is a not a good programing practice because it has several serious disadvantages.

(1) with out effecting remaining components we can't modify any component. And hence enhancement will become difficult.

(2) it surpresses reusability

(3) it reduces maintainability of the application.

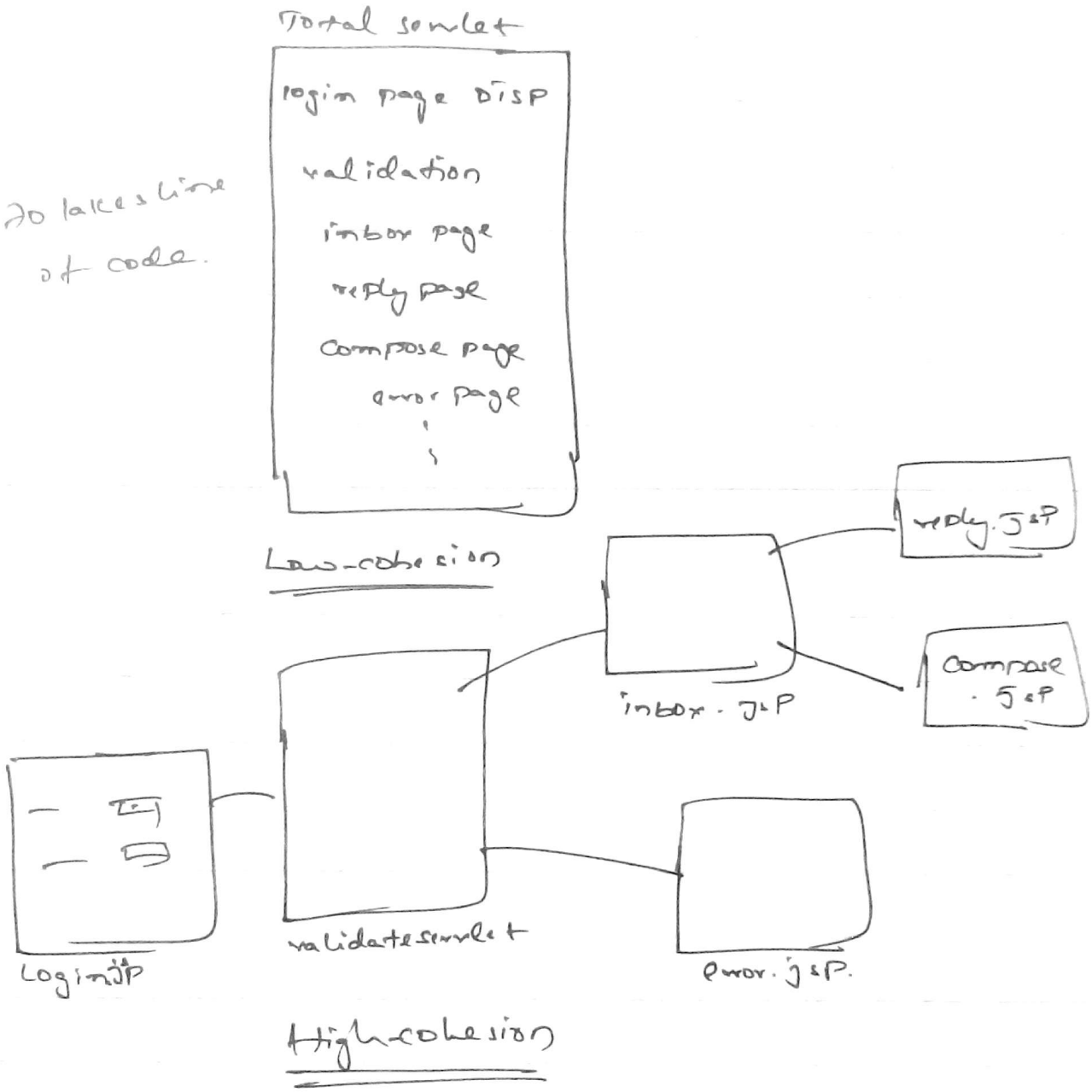
Hence we have to maintain dependency b/w the components as less as possible. That is loosely coupling is a good programing practice.

Cohesion For every component a clear well defined functionality is required (defined) then that component is said to be follow high cohesion.

High-Cohesion :- is always a good programing practice because it has several advantages

(1) without effecting remaining components we can modify any component. hence enhancement will become easy.

(2) it promots reusability of the code. (Whenever validation required we can reuse the same validate servlet with out re writing).



(2) it improves maintainability of the application.

NOTE: Loosely Coupling & and High cohesion are good programing practices

Object type Casting:-

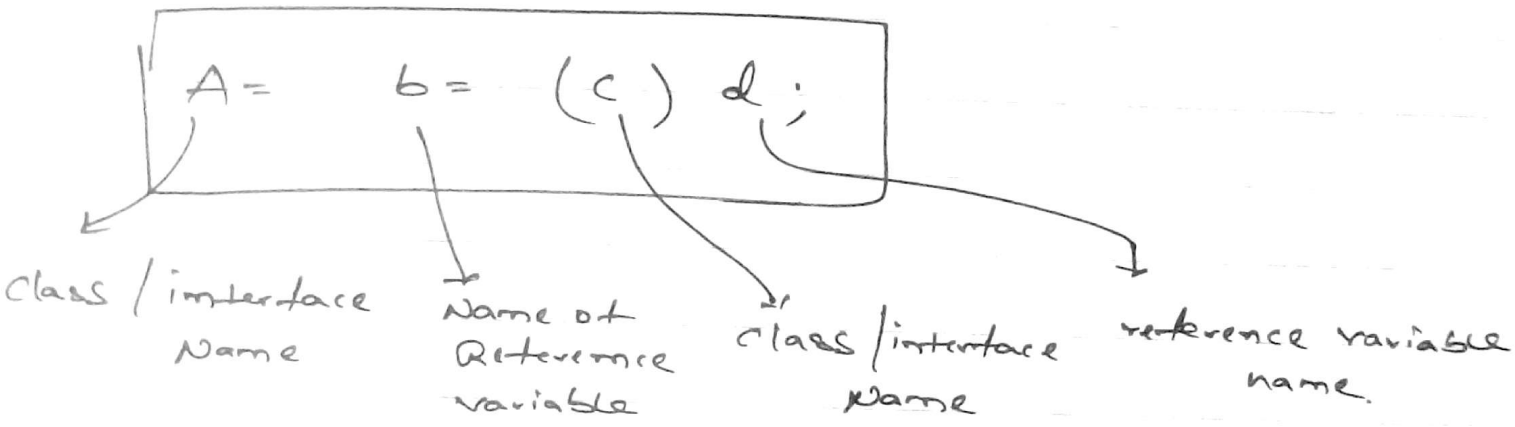
=) we can use parent reference to hold child object

Example

```
Object O = new String ("durga");
```

=) we can use interface reference to hold implemented class object

Ex: Runnable r = new Thread ();



Mantra 1: (compile time checking 1)

The type of "d" and 'c' must have some relation either child to parent (or) Parent to child (or) same type. otherwise we will get compile time error. saying

CE:

```
inconvertable type
found : d type
required : c
```

```
p.s.v s [] args
{
Ex:  Object O = new String("durga");
     StringBuffer sb = (StringBuffer)O;
}
```

```
Ex:  String s = new String("durga");
     StringBuffer sb = (StringBuffer)s;
```

CE: incortable types
found : java.lang.string
required : java.lang.stringBuffer.

Plantoa 2 (compiletime cheaking 2) :-

'C' must be either same (or) derived type of 'A'. otherwise we will get compile time error. saying in compatable types found 'C' required 'A'.

Ex:1
```
Object O = new String("durga");
StringBuffer sb = (StringBuffer) O;
```

Ex:2
```
Object O = new String("durga");
StringBuffer sb = (String) O;
```
CE: incompitable types
found: J.l. String
required: J.l. StringBuffer.

Mantra 3 (Runtime checking)

Runtime object type of 'd' must be either same or) derived type of 'c' otherwise we will get Runtime exception saying "classcast exception"

Ex:1
```
Object o = new String("durga");
StringBuffer sb = (StringBuffer) o;
```

Ex:2
```
Object o = new String("durga");
✓ Object o1 = (String) o;
```

```
Base2 b = new Derv();
```
CE: incompatible type found: Base2 required: Base1

✓① `Object o = (Base2) b;`

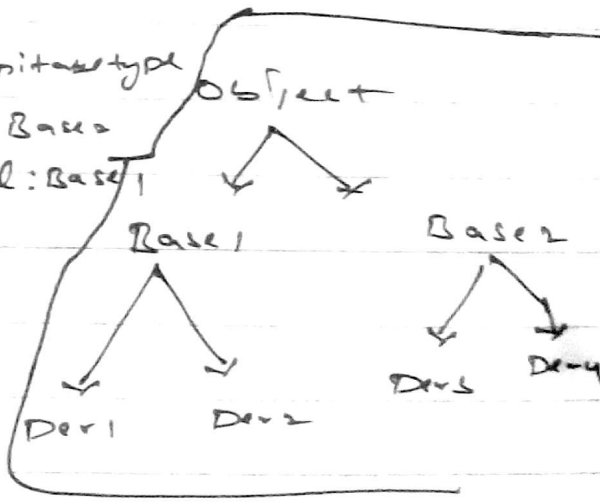
✗② `Object o = (Base1) b;`

✗③ `Object o = (Der3) b;`

✗④ `Obj Base2 b1 = (Base1) b;`

✗⑤ `Base1 b1 = (Derv) b;`

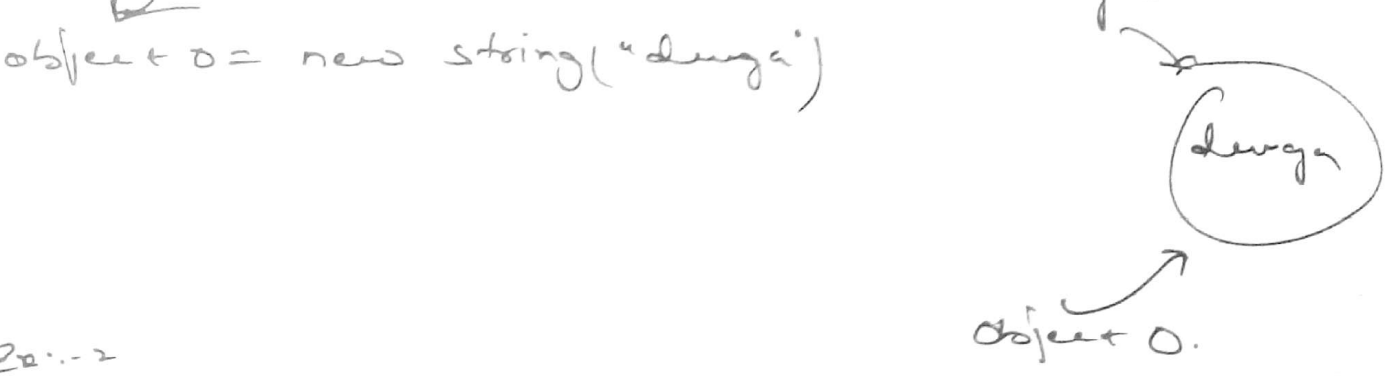
✗⑥ `Base b1 = (Der1) b;` → CE: incompatible type found: Base2 required: Der1



strictly speaking these type casting we are not creating any new object.

for the existing object we are providing another type of Reference variable. That is we are performing typecasting but not object casting.

Ex:
```
String s = new String("durga");
Object o = (Object) s;
```



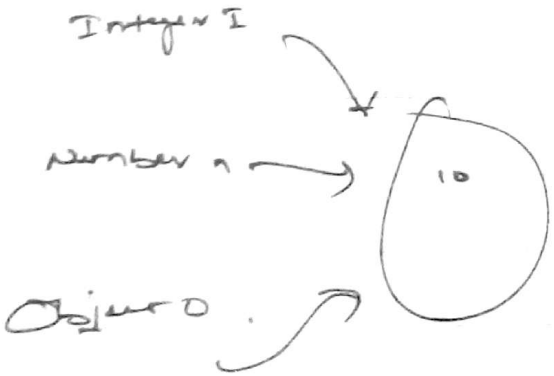
Ex:-2

```
Integer I = new Integer(10);
Number n = (Number) I;
Object o = (Object) n;
```

Object o = new Integer(10);

Number n = new Integer(10);

```
SOP(I==n); true
SOP(n==o); true
```



NOTE:- C c = new C();

B b = new (C); ← (B)C

A a = new (C); ← A((B)C)

Ex:1

C c = new C();

✓ c.m1();
✓ c.m2();
✓ ((P)c).m1();
✗ ((P)c).m2();

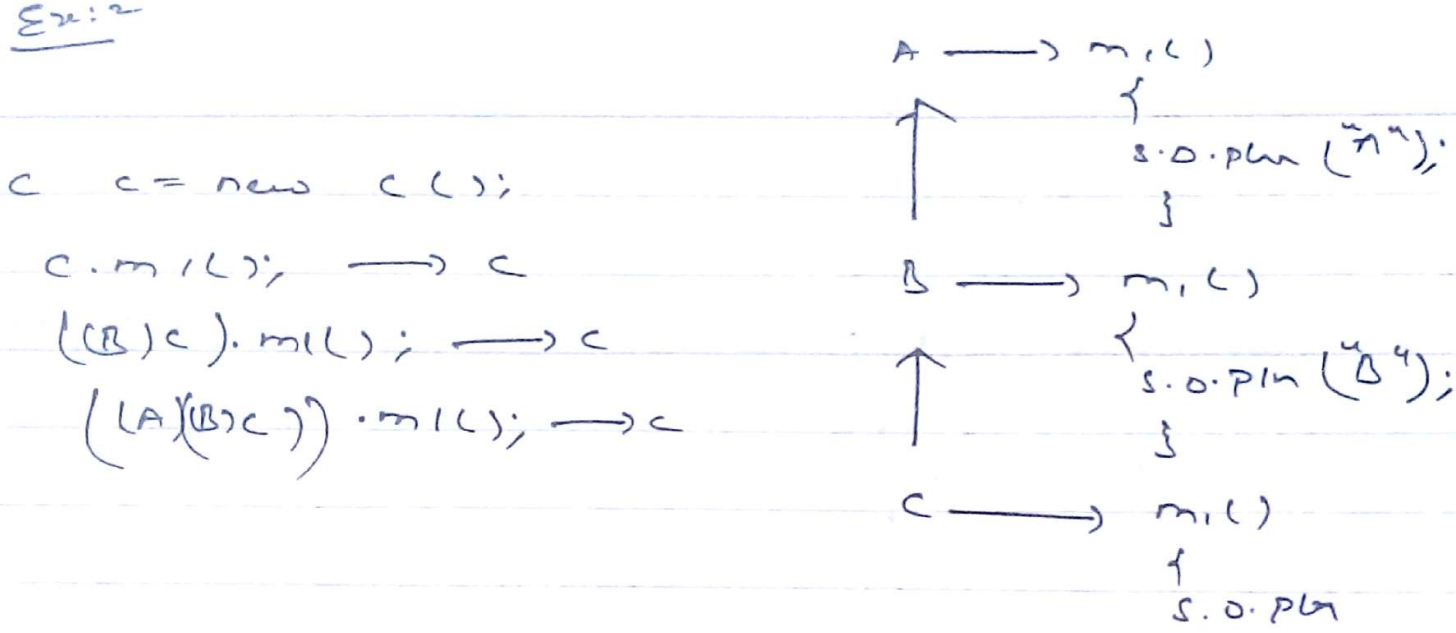
P → m1(){}
↑
C → m2(){}

P P = new C();
P.m1();

P P = new C();
P.m2();

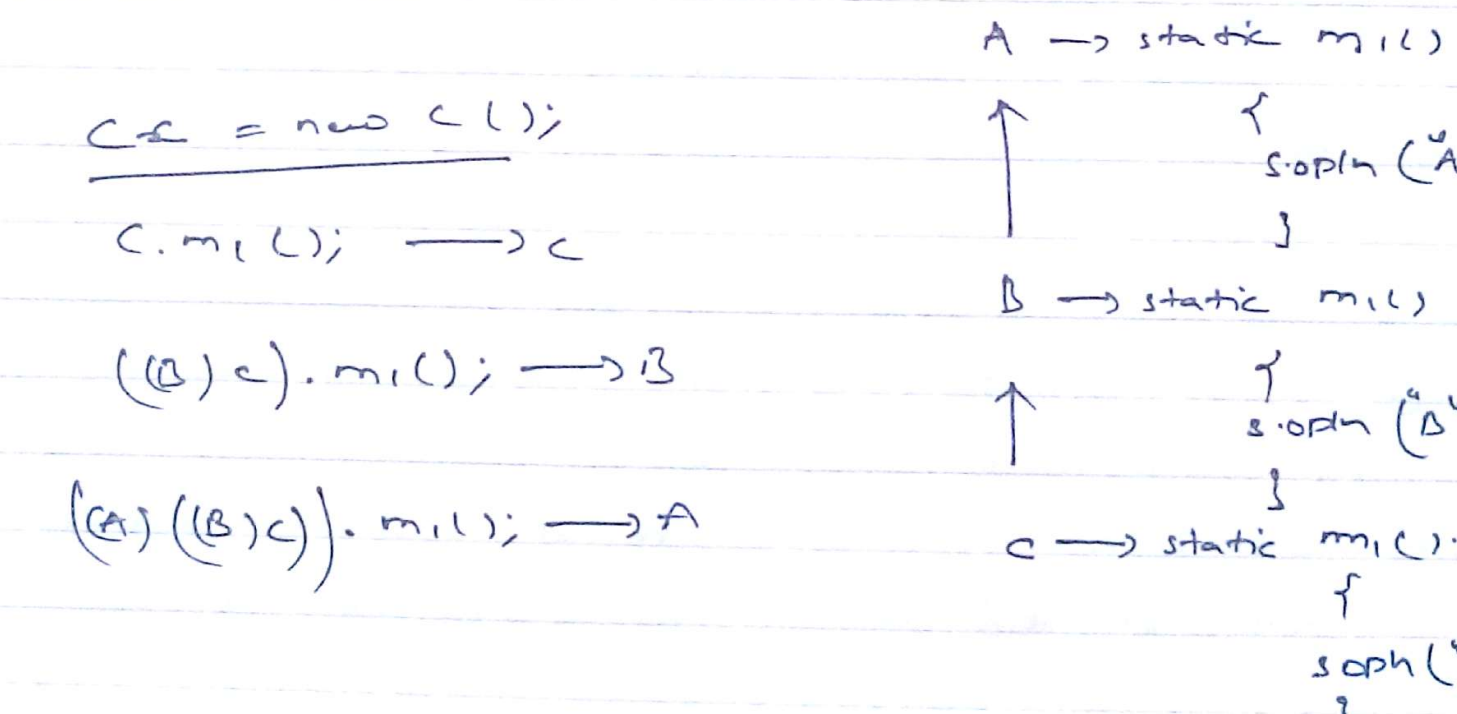
Reason: Parent Reference can be used to hold child Object but by using that Reference we can't call child specific methods. and we can call only methods available in parent class.

Ex:2



It is overriding and method Resolution is always based on Run time object type

Ex:3



It is method hiding and method Resolution it always based on Reference type.

Ex:-4

C c = new C();

S.opln(c.x); ⟶ 999

S.opln(((B)c)x); ⟶ 888

S.opln(((A(B)c)).x); ⟶ 777

A ⟶ int x = 777;

↑

B ⟶ int x = 888;

↑

C ⟶ int x = 999;

NOTE: variable Resolution is always based on Reference type but not based on Run time object.

Question 10

Static Control flow:-

When never we are executing a java class The following sequence of steps are will be executed as the part of static Control flow.

① Identification of static members from top to bottom [1 to 6]

② Execution of static variable assignments and static blocks from top to bottom [7 to 12]

③ Execution of main method [13 to 15]

```
class Base
{
① static int i = 10;                                   ⑦
② static
  {
    m1();  ⑧
    S.O.Pln("first static Block"); 10
  }
③ P S.V main (String[] args)
  {
    m1(); 13
    S.O.Pln("main method"); ⑮
  }
④ P S.V m1()
  {
    S.O.Pln(j);  ⑨  14
  }
⑤ static
  {
    S.OPln("secondStatic Block"); 11
  }
⑥ static int j = 20;  12
}
```

i = 0 [RIWO]
j = 0 [RIWO]
i = 10 [RLW]
j = 20 [RLW]

O/P

Java Base
①
First Static Block
Second Static Block
20
main method

RIWO [Read Indirectly write Only] :-

In side Static block if we are trying to Read a variable. That Read operation is called "direct Read".

if we are calling method and within that method if we are trying to Read a variable that read operation is call Indirect Read.

```
class Test
{
    static int i=10;
    static
    {
        m1();
        S.O.Pln(i); ----> Direct Read
    }
    P S.V m1()
    {
        S.O.Pln(i); -> Indirect Read
    }
}
```

i=0 [RIWO]

CE: illegal forward reference

if a variable is just identified by the JVM and original not yet assigned Then the variable is said to be in Read indirectly & write only state. [RIWO].

if a variable is in read indirectly write only state Then ~~That read operation is~~ ca we can't perform direct Read. but we can perform indirect Read. if we are trying to read directly Then we will get compile time error saying **illegal forward Reference**

Ex:1

```
class Test
{
static int x=10;
 static
 {
  s.o.pln(x);
 }
}
```

O/P= 10

```
class Test
{
static
 {s.o.pln(x);
 }
static int x= 10;
}
```

x=0 [RTW]

RE: NoSuchmethodError:main

CE: illegal forward Reference

~~Ex3~~:

```
class Test
{  static
    {m1();
    }
  ps.x m1()
    {s.o.pln(x))
    }
    static int x=10;
    }.
```

O/P 0

RE: No such method error: main.

Static block

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

Ex: 1 at the time of java class loading the corresponding native library should be loaded. hence we have to define this activity inside static block.

```
Ex:  class Test
     {
       static
       {
         system.loadLibrary("nativelibrarypath");
       }
     }
```

Ex: 2 after loading every database driver class we have to register driver class with driver manager. but inside database driver class there is a static block. To perform this activity and we are not responsible to register explicity

```
class DbDriver
{
  static
  {
    Register this driver
    with driver manager
  }
}
```

NOTE: with in a class we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

① without writing main method is it possible to print some statements to the Console?

Ans:- yes, by using static block.

Ex:
```
class Test
{
  static
  {s.opln("Hello I can print");
   System.exit(0);
  }
}
```
o/p: hello I can print

Ex:2

Q②: with out writing main method & static block is it possible to print some statements to the Console?

Ans: yes, of course There are multiple ways

```
class Test
{
 static int x = m1();
  public static int m1()
  {System.out.println("Hello I Can print");
   System.out(0);
   return 10;
  }
}
```

```
class Test
{
 static Test t = new Test();
  {s.opln("Hello I can print");
   System.out(0);
  }
}
```

```
class Test
{
 static Test t = new Test();
  Test()
  {
  s.opln("Hello I can
  System.out(0);
  }
}
```

NOTE: doom 1.7 version onwards main method it mandatory to start program execution. hence doom 1.7 version onwards without writing main method it is impossible to print some statements to the Console.

Static Control flow in parent to child relationship:-

whenever we are executing child class the following sequence of events will be executed automatically as the part of static control flow

[1 to 15]

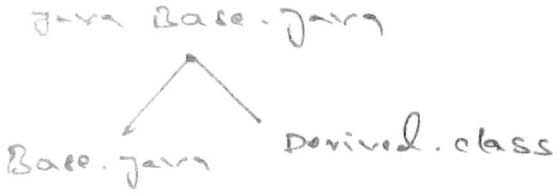
(1) Identification of static members doom parent to child

(2) Execution of static variable assignments and static blocks doom parent to child. [12 to 22]

(3) Execution of only child class main method. [23 to 25]

```
① class Base
② {
   static int i=0; (12)
   static
   {
   m1(); (13)
   s.o.pln("Base static Block"); (15)
   }
③  p s v main (static (or) args)
   {m1();
    s.o.pln("Base main");
④  p s.v m1()
   {so.pln(i); (14)
⑤  static int j=20; (16)
   }
```

```
   class Derived extends Base
⑥  {static int x=100; (17)
⑦  static
   {m2(); (18)
    s.o.pln("Derived First static Block"); (20)
   }
⑧  p.s.v main (string [] args)
   {
   m2(); (23)
   s.o.pln("Derived main"); (25)
⑨  p s.v m2()
   {s.o.pln(y); (24) (19)
⑩  static
   {s.o.pln("Derived second Block"); (21)
⑪  static int y=200; (22)
   }
```



o/p

0
Base static block
0
Derived First
Derived Second
200
Derived main

Java Base

0
Base static block
20
Base main

NOTE:- whenever we are loading child class automatically parent class will be loaded. but whenever we are loading parent class child class won't be loaded. (because parent class members by default available to the child class whereas child class members by default won't available to the parent)

Instance Control flow:-

Whenever we are executing a java class first static control flow will be executed.

In the static control flow if we are creating an object the following sequence of events will be executed. as the part of instance control flow.

1. Identification of Instance members from top to bottom (3 to 8)
2. execution of instance variable assignments and Instance blocks from top to bottom. (9 to 14)
3. Execution of Constructor (15)

Ex:

```
class Test
{
③   int i = 10;
④   m1();
    s.o.println("First Instance Block"); ⑫
⑤ }Test()
  {s.o.println("constructor"); ⑮
① }p.s.v main(String[] args)
  {
②   Test t = new Test(); Line 11
    s.o.println("main");
  }
⑥ public void m1()
  { s.o.println(j); ⑪
  }
⑦ {
    s.o.println("Second Instance Block");
⑧ }int j = 20;
  }
```

i = 0 [RIW 0]
j = 0 [RIW 0]
i = 10 [RLW]
j = 20 [RLW]

O/P

0
First start
Second Instance
constructor
main

If we comment Line 1 Then the o/p is main.

NOTE: ① Static control flow is one Time activity which will be performed at the time of class loading.

But Instance Control flow is not one time activity & it will be performed for object creation.

② Object creation is the most costly operation if there is no specific requirement Then it is not recomended To create Object.

Instance control flow in Parent to child relationship:-

whenever we are creating child class object the following sequence of events will be performed automatically as the part of Instance control flow

① Identification of Instance members from parent to child (1 to 14)

② Execution of Instance Variable assignments and Instance blocks only in Parent Class. [15 to 19]

③ Execution of Parent Constructor (20)

④ Execution of Instance variable assignments and Instance blocks in child class. [21 to 26]

⑤ Execution child constructor (27)

```
class Parent
{
(4) int i=10; (15)
}
(5) m1(); (16)
    s.o.pln("Parent IB"); (18)
}
parent().
{
s.o.pln("P C"); (20)
}
p.s.v main(String[] args)
{Parent p=new Parent();
 s.o.pln("Parent main")
}
(7) public void main()
{
s.o.pln(j); (17)
}
(8) int j = 20; (19)
```

i=0 [RLW]
j=0 ["] (6)
z=0 [u u]
y=0 [u u]
i=10 [RLW] (1)
j=20 [RLW]
k=100 [RLW]
y=100 [RLW]

```
class
child extends Parent
(9) int k = 100; (21)
(10) { m2();
   s.o.pln("C E I S"); (24)
}child()
   {s.o.pln("child constructor");} (27)
}
(2) P.s.v main(String[] args)
{
(3) child c=new child();
(28) s.o.pln("child main");
}
(12) public void m2()
{
s.o.pln(y); (23)
}
(13) s.o.pln("C S I B"); (25)
}
(14) int j = 200; 26
}
```

O/P Java child

0
P I B
P C
0
C F I B
C S I B
child constructor
child main

Eg:

```
class Test
{
  {s.o.pln ("FIB");
  }
  static
  {
  s.o.pln ("FSB");
  }
  Test()
  {s.o.pln ("constructor");
  }
PSV main (String[] args)
{Test t1 = new Test();
  s.o.pln ("main");
  Test t2 = new Test();
  }
  static
  {s.o.pln ("SSB");
  }{s.o.pln ("SIB");
  }
  }
```

o/p

FSB
SSB
FIB
SIB
Constructor
main
FIB
SIB
Constructor.

Ex:-

```
public class Intialization
{
private static String m1(String msg)
{
S.O.Pln(msg);
  Return msg;
 }
public Intialization()
{
 m = m1("1");
}
m = m1("2");
}
String m= m1("3");
③ P S V main(String[] args)
① {
Object o= new Intialization();
  }
 }
```

O/P

2
3
1

Ex:-

```
public class Intialization
{
Private static String m1(String msg)
{S.O.Pln(msg);
 Return msg;
}
Static String m=m1("4");
 {
  m = m1("42");
 }
 static
 { m= m1("43");
 }P.S.V main(String[] args)
  {Object obj = new Intialization
 }
```

O/P

1
3
2

* NOTE: From static area we can't access instance members directly because while executing static area JVM may not identify instance members

Ex:
```
class Test
{
    int x = 10;
    P S. V main (String[] args)
    { S.O.Pln(x); ⟶ CE: non-static variable x cannot be referenced from a static context
      Test t = new Test();
      S.O.Pln(t.x);
    }
}
```

** In How many ways we can create an object in java (or) In how many ways we can get object in java.

① By using new operator

```
Test t = new Test();
```

② By using newInstance() method:

```
test t = (Test) class.forName("Test").newInstance();
```

③ By using Factory method.

```
Runtime r = Runtime.getRuntime();
DateFormat df = DateFormat.getInstance();
```

④ By using clone method

```
Test t1 = new Test()
test t2 = (Test) t1.clone();
```

⑤ By using Deserialization

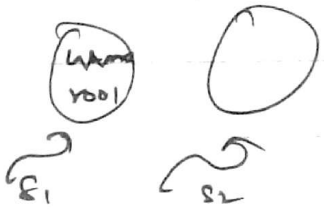
```
FileInputStream fir = new FIS ("abs.ser");
ObjectInputStream ois = new OIS (Fir);
Dog d2 = (Dog) ois.readObject();
```

These are the above are 5 standerd methods To get an object. (or) Create an object.

Constructors

once we creates an object Compulsory we should perform initialization. Then only the object is in a position to respond properly.

whenever we are creating an object some peice of the code will be executed automatically to perform initialization of the object. This peice of the code is nothing but "Constructor". Hense the main purpose of constructor is to perform initialization of an object.



ex:
```
class student
{
    string name;
    int rollno;
    student (string name, int rollno)
    {
        this.name = name;
        this.rollno = rollno;
    }
    p s v main (string[] arg)
    {
        student s1 = new student ("durga", 101);
        student s2 = new student ("Ravi", 102);
    }
}
```

Constructor ←

NOTE:- The main purpose of constructor is to perform initialization of an object. but not to create object.

Difference b/w Constructor & instance block :-

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

⇒ But other than initialization if we want to perform any activity for every object creation. Then we should go for instance block. (Like updating one entry in the data base for object creation (or) incrementing count value for every object creation etc..)

⇒ Both Constructor & instance block have their own purposes and replacing one concept with another concept may not work always.

⇒ Both Constructor and instance block will be executed for object creation but 1st instance block first followed Constructor.

Demo program to print number of objects created for a class

```
class Test
{
    static int count=0;
    {
        count++;
    }
    Test()
    {
    }
    Test(int i)
    {
    }
    Test(double d)
    {
    }
    public static void main(String[] args)
    {
        Test t1=new Test();
        Test t2=new Test(10);
        Test t3=new Test(10.5);
        S.o.println("The no of objects created:"+count);
    }
}
```

rules of writing Constructor :-

① name of the class and name of constructor must be matched.

② written type concept not applicable for constructor even void also.

③ By mistake if we are trying declare return type for the constructor then we won't get any compile time error. because compiler treats its as a method.

Ex: class Test
{
void Test
{
}
}

→ it is a method but not constructor.

Here it is legal (but stupid) to have a method whose name is exactly same as class name.

Ex:
```
class Test
{
void Test()
{System.o.println("method but not constructor");
}
public st, void m(String arg[])
{ Test t = new Test();
  t.test();
}
}.
```

The only applicable modifiers for constructors are public, private protected, default. if we are toying to use any other modifier we will get compile time error.

```
class Test
{
    static Test()
    {
    }
}
```

CE: modifier static not allowed here

Default Constructor:-

1) Compiler is responsible to generate default constructor (but not JVM)

2) if we are not writing any constructor Then only compiler will generate default constructor. i.e., if we are writing atleast one constructor. Then compiler won't generate default constructor. hence every class in Java can contain constructor it may be default constructor generated by compiler (or) customized constructor explicitly provided by programmer but not both simultaneously.