OOPs Concepts

Date __/__/____
Page ____

Modular-II

1. Data Hiding
2. Abstraction
3. Encapsulation
4. Tightly Encapsulated class

} Talks about "Security"

5. Is-A Relationship
6. Has-A Relationship
7. Method signature
8. Overloading
9. OverRiding
10. static control flow
11. Instance control flow
12. Constructors
13. Coupling
14. Cohesion
15. Type-Casting
16. singleton class

(*) Data Hiding

↝ outside person can't access our internal data directly
(Or)
Our internal data should not go out directly.
This OOP feature is nothing but Data Hiding

↝ After Validation / Identification outside person can access our internal Data.

Ex.1 After providing proper username & password, we can able to access our (gmail) - Inbox Information.

Ex.2 Even though we are valid customer of Bank, we can able to access our account info only, we can't access other's account info.

↝ By declaring data member (variable) as private, we can achieve Data Hiding

```
public class Account
{
    private double balance;

    public double getBalance()
    {
        //validation;
        return balance;
    }
}
```

↝ The main advantage of Data Hiding is, security.

(Note: HighLy recommanded to data member (variable) → private)

✱ Abstraction

→ Hiding internal implementation & just highlight the set of services, what we are offering is the concept of Abstraction.

Ex. → Through Bank ATM GUI screen Bank people are highlighting the set of services, what they are offering, without giving internal implementation.

↝ The main Advantages of Abstraction are:-

① we can achieve [security] baz we are not highlighting or internal imple.

② without affecting outside person, we can able to perform any type of changes in our internal system.
∴ [Enhancement] becomes Easy

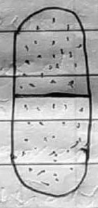
③ It improves [Maintainability] of application.

④ It improves [Easyness] to use our system.

~ By using Interfaces & Abstract classes, we can implement Abstraction.

Encapsulation

~ The process of Binding Data & corresponding methods into a single unit is nothing but Encapsulation



capsule

~ Ex. class student
{
Data members + Methods (Behavior)
}

~ If any component follows, Data Hiding & Abstraction, such type of component is said to be Encapsulated component

Encapsulation = Data Hiding + Abstraction

Ex.

```
public class Account
{
    private double balance;

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

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

Welcome to Balance Info.
Balance Enquiry
Update Balance
GUI Screen

~ Here, Data Hiding force us to use method to access data member which leads us to Data Encapsulation

~ The main Advantages of Encapsulation are:

1. we can achieve security.
2. Enhancement becomes Easy.
3. Improves Maintainability of application.

→ Despite having main advantage like security, Encapsulation has disadvantage like
- It Increses the LOC & slows down execution.

⊛ Tightly Encapsulated class

→ A class said to be tightly encapsulated if and only if each & every variable declared as private,
whether class contains corresponding getters & setters methods or not & whether these methods declared as public or not, these things are not required to check.

```
public class Account
{
    private double balance;
    public double getBalance()
    {
        return balance;
    }
}
```

This thing not matters

→ Which are Tightly Encapsulated class?

① class A ✓
```
{
    private int x=10;
}
```

② class B extends A ✗
```
{
    int y=20;
}
```

③ class C extends A ✓
```
{
    private int z=30;
}
```

④ class D extends B ✗
```
{
    private int w=40;
}
```

Bcz B has non-private data & It parent class is Not Tightly Encapsulated then No child is Tightly Encapsulated

⊛ IS-A Relationship

- It is also known as "Inheritance".
- The main Advantage of IS-A Rel^n ship is code reusability.
- By using "extends" keyword, we can implement IS-A Rel^n ship.

Ex:

```
class P
{
    public void m1()
    {
        s.o.p ("Parent");
    }
}
class C extends P
{
    public void m2()
    {
        s.o.p(" child");
    }
}

class Test
{
    p s v main (String[] args)
    {
```

```
① P p = new P();
   p.m1();  ✓
   p.m2();  ✗  → C.E: Cannot find symbol
                   symbol : m1(
                   location : class

② C c = new C();
   c.m1();  ✓
   c.m2();  ✓

③ P p1 = new C();
   p1.m1();  ✓
   p1.m2();  ✗

④ C c1 = new P();  → C.E: Incompatible types
       ✗               found: P
                       required: C
    }
}
```

- Whatever methods parents has by default available to child, Hence on the child reference, we can call both parent & child class method.
- Whatever methods child has not available to parent, Hence on parent reference, we can't call child specific methods.
- Parent reference can be used to hold

child object, But using that reference we can't call child specific methods But we can call the methods present in parent class

(↳ polymorphism & advantage of this approach will be in next pages)

↝ Parent ref. can be used to hold child object, But child ref cannot be used to hold parent object.

↝ Ex. Showing Code Reusability & Removal of Redundancy.

⇒ (Without Inheritance)

```
class Vloan          class Hloan          class Ploan
{                    {                    {
  // 300 methods      // 300 methods       // 300 methods
}                    }                    }
```

900 Methods ⇒ 90 Hours

(If changes occur in common methods → we has to change all.)

⇒ (With Inheritance)

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

```
class Vloan extends Loan          class Hloan extends Loan          class Ploan extends Loan
{                                 {                                 {
  // 50 methods                   // 50 methods                     // 50 methods
}                                 }                                 }
```

(400 Methods → 40 Hours)

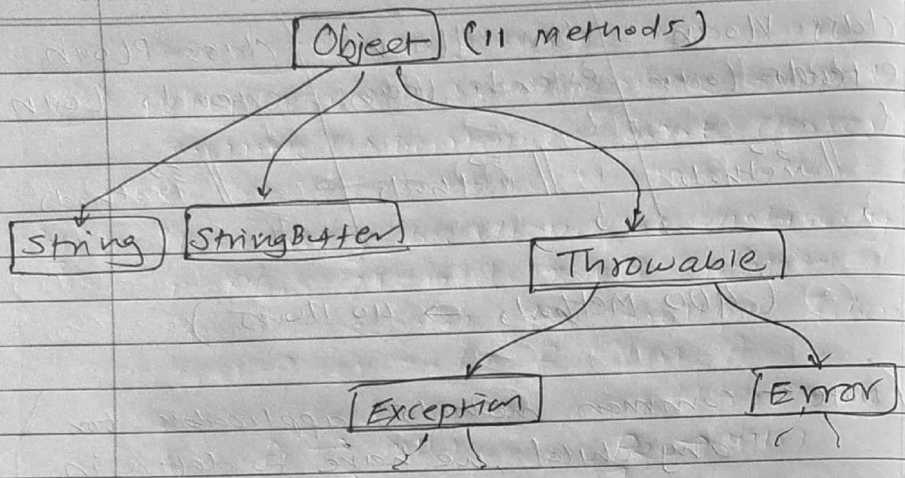
(Note :- common methods applicable for any child, we have to define in parent class.)

↝ Total Java API is implemented based on Inheritance concept,

The most common methods which are applicable for any Java-Object are defined in Object class & hence every class in java is the child class of Object either directly or indirectly so the Object class methods by default available to every Java-class without Re-writing

- Due to this reason, Object class act as Root for all java classes.

↝ Throwable class defines the most common methods which are required for every Exception or Error class, Hence this class act as Root for Java Exception Hierarchy.



⇒ Multiple Inheritance :-

→ A java class can't extend more than one class at a time.

Hence

Java won't provide support for Multiple Inheritance for classes

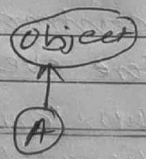
```
Class A extends B, C
{
}
```

(X)

C.E

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

```
class A
{
}
```

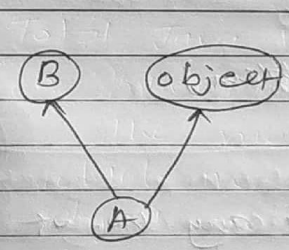


→ If our class extends any other class then our class is indirect child of Object.

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

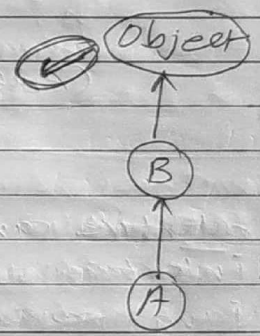
This will not happen

(X)



Multiple Inheritance

This will happen Internally.

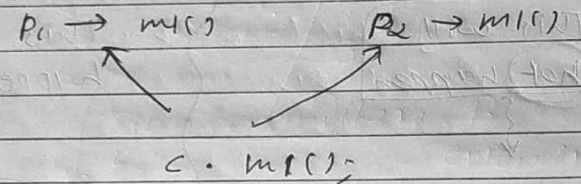


Multi-Level Inheritance

(Note : Either directly / Indirectly Java won't provide support for Multiple inheritance w.r.t classes.)

Q. Why java won't provide support for multiple Inheritance w.r.t classes?

Ans: There may be chance of ambiguity problem, Hence Java won't provide support for multiple Inheritance.



(Ambiguity problem)

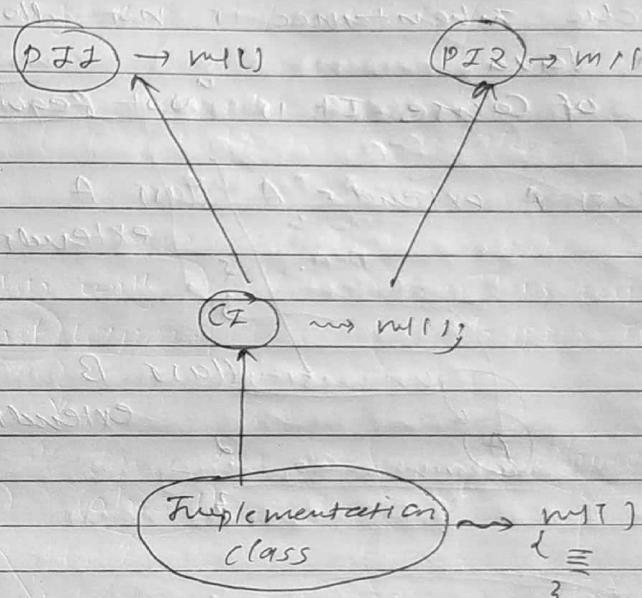
→ Interface can extend any no. of interfaces simultaneously, Hence java provides support for multiple Inheritance w.r.t interfaces.

```
Interface A        interface B
{                  {
}                  }

interface C extends A, B
{
}
```

Q. Why Ambiguity problem won't be there in interfaces.

Ans:



→ Even though, multiple method deleration are available, but implementation is unique, there is no chance of Ambiguity problem in Inheritance.

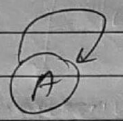
★ → Note: Strictly speaking through interfaces, we won't get any inheritance, as implementations are not there.

⇝ Cyclic Inheritance

→ Cyclic inheritance is not allowed in Java.
of course It is "Not Required"

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

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



C.E. Cyclic inheritance involved in A

⊛ Has-A Relationship

⇝ Has-A Relationship is also known as "composition / Aggregation".

⇝ There is no specific keyword to implement Has-A Rel"ship. But most of the time, we depends on new keyword.

⇝ The main Advantage of Has-A Rel"ship is "Reusability" of code.

⇝ Ex.

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

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

Car Has-A Engine reference

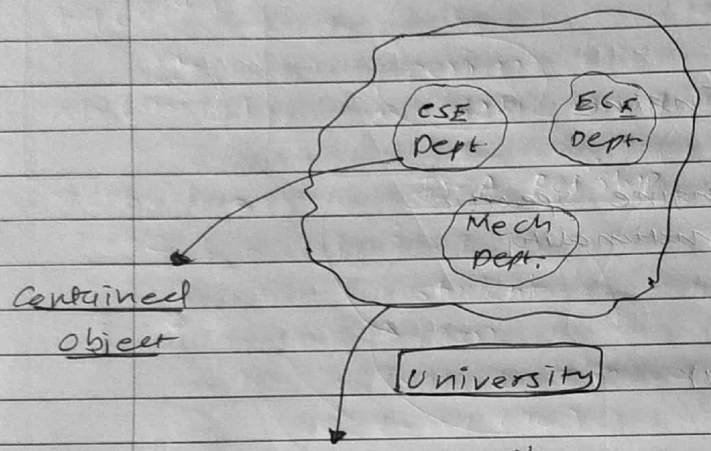
* Interview

⇝ Difference blw "Composition" & "Aggregation"

→ without existing container object if there is no chance of existing contained objects, then container & contained object are strongly associated & this strong association is known as "composition".

Ex. University consists of several Dept. without existing university, there is no chance of existing Dept.

Hence, university & Dept. are strongly associated ⇒ "composition"

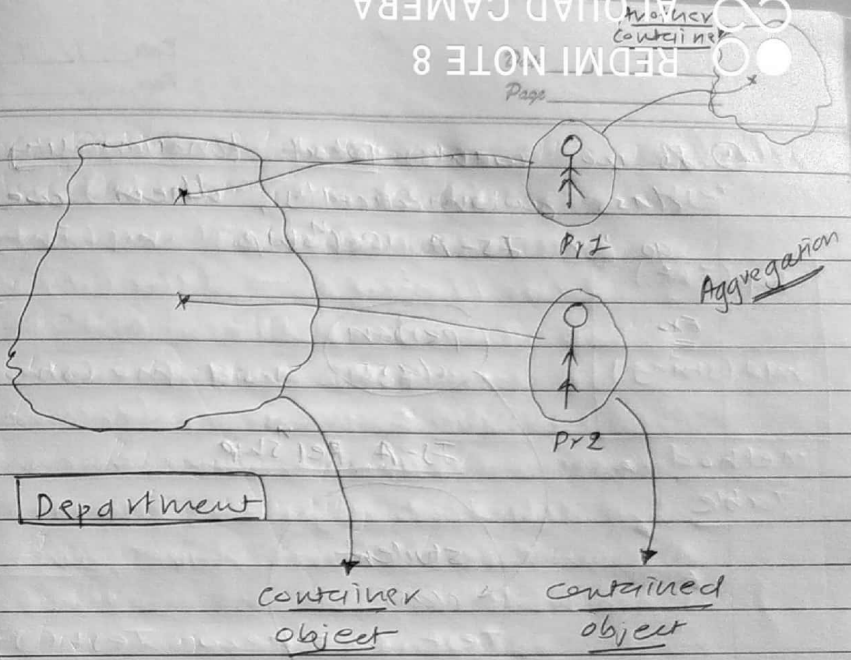


↝ without existing container object if there is chance of existing contained object, then container & contained objects are weakly associated this weak association is known as "Aggregation"

Ex. Dept. consists of several prof. without existing Dept, there may be chance of existing prof. object, Hence Dept. & prof. are weakly associated

⇓

"Aggregation"



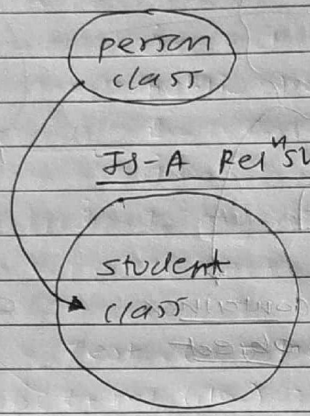
Note :- (1) In composition, objects are strongly associated.
whereas
In Aggregation, objects are weakly associated.

(2) In composition, container object holds directly contained object
Whereas
In Aggregation, container object holds just References of contained objects.

↝ IS-A (vs) Has-A

① If we want total functionality of class automatically, then we should go for IS-A Rel^{n}ship.

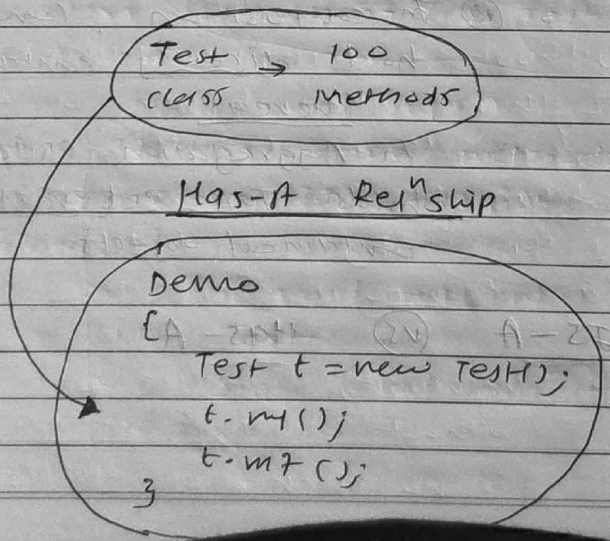
Ex.



- Complete Functionality of person class needed for student class.

② If we want only some part of functionality, then we should go for Has-A Rel^{n}ship.

Ex.



Here, Demo class needs only several methods of Test class.

✱ Method Signature

↝ In Java, method signature consists of
- method_name
- argument types

↝ Ex.

public static int m1 (int i, float f)

Method Declaration

m1 (int, float)

Method signature

(Note: return type is Not part of method signature in Java, unlike C & C++.)

★ ↝ compiler will use method signature to resolve method calls & compiler maintains "Method Table" for every clas

class Test

m1(int)
m2(String)

Method Table

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

```
Test t = new Test();
t.m1(10);      ✓
t.m2("abc");   ✓
t.m3(10.5);    ✗
```

C.E: Cannot find symbol
symbol: method m3(double)
location: class Test

⇝ within a class, two methods with same signature is not allowed.

Ex:

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

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

⊛ Overloading

⇝ Two methods are said to be overloaded if and only if both methods having
- same name & (MUST)
- Different Argument type. (MUST)

⇝ No Restrictions on return type (can change)

⇝ 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 name, which increases complexity of programming.

C//

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

⇒ But in Java, we can declare multiple methods with same name but different arguments types such methods are called overloaded methods.

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

- Having overloading concept in Java reduces complexity of programming.

⇒ Ex.

```
class Test
{
    public void m1()
    {
        s.o.p("no-arg");
    }
    public void m1(int i)
    {
        s.o.p("int-arg");
    }
    public void m1(double d)
    {
        s.o.p("double-arg");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();      // no-arg
        t.m1(10);    // int-arg
        t.m1(10.5);  // double-arg
    }
}
```

Overloaded methods

★ ⇒ In overloading, method resolution always taken care by compiler based on reference type, so overloading is also considered as

- Compile-Time Polymorphism (or)
- Static-polymorphism (or)
- Early Binding

⊛ Loop-holes in Overloading

Case-1 :- Automatic Promotion in Overloading

while resolving overloaded method, if exact match method is not available

then, we won't get compile-Time Error immediately, first it will promote argument to the next level & check whether matched method is available or not.

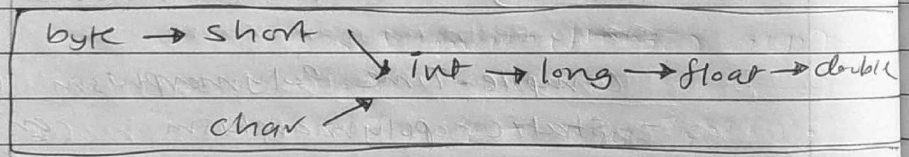
If matched method is available then it will be considered.

If matched method is not available then compiler promotes argument to the next level

This process will be continued until all possible promotions,

still if the matched method is not available, then we will get C.E.

The following are all possible promotions in overloading



This process is called "Automatic promotion" in overloading

Ex. class Test
{
public void m(int i)
{
S.O.P("int-arg");
}

public void m(float f)
{
S.O.P("float-arg");
}

P S V main(String[] args)
{
Test t = new Test();

Exact Match → t.m(10); // int-arg
t.m(10.5f); // float-arg

promotion → t.m('a'); // int-arg
t.m(10L); // float-arg

t.m(10.5); ~ C.E: cannot find symbol
Symbol: method m(double)
Location: class Test

}
}

~ Here, we can check like this
- int i = 10; ✓
- int i = 10.5; ✗ but float f = 10.5f; ✓
- int i = 'a'; ✓
- int i = 10L; ✗ but float f = 10L; ✓
- int i = 10.5; ✗ & float f = 10.5; ✗ } → C.E.

Case-2 :- (Child Get Higher priority, If both can execute.
↳ (Parent & Child)

↳ while resolving overloaded method compiler will always gives precedence for child type argument than compared with parent type argument

Ex. class Test
{
public void m1(String s)
{
S.O.P("String-version");
}
public void m1(Object o)
{
S.O.P("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
}
}

Object
|
String

★

Case-3 :- Ambiguity while Siblings are there

class Test
{
public void m1(String s)
{
S.O.P("String version");
}
public void m1(StringBuffer sb)
{
S.O.P("StringBuffer version");
}
p s v main(String[] args)
{
Test t = new Test();
t.m1("durga");
// String version
t.m1(new StringBuffer("d"));
// StringBuffer version
t.m1(null); (X)
// Both can take null
}
}

C.E. reference to m1() is ambiguous

Object
/ \
String StringBuffer

Case-4 :- Multiple Argument method (overloaded)

```
class Test
{
    public void m1(int i, float f)
    {
        s.o.p("int-float version");
    }
    public void m1(float f, int i)
    {
        s.o.p("float-int version");
    }

    p s v m(String[] args)
    {
        Test t = new Test();
        t.m1(10, 10.5f);
        //int-float version
        t.m1(10.5f, 10);
        //float-int version

        t.m1(10,10);
```

C.E. reference to m1 is ambiguous

```
        t.m1(10.5f, 10.5f);
```

C.E. cannot find symbol
s: method m1(float, float)
L: class Test

```
    }
}
```

Overloaded methods

Case-5 :- Overloading with var-arg method

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

        t.m1(10); // General method
    }
}
```

[This case discussed in var-arg Topic]

Case-6

```
class Animal
{
}
```

```
class Monkey
    extends Animal
{
}
```

```
class Test
{
    public void m(Animal a)
    {
        S.O.P("Animal version");
    }
    public void m(Monkey m)
    {
        S.O.P("Monkey version");
    }
    p s v m(String[] args)
    {
        Test t = new Test();
```

Overloaded Methods

① `Animal a = new Animal();`
`t.m(a);`
`//Animal version.`

② `Monkey m = new Monkey();`
`t.m(m);`
`//Monkey version.`

③ `Animal a = new Monkey();`
`t.m(a);`
`// Animal version.`

```
    }
}
```

→ Bcz, In overloading, method Resolution is done at compile time only, based on Reference type object don't play any Role.

⊛ Overriding

→ whatever methods parent has by-default available to child through inheritance,
If child class not satisfied with parent class implementation, then child is allowed to redefine the method based on its requirement.
This process is called "over-riding".

→ The parent class method which is overridden is called overridden method &
The child class method which is overriding is called overriding method.

→ Ex.

```
class Parent
{
    public void property()
    {
        S.O.P("cash + land + gold");
    }
    public void marry()
    {
        S.O.P("subh lakshmi");
    }
}
class C extends Parent
{
    public void marry()
    {
        S.O.P("Trisha");
    }
}
```

Overridden method

Overriding

overriding method

Date __/__/__
Page

⇒ Ex. class Test
{
psvm(String[] args)
{
① Parent p = new Parent();
p.marry();
//parent method

② C c = new C();
c.marry();
// child method

③ Parent p1 = new C();
p1.marry();
// child method
}
}

- In overriding, method resolution always taken care by JVM based on runtime object & Hence overriding is also considered as

"Runtime Polymorphism"
"Dynamic polymorphism"
"Late Binding"

Date / /
Page

Rules for Overriding :-

(1) → In overriding, "method names" & "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 V only.

From 1.5V onwards, we can take "co-variant return types" also.

According to this, child class method return type need not be same as parent class method type.

Ex.

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

Invalid till 1.4

Valid From 1.5V onwards

parent class method	Object	Number	String	double
Return Type				
child class method Return Type	Object, String, StringBuffer	Number, Integer	Object	int
	✓	✓	✗	✗

co-variant return type concept applicable only for object type but not for primitive types.

(3) ⇝ Parent class private methods not available to the child & hence, overriding concept not applicable for private method.

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()
    {}
}
```

✓ valid, But not overriding.

(4) ⇝ We can't override parent class "final" methods in child classes, If we try to override, we will get C.E.

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

✗ C.E. m1() in C cannot override m1() in P, overridden method is final

Date / / Page

(5) → 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

Ex.
```
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 method to the next level child classes.

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

parent method :	final	non-final	abstract
child method :	non-final/ final ✗	final ✓	non-abstract ✓

Synchronized	native	strictfp
non-synchronized ✓	non-native ✓	non-strictfp ✓

(6) → while overriding "we can't Reduce" scope of accessability (access modifier)
BUT
"we can increase the scope"

Ex.

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

(X)

C.E. m() in C cannot override m() in P; attempting to assign weaker access privileges; was public

private < default < protected < public

Parent class Method	public	protected	<default>	private
Child class method	public	protected/ public	<default>/ protected/ public	Overriding concept Not Applicable

(7) → If child class method throws any checked Exception, compulsory parent class method should throw the same checked Exception or its parent

Otherwise we will get compile Time Error.

BUT

There are no exceptions for unchecked Exception.

Ex.

```
class P
{
    public void m()
            throws IOException
    {}
}
class C extends P
{
    public void m() throws
            EOFException, InterruptedException
    {}
}
```

(X)

C.E. m() in C cannot override m() in P; Overridden method does not throw java.lang.InterruptedExcepti

1.) P: public void m() throws Exception
C: public void m()

✗ 2.) P: public void m()
C: public void m() throws Exception

✓ 3.) P: public void m() throws Exception
C: public void m() throws IOException

✗ 4.) P: public void m() throws IOException
C: public void m() throws Exception

✓ 5.) P: public void m() throws IOException
C: public void m() throws FileNotFoundException, IOException

✓ 6.) P: public void m() throws IOException
C: public void m() throw NPE, ...

★

Q. Why this Rule?

Ans 3

```
class P
{
    public void m() throws IOException
    {
    }
}
class C extends P
{
}
```

Outside user :-

```
P p = new C();
try
{
    p.m();
}
catch(IOException e)
{
}
```

⇒ Here, before overriding the parent class method, if outside user want to access an method, they need to handle the exception throw by that method, so (Here they handled IOException)

⇒ After override

```
class P
{
    public void m() throws IOException
    {}
}
class C extends P
{
    public void m() throws [          ]
    {}
}
```

As per outside user's code, now overriding method (child class method) get executed.

And if we put [Exception] inside that block, their old code get affected, bcz our child class throws ~~Exception~~ Exception & they handle IOException

So, we can put any childs of IOException in that box [ ].

→ Reducing Access modifier scope C.E. can also be understand by this type of solution.

→ Overriding w.r.t static method

① → we can't override a static method as non-static, otherwise we will get C.E

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

(+)

C.E. m1() in C cannot override m1() in P; overridden method is static

(X)

② → similarly, we can't override a non-static method as static

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

(X)

C.E. m1() in C cannot override m1() in P; overriding method is static

[This is like we are replacing Faculty & Student]

③ → If Both parent & Child class method are static, then we won't get any C.E.

It seems overriding concept applicable for static methods, but it is not "overriding"

It is "Method Hiding"

Ex.

```
class P
{
    public static void m1()
    {
        S.o.p("parent");
    }
}
class C extends P
{
    public static void m1()
    {
        S.o.p("child");
    }
}
```

It is Method Hiding

Not overriding

⇝ Method Hiding

⇝ All rules of "Method Hiding" are exactly same as "overriding" except the following differences:-

Method Hiding	Overriding
① Both parent & child class methods should be static	Both parent & child class method should be Instance (Non-static).
② Compiler is responsible for Method resolution based on Reference Type	JVM is responsible for method resolution based on Runtime object.
③ It is also known as -C.T. Polymorphism, -Static polymorphism, -Early Binding	It is also known as -R.T. Polymorphism, -Dynamic " -Late Binding

Ex.

```
class P
{
    public static void m1()
    {
        S.o.p("parent");
    }
}
class C extends P
{
    public static void m1()
    {
        S.o.p("child");
    }
}
```

It is Method Hiding

Overriding

Date / /
Page

```
class Test
{
    p s v main (String[] args)
    {
        P p = new P();
        p.m();          ~> (parent)

        C c = new C();
        c.m();          ~> (child)

        P p = new C();
        p.m();          ~> (parent)
    }
}
```

→ If both parent & child class methods are non-static, then it will become overriding & o/p is:- parent, child, child.

★

Q Why the words "overriding" & "Method Hiding" ?

Ans: We have one child object.

(we can use parent ref.) P p ——→ new C()

(we can use child ref.) C c ——→ new C()

Date / /
Page

Overriding :

→ Suppose, we have one Marker-Pen board,
so, we wrote something on board.
we rubbed that
and
we add/write new data.

→ This is called "Overriding"

→ Here, old data is gone & Not Available anymore.

Method Hiding :

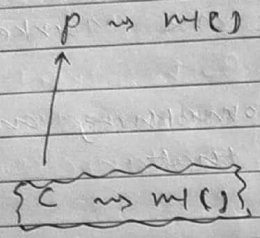
→ we have one chart pinned on wall.

We want to pin another chart, so we put that over old chart

Here, Now we can see new chart, but if we need, we can see old chart also.

→ old data is Available.

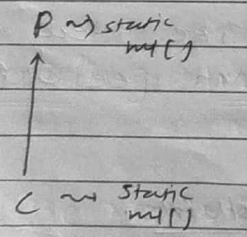
Overriding &



In overriding, For child object, Whether we use (parent Ref.) or (child Ref.), we will get child Method only,
child object can't call that already gone parent method.

Method Hiding &

In Method Hiding, For child object, If we use (Parent Ref.) parent method get executed & if we use (Child Ref.) child method get executed.



So, Here old copy only gets Hide But it is always available.

→ Overriding w.r.t Var-arg Methods

⇝ We can override var-arg method with another var-arg method only.

⇝ If we are trying to override with normal method then it will became Overloading but Not overriding.

Ex-

```
class P
{
    public void m(int... x)
    {
        S.O.P("parent");
    }
}
class C extends P
{
    public void m(int x)
    {
        S.O.P("child");
    }
}
class Test
{
    p s v m(String[] args)
    {
```

It is Overloading but not Overriding.

```
P p = new P();
p.m1();  ---> parent

C c = new C();
c.m1();  ---> child

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

}
}

Note: In the above program if we replace child method with var-arg method, then it will become over-riding.

→ In this case the o/p :
parent
child
child

→ Overriding w.r.t 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.

Ex:

```
class P
{
    int x = 888;
}
class C extends P
{
    int x = 999;
}
class Test
{
    p s v m main(String[] args)
    {
        P p = new P();
        S.O.p(p.x);  ---> 888

        C c = new C();
        S.O.p(c.x);  ---> 999

        P p1 = new C();
        S.O.p(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
			same

✱ Differences b/w Overloading & Overriding

property	Overloading	Overriding
① method names	must be same	must be same
② Argument types	must be different (Atleast order)	must be same (Including order)
③ method signature	must be different	must be same
④ Return types	No Restrictions	must be same → till 1.4V can be co-variant → from 1.4V
⑤ private, final, static methods	can be overloaded	Can't be overridden
⑥ Access Modifier	No restrictions	Scope can't reduce, can be increase.
⑦ throws clause	No restrictions	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 Excp
⑧ method Resolution	Always taken care by compiler based on reference type	Always taken care by JVM based on Object type
⑨ Also known as	- Compile T. P. - Static P. - Early Binding	- Run T P. - Dynamic P. - Late Binding

Note : In overloading, we have to check only
- method names (must be same) &
- argument type (must be diff.)

We are not require to check remaining like return type, acm etc

But, In Overriding, everything we have to check like method name, argument types, return types, Ac. mod., throws clause etc.
It is complex process.

⇝ Consider following method in parent class:

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

In the child class which of the following methods we can take:

(1) pubic void my(int i) ✓
Reason: overriding ✓

(2) public static int my(long l) ✓
Reason: overloading ✓

(3) public static void my(int i) ✗
Reason: overriding ✓, but static ✗

(4) public void my(int i) throws Exception ✗
Reason: overriding ✓, but throws clause ✗

(5) public void static abstract void my(double d); ✗
Reason: overloading ✓, All ok ✓
but static abstract ✗ illegal combi.

Note: In this type of question, We should always check (method name + method argu.) ⇒ method sign. first
And then after deciding, whether it is (overriding) or (overloading) we should check corresponding Rules

✱ Polymorphism

⇝ one name but multiple forms is the concept of polymorphism

⇝ Ex.1 method name is same but we can apply different types of argument → overloading

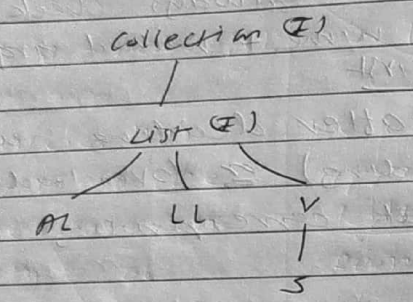
abs(int)
abs(long)
abs(float)

Ex.2 method signature is same, but in parent class one type of implementation & in child class another type of implementation → overriding

```
class P
{
    marry() { // subhlakshmi }
}
class C extends P
{   marry() { // charmi }
}
```

Date / / Page

Exp: Usage of parent reference to hold child is the concept of polymorphism

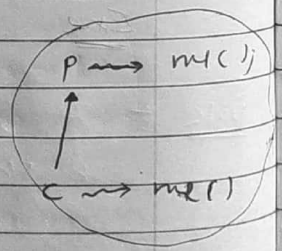


List l = new AL();
new LL();
new Stack();
new Vector();

★

→ 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 & we can't call child specific methods

Ex: `P p = new C();`



p.m1(); ✓

p.m2(); ✗

C.E: Cannot Find symbol
symbol: method m2()
Location: class P

Date / / Page

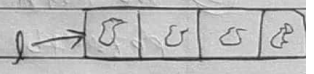
BUT, By using child reference we can call both parent & child class methods.

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

★ So, when we should go for parent reference to hold child object?

Ans: If we don't know exact Run-time type of object then we should go for parent reference.

Ex: 1st element present in ArrayList can be any type like
- Student object,
- Customer object,
- String object,
- StringBuffer object.

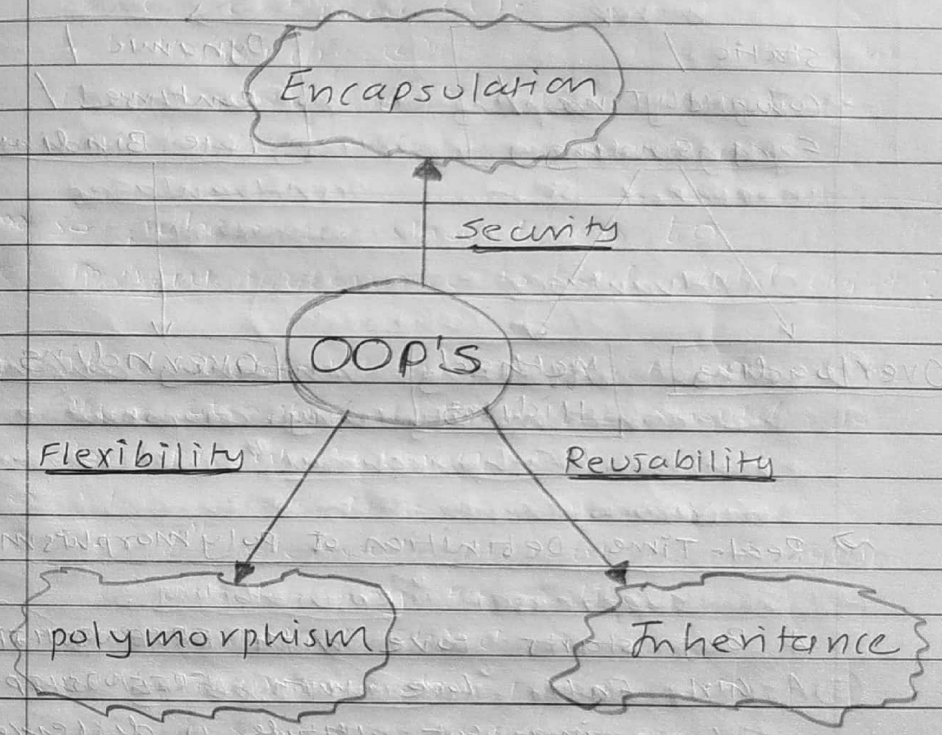


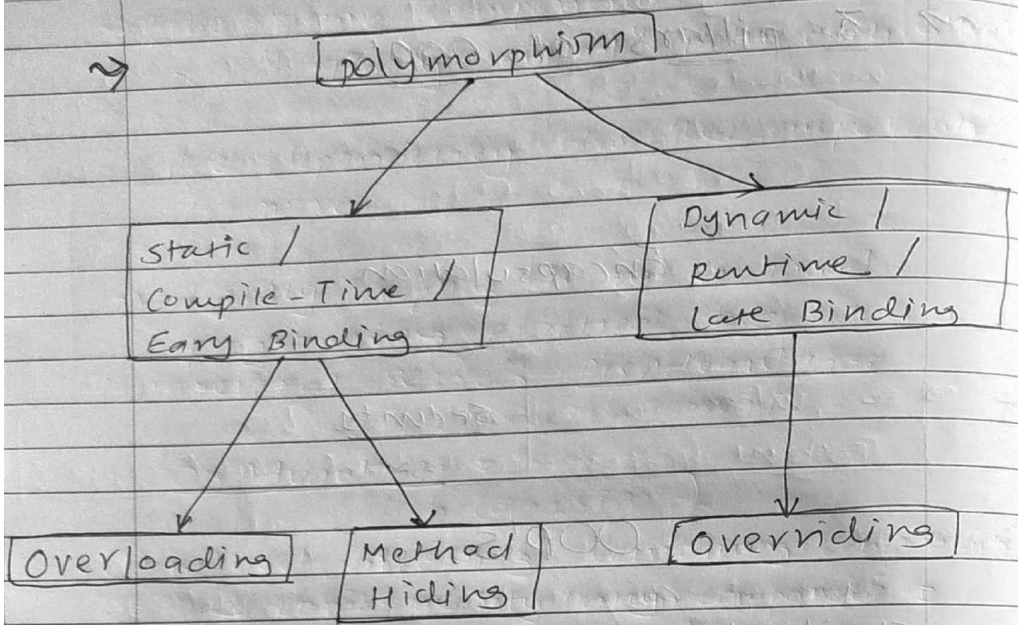
Hence the return type of get() method is Object, which can hold any object.

Object o = l.get(0);

C c = new C();	P p = new P();
eg. AL al = new AL();	eg. List l = new AL();
① We can use this approach if we know exact run-time type of object	We can use this approach if we don't know exact run-time type of object
② By using child ref., both parent class & child class method. → Advantage	By using parent ref., we can call only methods available in parent class, can't call child specific method → DisAdvantage
③ We can use child ref. to hold only particular child class object → DisAdvantage	We can use parent ref. to hold any child class object → Advantage

⇝ 3 pillars of OOP's





→ Real-Time Definition of polymorphism :-

A Boy starts love with FRIENDSHIP, But A Girl Ends love with FRIENDSHIP, here word is same but attitude is diffience, This is nothing but polymorphism.

(Not given right, just based on internet Waving)

✱ Coupling

→ The degree of dependency blw the components is called coupling.

- If dependency is More, then it is considered as Tightly Coupled.
- If dependency is Less, then it is considered as Loosely Coupled.

→ Ex:

```
Class A
{
    static int i = B.j;
}
Class B
{
    static int j = C.k;
}
Class C
{
    static int k = D.m1();
}
Class D
{
    public static int m1()
    {
        return 10;
    }
}
```

The above components are said to be "Tightly coupled" with each other

Bcz dependency blw the components is more.

↳ Tightly coupling is not a good programming practice, bcz it has several serious DisAdvantages

DisAdvantages :-

① without affecting remaining components we can't modify any component & hence "Enhancement" will become difficult.

② It suppresses "Reusability".

③ It reduces "Maintainability" of the application.

↳ Hence, we have to maintain dependency blw components as Less as possible.

i.e loosly coupling is a good programming practice.

⊛ Cohesion

↳ For every component a clear & well-defined functionality is ~~required~~ defined then that component is said to be follow "High Cohesion"

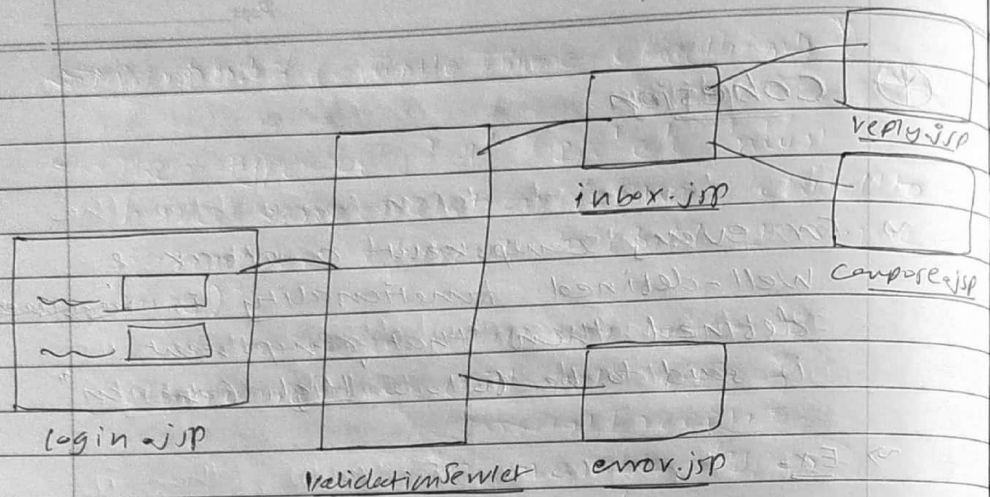
↳ Ex.

TotalServlet

- login page
- validation
- Inbox page
- Reply page
- compose page
- Error page

70 lakhs lines of code

(Low Cohesion)



(High Cohesion)

~ High Cohesion is always a Good programming practice, bcz it has several Advantages :-

1. Without affecting remaining components we can modify any component, Hence "Enhancement" will become Easy.
2. It promotes "Reusability" of code. (Wherever validation is required we can reuse, the same validateServer, without rewriting it.)
3. It improves "Maintainability" of application.

(Note :- Loose Coupling & High Cohesion are Good programming practices.)

⊛ Object Type-casting

~ we can use parent Ref. to hold child Object.

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

~ we can use Interface Ref. to hold Implemented class Object.

Ex. Runnable r = new Thread();

~ `A b = (C) d ;`

- A → class/interface name
- b → Name of Ref. variable
- C → class/interface name
- d → name of Ref. variable

→ Mantra1 (compile-Time Checking 1)

→ The type of 'd' & 'c' must have some relation (either child to parent or parent to child or same type)

otherwise we will get C.E.

```
incovertible types
found : d
required : c
```

Ex1 :-

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

M1 ✓

[Mantra1 ✓]

∵ Relation b/w (Object ↑ StringBuffer)

Ex2 :-

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

M1 ✗

[Mantra1 ✗]

∵ Relation ✗

∴ C.E. →

```
Inconvertible types
found : j.l.String
required : j.l.StringBuffer
```

→ Mantra2 (compile-Time Checking 2)

→ 'c' must be either same or derived type of 'A'

otherwise we will get C.E.

```
incompatible types
f : c
r : a
```

Ex1 :-

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

M2 ✓ M1 ✓

[Mantra2 ✓]

(∵ StringBuffer is same type as StringBuffer)

Ex2 :-

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

M2 ✗ M1 ✓

[Mantra2 ✗]

∵ Bcz String is neither same nor derived type of StringBuffer

C.E.

```
Incompatible Types
f : String
r : StringBuffer
```

Mantra 3 (Run-Time Checking)

→ Run-time object type of 'd' must be either same or derived type of 'c'

otherwise we will get R.E. ↓

ClassCastException

Ex 1 :-

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

m2 ⊖ m1 ⊖ RTC M3 ✗

[Mantra 3 ✗]

∵ Bcz underlying object of o is 'String' which is neither same nor derived type of 'StringBuffer'.

∴ R.E. → ClassCastException : j.l.String cannot be cast to j.l.StringBuffer

Ex 2 :-

```
Object o = new String("abc");
Object o1 = (String) o;
```

m2 ✓ m1 ⊖

→ Base2 b = new Der4();

Object

Base1 Base2

Der1 Der2 Der3 Der4

① Object o = (Base2) b; ✓

② Object o = (Base1) b; M1 ✗ (InCov.)

③ Object o = (Der3) b; M3 ✗ (CCE)

④ Base2 b1 = (Base1) b; M1 ✗ (InCov.)

⑤ Base1 b1 = (Der4) b; M2 ✗ (InCovp.)

⑥ Base1 b1 = (Der1) b; M1 ✗ (InConv.)

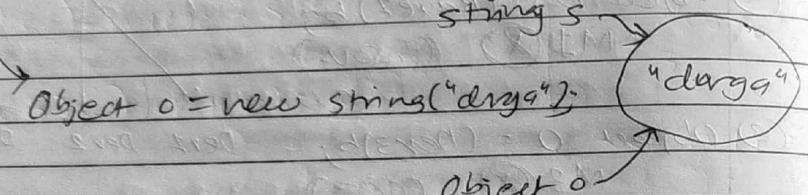
→ Strictly speaking, Through type-casting we are not creating any new object

For the existing object, we are providing another type of Ref. variable

i.e we are performing type-casting but not object-casting

Ex1:-

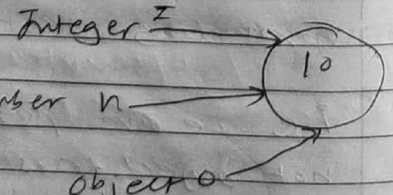
```
String s = new String("durga");
Object o = (Object)s;
```



Ex2:-

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

  S.o.p(I == n); true
  S.o.p(n == o); true
```



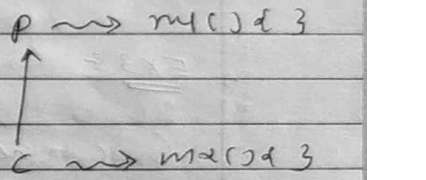

```
Number n = new Integer(10);
Object o = new Integer(10); }
```

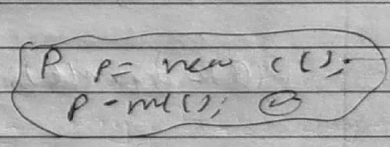
Note :-

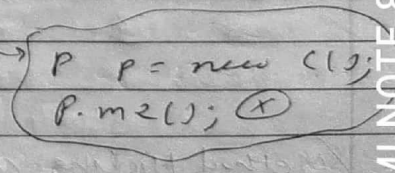
```
C c = new C();
(B) c;
B b = new C();
A(B) c;
A a = new C();
```


~> Ex. 1

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







(In polymorphic object we can't call child specific methods)

Date / / Page

⇝ Ex:-

```
C c = new C();
c.m1();  ⇝ C

(B)c).m1();  ⇝ C

(A((B)c)).m1();  ⇝ C
```

```
A ⇝ m1()
{
  s.o.p("A");
}

B ⇝ m1()
{
  s.o.p("B");
}

C ⇝ m1()
{
  s.o.p("C");
}
```

(Overriding concept
In which method resolution is based on Runtime object)

⇝ Ex3 :

```
C c = new C();
c.m1();  ⇝ C

((B)c).m1();  ⇝ B

(A((B)c)).m1();  ⇝ A
```

```
A ⇝ static m1()
{
  s.o.p("A");
}

B ⇝ static m1()
{
  s.o.p("B");
}

C ⇝ static m1()
{
  s.o.p("C");
}
```

(Method Hiding concept
∵ static method
∴ In which method resolution is based on Ref. type)

⇝ Ex.4 :

```
C c = new C();

s.o.p(c.n);  → 999

s.o.p(((B)c).n);  → 888

s.o.p(((A((B)c)).n);  → 777
```

```
A ⇝ int n=777;

B ⇝ int n=888;

C ⇝ int n=999;
```

(Not overriding for variable
variable resolution is based on Ref. type only)

✱ static Control flow

⇝ whenever we are executing a Java class, the following sequence of steps / activities will be executed as a part of static control flow :-

- ① Identification of static members from top to bottom. [here.. 1 to 6]
- ② Execution of static variable assignments & static blocks from top to bottom [here.. 7 & 8]
- ③ Execution of Main Method. [here.. 13 to 15]

Ex.

```
class Base
{
    ① static int i = 10;                    (7)
    ② static
    {
        m();                                (8)
        S.O.p("First static Block");        (10)
    }
    ③ p s v main(String[] args)
    {
        m();                                13
        S.O.p("main method");               15
    }
    ④ p s v m()
    {
        S.O.p(j);                           (9)
    }                                       19
    ⑤ static
    {
        S.O.p("second static Block");
    }                                       (11)
    ⑥ static int j = 20;                    (12)
}
```

i = 0 &	[RIWO]	⇝ At (1)
j = 0	[RIWO]	⇝ At (6)
i = 10	[R & W]	⇝ At (7)
j = 20	[R & W]	⇝ At (12)

O/P :- Java Base

```
0
First static Block
second static Block
20
main method
```

⇝ RIWO :- Read Indirectly Write Only

⇝ Inside static block if we are trying to read a variable, that read operation is called
↳ Direct Read

⇝ If we are calling a method from static block & we are trying to read that variable within that method, that read operation is called
↳ Indirect Read

```
class Test
{
    static int i=10;

    static
    {
        m1();
        S.O.P(i); ---> Direct Read
    }

    p s v m1()
    {
        S.O.P(i); ---> Indirect Read
    }
}
```

~ If a variable just identified by the JVM & original value not yet assigned, then the variable is said to be in RIWO state.

~ If a variable is in RIWO state then ~~Read~~ op we can't perform Direct Read, But we can perform Indirect Read.

~ If we are trying to read directly then we will get C.E.

Illegal forward reference

~ Ex-1 :-

```
class Test
{
    ① static int x=10;        ③
    ② static
      {
          S.O.P(x); ④
      }
}
```

O/P :- 10
R.E. NoSuchMethodError: main.

~ Ex-2 :-

```
class Test
{
    ① static
      {
          S.O.P(x); ③ ---> C.E.
      }

    ② static int x=10;
}
```

O/P :- x=0 [RIWO] ~> At ②

At ③ ~> C.E Illegal forward

Date / /
Page

⇒ Ex-3 :-

```
class Test
{
    (1) static
    {
        m1();  (4)
    }

    (2) p s v m1()
    {
        s.o.p(x);  (5)
    }

    (3) static int x=10;  (6)
}
```

x=0 [RIWO] ⇝ At (3)

(O/P :- 0)

x=10 [R&W] ⇝ At (6)

(O/P :- R.E. NoSuchMethodError: main)

⊛ static block

⇝ static blocks will be executed at the time of class loading

Hence,

At the time of class loading, if we want to perform any activity we have to define that inside static block.

Ex1:- At the time of Java class loading, the corresponding native library should be loaded.

Hence,

we have do define this activity inside static block.

```
class Test
{
    static
    {
        System.loadLibrary
            ("native library path");
    }
}
```

In Object class, Thread class also there is loading of native methods in static block.

Ex2 : After loading Every Database driver class, we have to register driver class with driver manager.

But inside database driver class there is static block to perform this activity & we are not responsible to register explicitly.

```
class DbDriver
{
    static
    {
        // Register this Driver with Driver manager
    }
}
```

① load Driver class — class.forName
② get Conn. Object
③ Prepare stmt obj
④ execute query
⑤ use ResultSet

↑ In this process we don't register Driver class explicitly, JVM does it.

Note : within a class, we can declare any no. of static blocks, but all these static blocks will be executed from top to bottom.

→ surprise note regarding this Q. is behind page.

Q. without writing main() method is it possible to print some statement on the console?

Ans : Yes, by using static block.

```
class Test
{
    static
    {
        s.o.p("Hello, I can print");
        System.exit(0);
    }
}
```

O/P : Hello, I can print

Q. without writing main() method & static block, Is it possible to print some statement on the console?

Ans : Yes, there are multiple ways :

Way 1 :

```
class Test
{
    static int x = m1();

    public static int m1()
    {
        s.o.p("Hii");
```

System.exit(0);
return 10;
}
}

o/p : Hii

[Way 2] : class Test
{
static Test t = new Test();
{
s.o.p("Hello");
System.exit();
}
}

Instance block

o/p : Hello

[Way 3] : class Test
{
static Test t = new Test();
Test()
{
s.o.p("Hello Hii");
} System.exit(0);
}

o/p : Hello Hii

★ Note : From 1.7 version onwards, main() method is mandatory to start a program execution

Hence, From 1.7 V onwards without writing main method it is impossible to print some statements on 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 a part of static control flow.

① Identification of static members from parent to child & top to bottom. [here.. 1 to 11]

② Execution of static variable assignments & static blocks from parent to child & top to bottom [here.. 12 to 22]

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

File

Base.java

```
class Base
{
① static int i = 10;                    (12)

② static
  {
     m1();                              (13)
     S.O.P("Base static Block");        (15)
  }

③ p s v main(String[] args)
  {
     m1();
     S.O.P("Base Main");
  }

④ p s v m1()
  {
     S.O.P(j);                          (14)
  }

⑤ static int j = 20;                    (16)
}
```

```
class Derived extends Base
{
⑥ static int x = 100;                   (17)

⑦ static
  {
     m2();                              (18)
     S.O.P("Derived 1st static Block");
  }                                     (20)

⑧ p s v main(Str[] args)
  {
     m2();                              23
     S.O.P("Derived Main");             25
  }

⑨ p s v m2()
  {
     S.O.P(y);                          (19), 24
  }

⑩ static
  {
     S.O.P("Derived 2nd static Block");
  }                                     (21)

⑪ static int y = 200;                   (22)
}
```

Date / / Page

9=0 [RIWO] ⇝ At (1)
j=0 [RIWO] ⇝ At (5)
x=0 [RIWO] ⇝ At (6)
y=0 [RIWO] ⇝ At (11)

i=10 [RIW] ⇝ At (12)
j=20 [RIW] ⇝ At (16)
x=100 [RIW] ⇝ At (17)
y=200 [RIW] ⇝ At (22)

javac Base.java

Base.class — Derived.class

java Base

O/P :
0
Base static Block
20
Base Main

java Derived

O/P :
0
Base static Block
0
Derived 1st static Blk
Derived 2nd static Block
200
Derived 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 (Bcz parent class members by default available to child class whereas, child class members won't be available to parent.

⊛ Instance control flow

⇝ whenever we are executing a java class,
→ First of all 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 a part of instance control flow.

① Identification of instance members from top to bottom. [here.. 3 to 8]

② Execution of instance variable assignments & instance block from top to bottom. [here.. 9 to 14]

③ Execution of constructor. [here.. 15]

When we create an

```
class Test
{
③ int i = 10;          ⑨
④ {
      m();             ⑩
      s.o.p("First Instance block");
                       ⑫
  }
⑤ Test()
  {
      s.o.p("constructor");   15
  }

1 p s v main(String[] args)
  {
2     Test t = new Test();
16    s.o.p("main");
  }

⑥ public void m()
  {
      s.o.p(j);        ⑪
  }
⑦ {
      s.o.p("second Instance block");
                       ⑬
  }
⑧ int j = 20;          ⑭
}
```

i = 0 [RIWO] ⇝ At ③
j = 0 [RIWO] ⇝ At ⑧
i = 10 [R1W] ⇝ At ⑨
j = 20 [R1W] ⇝ At ⑭

O/p: Java Test

```
0
First Instance Block
Second Instance Block
Constructor
main
```

(Note: If we comment ②, the the o/p will be: Main only.)

★ (Note: 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 every object creation.)

Scanned with CamScanner

REDMI NOTE 8
AI QUAD CAMERA

→ Object creation is the most costly operation, If there is no specific requirement then its not recommanded 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 a part of Instance control flow

- ○ (1) Identification of instance members from parent to child [here.. 4 to 14]
- ○ (2) Execution of instance variable assignment & instance blocks only in parent class. [here. 15 to 19]
- ___ (3) Execution of parent constructor. [here.. 20]
- ○ (4) Execution of instance variable assignments & instance blocks in child class [here.. 21 to 26]
- ___ (5) Execution of child class constructor. [here.. 27]

```
class Parent
{
    (4) int i = 10;  (15)

    (5) {
            m1();  (16)

    (18) s.o.p("P I 13");
    }

    (6) Parent()
    {
        s.o.p("P constructor");  (20)
    }

    (1) {P s v main(String[] args)}
    {
        Parent p = new Parent();

        s.o.p("P main");
    }

    (7) public void m1()
    {
        s.o.p(j);  (17)
    }

    (8) int j = 20;  (19)
}
```

```
class Child
{
(9)  int x = 100;                                   (21)

(10) {
        m2();                                       (22)

        S.O.P("C F I B");                           (24)
     }

(11) Child()
     {
        S.O.P("C constructor");                     (27)
     }

{2}  {P S V main(String[] args)}
     {
{3}     {Child c = new Child();}

        S.O.P("C main");
     }

(12) public void m2()
     {
        S.O.P(y);                                   (23)
     }

(13) {
        S.O.P("C S I B");                           (25)
     }

(14) int y = 200;                                   (26)
}
```

i = 0	[R I W O]	⇝	At (4)
j = 0	[R I W O]	⇝	At (8)
x = 0	[R I W O]	⇝	At (9)
y = 0	[R I W O]	⇝	At (14)
i = 10	[R & W]	⇝	At (15)
j = 20	[R & W]	⇝	At (19)
x = 100	[R & W]	⇝	At (21)
y = 200	[R & W]	⇝	At (26)

javac Parent.java

Parent.class Child.class

java Child

O/P :-

```
O
P I B
P constructor
O
C F I B
C S I B
C constructor
C Main
```

Examples including both static & instance control flows in single program.

```
class Test
{
    {
        s.o.p ("FIB");
    }
    static
    {
        s.o.p ("FSB");
    }
    Test()
    {
        s.o.p ("constructor");
    }
    p s v main (Str args)
    {
        Test t1 = new Test();
        s.o.p ("main");
        Test t2 = new Test();
    }
    static
    {
        s.o.p ("SSB");
    }
    {
        s.o.p ("SIB");
    }
}
```

O/P :-

```
FSB
SSB
FIB
SIB
constructor
main
FIB
SIB
constructor
```

```
public class Initialization
{
① private static String m1 (String msg)
    {
        s.o.p (msg);
        return msg;
    }
④ public Initialization()
    {
⑨       m = m1("i");
    }
⑤   {
⑦       m = m1("2");
    }
⑥,⑧ String m = m1("3");

② p s v main (Str— args)
    {
③       object o = new Initialization();
    }
}
```

Date / / Page

O/P : 2 (At ⑦)
3 (At ⑧)
1 (At ⑨)

① ② }
—
③ }
④ ⑤ ⑥ }
⑦ ⑧
⑨ }

m = null (At ⑤)
m = 2 (At ⑦)
m = 3 (At ⑧)
m = 1 (At ⑨)

→

```
public class Initialization2
{
①   private static string m1(String msg)
    {
        S.O.P(msg);
        return msg;
    }
②⑤  static String m = m1("1");

⑧   {
⑨       m = m1("2");
    }

③   static
    {
⑥       m = m1("3");
    }

④   p s v main (Str args)
    {
⑦       Object obj = new Initialization2();
    }
}
```

O/P : 1 (At ⑤)
3 (At ⑥)
2 (At ⑨)

① ② ③ ④ }
⑤ ⑥ }
⑦
⑧ }
⑨

m = null (At ②)
m = 1 (At ⑤)
m = 3 (At ⑥)
m = 2 (At ⑨)

Note : From static Area we can't access Instance members directly bcz while executing static area may not identified instance members

Ex:

```
class Test
{
    int x = 10;
    p s v main (Str args)
    {
        S.O.P(x);
    }
}
```

⑦ C.E non-static variable x cannot be referenced from a static context

```
Test t = new Test();
S.O.P(t.x);
```

Scanned with CamScanner

Interview ★

Q. In how many ways we can create an object in Java?
(or)
In how many ways we can get object in Java?

Ans: There are 5 standard ways

(i) By using new operator

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

(ii) By using newInstance() method

```
Test t = (Test) Class.forName("Test").
                newInstance();
```

(iii) By using Factory method

```
Runtime r = Runtime.getRuntime();

DateFormat df = DateFormat.getInstance();
```

Note: Factory Method: By using class name, we are calling a method, which returns object of same class, Internally it uses new operator only.

(iv) By using clone() method

```
Test t1 = new Test();

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

(v) By using De-serialization

```
FileInputStream fis =
        new FileInputStream("Abc.txt");

ObjectInputStream ois =
        new ObjectInputStream(fis);

Dog d = (Dog) ois.readObject();
```

✱ 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 piece of code will be executed automatically to perform initialization of the object.

This piece of code is nothing but constructor.

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

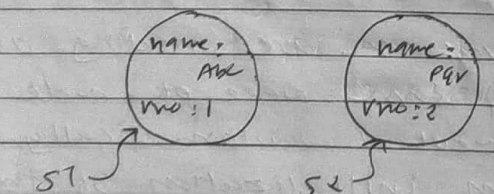
```
class student
{
    string name;
    int rno;

    Student (String name, int rno)
    {
        this.name = name;
        this.rno = rno;
    }

    p s v main (string[] args)
    {
        student s1 =
                new student("Abc", 1);

        Student s2 =
                new student("pqr", 2);
    }
}
```

Constructor →



(Note: The main purpose of constructor is to initialize the object not to create the object.)

⇒ Difference blw 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 database for every object, or incrementing count value for every object creation etc...)

→ Both constructor & Instance block have their own purposes & Replacing one concept with another concept may not work always.

→ Both constructor & Instance block will be executed for every object creation, BUT Instance block first

followed by constructor.

⇒ Demo program to print No. of objects created for class.

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

    {
        count++;
    }

    Test()
    {
    }

    Test(int i)
    {
    }

    Test(double d)
    {
    }

    p s v main(String args)
    {
        Test t1=new Test();
        Test t2=new Test(10);
        Test t3=new Test(10.5);

        S.O.P("No. of objects
            created:"+count);
    }
}
```

⇒ Rules for Writing Constructor

① Name of class & Name of the constructor must be matched.

② return type concept is not applicable for constructor, even for void also.

By mistake if we are trying to declare return type for constructor, we won't get any C.E, bcz compiler treates it as a method.

```
class Test
{
    void Test()
    {
        S.O.P("method but not constr.");
    }
}
```

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

③ The only Applicable modifiers for constructors are :-
(public, private, default, protected)

If we are trying to use any other modifier we will get C.E.

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

→ C.E. modifier static not applicable here

⇒ default Constructor

→ compiler is responsible to generate default constructor but Not JVM.

→ 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.

→ prototype of default constructor :

Conclusion Note :

- Every Default constructor (which is provided by compiler) is always No-arg.

But

Every No-arg constructor is Not Default.

- If we write constructor with No-argument in our code, it is not default constructor.

```
class Test
{
    Test()
    {
        // any code
    }
}
```

Write by programmer

```
class Test
{
    Test()
    {
        super();
    }
}
```

By compiler

① It is always No-arg constructor

② The access-modifier of default constructor is exactly same as class's access-modifier (Applicable only for public, default)

③ It contains only one line :

Date / /
Page

super();

It is a ~~zero~~ no-argument call to super class constructor

programmer's code	compiler-generated code

class Test
{
}
}
	class Test
	{
	Test()
	{
	super();
	}
	}

public class Test
{
}
}
	public class Test
	{
	public Test()
	{
	super();
	}
	}

public class Test
{
void Test()
{
}
}
	public class Test
	{
	public Test()
	{
	super();
	}
	void Test()
	{
	}
	}

Date / /
Page

programmer's code	compiler-generated code

class Test
{
Test()
{
S.o.p("A");
}
}
	class Test
	{
	Test()
	{
	super();
	S.o.p("A");
	}
	}

class Test
{
Test(int i)
{
super();
}
}
	class Test
	{
	Test(int i) NO change
	{
	super();
	}
	}

class Test
{
Test()
{
this(10);
}
Test(int i)
{
}
}
	class Test
	{
	Test()
	{
	this(10);
	}
	Test(int i)
	{
	super();
	}
	}

→ The First line inside every constructor should be either super() or this()

And if we are not write anything then compiler will always place.
super()

→ **Conclusions :**

case-1 : we can take super() / this() only in first line of constructor, if we are trying to take any where else, we will get C.E.

```
class Test
{
    Test()
    {
        S.O.P("constructor");
        super();
    }
}
```

C.E. call to super must be first statement in constructor

case-2 : within a constructor, we can take either super() (or) this(), but not both simultaneously.

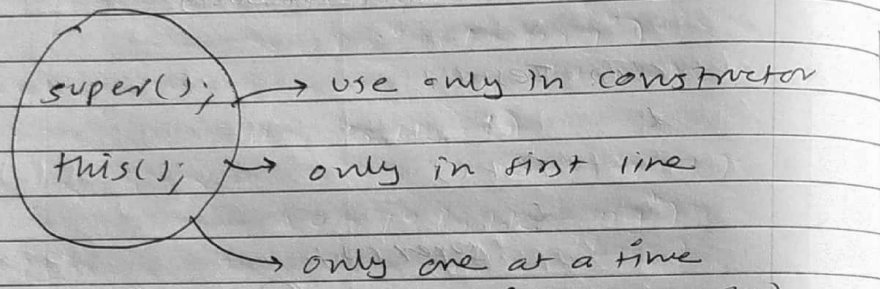
```
class Test()
{
    Test()
    {
        super();
        this();
    }
}
```

C.E call to this must be first statement in constructor

case-3 : we can use super() / this() only inside constructor, if we are trying to use outside of constructor, we will get C.E.

```
class Test
{
    public void m()
    {
        super();
        S.O.P("Hello");
    }
}
```

i.e we can call constructor directly from another constructor only.



⇒ super(), this()	super, this
① There are constructor calls to call super class & current class constructor	These are keywords to refer super class & current class instance members.
② we can use only in constructors as first line	we can use anywhere except static area
③ we can use only once in constructor	we can use any number of times.

```
class Test
{
    p s v main(String[] args)
    {
        s.o.p(super.abc());   (X)
    }
}
```

C.E: non-static variable super cannot be referenced from a static content

⇒ Overloaded constructors

⇒ within a class, we can declare multiple constructors, & all these constructors having same name but different arguments

Hence

All these constructors all known as overloaded constructors.

Hence

Overloading concept applicable for constructor.

Ex: class Test

```
{
    Test()
    {
        this(10);
        s.o.p("no-arg");
    }
```

```
Test(int i)
{
    this(10.5);
    s.o.p("int-arg");
}
Test(double d)
{
    s.o.p("double-arg");
}
p s v main(String[] args)
{

    Test t = new Test(1);
    // double-arg
       int-arg
       no-arg

    Test t1 = new Test(10);
    // double-arg
       int-arg

    Test t2 = new Test(10.5);
    // double-arg

    Test t3 = new Test(10L);
    // double-arg
}
}
```

→ For constructors, Inheritance & overriding concepts are Not Applicable

BUT

overloading concept is Applicable

↳ Every class in Java, including abstract class can contain constructors

BUT

Interfaces can't contain constructors.

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

→ Conclusions

Case 1: Recursive method call is a Run-time Exception saying "StackOverflowError"

But, in our program if there is a chance/possibility of recursive constructor invocation, then code won't compile & we will get CE.

```
class Test
{
    p s v m1()
    {
        m2();
    }
    p s v m2()
    {
        m1();
    }
    p s v main(Str arg)
    {
        m1();
        S.o.p("Hello");
    }
}
```

(R.E.) StackOverflowError

```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int i)
    {
        this();
    }
    p s v main(Str arg)
    {
        S.o.p("Hello");
    }
}
```

(C.E.) Recursive constructor invocation

Compiler inserts

case-2 :

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

Compiler inserts

```
P()
{
    super();
}
C()
{
    super();
}
```

✓ BCZ

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

→ Compiler inserts

```
P()
{
    super();
}
C()
{
    super();
}
```

BCZ

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

→ Compiler inserts

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

✗ BCZ

But, here no constructor is there with no-arg

(C.E.) cannot find symbol
Symbol: constructor P()
location: class P

Note :

① If parent class containing any argument constructors, then while writing child classes we have to take special care w.r.t constructors.

★ ② Whenever we are writing any argument constructor, it is highly recommanded to write no-arg constructor also.

case-3 : class P
{
P() throws IOException
{
}
}
class C extends P
{
}

C() { super(); }

(X) Bcz, compiler will add

C.E: unreported exception java.io.IOException in default constructor

class P
{
P() throws IOException
{
}
}
class C extends P
{
C() throws IOException/Exception/Throwable
{
super();
}
}

Note : If parent class constructor throws any checked exception, compulsory child class constructor should throw the same (or) its parent Exception, otherwise we will get C.E

⇒ we can't handle exception by try-catch, bcz, if we do so, super() will not be the first statement

⇒ which of the following is valid?

① The main purpose of the constructor is

Create an object ⊕.

② The name of constructor need not to be same as class name ⊕

③ default constructor generated by JVM ⊕

Interview ★

⊛ singleton classes

↝ For any java class, if we allow to create only one object, such type of class is called singleton class.

Ex.
- Runtime
- BusinessDelegate
- ServiceLocator

↝ Advantage :-

- If several people have same requirement, then its not recommand to create a separate object for every requirement.

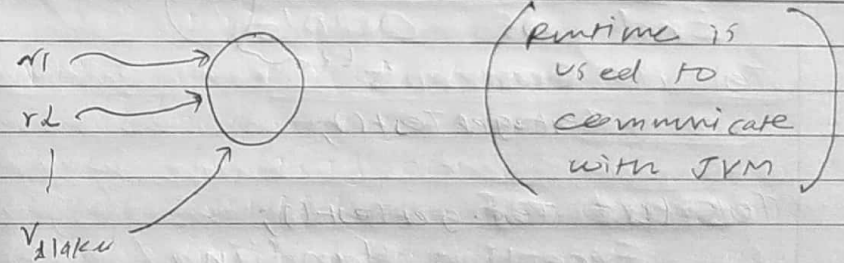
We have to create only one object & we can reuse same object for every similar requirement.

so that performance & memory utilization will be improved.

- This is the central idea of singleton classes.

Ex.

```
Runtime r = Runtime.getRuntime();
Runtime r2 = Runtime.getRuntime();
:
:
Runtime rLakh = Runtime.getRuntime();
```



⊕ ↝ How to create our own singleton class?

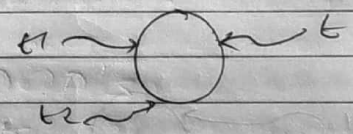
Ans :- we can create our own singleton classes & for this we have to use private constructor, private static variable & public Factory Method.

Approach 1 :-

```
class Test
{
    private static Test t = new Test();

    private Test()
    {
    }

    public static Test getTest();
    {
        return t;
    }
}
```




```
Test t1 =
        Test.getTest();

Test t2 = Test.getTest();
```

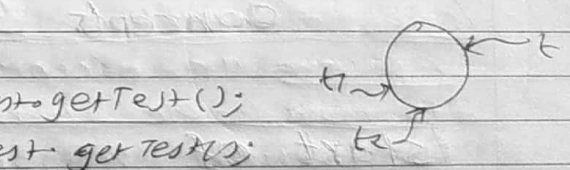
⇒ Runtime class internally implemented in this approach.

Approach 2 :- Creates object only when first request comes.

```
class Test
{
    private static Test t = null;

    private Test()
    {
    }

    public static Test getTest()
    {
        if (t == null)
        {
            t = new Test();
        }
        return t;
    }
}
```




```
Test t1 = Test.getTest();
Test t2 = Test.getTest();
```

Q. Class is not final, but we are not allowed to create child classes, How it is possible?

Ans :- By declaring every constructor as private we can restrict child class creation

Date __/__/____
Page _____

```
class P
{
    private P()
    {
    }
}
```

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

(X) → BCZ compiler adds

CE: P() has private access in class P

End of OOP's Concepts

Start of Exception Handling