QUESTIONS	ANSWERS

Data Hiding	Outside person can't access our internal data directly or our internal data should not go out directly, this OOP feature is nothing but Data Hiding. After validation or authentication outside person can access our internal data, for Example: After providing proper username and password we can able to access our gmail inbox information. Ex 2: Even though we are valid customer of the bank we can able to access our account information and we can't access other's account information.
	By declaring data member (Variable) as private we can achieve Data Hiding.
	Public class Account {
	private double balance;
	public double getBalance () {
	//VALIDATION;
	return balance;
	}
	}
	The main advantage of data hiding is security.
	NOTE: It is highly recommended to declare data member (Variable) as private.

Abstraction.	Hiding internal implementation and just highlight the set of services what we are offering is a concept of Abstraction. Through bank ATM GUI screen, bank people are highlighting the set of services what they are offering without highlighting internal implementation. The Main Advantages of Abstraction are: 1.) Security: We can achieve security because we are not highlighting our internal implementation. 2.) Without affecting outside person we can able to perform any type of changes in our internal system and hence enhancement will become easy. 3.) It improves maintainibility of the application. 4.) It improves easiness to use our system.

BY using interfaces and abstract classes we can implement Abstraction.

Encapsulation.

The process of binding data and corresponding methods into a single unit is nothing but encapsulation.
Example:

```
class Student {
        data Members;
                +
        Methods (Behaviour);
                }
```

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

Encapsulation = Data Hiding + Abstraction

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

The Main Advantages of Encapsulation are:
1.) We can achieve security.
2.) Enhancements will become easy.
3.) It improves maintainabillty of the application.
The main advantage of Encapsulation is we can achieve Security but the main disadvantage of Encapsulation is it increases length of the code and slows down execution.

Tightly Encapsulated Class

A class is said to be Tightly Encapsulated if and only if each and every variable declared as private, whether class contains corresponding getter and setter methods are not and whether these methods are decalred as public or not this things we are not required to check.

example:

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

Which of the following classes are tightly encapsulated ?

```
class A {                                       //Tightly Encapsulated
    private int x=0;
      }
class B extends A {                             //Not Tightly Encasulated
   int y = 5;
                }
clas C extends A  {                             //Tightly Encapsulated
     private z=20;
                }
```

Next Example:

```
class A {                                       //NOT TIGHTLY ENCAPSULATED
  int a=5;
      }
class B extends A {                             //NOT TIGHTLY ENCAPSULATED
   private b= 6;
```

```
                    }
class C extends B {                         //NOT TIGHTLY ENCAPSULATED
      private int z=30;
                    }
```

NOTE: If the parent class is not tightly encapsulated then no child class is tightly encapsulated.

IS-A Relationship (Inheritance Concept)

* It is also Known as Inheritance.

* The main advantage of Is-A Relationship is Code Reusability.
* By using extends Keyword we can implement Is-A Relationship.

Example:

```
class P {
        public void m1 () {
             System.out.println ("PARENT");
                        }
        }
class C extends P {
        public void m2 () {
           System.out.println ("CHILD");
                        }
                    }

class Test {
   public static void main (String args[]) {
(1)   P p = new P();
        p.m1 ();
        p.m2 ();
```

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

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

(4) C c1 = new P ();

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

CE : Incompatible types
found : P
required : C

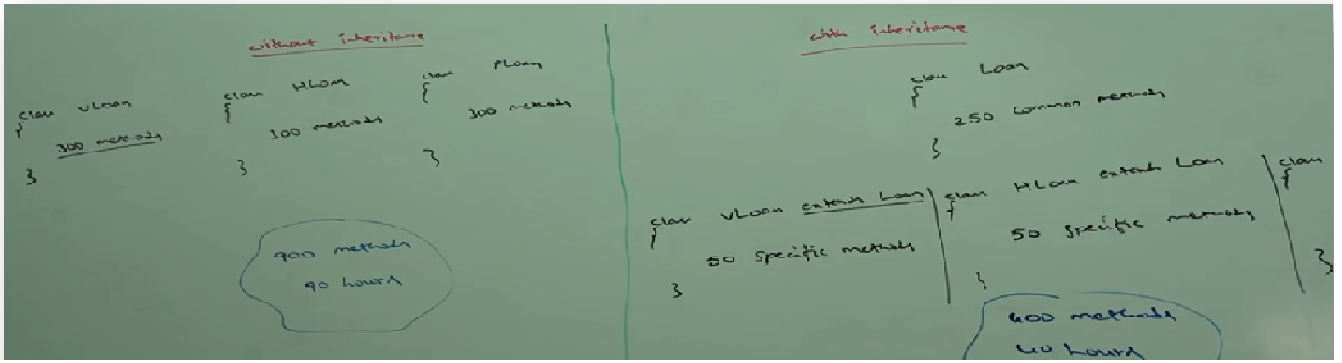
Conclusions:

1.) Whatever methods parent has bydefault available to the child and hence on the child reference we can call both parent class and child class methods.

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

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

4.) Parent reference can be used t hold child object but Child reference can not be used to hold parent object.



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

The Specific methods which are applicable for a particular child, we have to define in child class.
Total Java API is implemented based on inheritance concept.
The Most common methods which are applicable for any java object are defined in Object class and hence every class in java is the child class of object either directly or indirectly, So that object class methods by default available to every java class without rewritting. Due to this Object class access root for all java classes.

Throwable class defines the most common methods which are required for every exception and error classes hence this class acts 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 in classes.

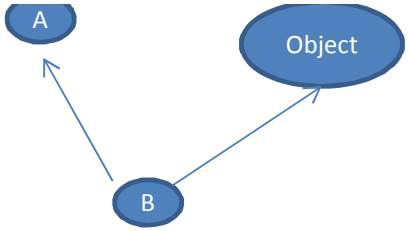
```
class A extends B, E {                    //Compile time error
        ……
                }
```

NOTE: 1.) If our class doesn't extends any other class then only our class is direct child class of Object.

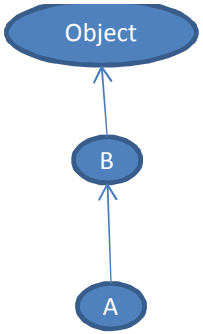
```
    Ex: class A {
            ……
                }
```

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

```
  ex: class A extends B {
            ……
                }
```



Multiple Inheritance [Wrong way]



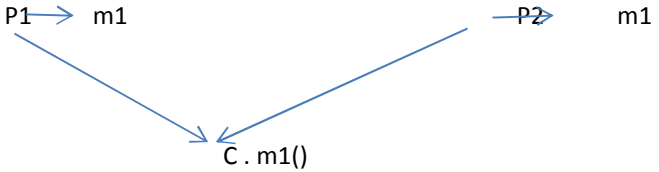
Multi Level Inheritance

[Right]

NOTE: Either directly or indirectly java won't provide support for inheritance with respect to classes.

Why Java won't provide support for Multiple inheritance.

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



AMBIGUITY PROBLEM

But interface can extend any number of interfaces simultaniously hence java provide support for Multiple Inheritance with respect to interfaces.

ex:

interface A {
……

interface B {
……

}

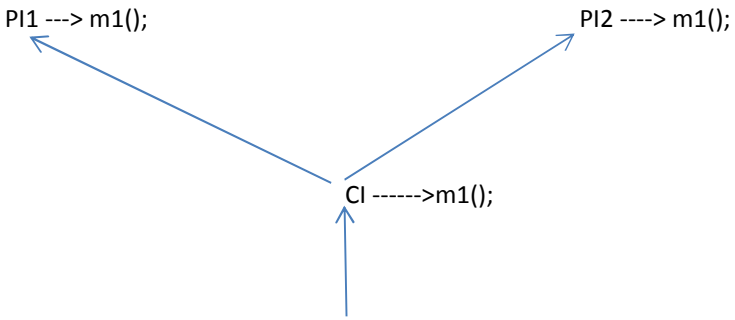
}

interface C extends A, B {

……

}

Why Ambiguity problem won't be there in interfaces ?



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

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

Cyclic Inheritance :- Cyclic inheritance is not allowed in java, ofcourse It is not required.

example:

class A extends A {

A

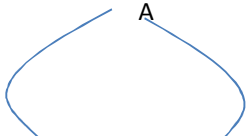
}

class A extends B {

A

}

class B extends A {



}

B

Has-A Relationship.

1.) Has-A relationship is also known as composition or Aggregation.
2.) There is no specific keyword to implement Has-A relation but most of the times we are depending on **new** keyword.

3.) The main advantage of Has-A relationship is the reusabiltiy of code.
ex:

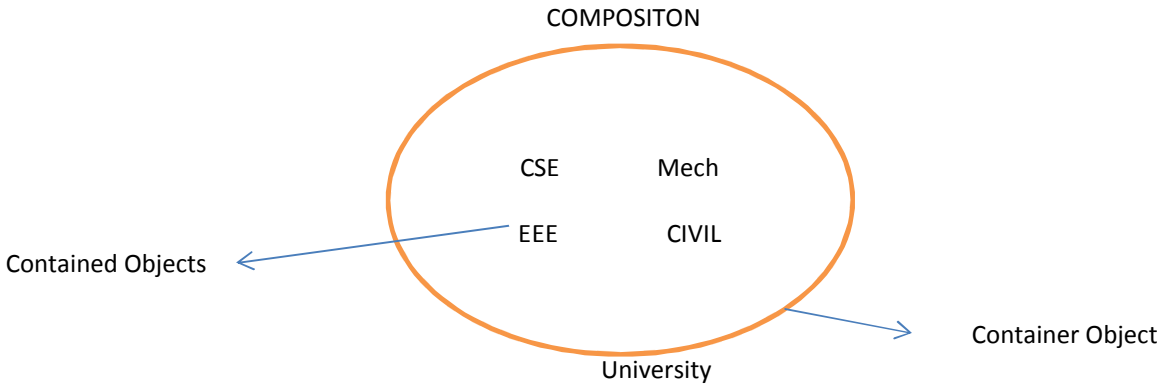
```
class Car   {
      Engine e = new Engine ();
          }
```

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

Car Has-A Engine reference

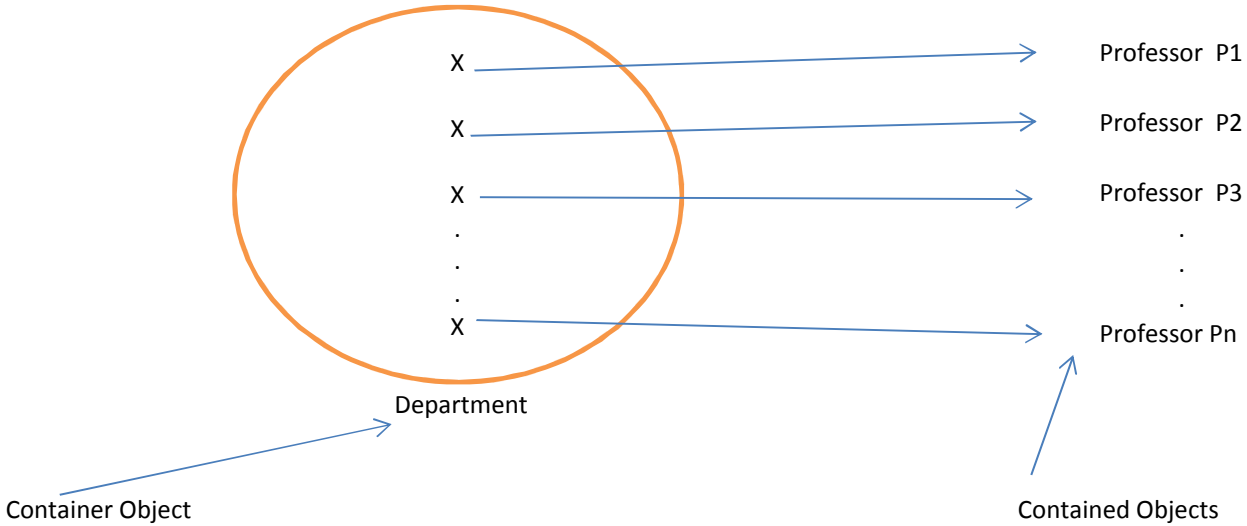
Difference between Composition and Aggregation.

Without existing Container object if there is no chance of existing contained objects then, Container and contained objects are Strongly associated and this strong association is nothing but Composition.
ex: University consist of several departments without existing university there is no chance of exisiting department hence university and departments are strongly associated and this strong association is nothing but Compositon.



Aggregation: Without Existing container object if there is chance of existing contained object then container and contained objects are weakly associated and this weak association is nothing but Aggregation.

Ex: Department consist of several professors without existing departments there maybe a chance of exisiting professor objects hence department and professor objects are weakly associated and this weak association is nothing but Aggregation.

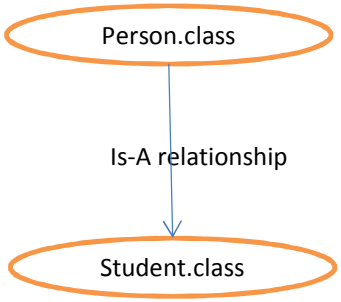


NOTE: 1.) In composition objects are Strongly Associated whereas in Aggregation Objects are Weakly Associated.

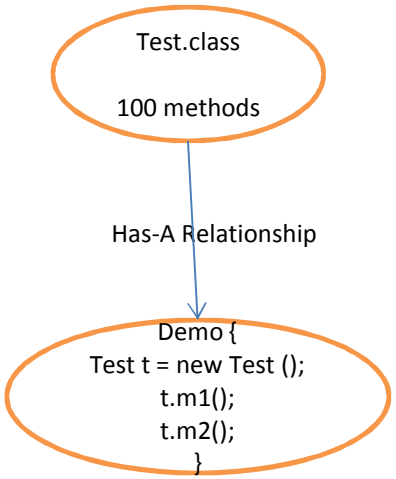
2.) In compositon container object holds directly contained objects whereas in aggregation container objects holds just the references of contained objects.

Is-A Vs Has-A

If we want total functionality of a class automatically then we should go for Is-A relationship.
Ex:



If we want part of the functionality then we should go for Has-A Relationship.
Ex:



METHOD SIGNATURE :- In Java method signature consist of method names followed by argument types.
ex: public static int m1 (int I, float f);
m1 (int, float)

Return type is not part of method signature in java.
Compiler will use method signature to resolve method calls.

Within a class two method with the same signature not allowed.

```
ex: public class Test {
        public int m1 (int i) {
                return I;
                }
        public int m2 (int x) {
                return x;
                }
        }
```

O/P : CE: m1(int) is already defined in Test.

OVERLOADING

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

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

C Language:	JAVA Language:	
abs (int i)	abs (int i);	Overloaded.
labs (long l)	abs (long l);	
fals (float f)	abs (float f);	

But in java we can declare mutiple methods with same name but different argument types. Such types of methods are called Overloaded methods. Having overloading concept in java reduces complexity of programming.

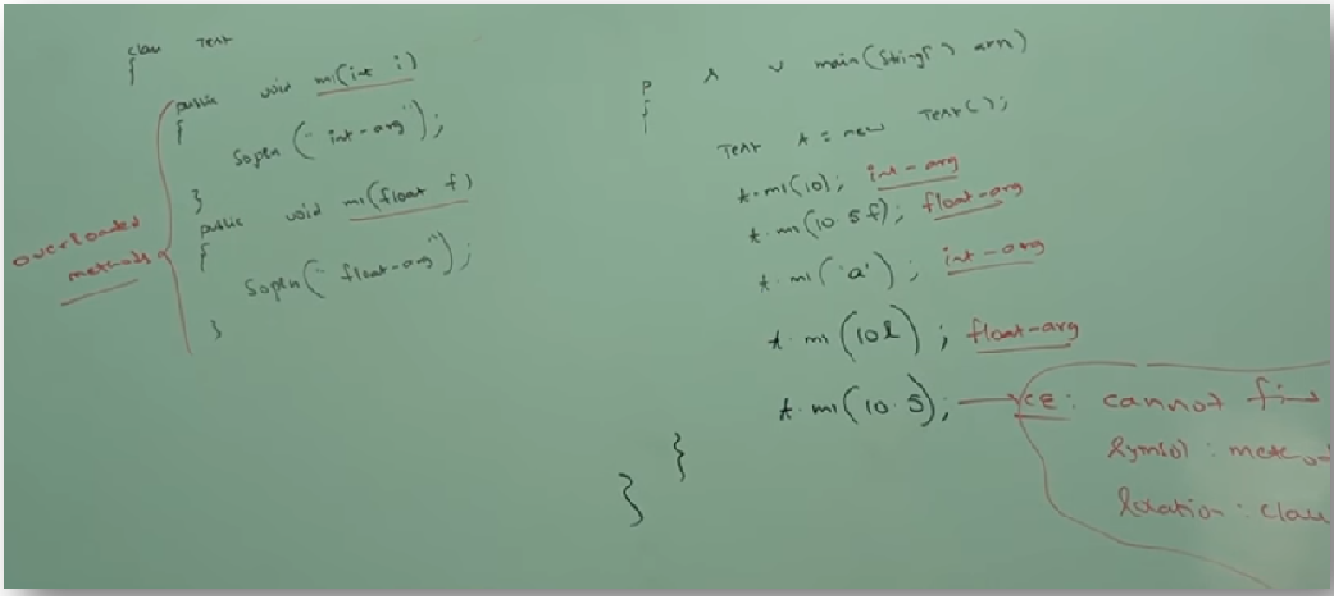
```
EX: public class Test {
        public void m1 () {
                S.O.P ("NO-Arg");
                }
        public void m1 (int i) {
                S.O.P ("INTEGER-Arg");
                }
        public void m1 (float f) {
                S.O.P ("Float- Arg ");
                }
        }
public static void main (String args[]) {
        Test t = new Test ();
        t.m1 ();                        // NO-Arg
        t.m1 (5);                       // INTEGER-Arg
        t.m1 (5.2);                     // Float-Arg
                                }
```

**** In Overloading method resolution always takes care by Compiler based on reference type hence, Overloading is also considered as Compile time Polymorphism OR Static Polymorphism OR Early Binding.

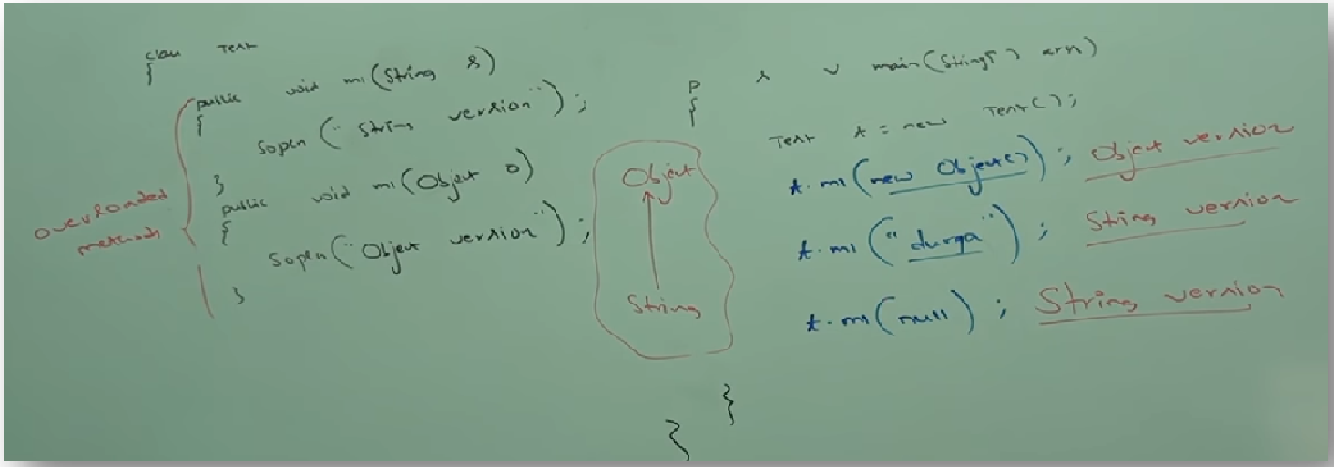
CASE: 1=> Automatic promotion in overloading

While resolving overloaded methods if exact match method is not available then we won't get any compile time error immideately, first it will promote argument to the next level and check whether matched method is available or not, if matched method is available then it will be considered, and if the matched method is not available then compiler promotes argument once again to the next level, this process will be continued untill all possible promotions, still if the matched method is not available then we will get compile time error.

The Following are all possible promotions in overloading :
byte->short->int->long->float->double
char->int
this process is called automatic promotion in overloading.
Example:

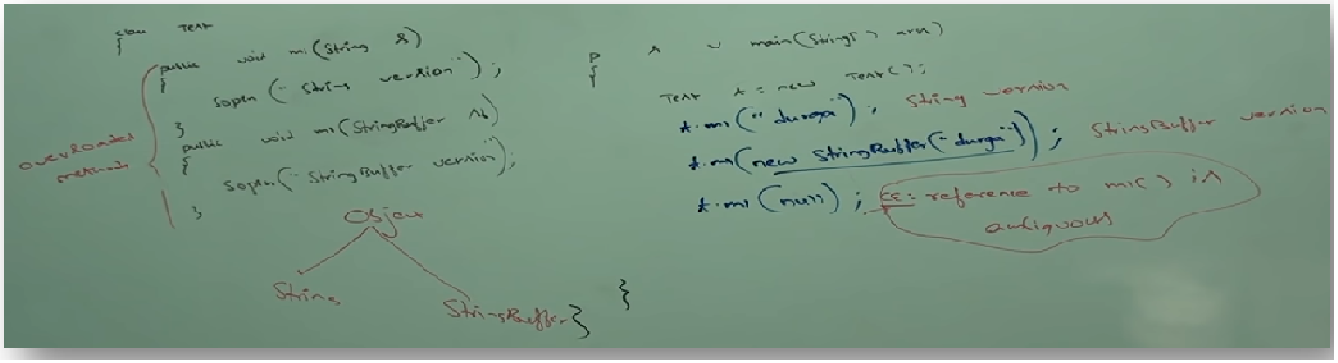


CASE: 2=>

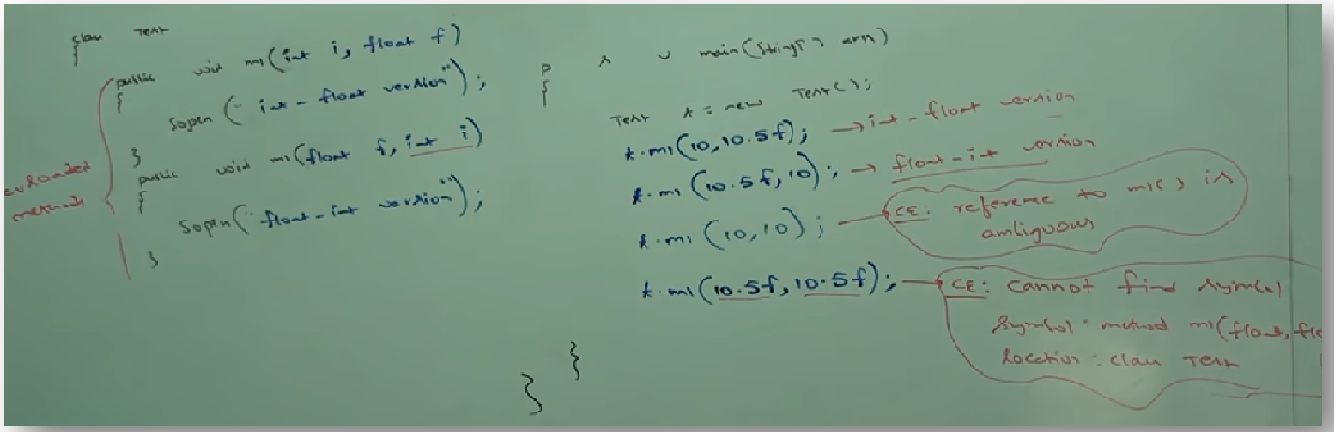


NOTE: While resolving overloaded methods compiler will always gives the precedence for child type argument and compared with parent type argument.

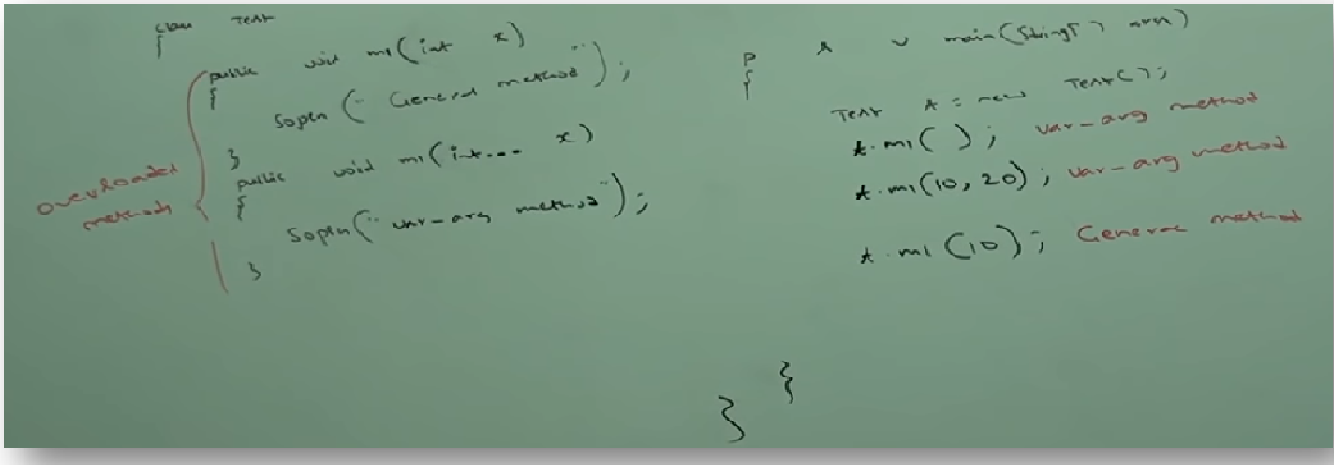
CASE: 3=>



CASE: 4=>

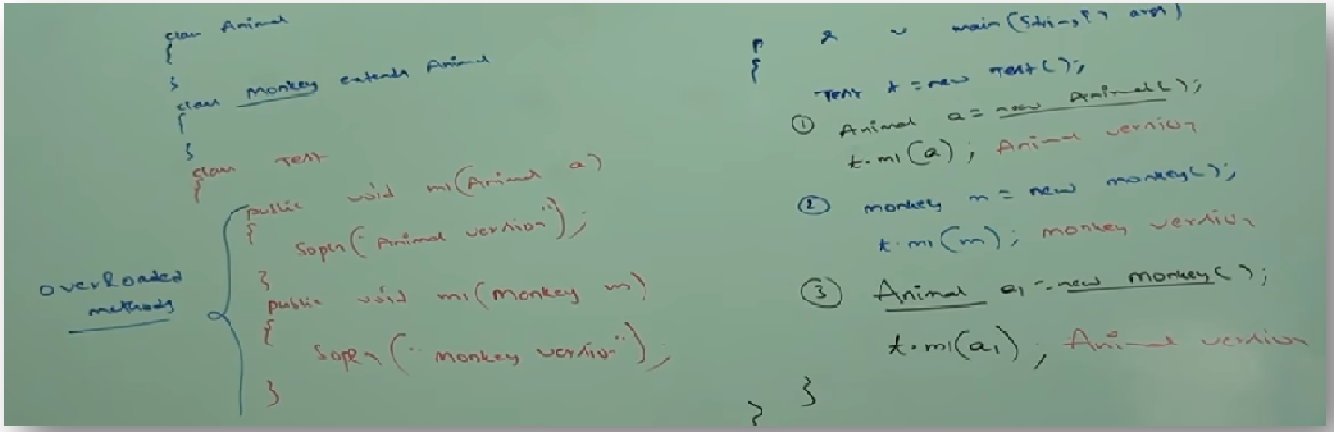

class Test
public void m(int i, float f)
Sopln("int - float version");
public void m(float f, int i)
Sopln("float - int version");
overloaded method
psvm(String[] args)
Test t = new Test();
t.m(10,10.5f); → int - float version
t.m(10.5f,10); → float - int version
t.m(10,10); → CE: reference to m() is ambiguous
t.m(10.5f,10.5f); → CE: cannot find symbol
Symbol: method m(float,fl
location: class Test

CASE: 5 =>


class Test
public void m(int x)
Sopln("General method");
public void m(int... x)
Sopln("var-arg method");
overloaded method
psvm(String[] args)
Test t = new Test();
t.m(); var-arg method
t.m(10,20); var-arg method
t.m(10); General method

In general, var-arg method will get least priority i.e. if no other method matched then only var-arg method will get the chance, it is exactly same as default case inside a switch.

CASE: 6 =>



NOTE: In overloading method resolution always takes care by compiler based on reference type. In overloading runtime object won't play any role.

OVERRIDING

Whatever methods parent has bydefault available to the child through inheritance, if child class not satisfied with the parent class implementation then child is allowed to redfine that method based on it's requirement, this process is called Overriding.

The parent class method which is overriden is called overriden method and child class method which is overriding is called overriding method.

Ex:

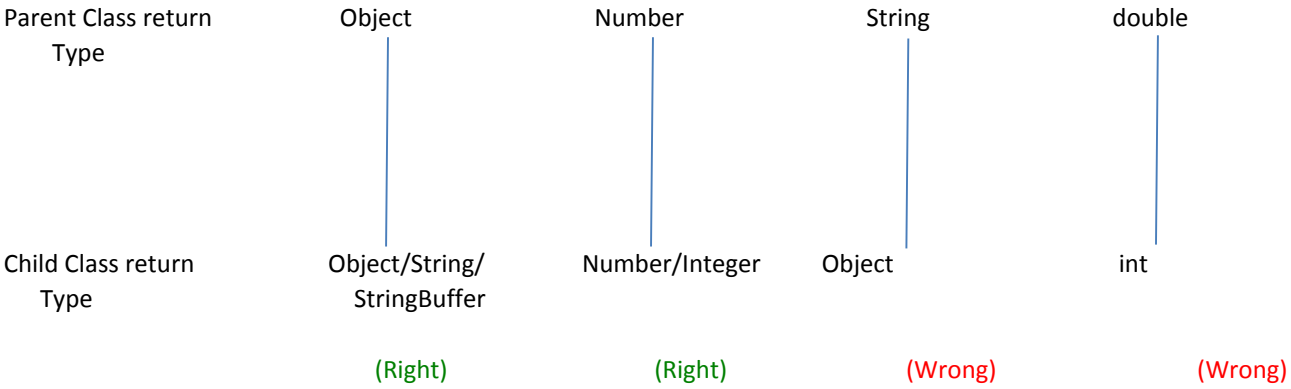
```
class P {
    public void property () {
                S.O.P ("cash+car");
                                    }
   public void marry () {                                  //Overriden method
                S.O.P ("Shubhlaxmi" );
                                    }
           }
class C extends P {
     public void marry () {                                 // Overriding method
                S.O.P. ("KATRINA" );
                                    }
              }
class Test {
      public static void main (String[] rgs ) {
              1.) P p = new P ();
                    p.marry ();                             // Parent class method will be called
              2.) C c = new C ();
                     c.marry ();                            // Child class method will be called
             3.) P p = new C ();
                    p.marry ();                             // Child class method will be called
                                                        }
              }
```

*** In Overriding method resolution always takes care by JVM based on Runtime Object and hence Overriding is also considered as Runtime Polymorphism or Dynamic Polymorphism or Late Binding.

Rules For Overriding: 1.) In Overriding method names and argument types must be matched i.e. Method signatures must be same.

2.) In Overriding return types must be same but this rule is applicable until 1.4 version only, from 1.5 version onwards we can take Co-Varient return types, According to this child class method return type need not be same as parent method return type. It's child type also allowed.

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



Co-Varient return type concept applicable only for object types but not for primitive types.

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

Based on our requirment we can define exactly same private method in child class, it is valid, but not overrding.
Example:

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

4.) We can't override parent class final method in child classes, if we are trying to override we will get compile time error. Example:

```
class P {
      public final void m1 () {
                                       }
             }
class C extends P {
     public void m1 () {            //CE: m1 () in C cannot be overriden in P. Overriden method is final.
                              }
                               }
```

5.) Parent class abstract method we should override in child class to provide implementation. Example:

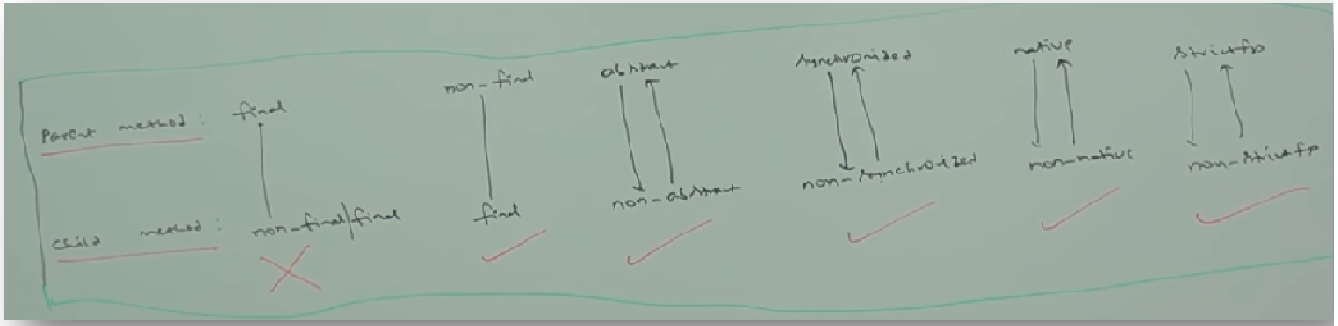
```
abstract class P {
        public abstract void m1() ;
                        }
public class C extends C {
       public void m1() {
                                    }
                                       }
```

6.) We can override non-abstract method as abstract . Example:

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

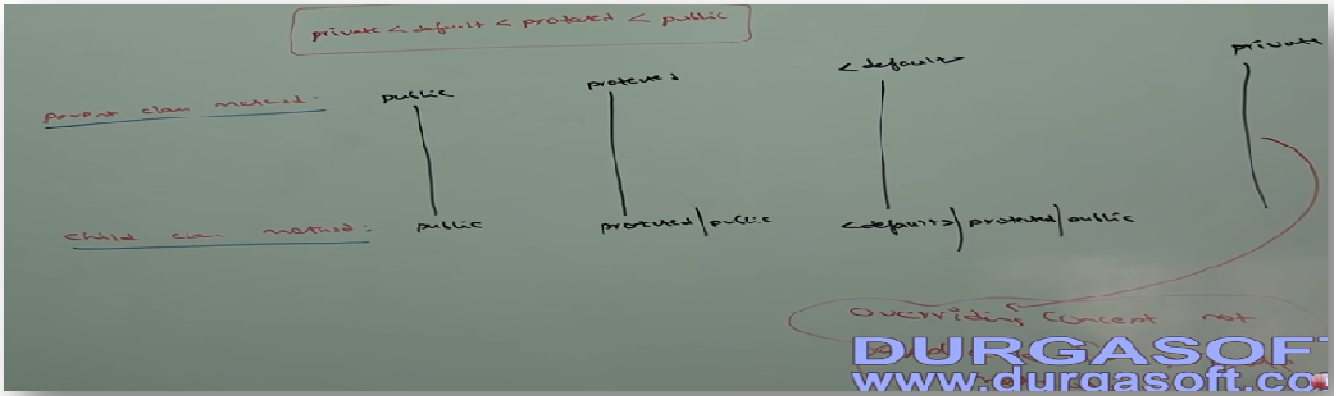
The main advantage of this approach is we can stop by availability of parent method implementation to the next level child classes.

7.) In Overriding the following modifiers won't keep any restriction:
1.) Synchronized 2.) native 3.) strictfp

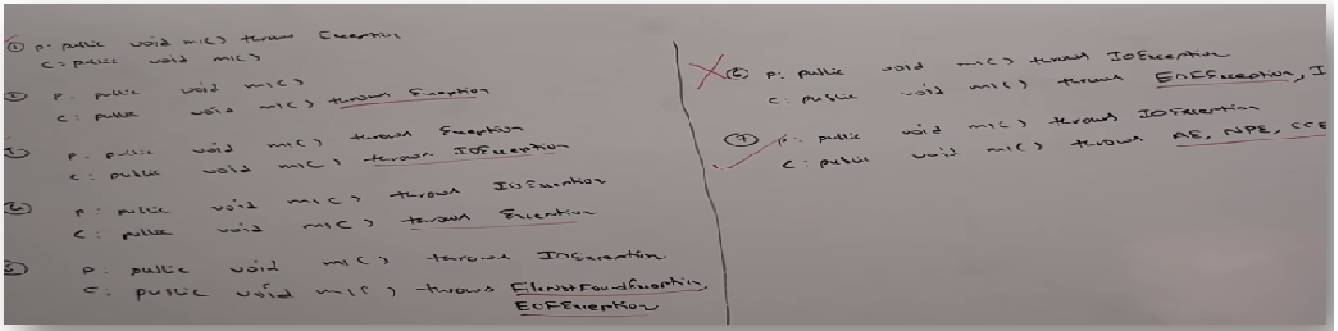


While overriding we can't reduce scope of access modifier but we can increase the scope.

```
class P {
       public void m1() {}
          }
class C extends P {                  // CE: m1() in C can not override m1() in P.
         void m1() { }                          attempting to provide weaker access privileges ; was public.
```



8.) If Child class method throws any checked exceptions compulsary parent class method should throw the same checked exception or it's parent otherwise we will get compile time error but there are no restrictions for Unchecked exceptions.



9.) Overriding with respect to static method :

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

```
ex: class P {
    public static void m1() {}
            }
class C extends P {
    public void m1 () {}                    //CE:  m1() in C cannot override m1() in P; Overriden method is static.
                        }
```

(ii) Similarly we can't override a non-static method as static.

```
ex: class P {
        public void m1() {}
                }
class C extends P {
      public static void m1() {}          // CE: m1() in C cannot override m1() in P; Overriding method is static.
                        }
```

(iii) If both parent and child class methods are static then we won't get any compile time error, it seems Overriding concept applicable for static methods but it is not overriding and it is method hiding.

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

It is method hiding but not method overriding.

METHOD HIDING

All rules of method hiding are exactly same as overriding except the following differences :

METHOD HIDING	OVERRIDING
1.) Both parent and child class method should be static	Both parent and child class method should be non-static.
2.) Compiler is responsible for method resolution based on reference type.	JVM is always responsible for method Resolution based on runtime object.
3.) It is also known as Compile time polymorphism or static polymorphism or early binding.	It is also known as runtime polymorphism Or Dynamic polymorphism or late binding.

EXAMPLE:

```
class P {
     public static void m1() {
           S.O.P ("Parent");
                                  }
        }
class C extends P {
    public static void m1 () {
          S.O.P ("Child");
                                 }
                        }
class Test {
      P.S.V.main (String args[]) {
          P p = new P();
            p.m1();                          //Parent
         C c = new C ();
           c.m1 ();                          // Child
        P p1 = new C ();
          p1.m1();                           // Parent
                                    } }
```

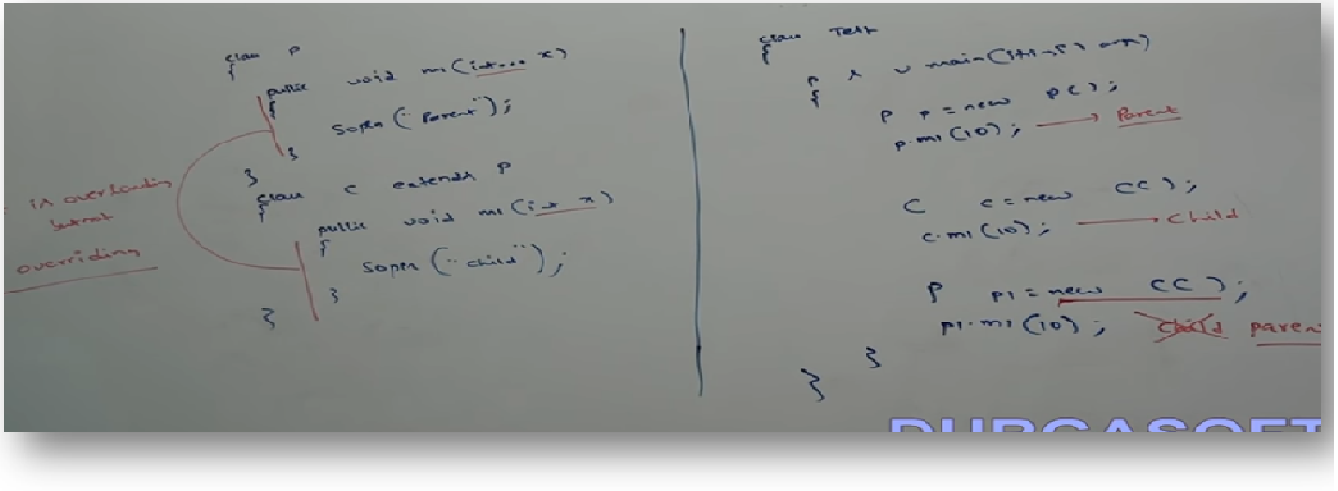
} It is method hiding but not method overriding.

If Both Parent and child class methods are non-static then it will become overriding, in this case output will Parent Child Child.

Overriding With Respect to Var-Args Methods

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

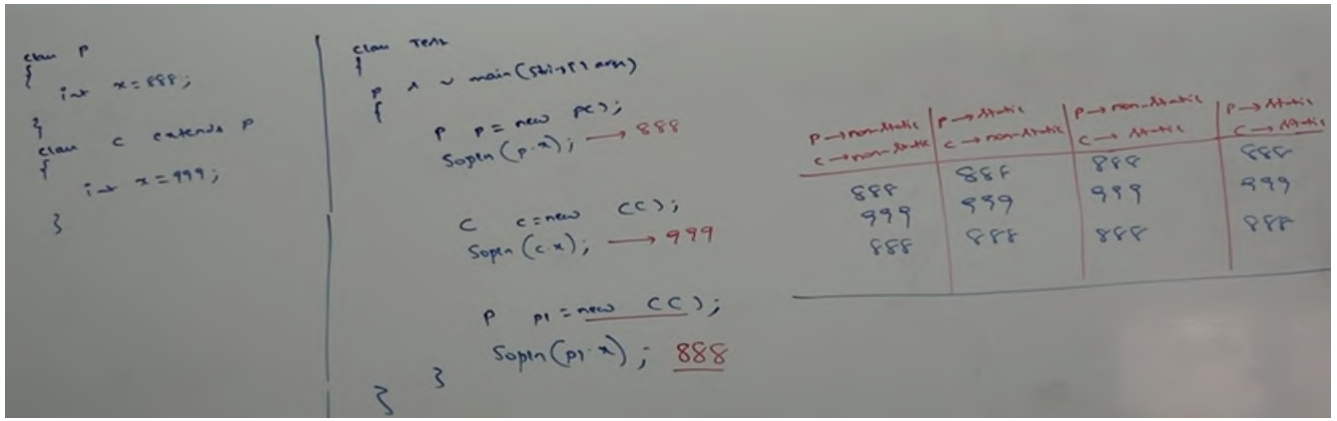
ex:



In the above program, if we replaced child method with var-arg method then it will become Overriding, in these case the output is : Parent Child Child

Overriding With Respect To Variables

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



COUPLING : The degree of dependency between the components is called Coupling.

If Dependency is more than it is considered as tighlty coupling, and if dependency is less then it is considered as losely coupling.

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 0;
```

```
        }
    }
```

The above component are said to be tightly coupled with each other because dependency between the components is more.

Tightly Coupling is not a good programming practice because it has several disadvantages.

1) Without affecting remaining components, we can't modify any component and hence enhancement will become difficult.

2) It suppresses reusability .

3) It reduces maintainability of the application.

Hence we have to maintain dependency between the components as less as possible. i.e Looesely Coupling is a good programming practice.

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

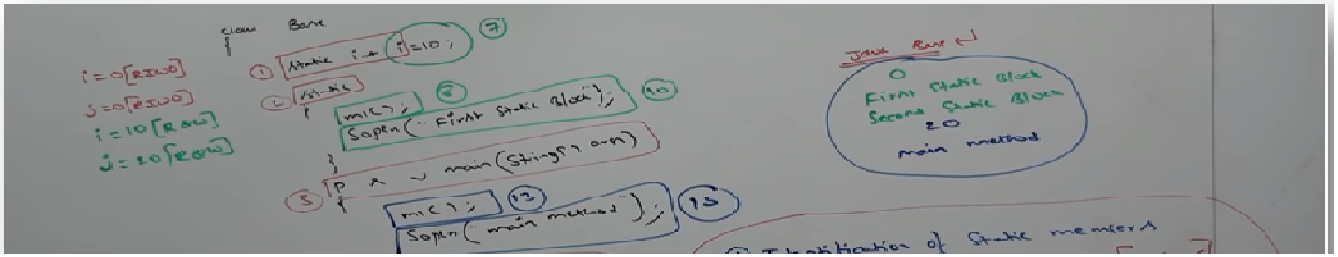
static Control Flow

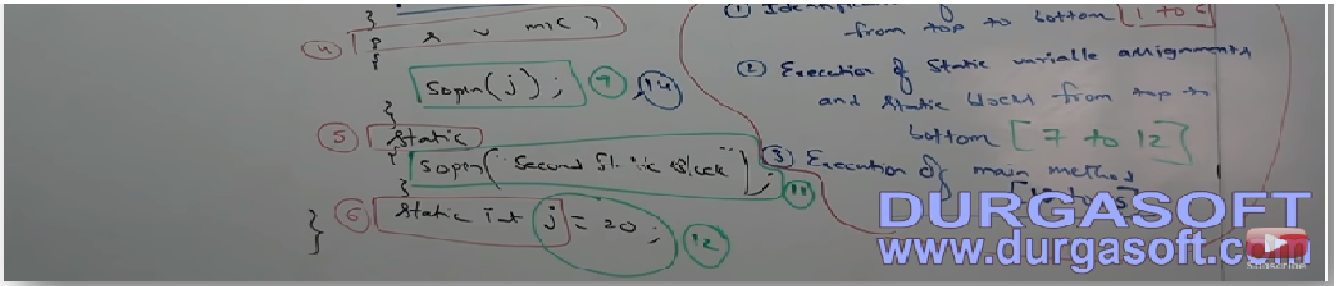
Whenever we are executing a java class the following sequence of steps will be executed as the part of static control flow.

1) Identification of static members from Top to Bottom.

2) Execution of static variable assignments and static blocks from Top to Bottom.

3) Execution of 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 and within that method if we are trying to read a variable that read operation is called indirect read.

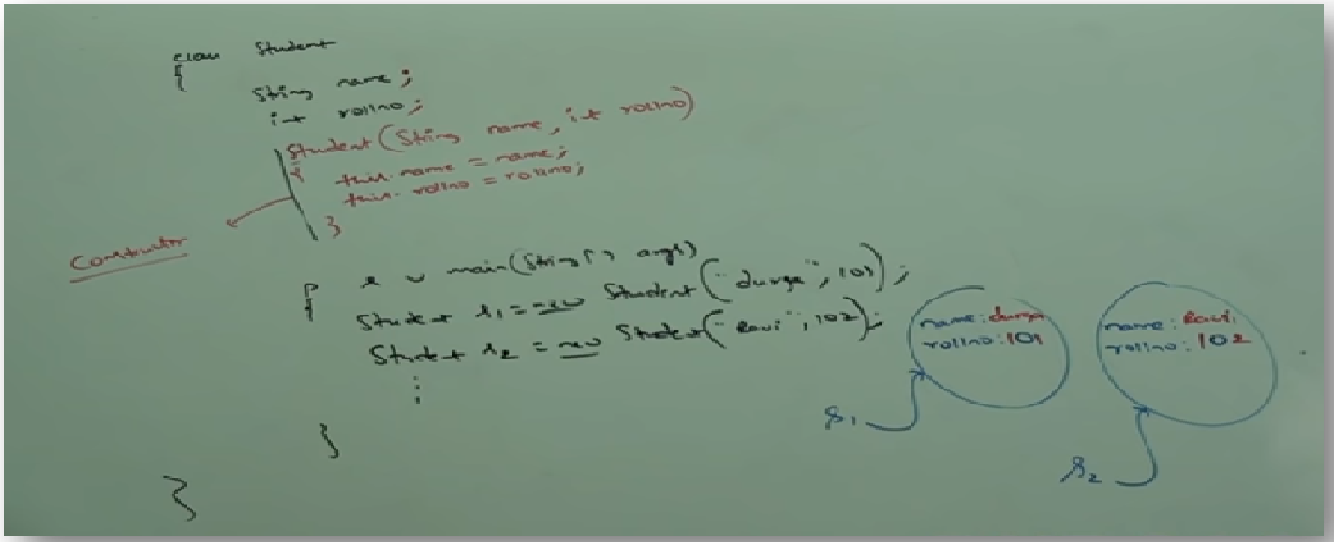
```
class Test {
          static int i=10;
         static  {
            m1();
            s.o.p (i);                    ----------Direct Read
                  }
        public static void m1() {
                        S.o.p.ln (i);  ----------Indirect Read
                                        }
                }
```

If a variable is just identified by the JVM and original value not yet assigned then the variable is said to be in read indirectly write only state (RIWO).
If a Variable is in Read Indirectly Write Only state then we can't perform direct read but we can perform indirect read.

If we are trying to read directly then we will get compile time error saying IllegalForwardReference.

CONSTRUCTORS

Once we creates an object compulsary we should perform initialization then only the object is in a position to respond properly.
Whenever we are creating an object some piece of the code will be executed automatically to perform intialization of the object, this piece of the code is nothing but Constructor.
Hence the main purpose of a Construtor is Initialization of an Object.



The Main purpose of constructor is to perform initialization of an object but not to create an Object

Difference Between Constructor and Instance block.	The main Purpose of constructor is to perform intialization of an object. But Other than initialization if we want to perform any activity for every object creation then we should go for Instance block (Like updating one entry in the database for every object creation or incrementing count value for every object creation etc.)

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

Both Constructor and instance block will be executed for every object creation but instance block first followed by constructor.

Demo Program to print number of objects created for a class :

```
class Test {
    static int count =0;
      {
          count ++;
      }
       Test () {
                  }
       Test (int i) {

                           }
      Test (double d) {

                                  }
                }
public static void main (String arg[]) {
                              Test t1 = new Test ();
                              Test t2 = new Test (5);
                              Test t3 = new Test (10.5);
                               System.out.println ("Number of objects created is "+ count);
                                                    }
```

Rules of Writing constructors:

1.) Name of the class and name of the constructors must be same.
2.) Return type concept not applicable for constructor even void also.
If we are trying to declare retrun typr for the cosntructor then we won't get any compile time error because compiler treats it as a method.

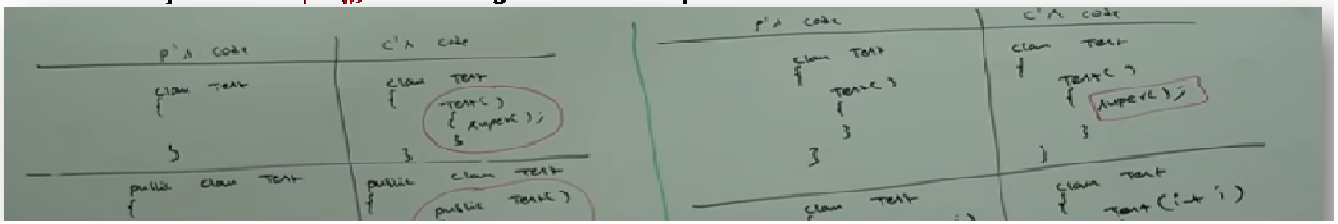
```
class Test {
      void Test () {
        S.o.p ("Method but not Constructor");          // It is a method not Constructor
                        }
              }
```

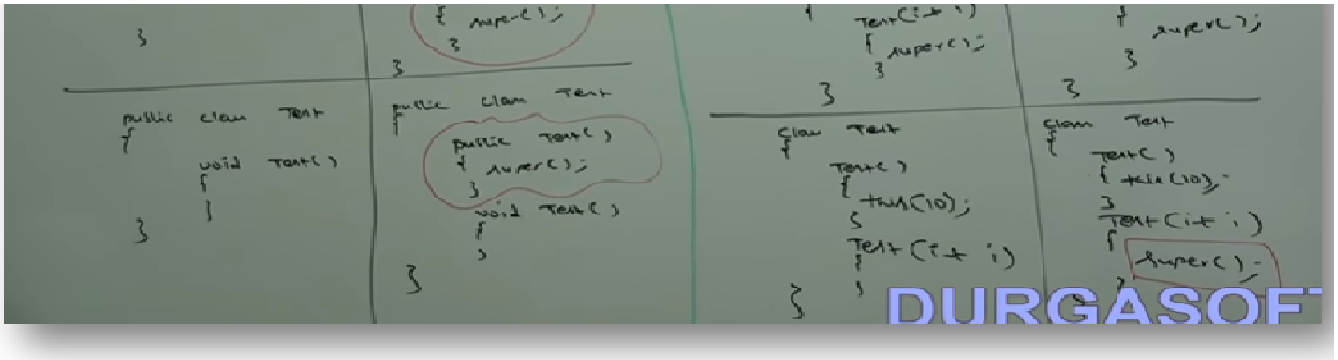
Hence it is legal (but stupid) to have a method whose name is exactly same as class name.

The Only applicable modifiers for constructors are public, private, protected, default. If we are trying to use any other modifier we will get compile time error.

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 at least one constructor then compiler won't generate default constructor, hence every class is java can contain constructor, it maybe default constructor generated by compiler or customized constructor explicitly provided by programmer but not both simultaniously.

Prototype of Default Constructor:- It is always No-Arg construtor .
The access modifier of default construtor is exactly same as access modifier of class.(This rule is applicable only for public and default).
It contains only one line super(); It is a no-argument call to super class construtor.





The first line inside every construtor should be either super() OR this(), and if we are not writing anything then compiler will always plays super().

CASE: 1=> we can take super() or this() only in first line of construtor, if we are trying to take anywhere else we will get compile time error. Ex:

```
class Test {
        Test () {
                System.out.println ("Construtor");
                super();
               }
            }
```

CE: call to super() must be the first statement in Construtor.

CASE: 2 => Within the construtor either super() or this() but not both simultaniously.

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

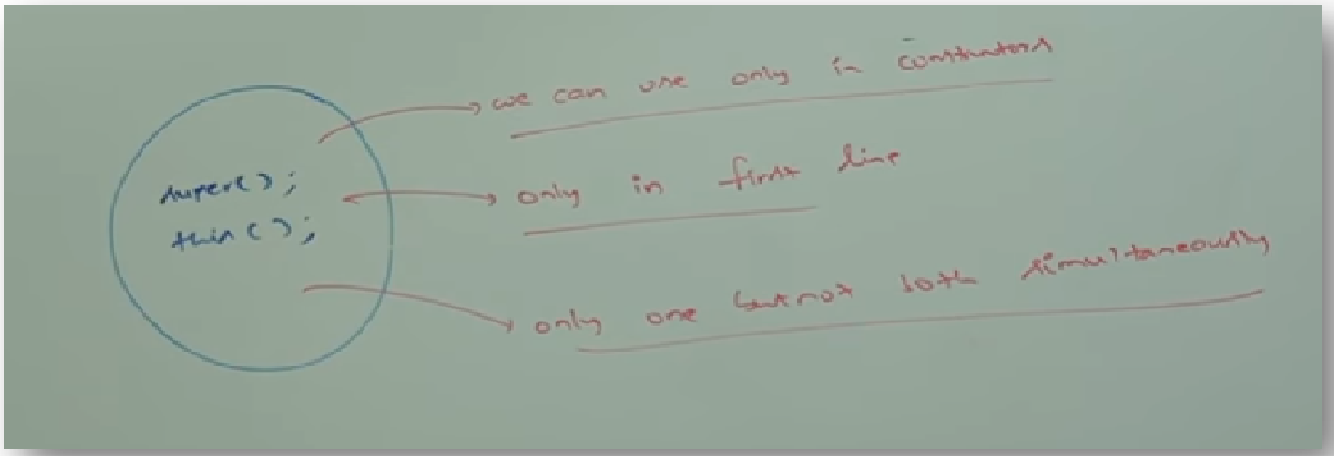
CE: call to this() must be the first statement in the construtor.

CASE :3 => We can use super() or this () only inside a construtor, If we are trying to use outside of construtor we will get compile time error.

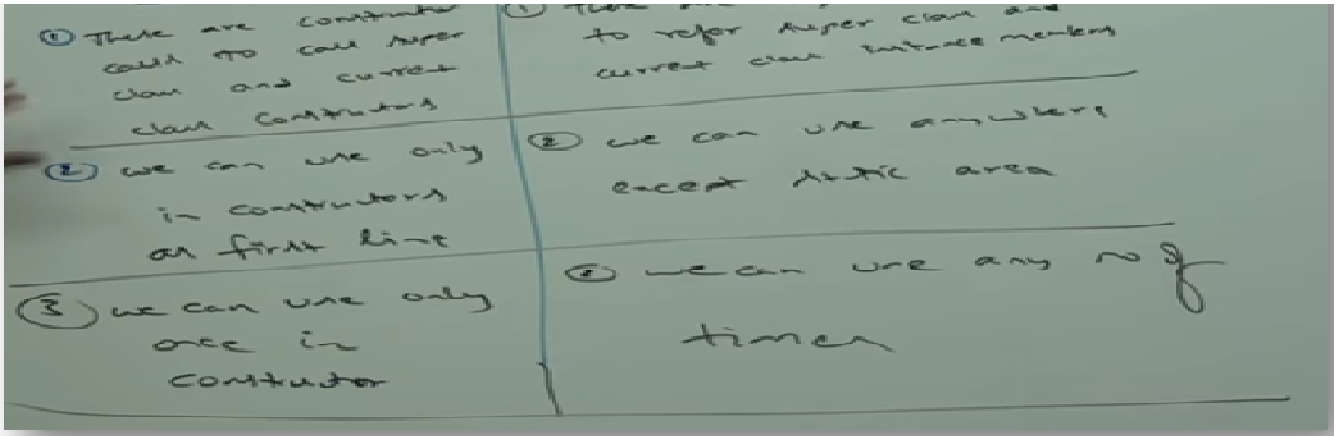
```
class Test {
        public void m1 () {
                super();
                System.out.println ("METHOD INSIDE");
                                }
                }
```

CE: call to super must be the first statement in the constructor.

IE. We can call a constructor directly from another construtor only.



Difference between Super(), this() and super, this:-



Ex:

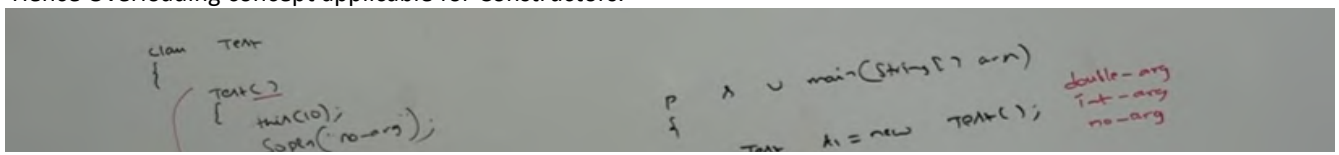
```
class Test {
        p.s.v.main (String arg[]){
                S.o.p (super.hashCode());
                                        }
                }
```

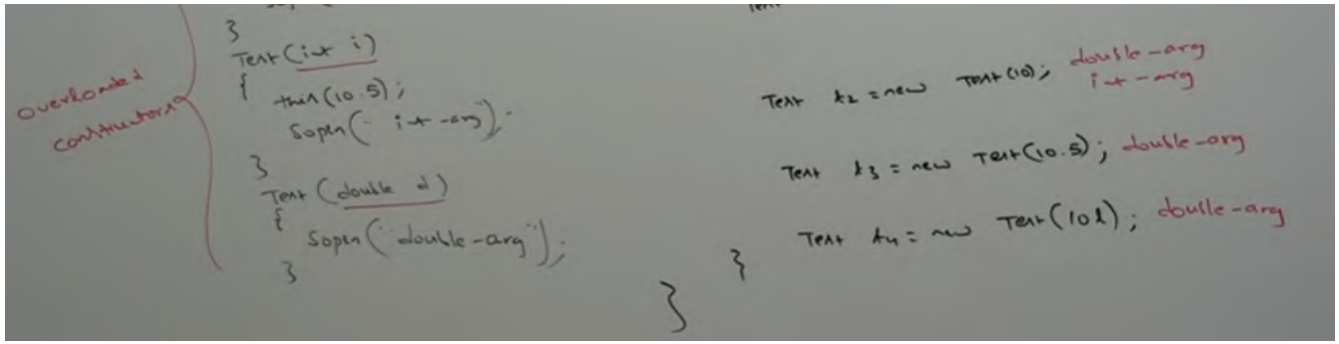
CE: non-static variable super can not be refered from a static content.

OVERLOADED CONSTRUCTOR

Within a class we can declare multiple constructors and all these construtors having same name but different type of arguments hence all these construcotrs are considered as Overloaded Constructors.

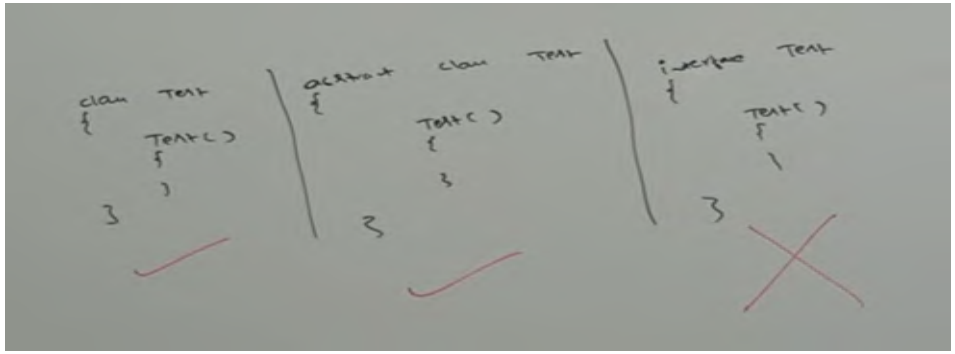
Hence Overloading concept applicable for Constructors.





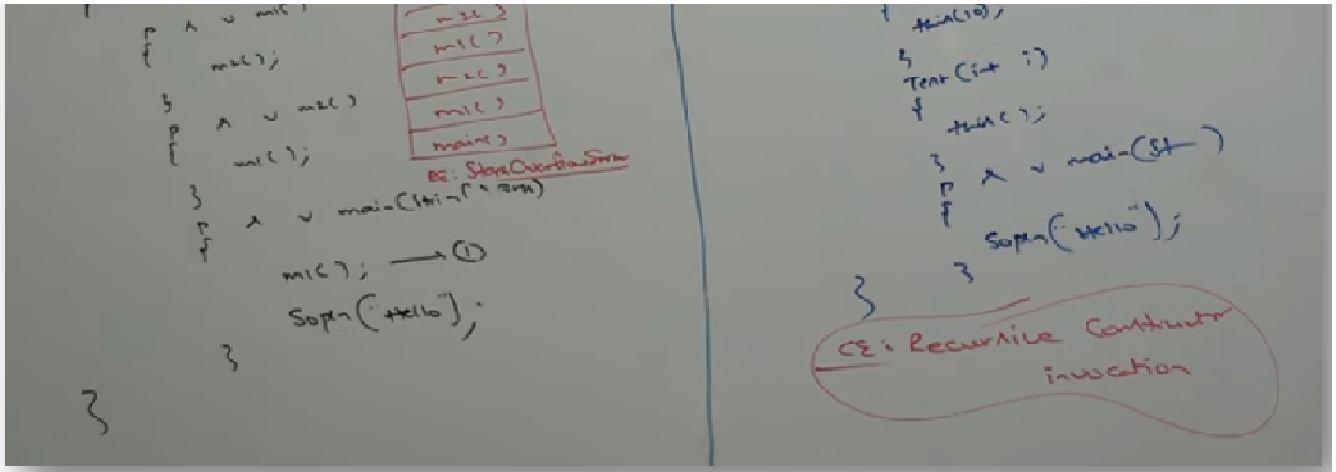
For Constructors Inheritance and overriding concepts are not applicable but overloading concept is applicable.

Every class in Java including abstract class can contain constructors but interface can not contain constructor.

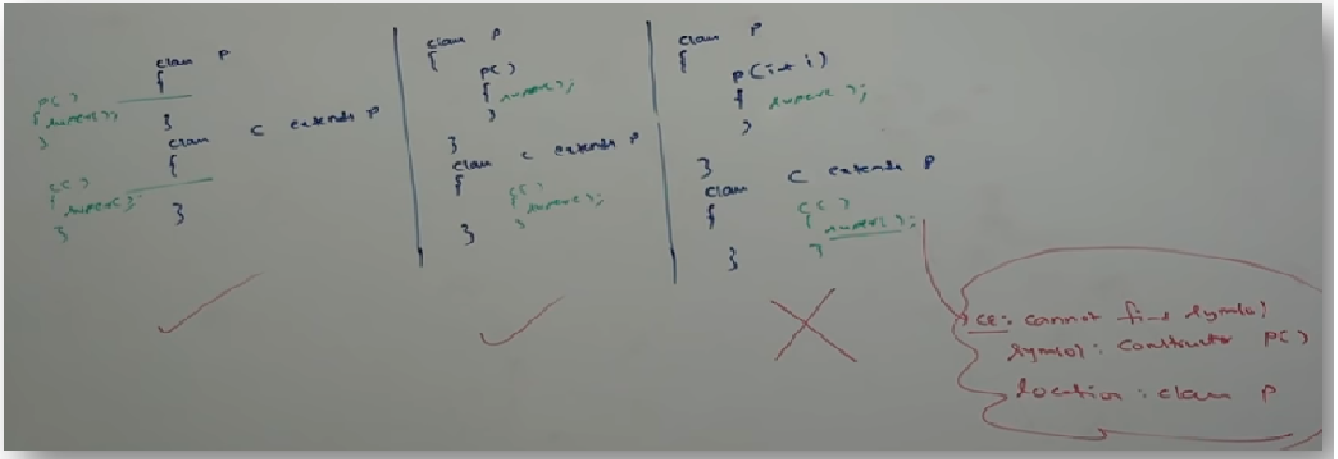


CASE 1 :=> Recursive method call is a Runtime Exception saying StackOverFlowError.

But In our program if there is a chance of recursive constructor invocation then the code won't compile and we will get compile time error.


m1();
m1();
m1();
m1();
main()
SOE: StackOverflowError
m1();
Sopln("Hello");
Test(int i)
this(10);
Sopln("Hello");
CE: Recursive Constructor invocation

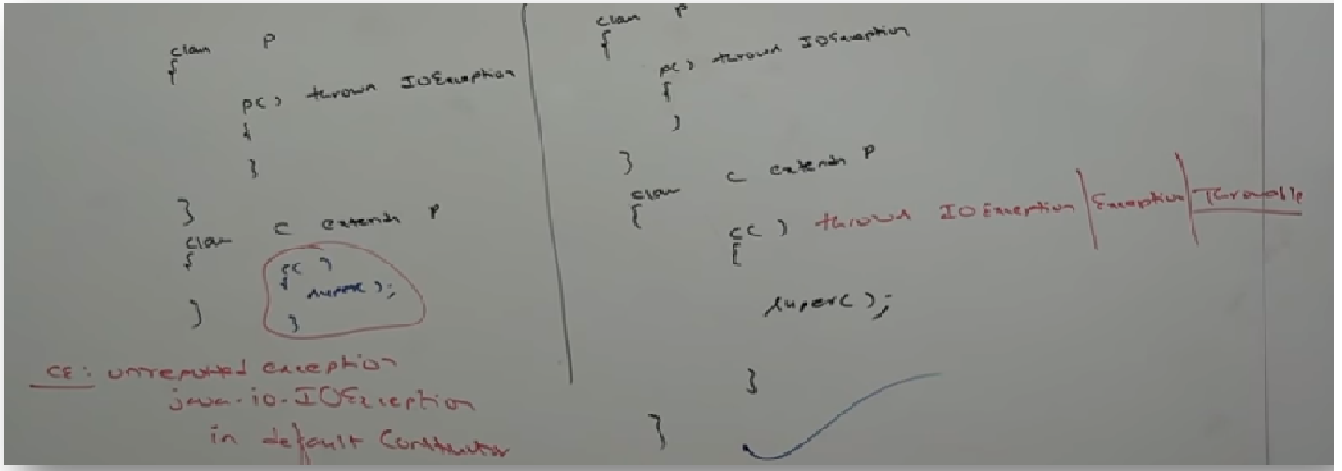
CASE 2 :=>


class P
class C extends P
CE: cannot find symbol
symbol: Constructor PC()
location: class P

If Parent class contains any argument constructor then while writing child classes we have to take special care with respect to constructors.

**** Whenever we are writing any argument constructor, it is highly recommended to write no-arg constructor also.

CASE 3:=>



If parent class constructor throws any checked exception compulsary child class constructor should throw the same checked exception or it's parent, otherwise the code won't compile.

Which of the following statement is valid :

1.) The main purpose of constructor is to create an object INVALID

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

3.) The name of the constructor need not be same as class name. INVALID

4.) Return type concept applicable for constructors but only void. INVALID

5.) we can apply any modifier for constructor. INVALID

6.) Default constructor generated by JVM. INVALID

7.) Compiler is responsible to generate default constructor VALID
8.) Compiler will always generate default constructor. INVALID
9.) If we are not writing no-arg constructor then compiler will generate default constructor. INVALID
10.) Every no-argument constructor is default constructor. INVALID
11.) Default constructor is always no-arg constructor. VALID
12.)The first line inside every constructor should be either super() or this(). If we are not wrting anything then compiler will generate this(). INVALID
13.) For constructors both overloading and overriding concepts are applicable. INVALID
14.) For constructors inheritance concept applicable but not overriding . INVALID
15.) Only concrete classes can contain constructors but abstract classes cannot. INVALID
16.) Interface can contain constructors. INVALID
17.) Recursive constructor invocation is a runtime exception. INVALID
18.) If parent class construtor throws some checked exception then compulsary child class contructor should throw the same checked exception or it's child. INVALID

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, BusinessDelegate, ServiceLocator etc.

ADVANTAGE OF SINGLETON CLASS :-

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

We have to create only one object and we can resuse same object for every similar requirement so that performance and memory utilizations will be improved.

This is the central idea of singleton classes.

ex:

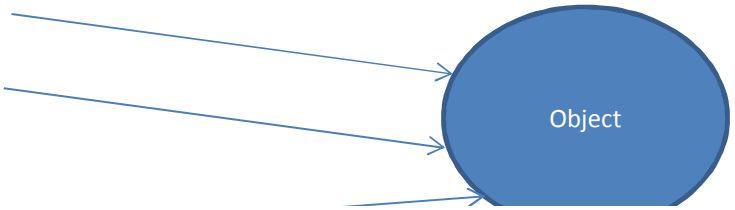
Runtime r1 = Runtime.getRuntime();

Runtime r2 = Runtime.getRuntime();

. . .

. . .

. . .

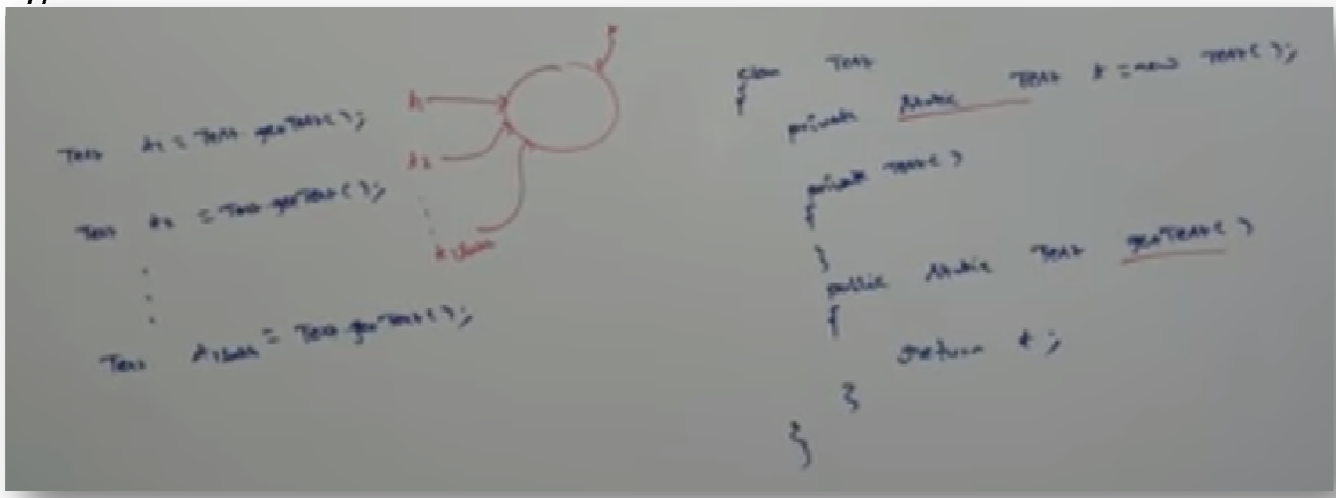


Runtime rlakh = Runtime.getRuntime();

How to create our own Singleton classes :-

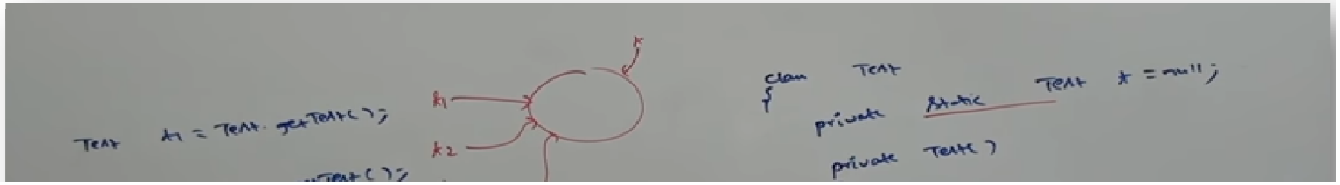
We can create our own singleton classes for this we have to use private constructor and private static variable and public factory method.

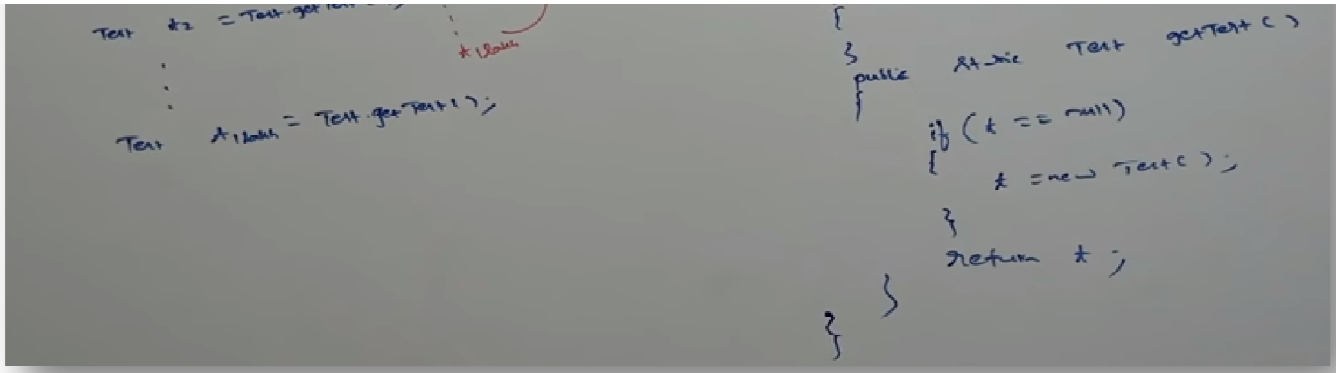
Approach 1 :=>



NOTE: Runtime class is internally implemented by using this approach.

Approach 2 :=>





At any point of time for Test class we can create only one object hence, Test class is singleton class.

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

By declaring every constructor as private, we can restrict child class creation.

EX:

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

For the above class it is impossible to create child class.

Abstract class

Abstraction is a process of hiding the implementation details and showing only functionality to the user.
There are two ways to achieve abstraction in java

Abstract class (0 to 100%)
Interface (100%)

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have final methods which will force the subclass not to change the body of the method.
- It can have Cnstructors and static method also.

Abstract Method :- A method which is declared as abstract and does not have implementation is known as an abstract method.

Ex:

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Interface in Java ?

An interface in Java is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also represents the IS-A relationship.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have default and static methods in an interface.

Since Java 9, we can have private methods in an interface.

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Ex:

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Marker OR Tagged Interface in Java ?

An interface which has no member is known as a marker or tagged interface, for example, Serializable, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{
}
```

Abstract class Vs. Interface ?

Abstract class and Interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

new Vs. Constructor

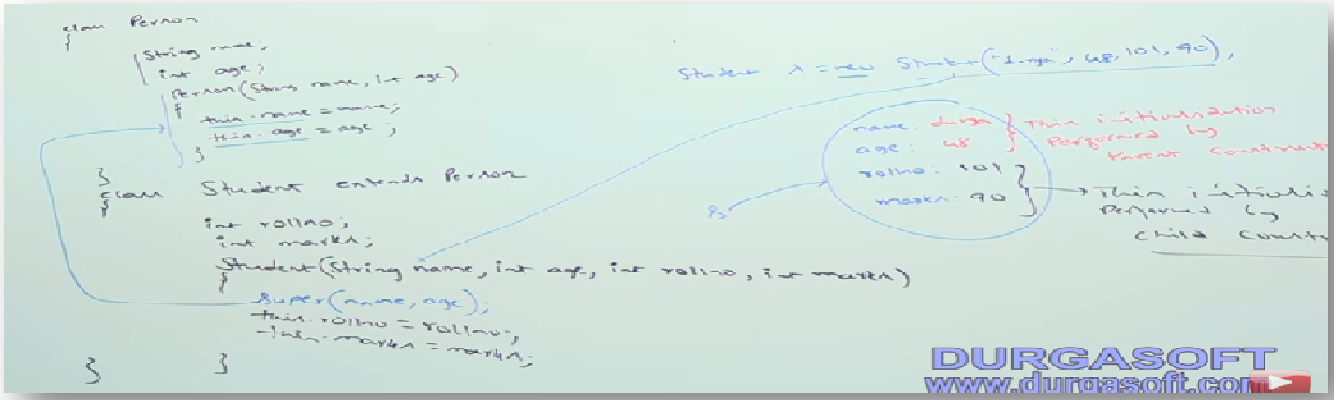
The main objective of new operator is to create an Object.

The main purpose of constructor is to initialize Object.

First Object will be created by using new Operator and then initialization will be performed by constructor.

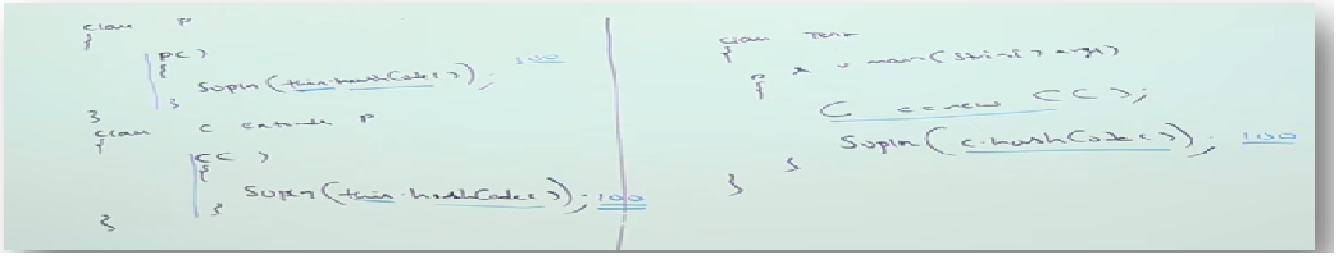
Child Object Vs Parent Constructor

Whenever we are creating child class obejct automatically Parent Constructor will be executed to perform initialization **for the instance variables which are inheriting from parent.**



In the above program both parent and child constructor executed for child object initialization only.

Whenever we are creating child classs object, Parent constructor will be executed but parent object won't be created.



	In the above example we just created only child class object but both parent and child constructors executed for that child class object
Need of Abstract class constructor	Anyway we can't create object for abstract class either directly or indirectly, but abstract class can contain construtor, what is the need ? The main objective for abstract class constructor is to perform initialization for the instance variables which are inheriting from abstract class to the child class. Whenever we are creating child class object automatically abstract class constructor will be executed to perform initialization for the instance variables which are inheriting from abstract class (Code Reusability).
Abstract class vs Interface with respect to constructor	Anyway we can't create object for abstract class and interface but abstract class can contain Constructor but interface doesn't , WHY ? ANS: The main purpose of constructor is to perform intialization of an object i.e to perform initialization for instance variables. Abstract class can contain instance variables which are required for child class object to perform intialization for these instance variables constructor concept is required for abstract classes. Every varibale present inside interface is always public static final whether we are declaring or not, hence there is no chance of existing instance variables inside interface because of this constructor concept not required for interfaces.
Abstract class Vs Interface	If everything is abstract, it is highly recommended to go for interface but not abstract class. We can replace interface with abstract class but it is not a good programming practice (This is something like recruting IAS officer for Sweeping purpose) While implementing interface we can extend any other class and hence we won't miss inheritance benefit. While extending abstract class we can't extend any other class and hence we are missing inheritance benefit.

JAVA Inner class.

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Advantage of java inner classes :-

1) It can access all the members (data members and methods) of outer class including private.
2) Nested classes are used to develop more readable and maintainable code.
3) It requires less code to write.

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

* Non-static nested class (inner class)
  (i) Member inner class
  (ii) Anonymous inner class
  (iii) Local inner class
* Static nested class

Member Inner Class :- A class created within class and outside method.

Anonymous Inner Class :- A class created for implementing interface or extending class. Its name is decided by the java compiler.

Local Inner Class :- A class created within method.

Static Nested Class :- A static class created within class.

Nested Interface :- An interface created within class or interface.

MEMBER INNER CLASS

A non-static class that is created inside a class but outside a method is called member inner class.
Example:

```
class TestMemberOuter{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter obj=new TestMemberOuter();
  TestMemberOuter.Inner in=obj.new Inner();
  in.msg();
 }
}
```

ANONYMOUS INNER CLASS

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1) Class (may be abstract or concrete).
2) Interface

Example Using Class:

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

Example Using Interface :

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Local Inner Class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Example:

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Rules For Local Inner Class :

1) Local variable can't be private, public or protected.

2) Local inner class cannot be invoked from outside the method.

3) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.

Static Nested Class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

It can access static data members of outer class including private.

Static nested class cannot access non-static (instance) data member or method.

static nested class example with instance method :

```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

static nested class example with static method :

```
class TestOuter2{
  static int data=30;
  static class Inner{
   static void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter2.Inner.msg();//no need to create the instance of static nested class
  }
}
```

If you have the static member inside static nested class, you don't need to create instance of static nested class.

Nested Interface

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class

Nested interfaces are declared static implicitely.

Example Nested Interface Declared inside of an Interface :

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly.

Example of nested interface which is declared within the class :

```
class A{
  interface Message{
    void msg();
  }
}

class TestNestedInterface2 implements A.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  A.Message message=new TestNestedInterface2();//upcasting here
  message.msg();
 }
}
```

Can we define a class inside the interface?

Yes, If we define a class inside the interface, java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
interface M{
  class A{}
}
```