28/4/11

obm Sai Ram prabu.

Oops Concept

① Data Hiding :-

→ Hiding of the data, So that outside person Cant access our data directly.

→ By using private modifier we Can implement Data Hiding.

Ex:-

```
Class Account
{
  private double balance = 1000;
}
```

→ The main Advantage of Data Hiding is we Can acheive Security.

② Abstraction :-

→ Hiding internal implementation details & just highlate the set of Services what we are offering, is called "Abstraction."

Ex:-

↳ By Bank ATM machine, Bank people will highLate the Set of Services what they are offering without Highlating internal implementation. This Concept is nothing but Abstraction.

→ By Using interfaces & abstract classes we Can acheive abstraction.

→ The main Advantages of Abstraction are.

1) We Can acheive Security as no one is allowed to know our internal implementation.

2) With out effecting outside person we Can change our internal implementation Hence Enhancement will become very easy.

→ The main disadvantage of Encapsulation is it increases the length of the code & slows down execution.

4) Tightly Encapsulated class :-

→ A class is said to be tightly encapsulated iff every data member declared as the private.

→ Wheather the class contains getter & setter methods are not & wheather those methods declared as public or not these are not required to check.

Ex:-

```
Class A
{
  private int balance;
  public int getBalance()
  {
    return balance();
  }
}
```

Ex:- Which of the following classes are Tightly Encapsulated.

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

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

✓
```
Class C extends A
{
  Private int z=30;
}
```

③ It improves modularity of the application. meaning?

3) Encapsulation :-

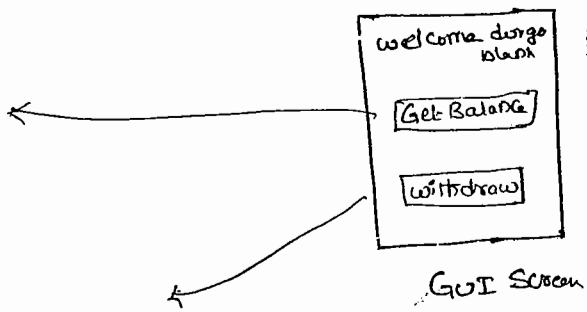
↳ Encapsulating data & corresponding methods (behaviour) into a single module is called "Encapsulation".

→ If any Java class follows Data Hiding & Abstraction such type of class is said to Encapsulated class.

> Encapsulation = Data Hiding + Abstraction

Ex:-

```
class Account
{
  private double balance;
  public double getBalance()
  {
    // validate User
    return balance;
  }
  public void setBalance(double balance)
  {
    // validate User
    this.balance = balance;
  }
}
```



→ Hiding data behind methods is the central concept of Encapsulation

→ the main advantages of Encapsulation are ① we can acheive Security.

② Enhancement will become very easy.

③ improves modularity of the application.

Ex 3:- 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 A
{
  private int z=30;
}
```

Q) by default what modifier to variables?

Conclusion:-

→ If parent class is not tightly Encapsulated then no child class is Tightly Encapsulated.

5) IS-A Relationship :-

→ It is also known as Inheritance

→ By using extends keyword we can implement IS-A Relationship

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

Ex:-

```
Class P
{
  public void m1()
  {
    ====
  }
}
Class C extends P
{
  public void m2()
  {
  }
}
```

```
Class Test
{
 P.S.V.m(String[] args)
 {
```

Case1: P p = new P();

p.m1(); ✓

p.m2(); X → C.E:- Cannot find Symbol

Symbol : method m2()

location : class P

Case2: C c = new C();

c.m1(); ✓

c.m2(); ✓

* Case3: P P1 = new C();

P1.m1(); ✓

P1.m2(); X C.E:

* Case4: C C1 = new P(); X C.E:- incompatible types

found : P

required : C

Conclusion(1):-

① what ever the parent class has by default available to the child. Hence with the help of child class reference we can call both parent & child class methods.

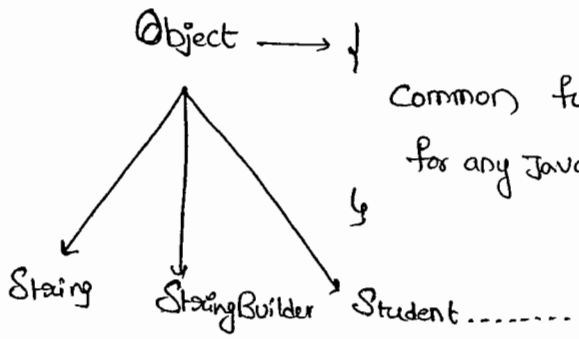
② what ever the child has by default not available to the parent hence on the parent class reference we can call only parent class methods & we cant call child specific methods.

③ parent class reference can be used to hold child class objects by using that reference we can call only parent class methods but we can't call child specific methods.

④ We can't use child class reference to hold parent class objects.

Ex:-

① The common functionality which is required for any java classes is defined in Object class and by keeping that class as super class it's functionality by default available to every java classes.



Ex:- the common functionality which is required for all Exceptions & Errors is defined in Throwable class as Throwable is parent for all Exceptions & Errors, it's functionality will be available automatically to every child not required to rewrite.

Q) Do 'Throwable' has 'Object' as parent class?
so yes

→ Java won't provide support for multiple inheritance but through interfaces it is possible.

ex:- class A extends B, C ✗ C.E:

But interface A extends B C ✓

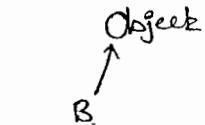
→ Every class in Java is the child class of Object.

→ If our class doesn't extend any other class then only it is the direct child class of Object.

Ex:-
```
Class A
{
}
```


→ If our class extend any other class then our class is not directly child class of Object.

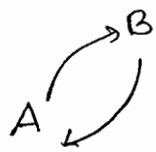
Ex:-
```
class A extends B
{
}
```



(multi level inheritance).

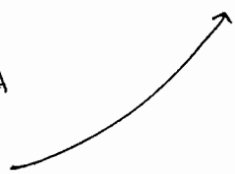
→ Cyclic inheritance is not allowed in Java

Ex:- ①
```
class A extends B
{
}
class B extends A
{
}
```
X



C.E:- cyclic inheritance involving A

②
```
class A extends A
{
}
```



6) Has-A Relationship :-

→ Has-A Relationship is also known as "Composition or Aggregation".

→ There is no specific keyword to implement Has-A Relationship the mostly we are using 'new keyword'.

→ The main advantage of Has-A Relationship is Reusability or (Code Reusability)

Ex:-

```
Class Car
{
  Engine e = new Engine();
}
```

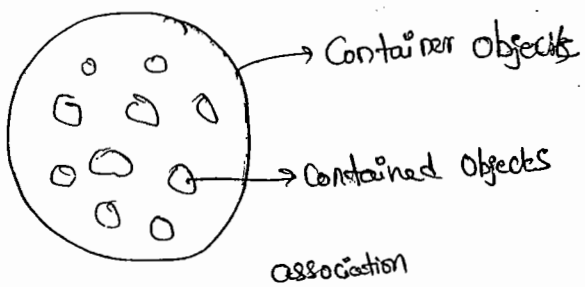
```
Class Engine
{
  // Engine specific functionality
}
```

Class Car has Engine reference.

→ The main disadvantage of Has-A Relationship is it increases dependency b/w the classes and creates maintaince problems.

Composition Vs Aggregation :-

→ In the case of Composition whenever Container objects is destroyed all Contained objects will be destroyed automatically. i.e, without Existing Container Object there is no chance of existing Contained object i.e Container & Contained objects having strong association



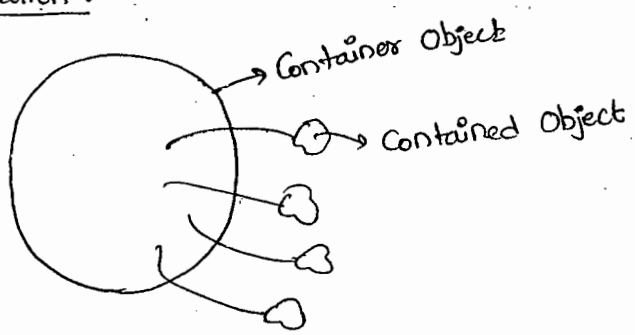
Ex:-

→ University is Composed of Several departments

→ whenever ~~you~~ are closing University automatically all departments will be closed. the relationship blw University Object & department object is Strong association which is nothing but Composition.

Aggregation :-

→ whenever Container Object destroyed, there is no gaurantee of destruction of Contained Objects i.e, without existing Container Object there may be a chance of existing Contained Object. i.e, Container object just maintains References of Contained Objects. This relationship is Called Weak association which is nothing but "Aggregation".



Ex:-

→ Several proofices will work in the department

→ whenever we are closing the department Still there may be a chance of existing paroofesers. the Relationship blw department & professor is Called weak association which is nothing but Aggregation.

```
public void m1(int i)
{
  S.o.pln("int-arg");
}
public void m1(float f)
{
  S.o.pln("float-arg");
}
p.s.v.m(———)
{
  Test t = new Test();
    t.m1();   // no-arg
    t.m1(10); // int-arg
    t.m1(10.5f); // float-arg
}
```

*→ In overloading method resolution always takes care by compiler based on reference type. Hence overloading is also considered as compiletime polymorphism (or) static polymorphism (or) Early binding

→ In overloading reference type will play very important role & runtime object will be dummy.

Case1:-

*Automatic promotion in overloading:-

→ In overloading method resolution, if the matched method with specified argument type is not available then compiler won't raise

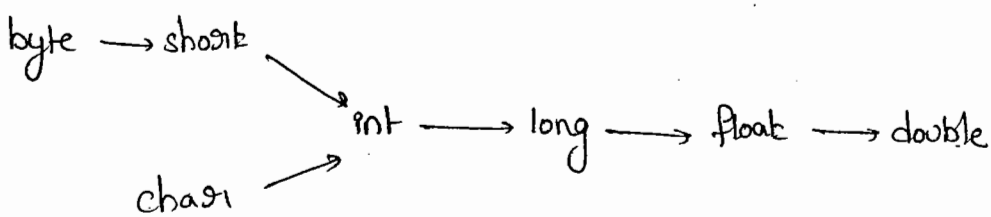
any Error immediately. first it promotes that arrguement to the next Level and checkes for matched method.

→ If the matched method is available then it will be Considered and if it is not available then Compiler once again promotes this arrguement to the next Level.

→ This process will be Continued untill all possible promotions after Completing all promotions Still if the matched method is not available then only we will get C.E.

→ this ~~fail~~ is Called Automatic promotion in overloading.

→ The following are Various possible promotions in Overloading.



Case 1% -

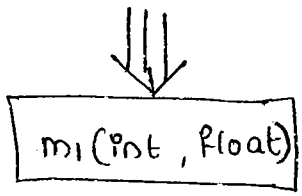
Ex:-

```
Class Test
{
  public void m1(int i)
  {
    S.o.pln("int-arg");
  }
  public void m1(float f)
  {
    S.o.pln("float-arg");
  }
  P.S.V.m(String[] args)
  {
    Test t = new Test();
```

Method Signature :-

→ method Signature Consists of name of the method & arrguement-List.

Eg:- public void m1(int i, float f)



→ In Java return type is not part of method Signature.

→ Compile will always use method Signature while resolving method Calls

→ with in the same class Two methods with the same Signature not allowed. Otherwise coe will get CompiletimeError.

Eg:-
```
class Test
{
  public void m1(int i)
  {
  }
  public int m1(int i)
  {
    return 10;
  }
}
```

m1(int) is the method Signature.

```
Test t = new Test()
t1.m1(10);
```

C.E:- m1(int) has already defined in Test

Overloading

Overloading:-

→ Two methods are said to overloaded iff method names are same but arguements are different.

→ Lack of overloading in 'C' increases complexicity of the program. In C, language if there is a change in method arguement type. Complusary we should go for new method name.

Ex!- abs() ⟶ int
labs() ⟶ long
fabs() ⟶ float

→ But in Java two methods having the same name with different arguements is allowed & these methods are considered as overloaded methods. Ex!- abs(int)
abs(long)
abs(float)

→ Having overloading concept in Java simplifies the programming

Ex!-
```
Class Test
{
  Public void m1()
  {
    S.o.pln("no-arg");
  }
```

Case1:-
→ In Overloading most specific version will get highest priority.
what does it mean?

Case2:-

Ex!-

```
Class Test
{
 Public void m1(StringBuffer sb)
 {
   S.o.pln("StringBuffer-arg");
 }
 Public void m1(String s)
 {
   S.o.pln("String-version");
 }
 public . s . v . m (——)
 {
   Test t = new Test();
   t.m1(new SB("durga")); //StringBuffer-args
   t.m1("durga"); // String version
   t.m1(null); // C.E!- Reference m1() is ambiguity.
 }
}
```

By default 'String' constant of String class object type like integral constant of 'int' floating literal " "double

```
t.m1('a'); // int-arg
t.m1(10l); // float-arg
t.m1(10.5); x C.E:-
}
}
```

Cannot find Symbol
Symbol : method m1(double)
location : class Test

Case 2 :-

→ In overloading method resolution child-argument will get more priority than parent argument.

Ex:-

```
Class Test
{
① Public void m1(Object o)
  {
    S.o.pln("Object version");
  }
② Public void m1(String s)
  {
    S.o.pln("String version");
  }
P.S.V.m(—)
{
  Test t = new Test();
  t.m1(new Object()); // Object-version
  t.m1("durga"); // String-version (suppose ② statement takes // the o/p is Object
  t.m1(null); // String-version
}
}
```

Object
↑
String

→ Hence overridding is also known as "runtime polymorphism (or) dynamic polymorphism (or) Late binding".

→ Overriding method resolution is also known as "Dynamic method dispatch".

Rules for Overridding :-

① In overriding method names & arguements must be matched i.e, method Signatures must be matched.

② In overriding return type must be matched, But this rule is applicable untill 1.4 version, from 1.5 version onwards Co-variant (ఒక కాని బదులు మరోకటి) return types are allowed. according to this, child method return-type need not be same as parent method return-type. its child classes also allowed.

Ex:

```
Class P
{
  Public Object m1()
  {
    return null;
  }
}
Class C extends P
{
  Public String m1()
  {
    return null;
  }
}
```

It is valid in 1.5v, But invalid in 1.4v

Ex:-

parent method return type	Object	Number	String	double
child method return types	String ✓	Integer ✓	Object ✗	int ✗

→ Co-variant return type concept is applicable only for object type but not for primitive types.

③ We can't override parent class final method. But we can use it as it is

④ private methods are not visible in child classes Hence Overridding concept is not applicable for private methods.

⑤ → Based on our requirement we can declare the same Parent class private method in child class also it is valid but it is not overriding.

ex:-

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

not overriding

✓ ⑯ Compiles fine but not overriding.

→ For parent class abstract methods we should override in child class to provide implementation.

⑰ → We can override parent class non-abstract method as abstract in child class to stop parent class method implementation availability to the child classes.

eg):-

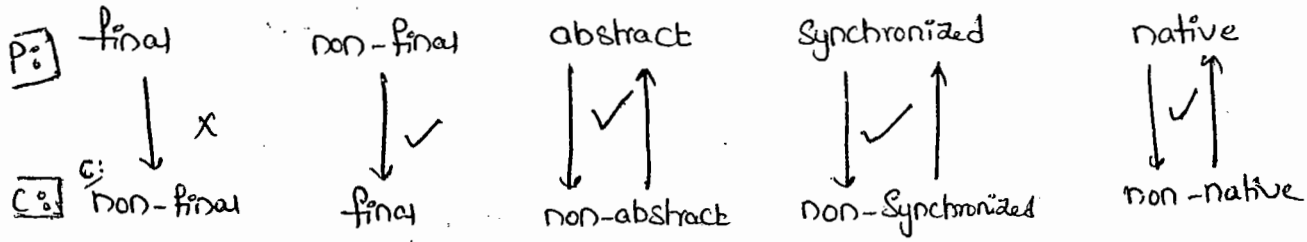
```
class P
{
  public void p()
  {
  }
}
abstract class C extends P
{
  public abstract void p();
  {
  }
}
```

→ The following modifiers won't play any restrictions in overriding

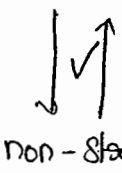
① native

② Synchronized

③ strictfp



→ strictfp



→ while overriding we can't decrease scope of the modifiers but we can increase the following are various acceptable overridings

private < default < protected < public

p) Public → public
protected → protected/public
default → default/protected/public
private → private method can't be override

Eg:-
```
Class P
{
 public void m1() { }
}
Class C extends P
{
 protected void m1() X
 {
 }
}
```
C.E:- m1 in C can't override in C

→ This rule is applicable while implementing interface methods also.

→ whenever we are implementing any interface method compulsary it should be declared as public. because every interface method is public by default.

Ex:-
```
interface Intef
{
  void m1();
}
Class Test implements Intef
{
  void m1()   X C.E:-
  { }
}
```
if we declare public we won't get any C.E

(#) → If child class method throws some checked Exception then Compulsary Parent class method should throw the same checked Exception or its parent class Exception. otherwise we will get C.E.

→ But there is no rule for unchecked Exception.

Ex:-①
```
Class P
{
  Public void m1()
  {
  }
}
Class C extends P
{
  Public void m1() throws Exception   ✗
  {
  }
}
```
C.E!- m1() in C can't override m1() in P; Overridden method doesnot throw Exception.

Ex②:-

ⓐ ✓
P: public void m1() throws IOException
C: public void m1()

ⓑ ✗
P: public void m1()
C: public void m1() throws IOException

ⓒ ✓
P: public void m1() throws Exception
C: public void m1() throws IOException

ⓓ ✗
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

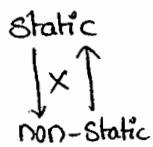
⑧ P: public void m1()

✓ C: public void m1() throws AE, NPE

Overriding w.r.t Static method :-

→ We can't override a static method as non-static.

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

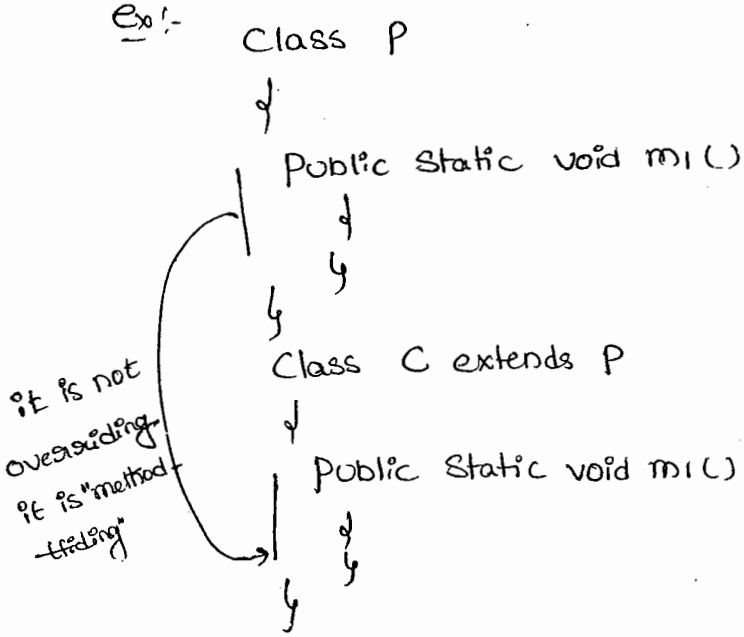


C.E:- m1() is Can't override m1() in P;
overridden method is static.

→ Similarly, we can't override non-static method as static

→ If both parent & child class method ~~class~~ are static then we wont to get any C.E it seems to be overriding is happen, but it is not overriding. it is "Method Hiding".

Ex:-



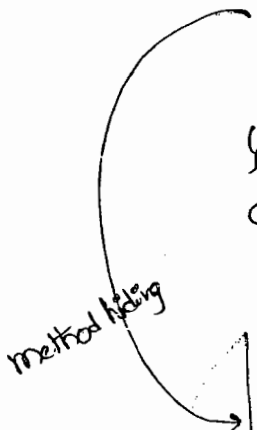
Method Hiding :-

→ All rules of Method Hiding are Exactly Same as Overriding Except the following difference.

Method hiding	Overriding
1) Both methods should be static	1) Both methods should be non-static
2) Method resolution takes care by Compiler based on Reference type.	2) Metho resolution always takes care by JVM based on Runtime object.
3) It is considered as Compiletime polymorphism or static polymorphism or early Binding	3) It is considered as Runtime 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()
  {
    So.pln("child");
  }
}
```




```
Class Test
{
  p.s.v.m(—)
  {
    P  p = new P();
       p.m1();   → parent

    C  c = new C();
       c.m1();   → child

    P  p1 = new C();
       p1.m1();  parent
  }
}
```

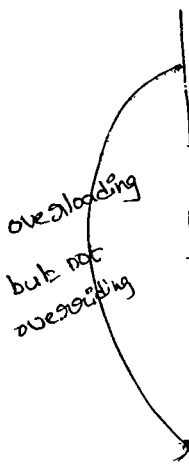
→ If both methods are non-static then it will become overriding in this case the o/p is: Parent
Child
Child

Overriding w.r.t Var-arg methods :-

→ We can't override a Var-arg method with general method. If we are taying to override it will become overloading but not overriding.

→ A Var-arg method should be overriden with Var-arg method only.

Ex:-




```
Class P
{
  public void m1(int... i)
  {
    S.o.pln("parent");
  }
}
Class C extends P
{
  public void m1(int i)
  {
    S.o.pln("child");
  }
}
Class Test
{
  p.s.v.m(——→
  {
    P p = new P();
    p.m1(10); // Parent
    C c = new C();
    c.m1(10); //child
    P p1 = new C();
    p1.m1(10); //parent
  }
}
```

→ If both parent & child class methods are Var-arg Then it will becomes overriding in this case o/p is parent child ~~parent~~ child

Overriding w.r.t Variables :-

→ Overriding Concept is not applicable for variables.

→ Variable resolution always takes care by Compiler based on reference type. Runtime object won't to play any role in variable resolution.

Eg:-

```
Class P
{
  int x=888;
}
Class C extends P
{
  int x=999;
}
Class Test
{
  P.S.V.m(------>)
  {
    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.
  }
}
```

~~both static~~

both static o/p 888 | both instance o/p 888 | one static & one inst o/p 888

→ whether the variables are static or non-static there is no change in result.

difference b/w overloading & overriding:-

Property	Overloading	Overriding
① method names	must be same	must be same
② arguements	must be different (at least order)	must be same (including order).
③ method signature	must be different	must be same.
④ return type	No restrictions	must be same until 1.4V but from 1.5V onwards Co-variant return types are allowed.
⑤ private, static & final methods	Can be overloaded	Can't be overridden
⑥ access modifiers	No restrictions	Scope we can't decrease the scope.
⑦ throws clause	No restrictions	Size & level of checked exceptions we can't increase but we can decrease. 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 Also known as	Compile-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) & arguements (must be diff.) All remaining teams like (Return type, throws clause, Access modifiers e.t.c) are not required to check.

→ But in overriding we have to check each & every thing.

Q) Consider the following method declaration in parent class which of the following methods allowed in child class?

P: public void m1(int i) throws IOException

Overriding ① public void m1(int i)

overloading ② public void m1() throws Exception

overloading ③ public static int m1(double d) throws IOException

C.E ✗ ④ public int m1(int i)

C.E ✗ ⑤ public synchronized void m1(int i) throws Exception

overloading ⑥ public static void m1(int... i) throws Exception

C.E ✗ ⑦ public native abstract void m1().throws Exception.

Polymorphism

↳ poly → many

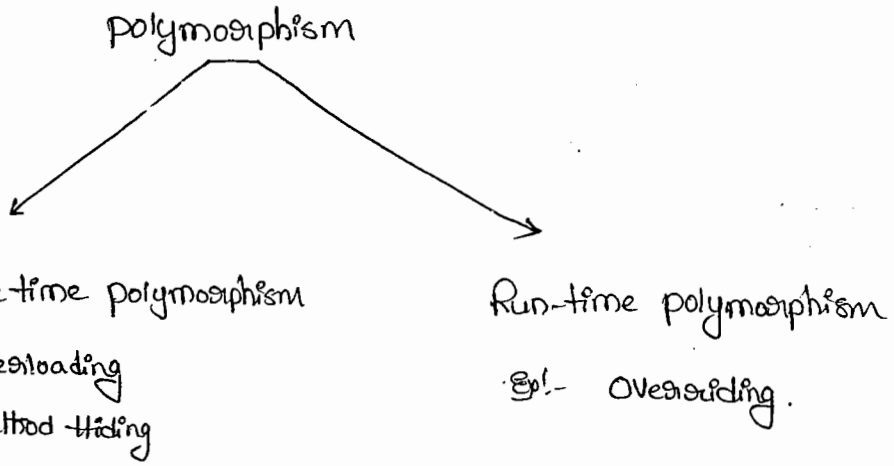
morphs —means→ forms

i.e polymorphism means many forms

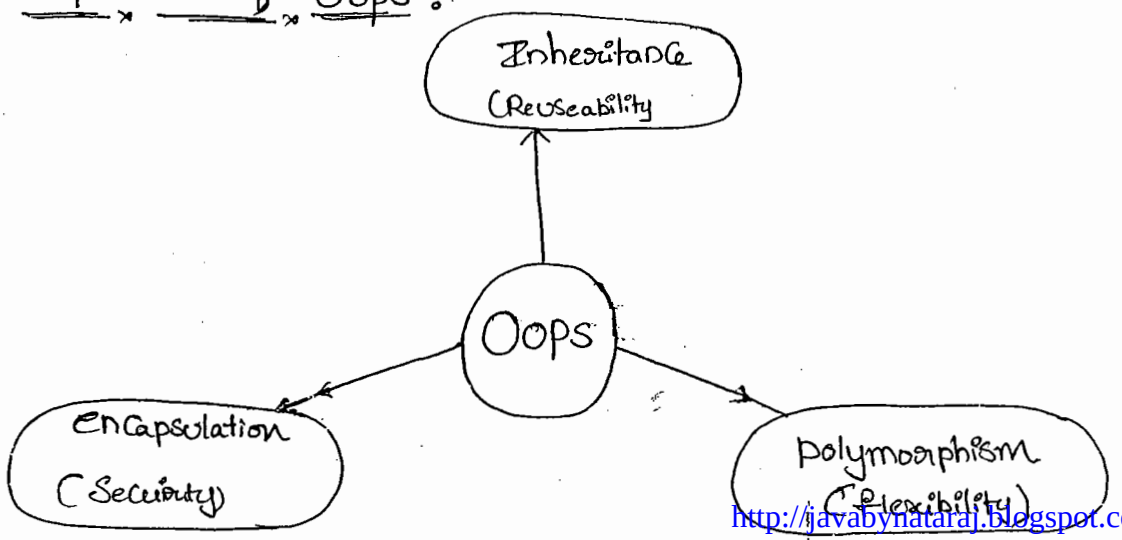
→ we can use same name to represent multiple forms in polymorphism.

Ex:- → In overriding we can have a method with one type of implementation in parent, but different type of implementation in child class.

→ There are 2 types of polymorphism.



3 pillars of Oops :-



funny diffination of polymorphism :-

→ A boy uses the coord FRIENDSHIP to Starts LOVE, but girl uses the Same coord to ~~close~~ ends. Same coord but different attitudes. This behaviour is nothing but polymorphism.

3) Static Control flow :-

Ex:-

```
Class Base
{
① Static int x=10;                                   7
② Static
  {
    m1();                                            8
    S.o.pln("FSB");                                  10
  }
③ Public Static void main(String[] args)
  {
    m1();                                            13
    S.o.pln("main method");                          15
  }
④ public static void m1()
  {
    S.o.pln(y);                                      9, 14
  }
⑤ Static
  {
    S.o.pln("SSB");                                  11
  }
⑥ Static int y=20;                                   12
}
```

x = 0 [RIWO]
y = 0 [RIWO]
x = 10 [R&W]
y = 20 [R&W]

O/P:-
0
FSB
SSB
20
main method

Process:-

→ when ever we are taying to execute a Java class first that .class file should be loaded, at the time class loading the following actions will be performed automatically.

① identification of Static members from Top to bottom. (1 to 6)

② Execution of Static variable assignments & Static blocks from top to bottom (7 to 12)

③ Execution of main method. (13 to 15)

Read Indirectly write only state (RIWOS)

→ If a variable is in Read indirectly write only state then we Can't perform read operation directly otherwise we will get Compile-time Error saying "Illegal-forward-Reference".

Ex:-

```
Class Test
{
①  Static int x=10;
②  Static
   {
      S.o.pln(x);
      System.exit(0);
   }
}
```

o/p :- 10

```
Class Test
{
①  Static
   {
      S.o.pln(x);
   }
②  Static int x=10;
}
```

C.E:- Illegal-forward reference.

Static block :-

→ At the time of class loading if we want to perform any activity we have to define that activity inside Static block because Static blocks will be Executed at the time of class loading.

→ with in a class we can take any no. of Static blocks but all these Static blocks will be Executed from top to bottom.

Ex(1) :-

→ After loading JDBC driver class we have to register driver with driverManager but every Driver class Contains a Static block to perform this activity at the time of Driver class loading automatically we are not responsible to perform register Explicitly.

```
Ex:  Class Driver
     {
       Static
       {
         Register this Driver with DM
       }
     }
```

Ex(2) :- Advantage :

→ At the time of class loading (Compulsory) we have to load the Corresponding native libraries hence we can define this step inside Static block.

```
Ex:-  Class Native
      {
        Static
        {
          System.loadLibrary("native Library path");
        }
      }
```

Static Control flow in parents child classes :-

```
Class Base
{
① Static int x=10; ⑫
② Static
  {
    m1(); ⑬
    S.o.pln("Base SB"); ⑮
  }
③ Public static void main(——)
  {
    m1();
    S.o.pln("Base main");
  }
④ public static void m1()
  {
    S.o.pln(y); ⑭
  }
⑤ Static int y=20; ⑯
}
Class Derived extends Base
{
⑥ Static int i=100; ⑰
⑦ Static
  {
    m2(); ⑱
    S.o.pln("DFSB"); ⑳
  }
⑧ Public static void main(——)
  {
    m2(); ㉓
    S.o.p("Derived main"); ㉕
  }
```

⑨ Public Static void m2()
{
S.o.pln(j); ⑲ ㉔
}

⑩ Static
{
S.o.p("DSSB"); ㉑
}

⑪ Static int j = 200; ㉒
}

> java Derived

O/P:-
0
Base SB
0
DFSB
DSSB
200
Derived main

x = 0 [RI W0]
y = 0 [RI W0]
i = 0 [RI W0]
j = 0 [RI W0]
x = 10 [R W]
y = 20 [R & W]
i = 100 [R & W]
j = 200 [R & W]

> java Base

0
Base SB
20
Base main.

process:-

> javac Derived.java

Base.class Derived.class

> java Derived

(1) Identification of Static member from parent to child [1 to 11]

(2) Execution of Static variable assignments & static blocks from parent to child [12 to 22]

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

(because main() method of parent class is overriding in child class, then child class main() method executed)

Process :-

→ whenever we are trying to load child class then automatically parent class will be loaded to make parent class members available to the child class. Hence whenever we are executing child class the following is the flow with respect to static members step.

(1) Identification of static members from parent to child

(2) Execution of static variable assignments & static blocks from parent to child.

(3) Execution of only child class main method. [If the child class wont contain main method then automatically parent class main() method will be executed].

Note :-

When ever we are loading child class automatically parent class will be loaded. But when ever we are loading parent class child class wont be loaded.

Instance Control flow :-

```
class Parent
{
③  int x=10; ⑨
④  {
     m1(); ⑩
     S.o.p("FIIB"); ⑫
   }
⑤  Parent()
   {
     S.o.pln("Constructor"); ⑮
   }
①  Public static void main(String[] args)
   {
②    Parent p = new Parent();
     S.o.pln("main");
   }
⑥  Public void m1()
   {
     S.o.pln(y); ⑪
   }
⑦  {  ——→ instance block
     S.o.pln("SIIB"); ⑬
   }
⑧  int y=20; ⑭
}
```

x = 0 [R I W O]
y = 0 [R I W O]
x = 10 [R W]
y = 20 [R W]

O/P :-

```
0
FIIB
SIIB
Constructor
main
```

Process :-

→ whenever we are Creating an Object the following Sequence of events will be performed automatically.

(1) Identification of instance member from Top to bottom [1 to 8]

(2) Execution of instance variable assignments & instance blocks from top to botom [9 - 14].

(3) Execution of Constructor [15]

* Note :-

→ Static Control flow is only one time activity and it will be performed at the time of class loading But instance control flow is not one time activity for every object Creation it will be executed.

Instance Control flow from parent to child :-

```
Class Parent
{
③ int x=10; ⑮
④ {
    m1(); ⑯
    S.o.pln("parent"); ⑱
  }
⑤ Parent()
  {S.o.pln(" parent Constructor"); ⑳
  }
```

```
① public static void main(———)
{
② Parent p = new Parent();
   S.o.pln("child main");   ㉑
}
⑥ public void m1()
{
   S.o.pln(y);   ⑰
}
⑦ int y=20;   ⑲
}
Class child extends Parent
{
⑩ int i=100;   ㉒
⑪ {
     m2();   ㉓
     S.o.pln("CIIB");   ㉕
   }
⑫ Child()
   {
     S.o.pln("child constructor");   ㉘
   }
⑧ public static void main(———)
   {
⑨   child c = new child();
     S.o.pln("child main");   ㉙
   }
⑫ public void m2()
   {
     S.o.pln(j);   ㉔
   }
```

```
0
Parent
Parent Constructor
0
CIIB
CSIIB
Child Constructor
Child main.
```

```
{
    S.o.pln("CSIIB");  (26)
}
int j = 200;  (27)
}
```

Process :-

→ when ever Sequence of we are creating child class object the following Sequence of ~~execute~~ events will be performed automatically.

(1) Identification of instance member from parent to child.

(2) Execution of instance variable assignments & instance blocks only in parent class.

(3) Execution of parent class Constructor.

(4) Execution of instance variable assignments & instance blocks only in child class.

(5) Execution of child class Constructor.

>java child

>java parent

Constructors :-

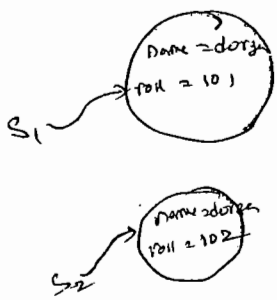
→ Object Creation is not enough Compulsary we should perform initialization then only that Object is in a position to provide response properlly.

→ when ever we are creating an Object some peace of the code will be executed automatically to perform initialization this place of code is nothing but constructor. Hence the main objective of Constructor is to perform initialization for the newly created Object.

Eg:-

```
Class Student
{
① int rollno;
② String name;
  Student (String name, int rollno)
  {
      this.name = name;
      this.rollno = rollno;
  }
  Public Static void main (String[] args)
  {
    Student s1 = new Student ("durga", 101);
    Student s2 = new Student ("raghu", 102);
  }
}
```

null.



Instance block Vs Constructor :-

→ At the time of object Creation if we want to perform initialization of instance variable then we should go for Constructor.

→ Other than initialization activity if we want to perform any activity at the time of object Creation then we should go for instance block.

→ We Can't replace Constructors with instance block because Constructor Can take arguement where as instance block Can't take agguements.

→ Similarly we Can't replace instance block with Constructor because a class Can Contain more than one Constructor. if we want to replace instance block with Constructor then in every Constructor we have to write instance blocks Code because at runtime which Constructor will be Called we can't Expect. It results duplicate & Creates maintaince problems.

Ex :-

```
class Test
{
    static int Count = 0;

    Test()
    {
        Count++;
    }

    Test(int i)
    {
        Count++;
    }

    p.s.v.m(---)
    {
        Test t1 = new Test();
        Test t2 = new Test(10);
    }
}
```

Only once required

if we Create instance

Rules to define Constructors :-

1) The name of the class & name of the Constructor must be matched.

2) Return type Concept is not applicable for Constructor even void also.

By mistake if we declare return type for the Constructor we wont get any Compiletime (or) runtime Errors, because Compiler treats it as method.

Ex:-
```
class Test
{
    void Test()        → It is a normal method but not Constructor
    {
    }
}
```

It is legal (for stupid to have a method whose name is exactly same as class name).

(3) The only applicable modifiers for Constructors are "public, private, protected, <default> [PPPD]", if we are trying to use any other modifier we will get Compile-time Error saying "modifier xxxx is not allowed here".

xxxx → static/ final/ Strictfp ...

Ex:-
```
class Test
{
    final Test()
    {
    }
}
```
✗ C.E: modifier final is not allowed here

Singleton classes :-

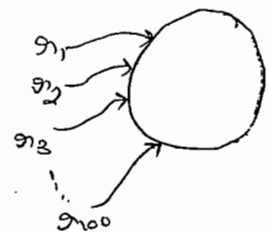
→ for any java class if we are allowed to create only one object such type of class is called "Singleton class".

Ex:- Runtime, ActionServlet (Structs 1.x)

BusinessDeligate (EJB), ServiceLocator (EJB) ---- e.t.c

→ The main advantage of Singleton is, instead of creating a separate object for every requirement we can create a single object and reuse the same object for every requirement this approach improves memory utilization & performance of the system.

```
Runtime r1 = Runtime.getRuntime()
Runtime r2 = Runtime.getRuntime()
                  ↳ Class   ↳ Static method
   :
Runtime r100 = Runtime.getRuntime()
```



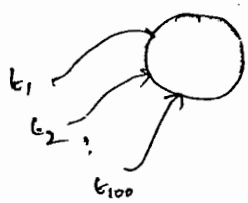
Creation of our own Singleton Class :-

→ We can create our own singleton classes also for this we have to use private constructor & factory method.

```
Ex:- class Test
     {
       private static Test t;

       private Test()
       {
       }
       public static Test getInstance()
       {
```

```
        if (t == null)
        {
          t = new Test();
        }
        return t;
    }
    public Object clone()
    {
      return this;
    }
}
```




```
Test t1 = Test.getInstance();
Test t2 = Test.getInstance();
  :
Test t100 = Test.getInstance();
Test t101 = Test.clone();
```

factory method:-

→ By using class name if we call any method & return same class object. then that method is consider as factory method.

Ex:-

```
Runtime r = Runtime.getRuntime();        → factory method
DateFormat df = DateFormat.getInstance();
                                  ↳ factory method.
Test t = Test.getInstance();
                          ↳ factory method
```

→ Similarly we can Create Doubleton, Tripleton ----- xxxton 120

Classes.

How to Create Doubleton class :-

Ex:-

```
Class Test
{
 private Static Test t1;
 private Static Test t2;
 private Test()
 {
 }
 Public Static Test getInstance()
 {
  if (t1 == null)
  {
    t1 = new Test();
    return t1;
  }
  else
  if (t2 == null)
  {
    t2 = new Test();
    return t2;
  }
 else
  {
    if (math.random() < 0.5)
    return t1;
   else
    return t2;
  }
}
```

Rule :-

Default Constructor :-

→ If we are not writing any Constructor then Compiler will always generate default Constructor.

→ If we are writing atleast one Constructor then Compiler won't generate default Constructor.

→ Hence a class can contain either programmer written Constructor (or) Compiler generated Constructor but not both simultaniously.

Prototype of default Constructor :-

1) It is always no argument Constructor.

2) The access modifier of default Constructor is same as class modifier but this rule is applicable public & <default>.

3) If Contains only one line, It is a no argument Call to Super class Constructor.

```
Test()
{
Super();
}
```

Programmers Code	Compiler generated Code
(1) `class Test { }`	(1) `class Test { Test() { super(); } }`
(2) `public class Test { }`	(2) `public class Test { public Test() { super(); } }`
(3) `Class Test { void Test() { } }` → It is not a constructor. It is a normal method.	(3) `Class Test { Test() { super(); } void Test() { } }`
(4) `Class Test { Test() { } }`	(4) `Class Test { Test() { super(); } }`
(5) `Class Test { Test() { this(10); } Test(int i) { } }`	(5) `Class Test { Test() { this(10); } Test(int i) { super(); } }`

(6)
```
Class Test
{
  Test(int i)
  {
    super();
  }
}
```

(c)
```
Class Test
{
  Test(int i)
  {
    super();
  }
}
```

super & this :-

→ The first line inside a constructor should be either super(arg) this().

→ If we are not writing any thing compiler will always places super().

Case (i) :-

We have to keep either super() (or) this() only as the first line of the constructor.

```
class Test
{
  Test()
  {
    S.o.p("Hi");
    super();   ✗ → C.E:- Call to super must be first statement in constructor.
  }
}
```

Case (ii) :-

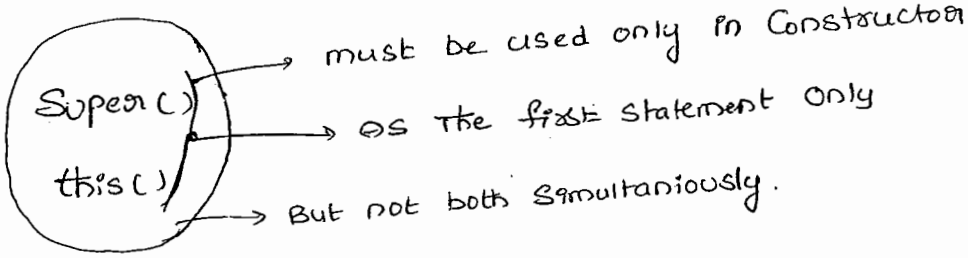
With in the constructor we can use either super() or this() but not both simultaniously.

```
Class Test
{
  Test()
  {
    super();   ✓
    this();    ✗ → C.E:- Call to this must be first statement in the constructor
  }
}
```

Case (iii) :-

→ We can use Super & this only inside Constructor if we are using any where else we will get Compile time error.

Ex:
```
class Test
{
  public void m1()
  {
    super();   X  → C.E:- Call to super must be first statement in the Constructor
    S.o.pln("Hi");
  }
}
```



this() :- To call Current class Constructors

super() :- To call parent class Constructors

Compiler provides default super() but not this().

super() this()	super this
(1) These are Constructor calls	1) These are key words to reference Super & Current class instance members
(2) We should use only in Constructors	2) We can use any where Except in static area.

Ex:-

```
Class Test
{
  p.s.v.m1()
  {
    S.o.pln(super.hashCode());  X
  }
}
```

↳ CE:- NON-static variable super can't be referenced from a static context

Constructor overloading:-

→ A class can contain more than one constructor with same name but with different arrguements & these constructors are considers as overloaded constructors.

Ex:-

```
Class Test
{
  Test(double d)
  {
    this(10);
    S.o.pln("double-args");
  }
  Test(int i)
  {
    this();
    S.o.pln("int-args");
  }
  Test()
  {
    S.o.p("No-args");
  }
  p.s.v.m(——)
  {
```

Test t_1 = new Test(10.5); ⟶ No-args
int-args
double-args

Test t_2 = new Test(10) ⟶ No-args
int-args

Test t_3 = new Test() ⟶ No-args

*** → Inheritance & overriding Concepts are not applicable for Constructors.

**** → Every class in java including abstract class also can contain Constructor. But interfaces can't have the constructors.

Class Test	abstract Class Test	interface Test
{ Test() { } } ✓	{ Test() { } } ✓	{ Test() { } } ✗

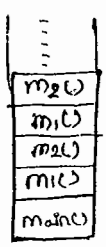
→ Case(i):-

→ Recursive method Call is always Runtime Exception where as Recursive Constructor invocation is a Compiletime Error.

ex:-

```
Class Test
{
  p.s.v.m1()
  {
    m2();
  }
  p.s.v.m2()
  {
    m1();
  }
  p.s.v.m(——)
  {
    S.o.p("Hello");
    m1();
  }
}
```

RE:- StackOverFlowError




```
Class Test
{
  Test()
  {
    this(10);
  }
  Test(int i)
  {
    this();
  }
  p.s.v.m(——)
  {
    S.o.pln("Hello");
  }
}
```

C.E:- Recursive Constructor invocation.

Case(ii) :-

Ex:-

```
class P
{
}
class C extends P
{
}
```

```
class P
{
  P()
  {
  }
}
class C extends P
{
}
```

```
class P
{
  P(int i)
  {
  }
}
class C extends P
{
  C()
  {
    super();
  }
}
```

C.E:- Can't find Symbol
Symbol : Constructor P()
location : class P.

Note:-

→ if the parent class contains some Constructors then while writing child class we have to take special care about Constructors.

→ when ever we are writing any arguement Constructor it is highely recommended to write no arguement Constructor also.

Case(iii) :-

→ If parent class Constructor throws Some Checked Exception Compulsary child class constructor should throw Same checked Exception or its parent other wise the code won't Compile.

```
class P
{
  P() throws IOException
  {
  }
}
```

```
class C extends P
{
}
```

C.E:- unreported Exception java.io.IOException must be declared

Ex:-
```
Class P
{
  P() throws IOException
  {
  }
}

Class C extends P
{
    C() throws IOException/Exception
    {
    }
}
```

Q) which of the following is True?

① Every class Contains Constructors ✓

② only Concreate Classes Can Contains Constructors but not abstract Classes ✗

③ the name of the Constructor need not be Same as class name ✗

④ return type is applicable for the Constructor ✗

⑤ the only applicable modifiers for Constructors are public & default ✗

⑥ If we are trying to declare return type for the Constructor we will get Compiletime Error ✗.

⑦ Compiler will always generate default Constructor ✗

⑧ the access modifier of the default Constructor is always default ✗.

⑨ the first Line inside every Constructor should be Super ✗.

⑩ the first Line " " should be Super(or this(), ✓
if we are not writing any thing Compiler will always place this() ✗

(11) Interface can contains Constructor ✗

(12) Both overloading & overriding concepts are applicable for Constructor ✗.

(13) Inheritance concept is applicable for Constructor ✗

Type-Casting

Type-Casting:-

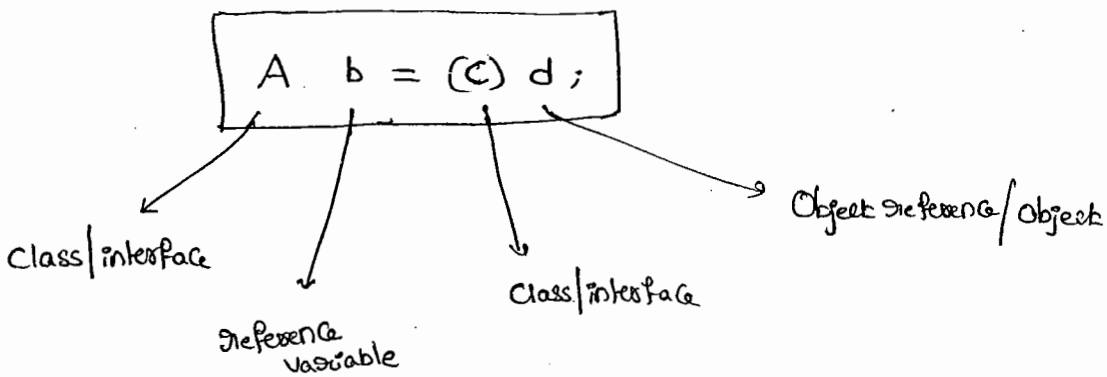
→ Parent class reference can be used to hold child class object

Eg:- Parent p = new child();

→ Similarly, interface reference can be used to hold implemented class object.

eg: Runnable r = new Thread();

Syntax:-



Compiler rule(1):-

→ C & type of d must have some relationship (either parent to child (or) child → parent (or) same type) otherwise we will get compiletime Error saying "inconvertable types found d type but required C type".

eg(1) :-

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

StringBuffer sb = (StringBuffer) o;
```

eg(2) :-

```
String s = new String("durga");

SB sb = (SB) s;
```

C.E :- inConvertable types

found :- java.lang.String

required : java.lang.SB

Compilerchecking rule 2 :-

→ C must be either same or derived type of A otherwise we will get Compile time Error saying "incompatable types"

found : C

required : A

Ex(1) :-

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

String s = (String) o;
```

Ex(2) :-

```
String s = new String("durga");

StringBuffer sb = (Object) s;
```

C.E :- incompatable types

found : Object

required : SB

Runtime checking

Rule 3 :-

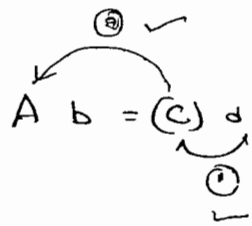
→ The underlying object type of 'd' must be either same or derived type of C. otherwise we will get runtime Exception saying "ClassCastException".

Ex:-

① Object o = new String("durga");

SB sb = (SB) o; ✗

Rule ① ✓
② ✓
③ ✗ (R.E):- CCE

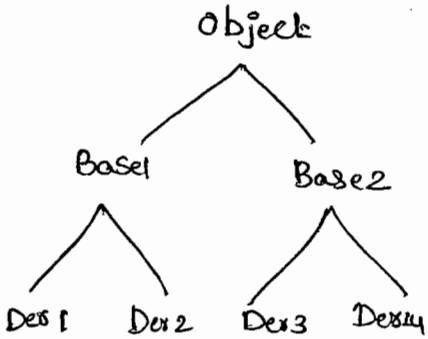


② Object o = new String("durga");

String s = (String) o; ✓

Rule ① ✓
② ✓
③ ✓

Ex:-

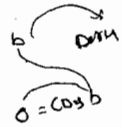


Ex:- ① Base2 b = new Der4();

② Object o = (Base2) b;

✗ ③ Object o = (Base1) b;

④ Base2 b1 = (Base2) o; →



✗ ⑤ Base1 b3 = (Der1)(new Der2());

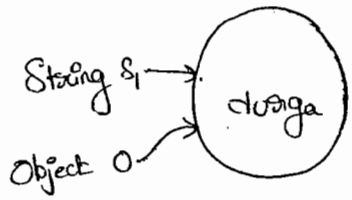
C.E:- inconvertiable types
found : Base2
required : Base1

(C.E:- inconvertiable types
found : Der2
required : Der1)

→ Strictly Speaking in type-casting just we are converting only type of object but not underlying object itself

Eg:-

```
String s1 = new String("durga");
Object o = (Object) s1;
S.o.pln(s1 == o); true
```



Ex:-

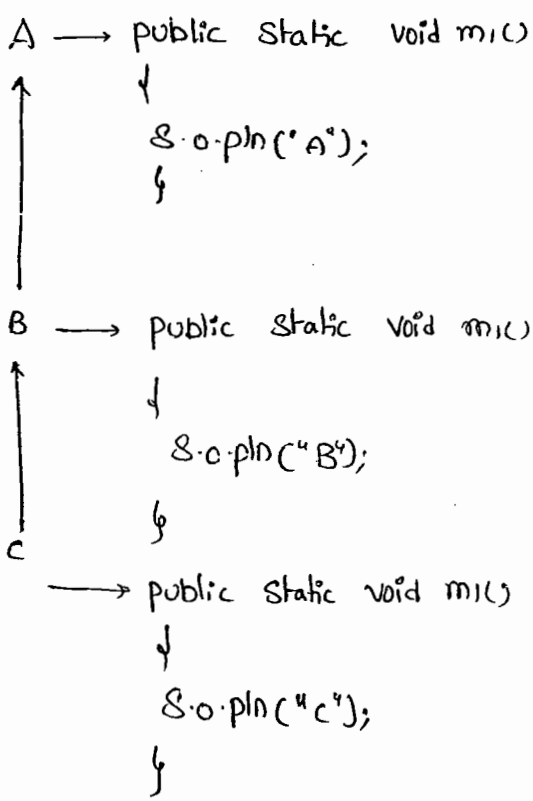
```
A ──→ public void m1()
      {
        S.o.pln("A");
      }

B ──→ public void m1()
      {
        S.o.pln("B");
      }

C ──→ public void m1()
      {
        S.o.pln("C");
      }
```

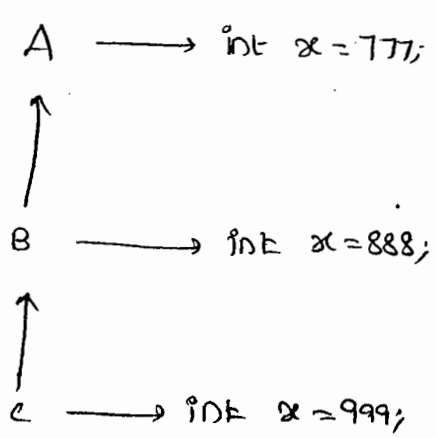
```
C c = new C();
c.m1();  o/p C
((B)c).m1();  → C  ──→ B b = new C();
                       b.m1();
((A)c).m1();  → C  ──→ A a = new C();
                       a.m1();
```

Eg 2:-



* C c = new C();
  c.m1(); // C
* ((B)c).m1(); // B
* ((A)c).m1(); // A

Eg 3 :-



C c = new C();

S.o.pln(c.x); 999

S.o.pln(((B)c).x); 888

S.o.pln(((A)((B)c)).x); 777

(because the overriding concept is not applicable for variables).

→ if we declare all variables as static then there is no chance to change the o/p.

Note!-

→ whether the variable is static or instance variable resolution should be done based on reference type but not based on runtime object.

Coupling

Coupling :-

→ The degree of dependency b/w the components is called "Coupling"

Ex!-

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

```
class B
{
static int j = C.m1();
}
```

```
class C
{
p.s.v.m m1()
{
return 0.k;
}
}
```

```
Class D
{
static int k=10;
}
```

→ the above components are said to be tightly coupled with each other. Tightly coupling is not recommended because it has several serious disAdvantages.

(1) With out effecting ~~any~~ remaining component we can't modify any component.

Hence, enhancement will become difficault.

2→ it reduces maintainability.

3→ It doesn't promote reusability.

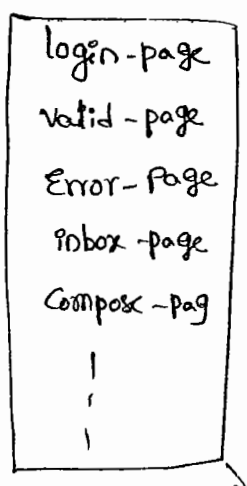
→ Hence it is highly recommended to maintain loosely coupling & dependency b/w the components should be as less as possible.

Cohesion

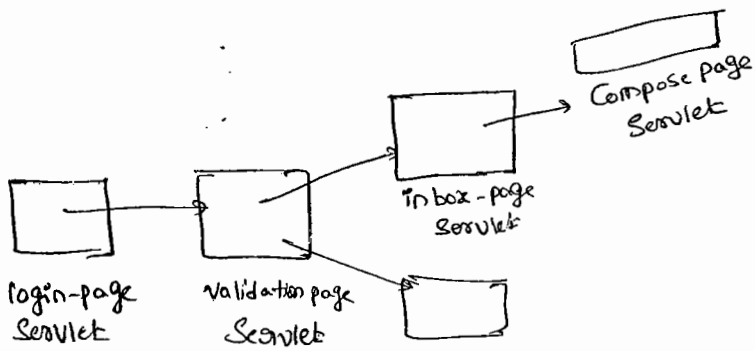
Cohesion :-

→ for every component a clear well-defined functionality we have to define, such type of component is said to be fallow high-cohesion

Ex:-



TotalServlet → Low-cohesion



→ High-Cohesion

→ High-Cohesion is always a good programming practice which has several Advantages.

(1) with out effecting remaining components we can modify any component hence enchancement will become very easy

② It improves maintainability of the application

③ It promotes reuseability of the code.

Ex:-

→ Where ever validation is required we can reuse the same validate Servlet without rewriting.

Note:-

Loosely Coupling & high-Cohesion are good programming practices.

=xox=