CORE JAVA

With

SCJP / OCJP

Study Material

Chapter 5: OOPS

DURGA M.Tech

(Sun certified & Realtime Expert)

Ex. IBM Employee

**Trained Lakhs of Students
for last 14 years across INDIA**

India's No.1 Software Training Institute

DURGASOFT

www.durgasoft.com Ph: 9246212143 ,8096969696

Object Oriented Programming (OOPS)

Agenda:

1. **Data Hiding**
2. **Abstraction**
3. **Encapsulation**
4. **Tightly Encapsulated Class**
5. **IS-A Relationship(Inheritance)**
   - **Multiple inheritance**
   - **Cyclic inheritance**
6. **HAS-A Relationship**
   - **Composition**
   - **Aggregation**
7. **Method Signature**
8. **Polymorphism**
   - **Overloading**
     - **Automatic promotion in overloading**
   - **Overriding**
     - **Rules for overriding**
     - **Checked Vs Un-checked Exceptions**
     - **Overriding with respect to static methods**
     - **Overriding with respect to Var-arg methods**
     - **Overriding with respect to variables**
     - **Differences between overloading and overriding ?**
   - **Method Hiding**
9. **Static Control Flow**
   - **Static control flow parent to child relationship**
   - **Static block**
10. **Instance Control Flow**
    - **Instance control flow in Parent to Child relationship**
11. **Constructors**
    - **Constructor Vs instance block**
    - **Rules to write constructors**
    - **Default constructor**
    - **Prototype of default constructor**
    - **super() vs this():**
    - **Overloaded constructors**
    - **Recursive functions**
12. **Coupling**
13. **Cohesion**
14. **Object Type Casting**
    - **Compile time checking**
    - **Runtime checking**

- **Difference between ArrayList l=new ArrayList() & List l=new ArrayList() ?**
- **In how many ways get an object in java ?**

- ▪ **Singleton classes**
- ▪ **Factory method**

Data Hiding :

- Our internal data should not go out directly that is outside person can't access our internal data directly.
- By using private modifier we can implement data hiding.

```
Example:
class Account  {
       private double balance;
       .....................;
       .....................;
}
```

After providing proper username and password only , we can access our Account information.

The main advantage of data hiding is security.
<u>Note:</u> recommended modifier for data members is private.

Abstraction :

- Hide internal implementation and just highlight the set of services, is called abstraction.
- By using abstract classes and interfaces we can implement abstraction.

<u>Example :</u>

By using ATM GUI screen bank people are highlighting the set of services what they are offering without highlighting internal implementation.

<u>The main advantages of Abstraction are:</u>

1. We can achieve security as we are not highlighting our internal implementation.(i.e., outside person doesn't aware our internal implementation.)
2. Enhancement will become very easy because without effecting end user we can able to perform any type of changes in our internal system.
3. It provides more flexibility to the end user to use system very easily.
4. It improves maintainability of the application.
5. It improves modularity of the application.
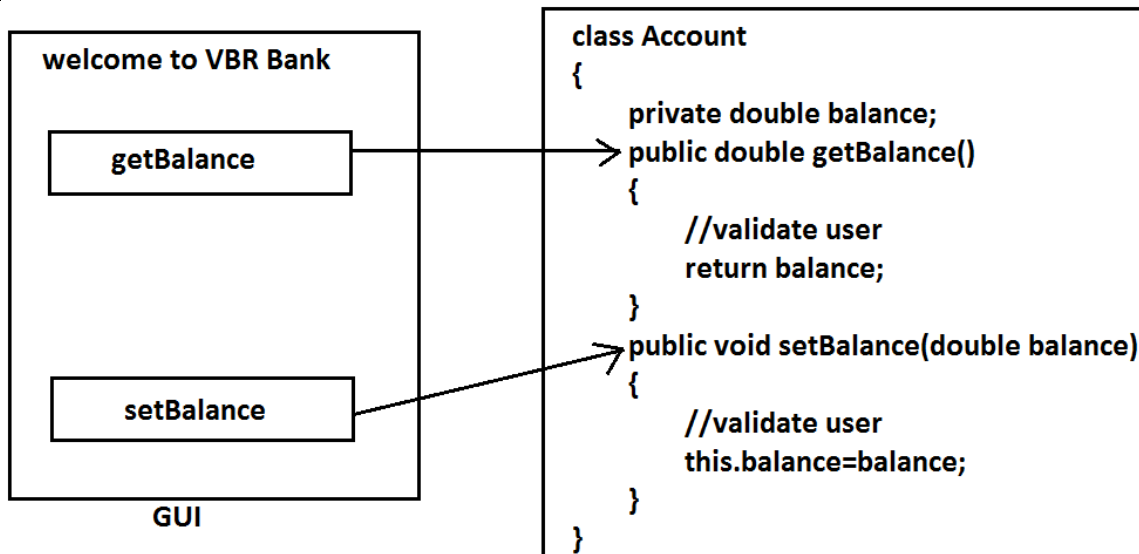6. It improves easyness to use our system.

By using interfaces (GUI screens) we can implement abstraction

Encapsulation :

- Binding of data and corresponding methods into a single unit is called Encapsulation .
- If any java class follows data hiding and abstraction such type of class is said to be encapsulated class.

Encapsulation=Datahiding+Abstraction
Example:

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

welcome to VBR Bank

getBalance

setBalance

GUI

Every data member should be declared as private and for every member we have to maintain getter & Setter methods.

The main advantages of encapsulation are :

1. We can achieve security.
2. Enhancement will become very easy.
3. It improves maintainability and modularity of the application.
4. It provides flexibility to the user to use system very easily.

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 every variable of that class declared as private whether the variable has getter and setter methods are not , and whether these methods declared as public or not, these checkings are not required to perform.

```
Example:
class Account   {
        private double balance;
        public double getBalance()  {
                return balance;
        }
}
```
Which of the following classes are tightly encapsulated?

class A

{

    private int x=10; (valid)

}

class B extends A

{

    int y=20;(invalid)

}

class C extends A

{

    private int z=30; (valid)

}

Which of the following classes are tightly encapsulated?
```
class A {
        int x=10;     //not
}
class B extends A {
        private int y=20;  //not
}
class C extends B {
        private int z=30;  //not
}
```
Note: if the parent class is not tightly encapsulated then no child class is tightly encapsulated.

IS-A Relationship(inheritance) :

1. Also known as inheritance.
2. By using "extends" keywords we can implement IS-A relationship.
3. The main advantage of IS-A relationship is reusability.

```
Example:
class Parent {
      public void methodOne(){ }
}
class Child extends Parent {
      public void methodTwo() { }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        Parent p=new Parent();
        p.methodOne();
        p.methodTwo();
        Child c=new Child();
        c.methodOne();
        c.methodTwo();
        Parent p1=new Child();
        p1.methodOne();
        p1.methodTwo();
        Child c1=new Parent();
    }
}
```

C.E: cannot find symbol
symbol : method methodTwo()
location: class Parent

C.E: incompatible types
found   : Parent
required: Child

Conclusion :

1. Whatever the parent has by default available to the child but whatever the child has by default not available to the parent. Hence on the child reference we can call both parent and child class methods. But on the parent reference we can call only methods available in the parent class and we can't call child specific methods.
2. Parent class reference can be used to hold child class object but by using that reference we can call only methods available in parent class and child specific methods we can't call.
3. Child class reference cannot be used to hold parent class object.
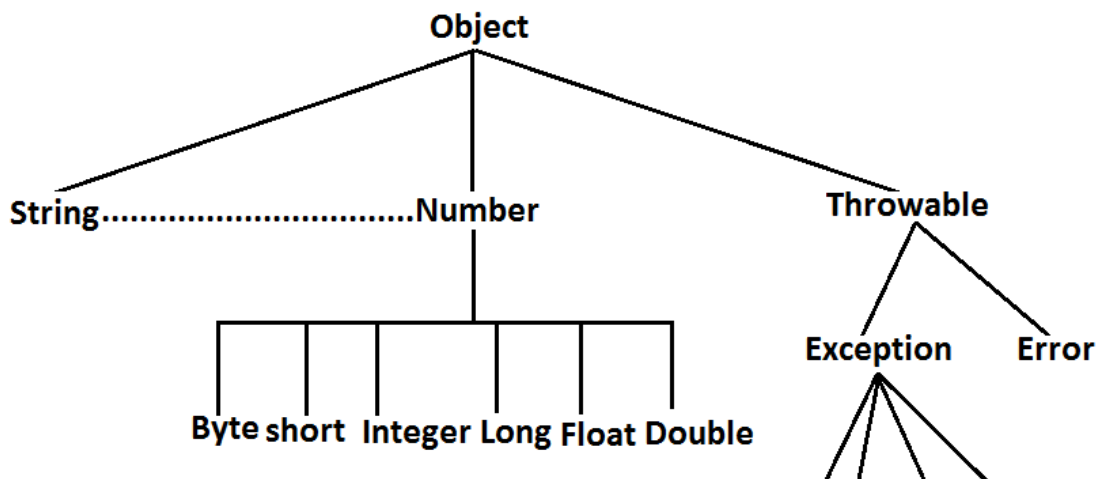
Example:

The common methods which are required for housing loan, vehicle loan, personal loan and education loan we can define into a separate class in parent class loan. So that automatically these methods are available to every child loan class.

```
Example:
class Loan {
        //common methods which are required for any type of loan.
}
class HousingLoan extends Loan {
        //Housing loan specific methods.
}
class EducationLoan extends Loan {
        //Education Loan specific methods.
}
```

- For all java classes the most commonly required functionality is define inside object class hence object class acts as a root for all java classes.
- For all java exceptions and errors the most common required functionality defines inside Throwable class hence Throwable class acts as a root for exception hierarchy.

Diagram:

Multiple inheritance :

Having more than one Parent class at the same level is called multiple inheritance.

Example:

```
Parent1              Parent2

        \           /
         \         /
          \       /
           Child
```
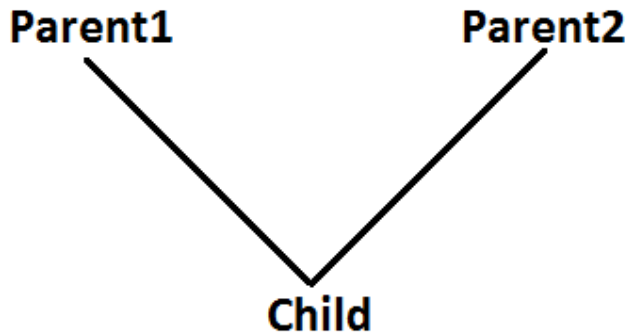
Any class can extends only one class at a time and can't extends more than one class simultaneously hence java won't provide support for multiple inheritance.

Example:

```
class A{}
class B{}                (invalid)
class C extends A,B
{}
```

But an interface can extends any no. Of interfaces at a time hence java provides support for multiple inheritance through interfaces.
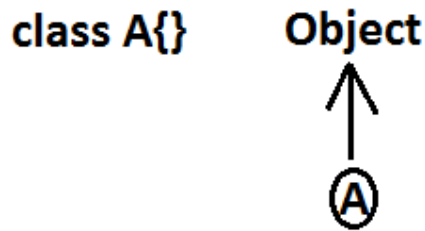
Example:

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

If our class doesn't extends any other class then only our class is the direct child class of object.
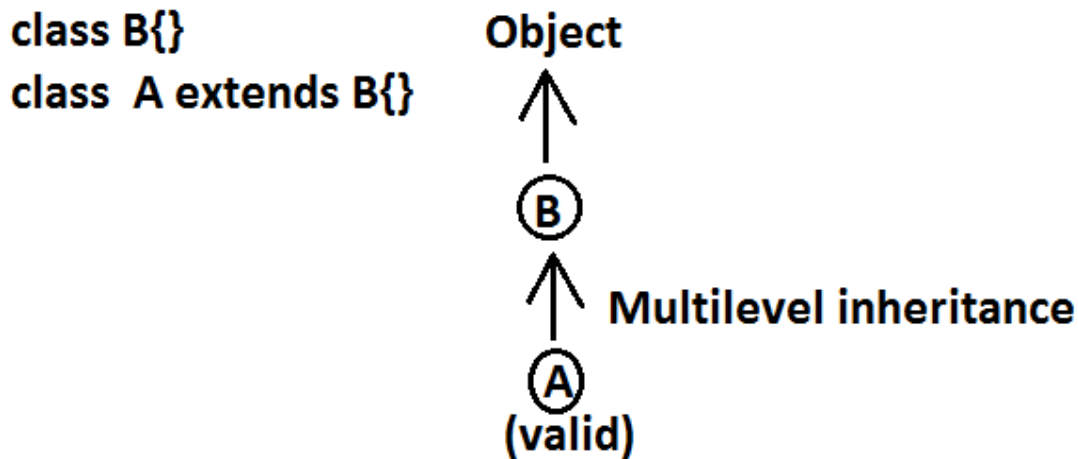
Example:

class A{}   Object
                      ↑
                      |
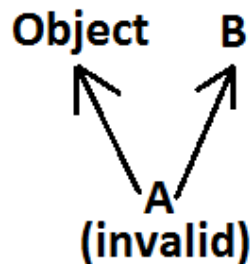   Ⓐ

If our class extends any other class then our class is not direct child class of object, It is indirect child class of object , which forms multilevel inheritance.

Example 1:

class B{}   Object
class A extends B{}   ↑
                          |
                         Ⓑ
                          ↑
                          | **Multilevel inheritance**
                         Ⓐ
                        (valid)

Example 2:

Object   B
  ↖   ↗
    ＼   ／
      Ａ
   (invalid)

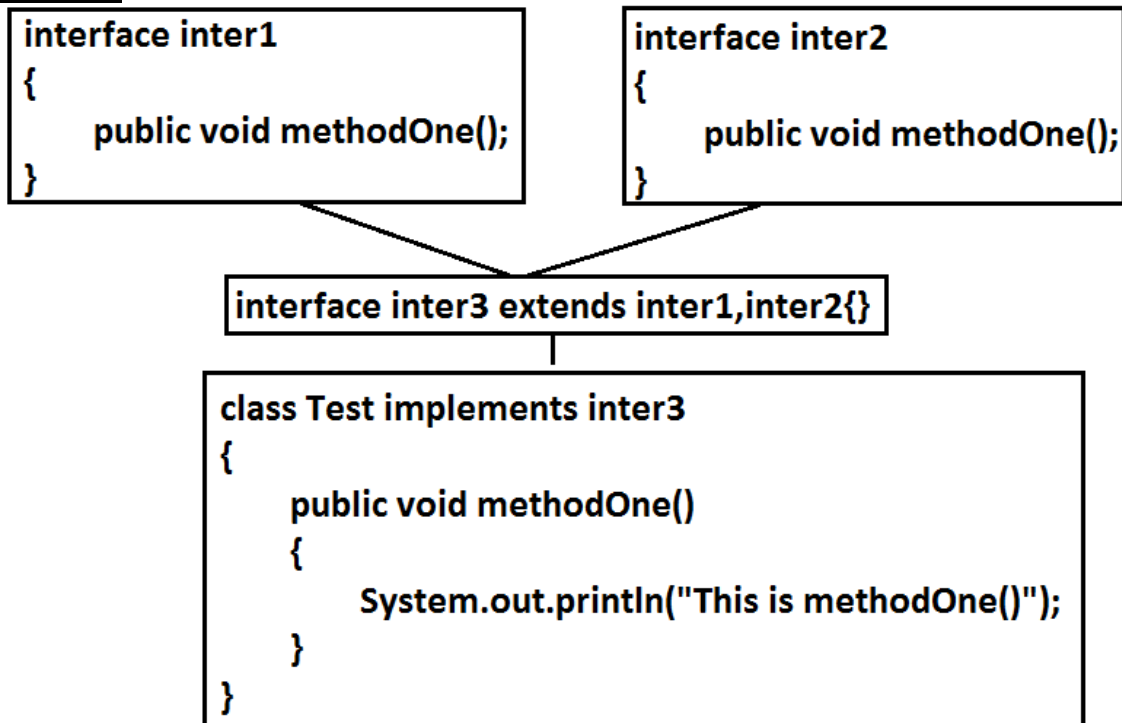**Why java won't provide support for multiple inheritance?**

There may be a chance of raising ambiguity problems.

Thed

Example:

Parent1 ⟶ methodOne()   Parent2 ⟶ methodOne()

c.methodOne(); ⟶ Ambiguity problem.
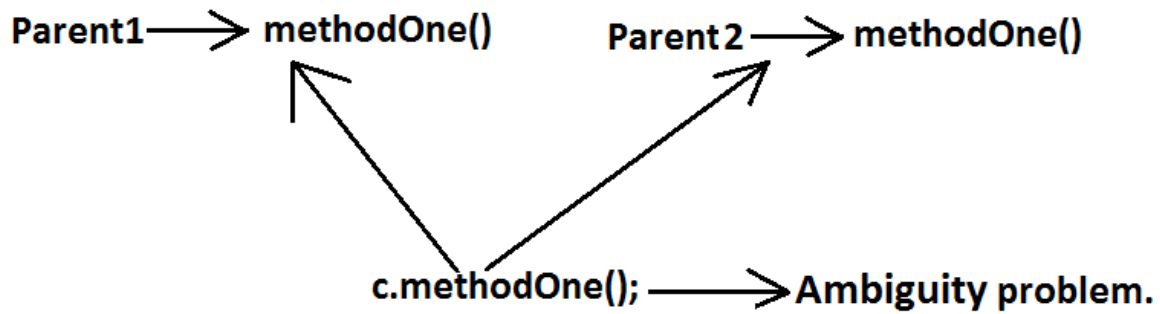
Why ambiguity problem won't be there in interfaces?

Interfaces having dummy declarations and they won't have implementations hence no ambiguity problem.

Example:

```
interface inter1
{
    public void methodOne();
}
```

```
interface inter2
{
    public void methodOne();
}
```

```
interface inter3 extends inter1,inter2{}
```

```
class Test implements inter3
{
    public void methodOne()
    {
        System.out.println("This is methodOne()");
    }
}
```
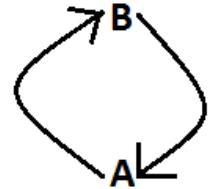
Cyclic inheritance :

Cyclic inheritance is not allowed in java.

Example 1:

```
class A extends B{}  (invalid)
class B extends A{}  C.E:cyclic inheritance involving A
```


Example 2:

```
class A extends A{}  ──C.E──>  cyclic inheritance involving A
```

HAS-A relationship:

1. **HAS-A relationship is also known as composition (or) aggregation.**
2. **There is no specific keyword to implement HAS-A relationship but mostly we can use new operator.**
3. **The main advantage of HAS-A relationship is reusability.**

```
Example:
class Engine
{
        //engine specific functionality
}
class Car
{
        Engine e=new Engine();
        //........................;
        //........................;
        //......................;
}
```

- **class Car HAS-A engine reference.**
- **The main dis-advantage of HAS-A relationship increases dependency between the components and creates maintains problems.**
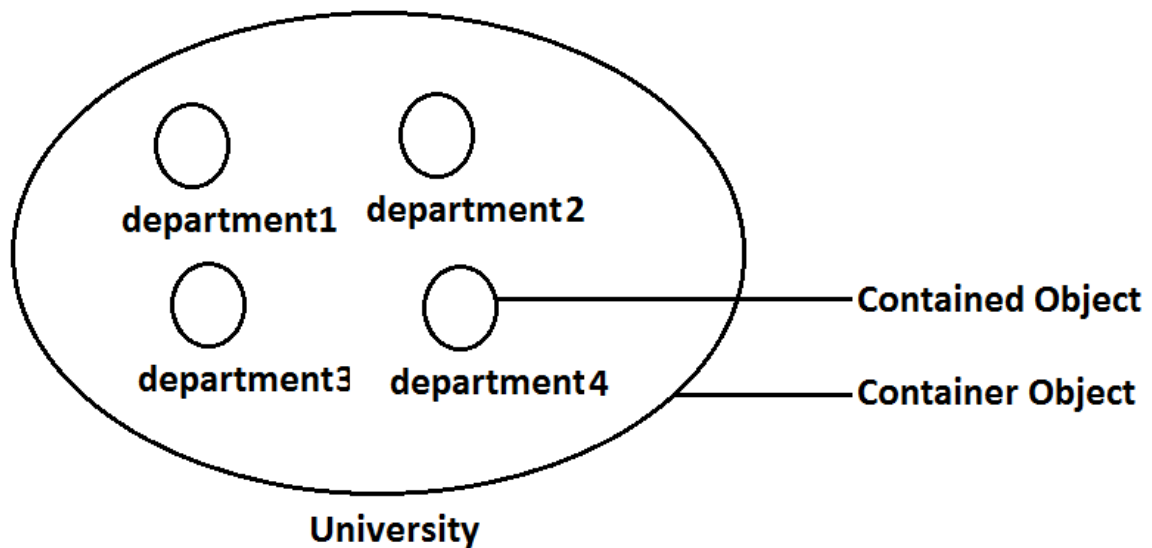
Composition vs Aggregation:

Composition:

Without existing container object if there is no chance of existing contained objects then the relationship between container object and contained object is called composition which is a strong association.

Example:

University consists of several departments whenever university object destroys automatically all the department objects will be destroyed that is without existing university object there is no chance of existing dependent object hence these are strongly associated and this relationship is called composition.

Example:

Aggregation :

Without existing container object if there is a chance of existing contained objects such type of relationship is called aggregation. In aggregation objects have weak association.

Example:

Within a department there may be a chance of several professors will work whenever we are closing department still there may be a chance of existing professor object without existing department object the relationship between department and professor is called aggregation where the objects having weak association.

Example:

Department
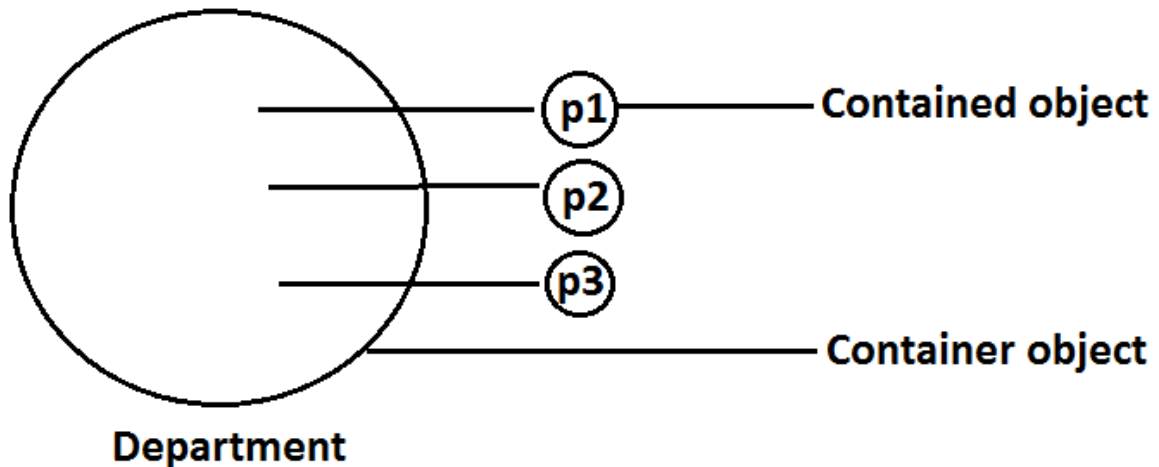
Note :

In composition container , contained objects are strongly associated, and but container object holds contained objects directly
But in Aggregation container and contained objects are weakly associated and container object just now holds the reference of contained objects.

Method signature:

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

Example:
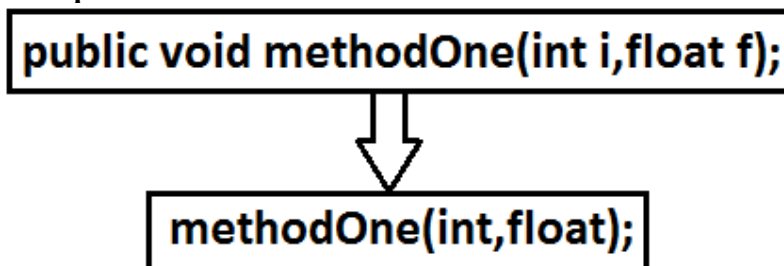
- In java return type is not part of the method signature.
- Compiler will use method signature while resolving method calls.

```
class Test {
public void m1(double d) {   }
public void m2(int i) { }
public static void main(String ar[]) {
 Test t=new Test();
 t.m1(10.5);
 t.m2(10);
 t.m3(10.5);   //CE
```

```
 }
 }
 CE : cannot find symbol
 symbol : method m3(double)
 location : class Test
```

Within the same class we can't take 2 methods with the same signature otherwise we will get compile time error.

```
Example:
public void methodOne() {  }
public int methodOne()  {
        return 10;
}
Output:
Compile time error
methodOne() is already defined in Test
```

Polymorphism:

Same name with different forms is the concept of polymorphism.

Example 1: We can use same abs() method for int type, long type, float type etc.

Example:

1. **abs(int)**
2. **abs(long)**
3. **abs(float)**

Example 2:
We can use the parent reference to hold any child objects.
We can use the same List reference to hold ArrayList object, LinkedList object, Vector object, or Stack object.

Example:

1. **List l=new ArrayList();**
2. **List l=new LinkedList();**
3. **List l=new Vector();**
4. **List l=new Stack();**

Diagram:
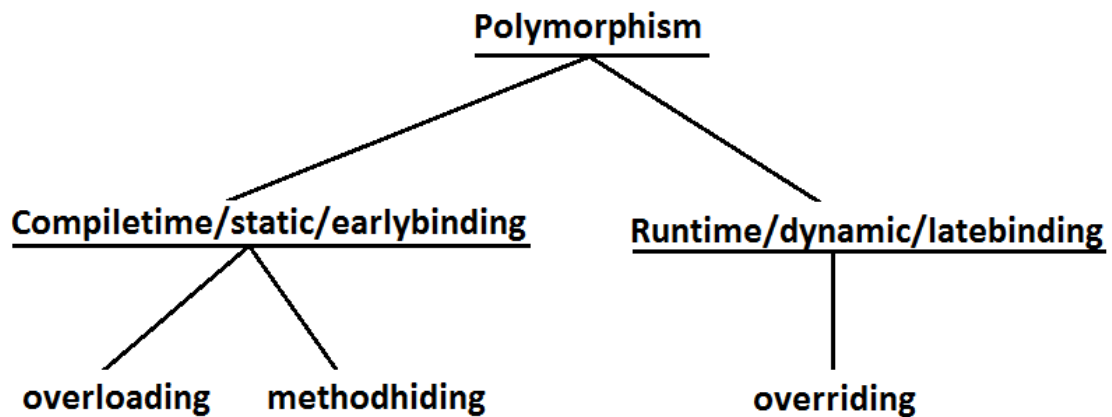
Diagram: 3 Pillars of OOPS
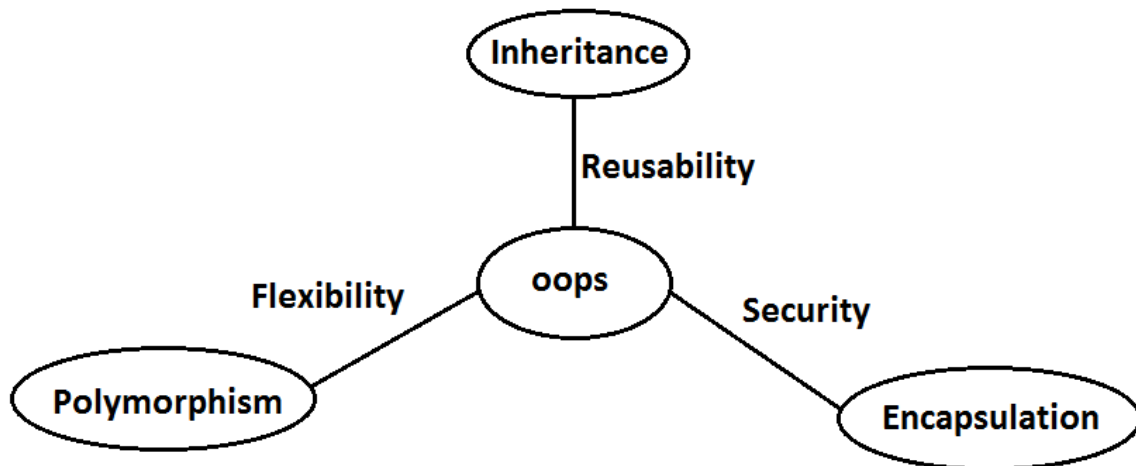
1) **Inheritance talks about reusability.**
2) **Polymorphism talks about flexibility.**
3) **Encapsulation talks about security.**

Beautiful definition of polymorphism:

A boy starts love with the word friendship, but girl ends love with the same word friendship, word is the same but with different attitudes. This concept is nothing but polymorphism.

Overloading :

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

2. In 'C' language we can't take 2 methods with the same name and different types. If there is a change in argument type compulsory we should go for new method name.
   Example :

abs() ——————for int type

labs() ———————for long type

fabs() ———————for float type

.

.

etc

3. Lack of overloading in "C" increases complexity of the programming.
4. But in java we can take multiple methods with the same name and different argument types.
   Example:

abs(int)

abs(long)

abs(float)

.

.

.

5. Having the same name and different argument types is called method overloading.
6. All these methods are considered as overloaded methods.
7. Having overloading concept in java reduces complexity of the programming.

```
8. Example:
9.  class Test {
10.         public void methodOne() {
11.                 System.out.println("no-arg method");
12.         }
13.         public void methodOne(int i) {
14.          System.out.println("int-arg method"); //overloaded methods
15.         }
16.         public void methodOne(double d) {
17.          System.out.println("double-arg method");
18.         }
19.         public static void main(String[] args)          {
20.                 Test t=new Test();
21.                 t.methodOne();//no-arg method
22.                 t.methodOne(10);//int-arg method
23.                 t.methodOne(10.5);//double-arg method
24.         }
25.  }
```

26. Conclusion : In overloading compiler is responsible to perform method resolution(decision) based on the reference type(but not based on run time

object). Hence overloading is also considered as compile time polymorphism(or) static polymorphism (or)early biding.

Case 1: Automatic promotion in overloading.

- **In overloading if compiler is unable to find the method with exact match we won't get any compile time error immediately.**
- **1st compiler promotes the argument to the next level and checks whether the matched method is available or not if it is available then that method will be considered if it is not available then compiler promotes the argument once again 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 compile time error. This process is called automatic promotion in overloading.**

The following are various possible automatic promotions in overloading.

Diagram :

byte ⟶ short
char ⟶ int ⟶ long ⟶ float ⟶ double
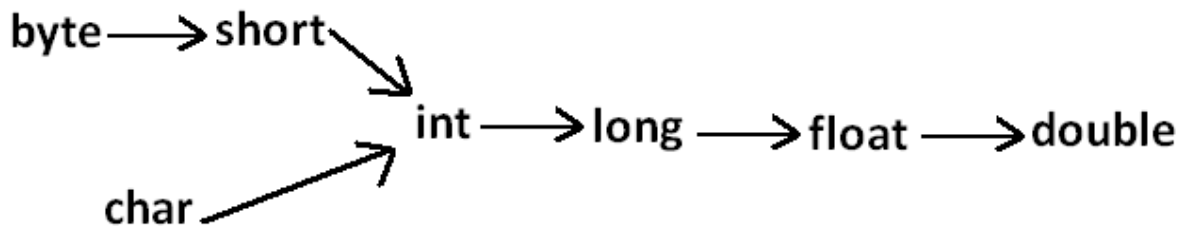
```
Example:
class Test
{
        public void methodOne(int i)
        {
                System.out.println("int-arg method");
        }
        public void methodOne(float f)         //overloaded methods
        {
                System.out.println("float-arg method");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                //t.methodOne('a');//int-arg method
                //t.methodOne(10l);//float-arg method
                t.methodOne(10.5);//C.E:cannot find symbol
        }
}
```
Case 2:
```
class Test
{
        public void methodOne(String s)
        {
```

```
                System.out.println("String version");
        }
        public void methodOne(Object o)          //Both methods are said to
                                                          //be
overloaded methods.
        {
                System.out.println("Object version");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                t.methodOne("arun");//String version
                t.methodOne(new Object());//Object version
                t.methodOne(null);//String version
        }
}
```

Note :

While resolving overloaded methods exact match will always get high priority,
While resolving overloaded methods child class will get the more priority than parent
class

Case 3:
```
class Test{
        public void methodOne(String s)        {
                System.out.println("String version");
        }
        public void methodOne(StringBuffer s) {
                System.out.println("StringBuffer version");
        }
        public static void main(String[] args){
        Test t=new Test();
        t.methodOne("arun");//String version
        t.methodOne(new StringBuffer("sai"));//StringBuffer version
        t.methodOne(null);//CE : reference to m1() is ambiquous
        }
}
```
Output:
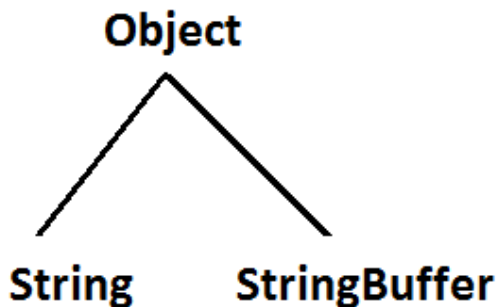
Case 4:
```
class Test {
public void methodOne(int i,float f) {
                System.out.println("int-float method");
        }
        public void methodOne(float f,int i)  {
                System.out.println("float-int method");
        }
        public static void main(String[] args){
```

```
                    Test t=new Test();
                    t.methodOne(10,10.5f);//int-float method
                    t.methodOne(10.5f,10);//float-int method
                    t.methodOne(10,10); //C.E:
                        //CE:reference to methodOne is ambiguous,
                //both method methodOne(int,float) in Test
                        //and method methodOne(float,int) in Test match
                    t.methodOne(10.5f,10.5f);//C.E:
                                    cannot find symbol
                                    symbol : methodOne(float, float)
                                    location : class Test

            }
}
```

Case 5 :
```
class Test{
        public void methodOne(int i)    {
                System.out.println("general method");
        }
        public void methodOne(int...i) {
                System.out.println("var-arg method");
        }
        public static void main(String[] args){
                Test t=new Test();
                t.methodOne();//var-arg method
                t.methodOne(10,20);//var-arg method
                t.methodOne(10);//general method
        }
}
```

In general var-arg method will get less priority that is if no other method matched then only var-arg method will get chance for execution it is almost same as default case inside switch.

Case 6:
```
class Animal{ }
class Monkey extends Animal{}
class Test{
        public void methodOne(Animal a)        {
                System.out.println("Animal version");
        }
        public void methodOne(Monkey m)        {
                System.out.println("Monkey version");
        }
        public static void main(String[] args){
                Test t=new Test();
                Animal a=new Animal();
                t.methodOne(a);//Animal version
                Monkey m=new Monkey();
                t.methodOne(m);//Monkey version
                Animal a1=new Monkey();
                t.methodOne(a1);//Animal version
        }
}
```

In overloading method resolution is always based on reference type and runtime object won't play any role in overloading.

Overriding :

1. Whatever the Parent has by default available to the Child through inheritance, if the Child is not satisfied with Parent class method implementation then Child is allow to redefine that Parent class method in Child class in its own way this process is called overriding.
2. The Parent class method which is overridden is called overridden method.
3. The Child class method which is overriding is called overriding method.

```
4. Example 1:
5.
6. class Parent {
7.   public void property(){
8.           System.out.println("cash+land+gold");
9.   }
10.        public void marry()    {
11.               System.out.println("subbalakshmi"); //overridden
    method
12.          }
13. }
14.  class Child extends Parent{                      //overriding
15.         public void marry()    {
16.               System.out.println("3sha/4me/9tara/anushka");
    //overriding method
17.          }
18. }
19.  class Test {
20.        public static void main(String[] args){
21.               Parent p=new Parent();
22.               p.marry();//subbalakshmi(parent method)
23.               Child c=new Child();
24.               c.marry();//Trisha/nayanatara/anushka(child method)
25.               Parent p1=new Child();
26.               p1.marry();//Trisha/nayanatara/anushka(child method)
27.          }
28. }
```

29. In overriding method resolution is always takes care by JVM based on runtime object hence overriding is also considered as runtime polymorphism or dynamic polymorphism or late binding.
30. The process of overriding method resolution is also known as dynamic method dispatch.

Note: In overriding runtime object will play the role and reference type is dummy.

Rules for overriding :

1. In overriding method names and arguments must be same. That is method signature must be same.
2. Until 1.4 version the return types must be same but from 1.5 version onwards co-variant return types are allowed.
3. According to this Child class method return type need not be same as Parent class method return type its Child types also allowed.
```
4. Example:
5. class Parent {
```

```
6.   public Object methodOne()        {
7.          return null;
8.   }
9. }
10.  class Child extends Parent {
11.         public String methodOne() {
12.                 return null;
13.          }
14.  }
15.
16.  C:> javac -source 1.4  Parent.java //error
```

It is valid in "1.5" but invalid in "1.4".

Diagram:
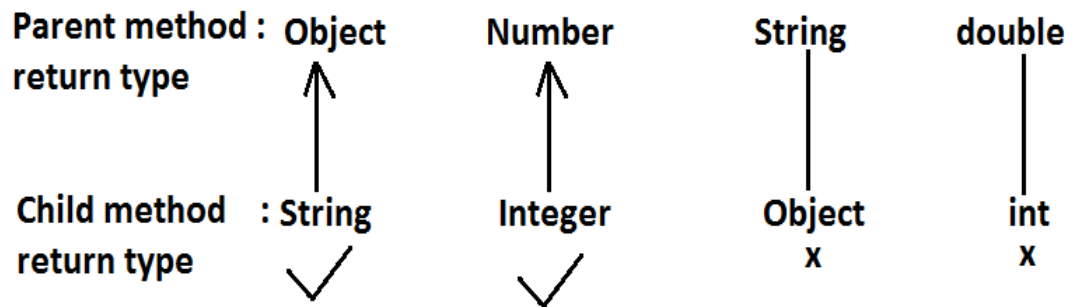
Co-variant return type concept is applicable only for object types but not for primitives.

17. **Private methods are not visible in the Child classes hence overriding concept is not applicable for private methods. Based on own requirement we can declare the same Parent class private method in child class also. It is valid but not overriding.**
    Example:

```
class Parent
{
        private void methodOne()
        {}
}                           ──── it is valid but not overriding.
class Child extends Parent
{
        private void methodOne()
        {}
}
```

Parent class final methods we can't override in the Child class.

```
18.  Example:
19.  class Parent {
20.        public final void methodOne()  {}
21.  }
22.  class Child extends Parent{
23.        public void methodOne(){}
24.  }
25.  Output:
26.  Compile time error:
27.   methodOne() in Child cannot override methodOne()
28.       in Parent; overridden method is final
```
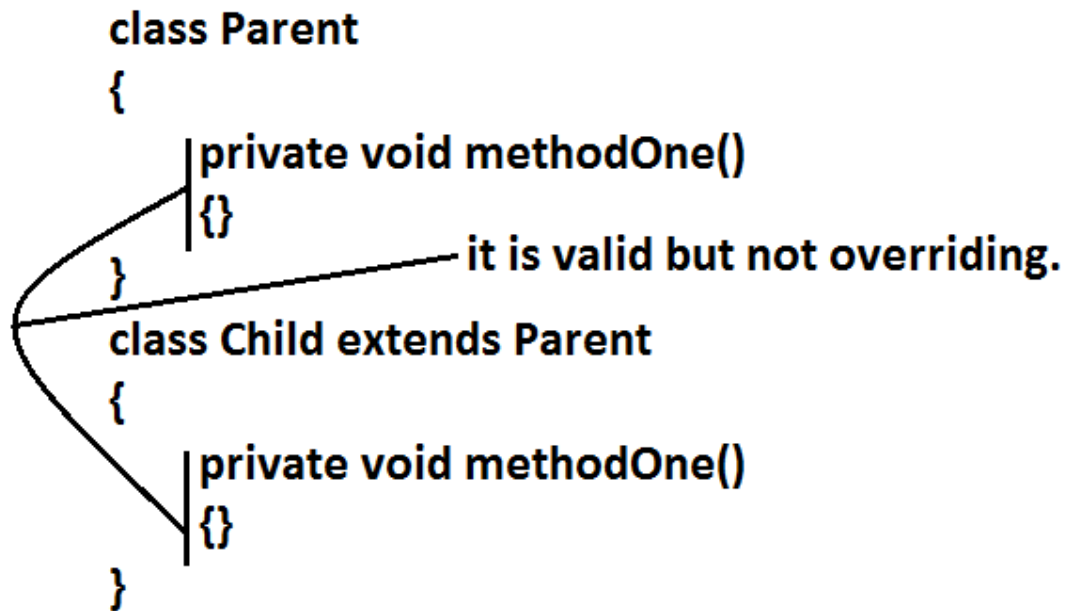
Parent class non final methods we can override as final in child class. We can override native methods in the child classes.

29. We should override Parent class abstract methods in Child classes to provide implementation.

```
30.  Example:
31.  abstract class Parent {
32.        public abstract void methodOne();
33.  }
34.  class Child extends Parent {
35.        public void methodOne() { }
36.  }
```

Diagram:
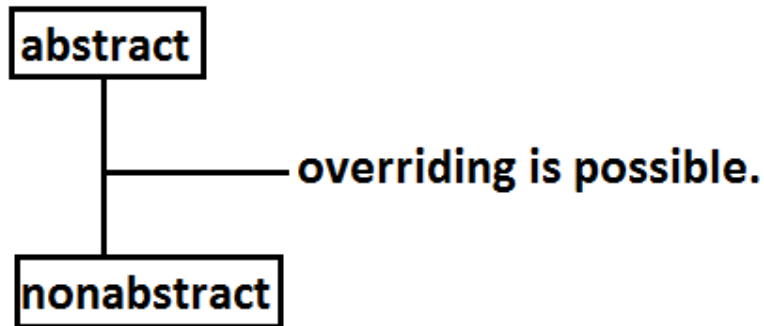
37. We can override a non-abstract method as abstract
this approach is helpful to stop availability of Parent method implementation to the next level child classes.

```
38.  Example:
39.  class Parent {
40.       public void methodOne()  { }
41.  }
42.  abstract class Child extends Parent {
43.       public abstract void methodOne();
44.  }
```

Synchronized, strictfp, modifiers won't keep any restrictions on overriding.

Diagram:
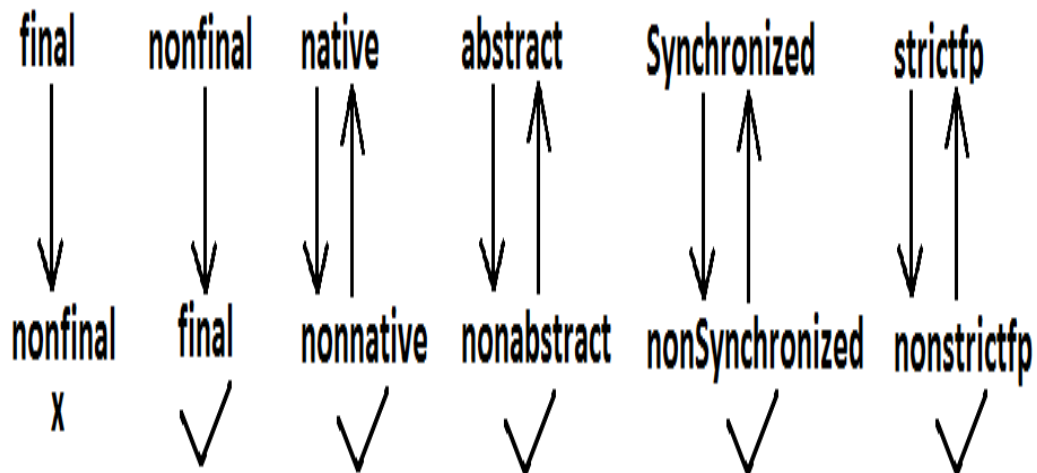
45. While overriding we can't reduce the scope of access modifier.
```
46.  Example:
47.  class Parent {
48.       public void methodOne() { }
49.  }
50.  class Child extends Parent {
51.       protected void methodOne( )  { }
52.  }
```

```
53.  Output:
54.  Compile time error :
55.  methodOne() in Child cannot override methodOne() in Parent;
56.   attempting to assign weaker access privileges; was public
```

Diagram:
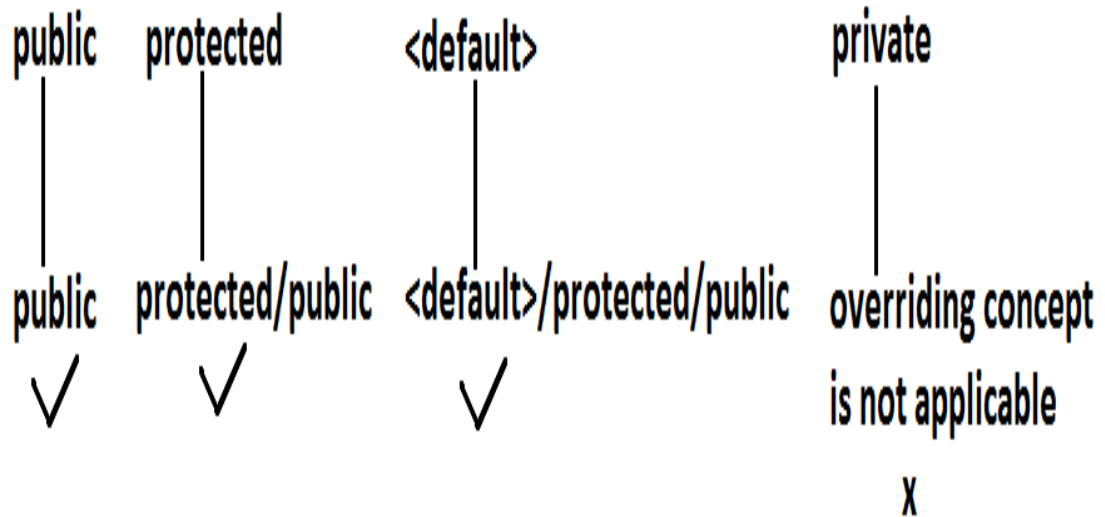
private < default < protected < public

Checked Vs Un-checked Exceptions :

- The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.
- The exceptions which are not checked by the compiler are called un-checked exceptions.
- RuntimeException and its child classes, Error and its child classes are unchecked except these the remaining are checked exceptions.

Diagram:

```
                              Object
                                ⇧
                            Throwable
```

```
Exception                                                          Error

RuntimeException        IOException  SQLException  ServletException        LinkageError
                                                                                      AssertionError
                                          EOFException              VirtualMachineError
        ArithmeticException               FileNotFoundException
        NullPointerException              InterruptedIOException              VerifyError
        IndexOutOfBoundsException
                ArrayIndexOutOfBoundsException              OutOfMemoryError  StackOverflowError
                StringIndexOutOfBoundsException
        IllegalArgumentException
                NumberFormatException
        ClassCastException
        IllegalStateException
```

Rule: While overriding if the child class method throws any checked exception compulsory the parent class method should throw the same checked exception or its parent otherwise we will get compile time error.

But there are no restrictions for un-checked exceptions.

```
Example:
class Parent {
     public void methodOne() {}
}
class Child extends Parent{
     public void methodOne()throws Exception {}
}
Output:
Compile time error :
methodOne() in Child cannot override methodOne() in Parent;
overridden method does not throw java.lang.Exception
```
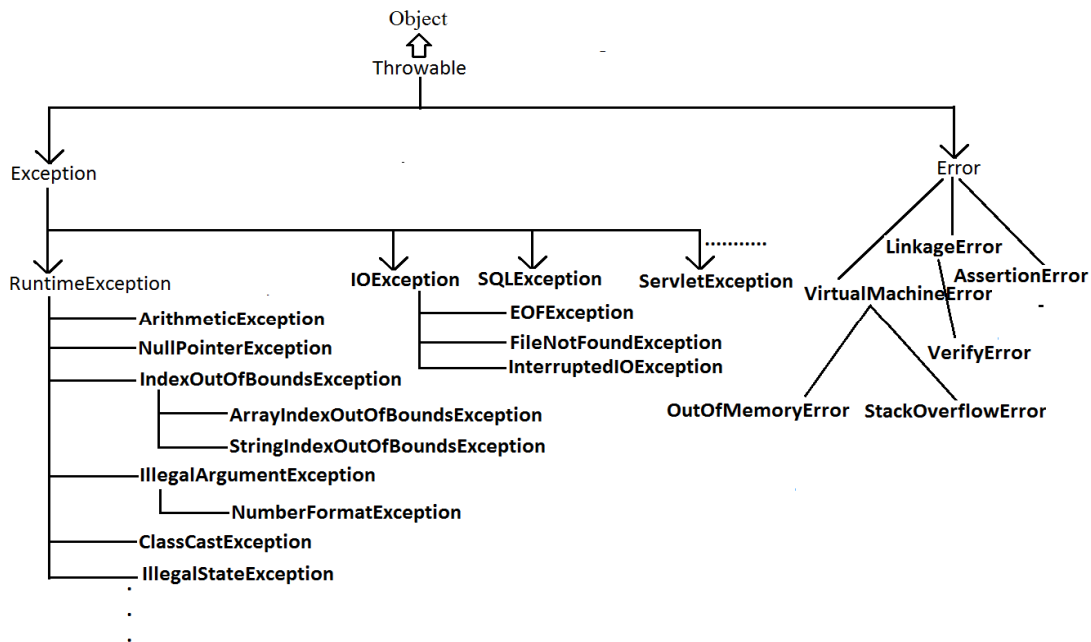
Examples :

① Parent: public void methodOne()throws Exception
   Child: public void methodOne() } valid

② Parent: public void methodOne()
   Child : public void methodOne()throws Exception } invalid

③ Parent: public void methodOne()throws Exception
   Child: public void methodOne()throws Exception } valid

④ Parent: public void methodOne()throws IOException
   Child: public void methodOne()throws Exception } invalid

⑤ Parent: public void methodOne()throws IOException
   Child: public void methodOne()throws EOFException,FileNotFoundException } valid

⑥ Parent: public void methodOne()throws IOException
   Child : public void methodOne()throws EOFException,InterruptedException } invalid

⑦ Parent: public void methodOne()throws IOException
   Child: public void methodOne()throws EOFException,ArithmeticException } valid

⑧ Parent: public void methodOne()
   Child: public void methodOne()throws
          ArithmeticException,NullPointerException,ClassCastException,RuntimeException } valid

Overriding with respect to static methods:

Case 1:

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

```
Example:
class Parent
{
public static void methodOne(){}
    //here static methodOne() method is a class level
}
class Child extends Parent
{
public void methodOne(){}
//here methodOne() method is a object level hence
    // we can't override methodOne() method
}

output :
CE: methodOne in Child  can't  override  methodOne() in Parent ;
  overriden method is  static
```
Case 2:

Similarly we can't override a non static method as static.

Case 3:
```
class Parent
{
     public static void methodOne() {}
}
class Child extends Parent {
     public static void methodOne() {}
}
```

It is valid. It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

METHOD HIDING :

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

Overriding	Method hiding
1. Both Parent and Child class methods should be non static.	1. Both Parent and Child class methods should be static.
2. Method resolution is always takes care by JVM based on runtime object.	2. Method resolution is always takes care by compiler based on reference type.
3. Overriding is also considered as runtime polymorphism (or) dynamic polymorphism (or) late binding.	3. Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early biding.

```
Example:
class Parent {
  public static void methodOne() {
    System.out.println("parent class");
  }
}
class Child extends Parent{
 public static void methodOne(){
  System.out.println("child class");
  }
}
class Test{
 public static void main(String[] args)       {
              Parent p=new Parent();
              p.methodOne();//parent class
              Child c=new Child();
              c.methodOne();//child class
              Parent p1=new Child();
              p1.methodOne();//parent class
        }
}
```

Note: If both Parent and Child class methods are non static then it will become overriding and method resolution is based on runtime object. In this case the output is

```
Parent class
Child class
Child class
```

Overriding with respect to Var-arg methods:

A var-arg method should be overridden with var-arg method only. If we are trying to override with normal method then it will become overloading but not overriding.

```
Example:
class Parent {
public void methodOne(int... i){
System.out.println("parent class");
}
}
class Child extends Parent {    //overloading but not overriding.
public void methodOne(int i) {
System.out.println("child class");
}
}
class Test {
public static void main(String[] args) {
Parent p=new Parent();
p.methodOne(10);//parent class
Child c=new Child();
c.methodOne(10);//child class
Parent p1=new Child();
p1.methodOne(10);//parent class
}
}
```

In the above program if we replace child class method with var arg then it will become overriding. In this case the output is

```
Parent class
Child class
Child class
```

Overriding with respect to variables:

- **Overriding concept is not applicable for variables.**
- **Variable resolution is always takes care by compiler based on reference type.**

```
Example:
class Parent
{
     int x=888;
}
class Child extends Parent
{
     int x=999;
}
class Test
{
        public static void main(String[] args)
```

```
        {
                Parent p=new Parent();
                System.out.println(p.x);//888
                Child c=new Child();
                System.out.println(c.x);//999
                Parent p1=new Child();
                System.out.println(p1.x);//888
        }
}
```

Note: In the above program Parent and Child class variables, whether both are static or non static whether one is static and the other one is non static there is no change in the answer.

Differences between overloading and overriding ?

Property	Overloading	Overriding
1) Method names	Must be same.	Must be same.
2) Argument type	Must be different(at least order)	Must be same including order.
3) Method signature	Must be different.	Must be same.
4) Return types	No restrictions.	Must be same until 1.4v but from 1.5v onwards we can take co-variant return types also.
5) private, static, final methods	Can be overloaded.	Can not be overridden.
6) Access modifiers	No restrictions.	Weakering/reducing is not allowed.
7) Throws clause	No restrictions.	If child class method throws any checked exception compulsory parent class method should throw the same checked exceptions or its parent but no restrictions for un-checked exceptions.
8) Method resolution	Is always takes care by compiler based on referenced type.	Is always takes care by JVM based on runtime object.
9) Also known as	Compile time polymorphism (or) static(or)early binding.	Runtime polymorphism (or) dynamic (or) late binding.

Note:

1. **In overloading we have to check only method names (must be same) and arguments (must be different) the remaining things like return type extra not required to check.**

2. **But In overriding we should compulsory check everything like method names, arguments, return types, throws keyword, modifiers etc.**

Consider the method in parent class
Parent: public void methodOne(int i)throws IOException

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

1. **public void methodOne(int i)//valid(overriding)**
2. **private void methodOne()throws Exception//valid(overloading)**
3. **public native void methodOne(int i);//valid(overriding)**
4. **public static void methodOne(double d)//valid(overloading)**
5. **public static void methodOne(int i)**
   Compile time error :

   methodOne(int) in Child cannot override methodOne(int) in Parent; overriding method is static

6. **public static abstract void methodOne(float f)**
   Compile time error :
   1. **illegal combination of modifiers: abstract and static**
   2. **Child is not abstract and does not override abstract method methodOne(float) in Child**

What is the difference between ArrayList l=new ArrayList() & List l=new ArrayList() ?

ArrayList al=new ArrayList(); [Child c=new Child();]	List l=new ArrayList(); [Parent p=new Child();]
If we know runtime object type exactly then we have to used this approach	If we don't know exact Runtime object type then we have to used this approach
By using child reference we can call both parent & child calss methods.	By using parent reference we can call only method available in parent class and child specific method we can't call.

- **We can use ArrayList reference to hold ArrayList object where as we can use List reference to hold any list implemented class object (ArrayList, LinkedList, Vector, Stack)**
- **By using ArrayList reference we can call both List and ArrayList methods but by using List reference we can call only List interface specific methods and we can't call ArrayList specific methods.**

IIQ : In how many ways we can create an object ? (or) In how many ways get an object in java ?

1. **By using new Operator :**
2. `Test t = new Test();`
3. **By using newInstance() :(Reflection Mechanism)**
4. `Test t=(Test)Class.forName("Test").newInstance();`
5. **By using Clone() :**
6. `Test t1 = new Test();`
7. `Test t2 = (Test)t1.Clone();`
8. **By using Factory methods :**
9. `Runtime r = Runtime.getRuntime();`
10. `DateFormat df = DateFormat.getInstance();`
11. **By using Deserialization :**
12. `FileInputStream fis = new FileInputStream("abc.ser");`
13. `ObjectInputStream ois = new ObjectInputStream(fis);`
14. `Test t = (Test)ois.readObject();`

Constructors :

1. Object creation is not enough compulsory we should perform initialization then only the object is in a position to provide the response properly.
2. Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of an object this piece of the code is nothing but constructor.
3. Hence the main objective of constructor is to perform initialization of an object.

```
Example:
class Student
{
        String name;
        int rollno;
        Student(String name,int rollno)  //Constructor
        {
                this.name=name;
                this.rollno=rollno;
        }
        public static void main(String[] args)
        {
                Student s1=new Student("vijayabhaskar",101);
                Student s2=new Student("bhaskar",102);
        }
}
```

Diagram:
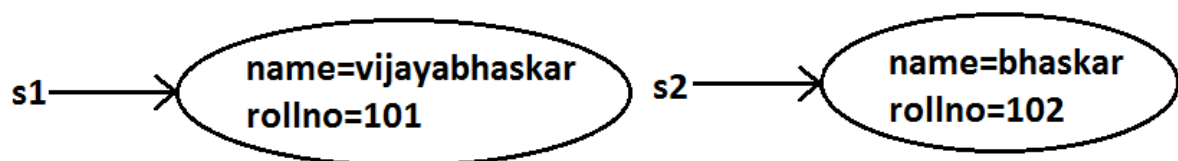
Constructor Vs instance block:

1. **Both instance block and constructor will be executed automatically for every object creation but instance block 1st followed by constructor.**
2. **The main objective of constructor is to perform initialization of an object.**
3. **Other than initialization if we want to perform any activity for every object creation we have to define that activity inside instance block.**
4. **Both concepts having different purposes hence replacing one concept with another concept is not possible.**
5. **Constructor can take arguments but instance block can't take any arguments hence we can't replace constructor concept with instance block.**
6. **Similarly we can't replace instance block purpose with constructor.**

Demo program to track no of objects created for a class:

```
class Test
{
        static int count=0;
        {
                count++;        //instance block
        }
        Test()
        {}
        Test(int i)
        {}
        public static void main(String[] args)
        {
                Test t1=new Test();
                Test t2=new Test(10);
                Test t3=new Test();
                System.out.println(count);//3
        }
}
```

Rules to write constructors:

1. **Name of the constructor and name of the class must be same.**
2. **Return type concept is not applicable for constructor even void also by mistake if we are declaring the return type for the constructor we won't get any compile time error and runtime error compiler simply treats it as a method.**
3. **Example:**
4. `class Test`
5. `{`
6. `  void Test()     //it is not a constructor and it is a method`
7. `  {}`
8. `}`
9. **It is legal (but stupid) to have a method whose name is exactly same as class name.**
10. **The only applicable modifiers for the constructors are public, default, private, protected.**
11. **If we are using any other modifier we will get compile time error.**

```
Example:
class Test
{
        static Test()
        {}
}
Output:
Modifier static not allowed here
```

Default constructor:

1. For every class in java including abstract classes also constructor concept is applicable.
2. If we are not writing at least one constructor then compiler will generate default constructor.
3. If we are writing at least one constructor then compiler won't generate any default constructor. Hence every class contains either compiler generated constructor (or) programmer written constructor but not both simultaneously.

Prototype of default constructor:

1. It is always no argument constructor.
2. The access modifier of the default constructor is same as class modifier. (This rule is applicable only for public and default).
3. Default constructor contains only one line. super(); it is a no argument call to super class constructor.

Programmers code	Compiler generated code
class Test { }	```class Test { Test() { super(); }}```
public class Test { }	```public class Test { public Test() { super(); }}```
```class Test{ void Test(){}}```	```class Test{ Test() { super();```

<table>
<tr>
<td>

</td>
<td>

```
}
void Test()
{}
}
```
</td>
</tr>
<tr>
<td>

```
class Test
{
      Test(int i)
      {}
}
```
</td>
<td>

```
class Test
{
      Test(int i)
      {
            super();
      }
}
```
</td>
</tr>
<tr>
<td>

```
class Test
{
      Test()
      {
            super();
      }
}
```
</td>
<td>

```
class Test
{
      Test()
      {
            super();
      }
}
```
</td>
</tr>
<tr>
<td>

```
class Test
{
      Test(int i)
      {
            this();
      }
      Test()
      {}
}
```
</td>
<td>

```
class Test
{
      Test(int i)
      {
            this();
      }
      Test()
      {
            super();
      }
}
```
</td>
</tr>
</table>

super() vs this():

The 1st line inside every constructor should be either super() or this() if we are not writing anything compiler will always generate super().

Case 1: We have to take super() (or) this() only in the 1st line of constructor. If we are taking anywhere else we will get compile time error.

```
Example:
class Test
{
      Test()
      {
            System.out.println("constructor");
            super();
      }
}
Output:
Compile time error.
Call to super must be first statement in constructor
```

Case 2: We can use either super() (or) this() but not both simultaneously.

```
Example:
class Test
{
      Test()
      {
            super();
```

```
                this();
        }
}
Output:
Compile time error.
Call to this must be first statement in constructor
```
Case 3: **We can use super() (or) this() only inside constructor. If we are using anywhere else we will get compile time error.**
```
Example:
class Test
{
        public void methodOne()
        {
                super();
        }
}
Output:
Compile time error.
Call to super must be first statement in constructor
```
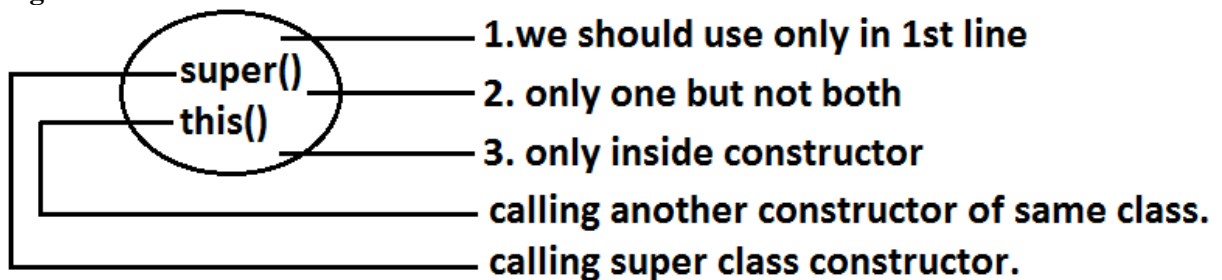
That is we can call a constructor directly from another constructor only.

Diagram:

Example:

super(), this()	super, this
These are constructors calls.	These are keywords
We can use these to invoke super class & current constructors directly	We can use refers parent class and current class instance members.
We should use only inside constructors as first line, if we are using outside of constructor we will get compile time error.	We can use anywhere (i.e., instance area) except static area , other wise we will get compile time error .

```
Example:
class Test
{
        public static void main(String[] args)
        {
                System.out.println(super.hashCode());
        }
}
Output:
Compile time error.
Non-static variable super cannot be referenced from a static context.
```

Overloaded constructors :

A class can contain more than one constructor and all these constructors having the same name but different arguments and hence these constructors are considered as overloaded constructors.

```
Example:
class Test {
 Test(double d){
    System.out.println("double-argument constructor");
       }
 Test(int i) {
       this(10.5);
       System.out.println("int-argument constructor");
       }
 Test() {
       this(10);
    System.out.println("no-argument constructor");
       }
 public static void main(String[] args) {
       Test t1=new Test();  //no-argument constructor/int-argument
                       //constructor/double-argument constructor
       Test t2=new Test(10);
              //int-argument constructor/double-argument constructor
       Test t3=new Test(10.5);//double-argument constructor
  }
}
```

- **Parent class constructor by default won't available to the Child. Hence Inheritance concept is not applicable for constructors and hence overriding concept also not applicable to the constructors. But constructors can be overloaded.**
- **We can take constructor in any java class including abstract class also but we can't take constructor inside interface.**

Example:

class Test
{
 	
    Test()
    {}
 	
}
valid
	abstract class Test
	{
	
	    Test()
	    {}
	
	}
	valid
		interface Test1
		{
		
		    Test1()
		    {}
		
		}
		invalid

We can't create object for abstract class but abstract class can contain constructor what is the need ?

Abstract class constructor will be executed for every child class object creation to perform initialization of child class object only.

Which of the following statement is true ?

1. **Whenever we are creating child class object then automatically parent class object will be created.(false)**
2. **Whenever we are creating child class object then parent class constructor will be executed.(true)**

```
Example:
abstract class Parent
{
        Parent()
        {
                System.out.println(this.hashCode());
                        //11394033//here this means child class object
        }
}
class Child extends Parent
{
        Child()
        {
                System.out.println(this.hashCode());//11394033
        }
}
class Test
{
        public static void main(String[] args)
        {
                Child c=new Child();
                System.out.println(c.hashCode());//11394033
        }
}
```

<u>**Case 1:**</u> **recursive method call is always runtime exception where as recursive constructor invocation is a compile time error.**

Note:

<u>**Recursive functions:**</u>

A function is called using two methods (types).

1. **Nested call**
2. **Recursive call**

Nested call:

- **Calling a function inside another function is called nested call.**
- **In nested call there is a calling function which calls another function(called function).**

```
Example:
public static void methodOne()
       {
               methodTwo();
       }
       public static void methodTwo()
       {
               methodOne();
       }
```

Recursive call:

- **Calling a function within same function is called recursive call.**
- **In recursive call called and calling function is same.**

```
Example:
public void methodOne()
       {
               methodOne();
       }
```
Example:

```
class Test
{
    public static void methodOne()
    {
        methodTwo();
    }
    public static void methodTwo()
    {
        methodOne();
    }
    public static void main(String[] args)
    {
        methodOne();
        System.out.println("hello");
    }
} R.E:StackOverflowError
```

```
class Test
{
    Test(int i)
    {
        this();
    }
    Test()
    {
        this(10);
    }
    public static void main(String[] args)
    {
        System.out.println("hello");
    }
}    C.E:recursive constructor invocation
```

Note: Compiler is responsible for the following checkings.

1. **Compiler will check whether the programmer wrote any constructor or not. If he didn't write at least one constructor then compiler will generate default constructor.**
2. **If the programmer wrote any constructor then compiler will check whether he wrote super() or this() in the 1st line or not. If his not writing any of these compiler will always write (generate) super().**
3. **Compiler will check is there any chance of recursive constructor invocation. If there is a possibility then compiler will raise compile time error.**

Case 2:

```
class Parent
{}
class Child extends Parent
{}        valid
```

```
class Parent
{
    Parent()
    {}
}
class Child extends Parent
{}
```

```
class Parent
{
    Parent(int i)
    {}
}
class Child extends Parent
{
    Child()
    {
        super();
    }
} output:
    compile time error
```

```
E:\scjp>javac Child.java
Child.java:10: cannot find symbol
symbol  : constructor Parent()
location: class Parent
        super();
```

- **If the Parent class contains any argument constructors while writing Child classes we should takes special care with respect to constructors.**
- **Whenever we are writing any argument constructor it is highly recommended to write no argument constructor also.**

Case 3:
```
class Parent
{
        Parent()throws java.io.IOException
        {}
}
class Child extends Parent
{}
Output:
Compile time error
Unreported exception java.io.IOException in default constructor.
```

```
Example:
class Parent
{
        Parent()throws java.io.IOException
        {}
}
class Child extends Parent
{
        Child()throws Exception
        {
                super();
        }
}
```

If Parent class constructor throws some checked exception compulsory Child class constructor should throw the same checked exception (or) its Parent.

Singleton classes :

For any java class if we are allow to create only one object such type of class is said to be singleton class.

```
Example:
1)      Runtime class
2)      ActionServlet
3)      ServiceLocator
4)      BusinessDelegate

Runtime r1=Runtime.getRuntime();
        //getRuntime() method is a factory method
Runtime r2=Runtime.getRuntime();
Runtime r3=Runtime.getRuntime();
...................................................
...................................................

System.out.println(r1==r2);//true
System.out.println(r1==r3);//true
```
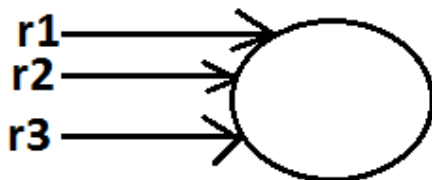
Diagram:

Advantage of Singleton class :
If the requirement is same then instead of creating a separate object for every person we will create only one object and we can share that object for every required person we can achieve this by using singleton classes. That is the main advantages of singleton classes are Performance will be improved and memory utilization will be improved.

## Creation of our own singleton classes:

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

```
Example:
class Test
{
        private static Test t=null;
        private Test()
        {}
        public static Test getTest()
                    //getTest() method is a factory method
        {
                if(t==null)
                {
                        t=new Test();
                }
                return t;
        }
}
class Client
{
        public static void main(String[] args)
        {
                System.out.println(Test.getTest().hashCode());//1671711
                System.out.println(Test.getTest().hashCode());//1671711
                System.out.println(Test.getTest().hashCode());//1671711
                System.out.println(Test.getTest().hashCode());//1671711
        }
}
```

Diagram:
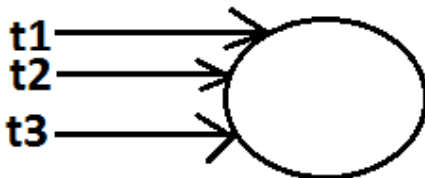
Note:

We can create any xxxton classes like(double ton,trible ton...etc)

```
Example:
class Test
{
        private static Test t1=null;
        private static Test t2=null;
        private Test()
        {}
        public static Test getTest()
                    //getTest() method is a factory method
        {
                if(t1==null)
                {
                        t1=new Test();
                        return t1;
                }
```

```
                    else if(t2==null)
                    {
                            t2=new Test();
                            return t2;
                    }
                    else
                    {
                            if(Math.random()<0.5) //Math.random() limit : 0<=x<1
                                  return t1;
                            else
                                  return t2;
                    }
            }
}
class Client
{
        public static void main(String[] args)
        {
                System.out.println(Test.getTest().hashCode());//1671711
                System.out.println(Test.getTest().hashCode());//11394033
                System.out.println(Test.getTest().hashCode());//11394033
                System.out.println(Test.getTest().hashCode());//1671711
        }
}
```

IIQ : We are not allowed to create child class but class is not final , How it is Possible ?

By declaring every constructor has private.
```
class Parent {
  private Parent() {
  }
```
We can't create child class for this class

Note : When ever we are creating child class object automatically parent class constructor will be executed but parent object won't be created.

```
class Parent {
 Parent() {
  System.out.println(this.hashCode()); //123
 }
}
class  Child extends Parent {
Child() {
 System.out.println(this.hashCode());//123
 }
}
class Test {
 public static void main(String ar[]) {
  Child c=new Child();
  System.out.println(c.hashCode());//123
```
Which of the following is true ?

1. **The name of the constructor and name of the class need not be same.(false)**
2. **We can declare return type for the constructor but it should be void. (false)**
3. **We can use any modifier for the constructor. (false)**
4. **Compiler will always generate default constructor. (false)**

5. The modifier of the default constructor is always default. (false)
6. The 1st line inside every constructor should be super always. (false)
7. The 1st line inside every constructor should be either super or this and if we are not writing anything compiler will always place this().(false)
8. Overloading concept is not applicable for constructor. (false)
9. Inheritance and overriding concepts are applicable for constructors. (false)
10. Concrete class can contain constructor but abstract class cannot. (false)
11. Interface can contain constructor. (false)
12. Recursive constructor call is always runtime exception. (false)
13. If Parent class constructor throws some un-checked exception compulsory Child class constructor should throw the same un-checked exception or it's Parent. (false)
14. Without using private constructor we can create singleton class. (false)
15. None of the above.(true)

Factory method:

By using class name if we are calling a method and that method returns the same class object such type of method is called factory method.

Example:
```
Runtime r=Runtime.getRuntime();//getRuntime is a factory method.
DateFormat df=DateFormat.getInstance();
```

If object creation required under some constraints then we can implement by using factory method.

Static control flow :

Example:

```
class Base
{
   ① static int  i=10;     ⑦
   ② static
      {
         methodOne();     ⑧
         System.out.println("first static block");     ⑩
      }
   ③ public static void main(String[] args)
      {
         methodOne();     ⑬
         System.out.println("main method");     ⑮
      }
   ④ public static void methodOne()
      {
         System.out.println(j);     ⑨,⑭
      }
   ⑤ static
      {
         System.out.println("second static block");     ⑪
      }
   ⑥ static int  j=20;     ⑫
   }
```

Analysis:

i=0[RIWO]
j=0[RIWO]
i=10[R&W]
j=20[R&W]
1.identification of static members from top to bottom[1 to 6]
2.execution of static variable assignments and static blocks from top to bottom[7 to 12]
3.execution of main method[13 to 15]

```
Output:
E:\scjp>javac Base.java
E:\scjp>java Base
0
First static block
Second static block
20
Main method
```

Read indirectly write only state (or) RIWO :

With in the static block if we are trying to read any variable then that read is considered as "direct read"
If we are calling a method , and with in the method if we are trying to read a method , that read is called Indirect read

If a variable is in RIWO state then we can't perform read operation directly otherwise we will get compile time error saying " illegal forward reference ".

Example:

```
class Test
{
    static int i=10;
    static
    {
        System.out.println(i);//10
        System.exit(0);
    }
}
```

```
class Test
{
    static
    {
        System.out.println(i);
    }
    static int i=10;
}
output:
compile time error:
illegal forward reference
```

```
class Test
{
    static
    {
        methodOne();
    }
    public static void methodOne()
    {
        System.out.println(i);
    }
    static int i=10;
}
output:
Runtime exception:
0
NoSuchMethodError: main
```

Static control flow parent to child relationship :

```
class Base
{
①  static int i =10;          ⑫
②  static
    {
        methodOne();          ⑬
        System.out.println("base static block");          ⑮
    }
③  public static void main(String[] args)
    {
        methodOne();
        System.out.println("base main");
    }
④  public static void methodOne()
        {
            System.out.println(j);          ⑭
        }
⑤  static int j=20;          ⑯
}

class Derived extends Base
{
⑥  static int x =100;          ⑰
⑦  static
    {
        methodTwo();          ⑱
        System.out.println("derived first static block");          ⑳
    }
⑧  public static void main(String[] args)
    {
        methodTwo();          ㉓
        System.out.println("derived main");          ㉕
    }
⑨  public static void methodTwo()
        {
            System.out.println(y);          ⑲,㉔
        }
⑩  static
    {
        System.out.println("derived second  static block");          ㉑
    }
⑪  static int y=200;          ㉒
}
```

Analysis:

i=0[RIWO]

j=0[RIWO]

x=0[RIWO]

y=0[RIWO]

i=10[R&w]

j=20[R&w]

x=100[R&w]

y=200[R&w]

```
Output:
E:\scjp>java Derived
0
Base static block
0
Derived first static block
Derived second static block
200
Derived main
Output:
E:\scjp>java Base
0
Base static block
20
Basic main
```

Whenever we are executing Child class the following sequence of events will be performed automatically.

1. **Identification of static members from Parent to Child. [1 to 11]**
2. **Execution of static variable assignments and static blocks from Parent to Child.[12 to 22]**
3. **Execution of Child class main() method.[23 to 25].**

Note : When ever we are loading child class autimatically the parent class will be loaded but when ever we are loading parent class the child class don't be loaded automatically.

Static block:

- **Static blocks will be executed at the time of class loading hence if we want to perform any activity at the time of class loading we have to define that activity inside static block.**
- **With in a class we can take any no. Of static blocks and all these static blocks will be executed from top to bottom.**

Example:

The native libraries should be loaded at the time of class loading hence we have to define that activity inside static block.

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

Ex 2 : Every JDBC driver class internally contains a static block to register the driver with DriverManager hence programmer is not responsible to define this explicitly.

```
Example:
class Driver
{
        static
        {
                //Register this driver with DriverManager
        }
}
```

IIQ : Without using main() method is it possible to print some statements to the console?
Ans : Yes, by using static block.

```
Example:
class Google
{
        static
        {
                System.out.println("hello i can print");
                System.exit(0);
        }
}
Output:
Hello i can print
```

IIQ : Without using main() method and static block is it possible to print some statements to the console ?
```
Example 1:
class Test
{
        static int i=methodOne();
        public static int methodOne()
        {
```

```
          System.out.println("hello i can print");
                    System.exit(0);
                    return 10;
          }
}
Output:
Hello i can print
```

Example 2:
```
class Test
{
       static Test t=new Test();
       Test()
       {
             System.out.println("hello i can print");
             System.exit(0);
       }
}
Output:
Hello i can print
```

Example 3:
```
class Test
{
       static Test t=new Test();
       {
       System.out.println("hello i can print");
       System.exit(0);
       }
}
Output:
Hello i can print
```

IIQ : Without using System.out.println() statement is it possible to print some statement to the console ?
```
Example:
class Test
{
       public static void main(String[] args)
       {
             System.err.println("hello");
       }
}
```

Note : Without using main() method we can able to print some statement to the sonsole , but this rule is applicable untill 1.6 version from 1.7 version onwards to run java program main() method is mandatory.
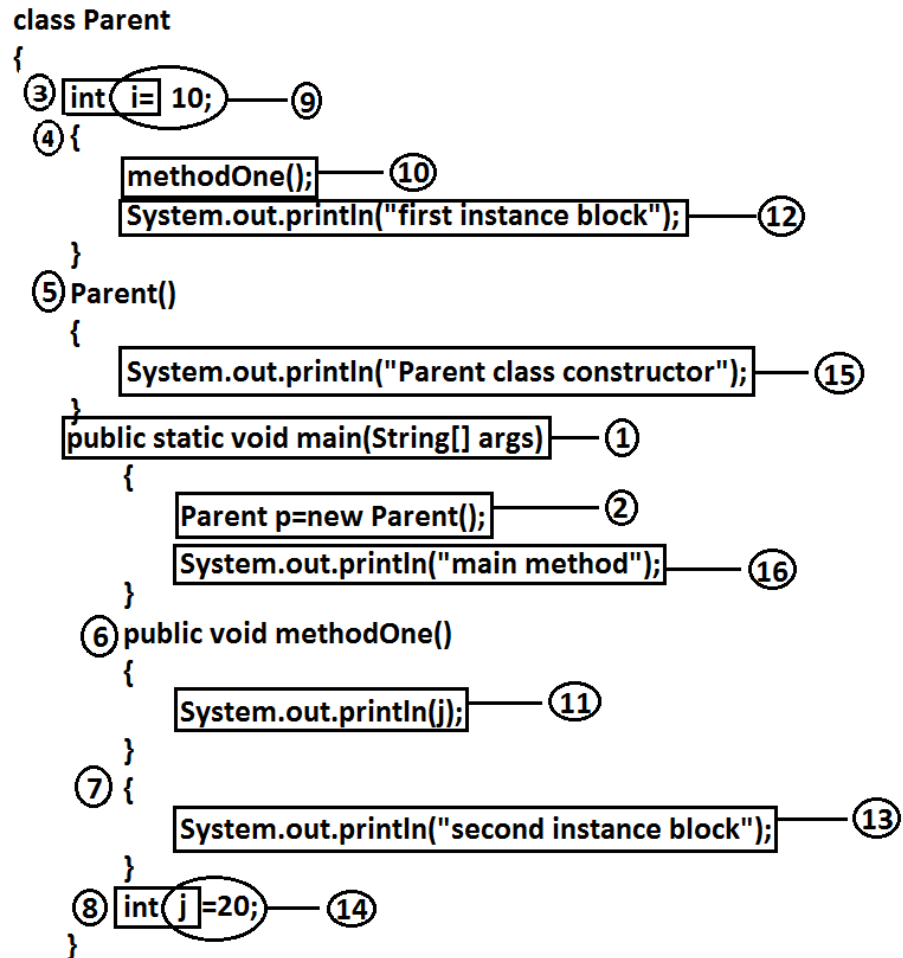
```
class Test {
 static {
  System.out.println("ststic block");
  System.exit(0);
  }
 }
```
It is valid in 1.6 version but invalid or won't run in 1.7 version

Instance control flow:

```
class Parent
{
③ int  i= 10;                                    ⑨
④ {
        methodOne();                             ⑩
        System.out.println("first instance block");   ⑫
    }
⑤ Parent()
    {
        System.out.println("Parent class constructor");   ⑮
    }
public static void main(String[] args)   ①
        {
            Parent p=new Parent();           ②
            System.out.println("main method");   ⑯
        }
⑥ public void methodOne()
        {
            System.out.println(j);           ⑪
        }
⑦ {
            System.out.println("second instance block");   ⑬
        }
⑧ int j =20;                                ⑭
    }
```

```
Analysis:
i=0[RIWO]
j=0[RIWO]
i=10[R&W]
j=20[R&W]
```

Output:

0
first instance block
second instance block
Parent class constructor

main method
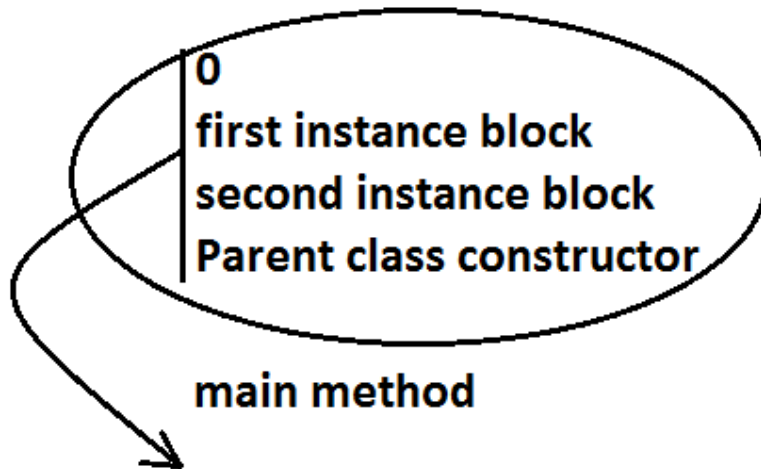
Whenever we are executing a java class static control flow will be executed. In the Static control flow Whenever we are creating an object the following sequence of events will be performed automatically.

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

Note: static control flow is one time activity and it will be executed at the time of class loading.
But instance control flow is not one time activity for every object creation it will be executed.

Instance control flow in Parent to Child relationship :

```
Example:
class Parent
{
        int x=10;
        {
                methodOne();
                System.out.println("Parent first instance block");
        }
        Parent()
        {
                System.out.println("parent class constructor");
        }
        public static void main(String[] args)
        {
                Parent p=new Parent();
                System.out.println("parent class main method");
        }
        public void methodOne()
        {
                System.out.println(y);
        }
        int y=20;
```

```
}
class Child extends Parent
{
        int i=100;
        {
                methodTwo();
                System.out.println("Child first instance block");
        }
        Child()
        {
                System.out.println("Child class constructor");
        }
        public static void main(String[] args)
        {
                Child c=new Child();
                System.out.println("Child class main method");
        }
        public void methodTwo()
        {
                System.out.println(j);
        }
        {
                System.out.println("Child second instance block");
        }
        int j=200;
}

Output:
E:\scjp>javac Child.java
E:\scjp>java Child
0
Parent first instance block
Parent class constructor
0
Child first instance block
Child second instance block
Child class constructor
Child class main method
```

Whenever we are creating child class object the following sequence of events will be executed automatically.

1. **Identification of instance members from Parent to Child.**
2. **Execution of instance variable assignments and instance block only in Parent class.**
3. **Execution of Parent class constructor.**
4. **Execution of instance variable assignments and instance blocks in Child class.**
5. **Execution of Child class constructor.**

Note: Object creation is the most costly operation in java and hence if there is no specific requirement never recommended to crate objects.

```
Example 1:
public class Initilization
{
        private static String methodOne(String msg)  //-->1
        {
                System.out.println(msg);
```

```
                return msg;
        }
        public Initilization()  //-->4
        {
                m=methodOne("1");  //-->9
        }
        {
                                        //-->5
                m=methodOne("2");               //-->7
        }
        String m=methodOne("3");        //-->6 , //-->8
        public static void main(String[] args) //-->2
        {
                Object obj=new Initilization();   //-->3
        }
}
```

Analysis:


```
Output:
2
3
1
Example 2:
public class Initilization
{
        private static String methodOne(String msg)  //-->1
        {
                System.out.println(msg);
                return msg;
        }
        static String m=methodOne("1");  //-->2,       //-->5
        {
                m=methodOne("2");
        }
        static                          //-->3
        {
                m=methodOne("3");                             //-->6
        }
        public static void main(String[] args)  //-->4
        {
                Object obj=new Initilization();
        }
}
Output:
1
3
2
```

We can't access instance variables directly from static area because at the time of execution of static area JVM may not identify those members.

Example:

```
class Test
{
    int i=10;
    public static void main(String[] args)
    {
        System.out.printl(i);
    }
} output:
```

compile time error
non-static variable i cannot be referenced from a static context

- But from the instance area we can access instance members directly.
- Static members we can access from anywhere directly because these are identified already at the time of class loading only.

Type casting:

Parent class reference can be used to hold Child class object but by using that reference we can't call Child specific methods.
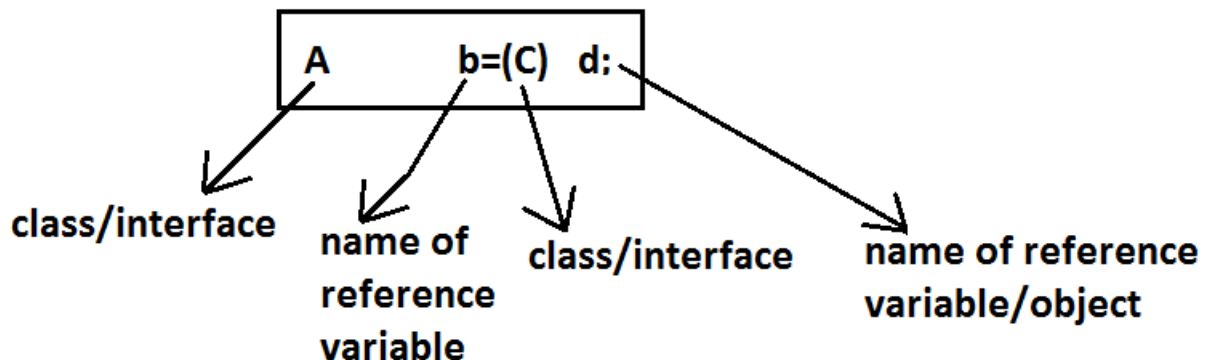
```
Example:
Object o=new String("ashok");//valid
System.out.println(o.hashCode());//valid
System.out.println(o.length());//
        C.E:cannot find symbol,
          symbol  : method length(),
        location: class java.lang.Object
```

Similarly we can use interface reference to hold implemented class object.

Example:
Runnable r=new Thread();
Type casting syntax:

A   b=(C) d;

class/interface   name of reference variable   class/interface   name of reference variable/object

Compile time checking :

Rule 1: The type of "d" and "c" must have some relationship [either Child to Parent (or) Parent to Child (or) same type] otherwise we will get compile time error saying inconvertible types.

Example 1:

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

Example 2:

```
String s=new String("bhaskar");
StringBuffer sb=(StringBuffer)s;
```
(inval id)

output:
compile time error
E:\scjp>javac Test.java
Test.java:6: inconvertible types
found   : java.lang.String
required: java.lang.StringBuffer
        StringBuffer sb=(StringBuffer)s;

Rule 2: "C" must be either same (or) derived type of "A" otherwise we will get compile time error saying incompatible types.
Found: C
Required: A
Example 1:

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

Example 2:

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

output:

compile time error
E:\scjp>javac Test.java
Test.java:6: incompatible types
found   : java.lang.String
required: java.lang.StringBuffer
        StringBuffer sb=(String)o;

Runtime checking :

The underlying object type of "d" must be either same (or) derived type of "C" otherwise we will get runtime exception saying ClassCastException.
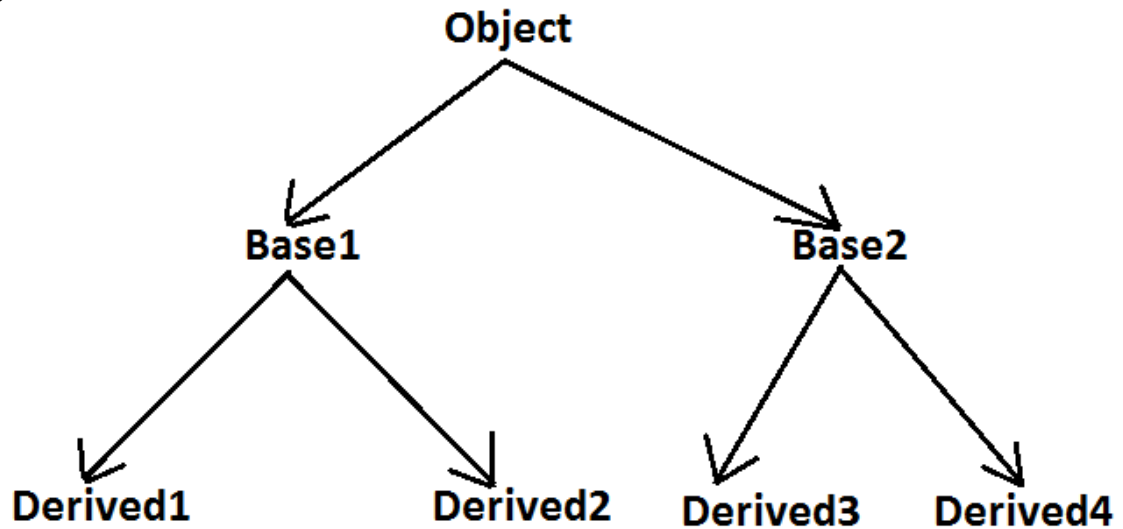
Example:

```
Object o=new String("bhaskar");
StringBuffer sb=(StringBuffer)o;
```
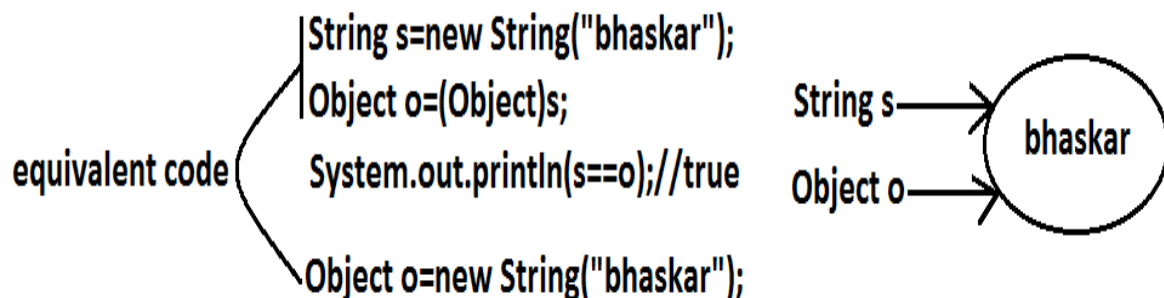
Runtime Exception:ClassCastException

Diagram:


```
Base1 b=new Derived2();//valid
Object o=(Base1)b;//valid
Object o1=(Base2)o;//invalid
Object o2=(Base2)b;//invalid
Base2 b1=(Base1)(new Derived1());//invalid
Base2 b2=(Base2)(new Derived3());//valid
Base2 b2=(Base2)(new Derived1());//invalid
```

Through Type Casting just we are converting the type of object but not object itself that is we are performing type casting but not object casting.

Through Type Casting we are not create any new objects for the existing objects we are providing another type of reference variable(mostly Parent type).

Example:

Example 1:

Parent ──→

```java
class Parent
{
    public void methodOne()
    {
        System.out.println("A");
    }
}
```
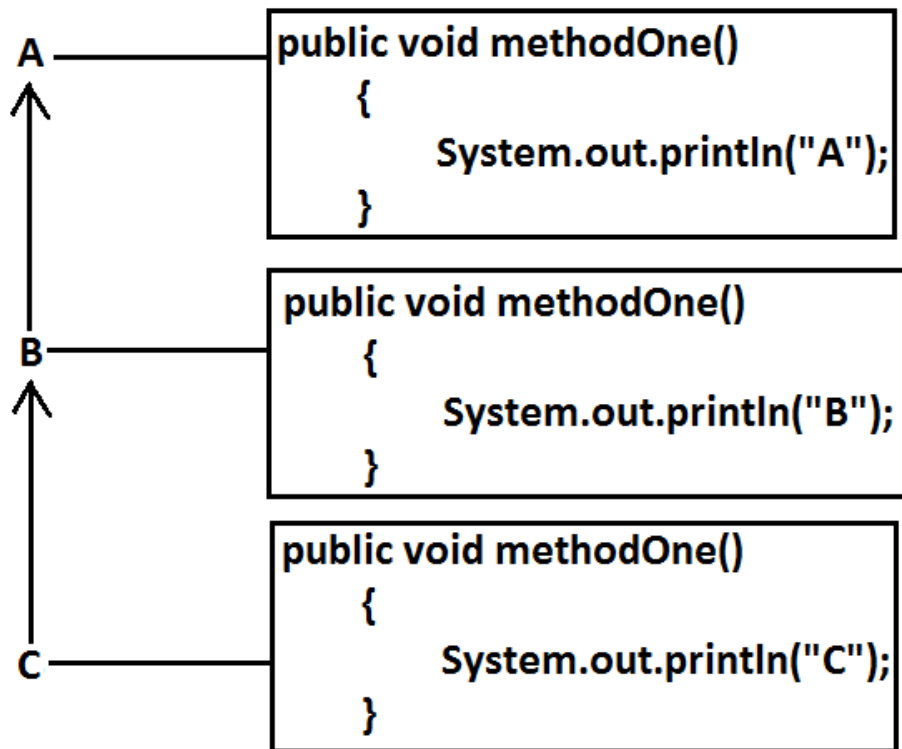
Child ──→

```java
class Child extends Parent
{
    public void methodOne()
    {
        System.out.println("B");
    }
    public void methodTwo()
    {
        System.out.println("C");
    }
    public static void main(String[] args)
    {
        Child c=new Child();
        c.methodOne();//B
        c.methodTwo();//C
        ((Parent)c).methodOne();//B
    }
}
```

```java
Parent p=new Parent();
((Parent)c).methodTwo();
```

output:
compile time error.
Child.java:25: cannot find symbol
symbol : method methodTwo()
location: class Parent
        ((Parent)c).methodTwo();

Example 2:

```
A ──────────  public void methodOne()
▲                 {
│                      System.out.println("A");
│                 }
│
│
B ──────────  public void methodOne()
▲                 {
│                      System.out.println("B");
│                 }
│
│
C ──────────  public void methodOne()
                  {
                       System.out.println("C");
                  }
```
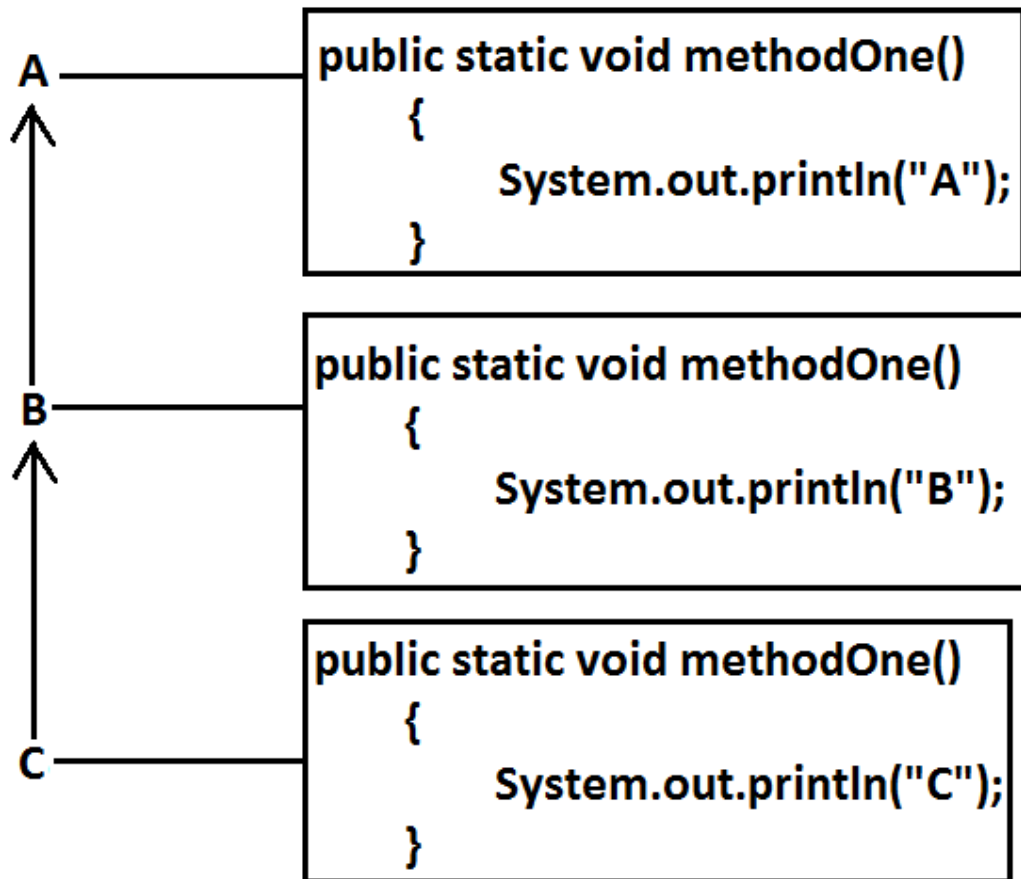
It is overriding and method resolution is based on runtime object.
```
C c=new C();
c.methodOne();//c
((B)c).methodOne();//c
((A)((B)c)).methodOne();//c
```

Example 3:

A ──────────
```
public static void methodOne()
    {
        System.out.println("A");
    }
```

B ──────────
```
public static void methodOne()
    {
        System.out.println("B");
    }
```

C ──────────
```
public static void methodOne()
    {
        System.out.println("C");
    }
```
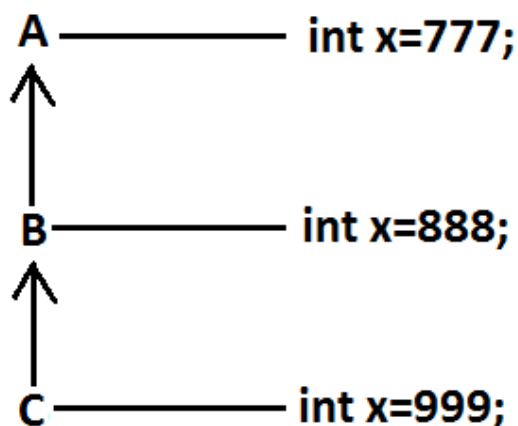
It is method hiding and method resolution is based on reference type.
```
C c=new C();
c.methodOne();//C
((B)c).methodOne();//B
((A)((B)c)).methodOne();//A
```

Example 4:

A ─────────── int x=777;

B ─────────── int x=888;

C ─────────── int x=999;

```
C c=new C();
System.out.println(c.x);//999
System.out.println(((B)c).x);//888
System.out.println(((A)((B)c)).x);//777
```

- **Variable resolution is always based on reference type only.**
- **If we are changing variable as static then also we will get the same output.**

Coupling :

The degree of dependency between the components is called coupling.
```
Example:
class A
{
        static int i=B.j;
}
class B extends A
{
        static int j=C.methodOne();
}
class C extends B
{
        public static int methodOne()
        {
                return D.k;
        }
}
class D extends C
{
        static int k=10;
        public static void main(String[] args)
        {
                D d=new D();
        }
}
```
The above components are said to be tightly coupled to each other because the dependency between the components is more.
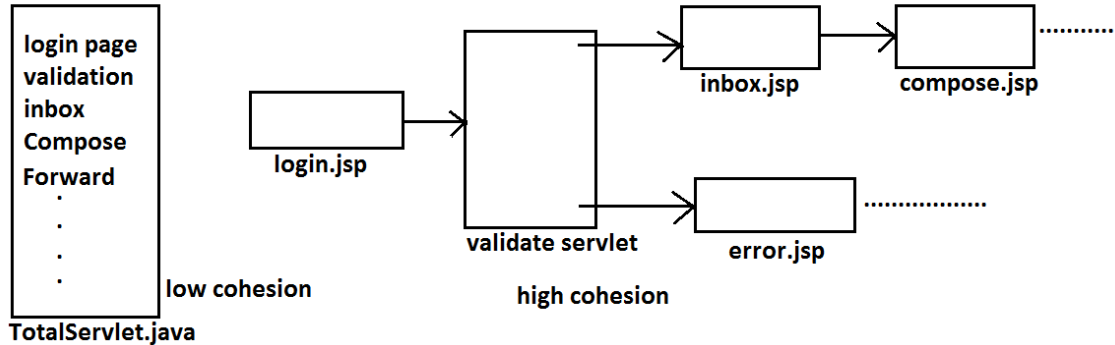
Tightly coupling is not a good programming practice because it has several serious disadvantages.

1. **Without effecting remaining components we can't modify any component hence enhancement(development) will become difficult.**
2. **It reduces maintainability of the application.**
3. **It doesn't promote reusability of the code.**

It is always recommended to maintain loosely coupling between the components.

Cohesion:

For every component we have to maintain a clear well defined functionality such type of component is said to be follow high cohesion.

Diagram:

High cohesion is always good programming practice because it has several advantages.

1. Without effecting remaining components we can modify any component hence enhancement will become very easy.
2. It improves maintainability of the application.
3. It promotes reusability of the application.(where ever validation is required we can reuse the same validate servlet without rewriting)

Note: It is highly recommended to follow loosely coupling and high cohesion.