## 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.**
**Note: 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.**

Example :

By using 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:

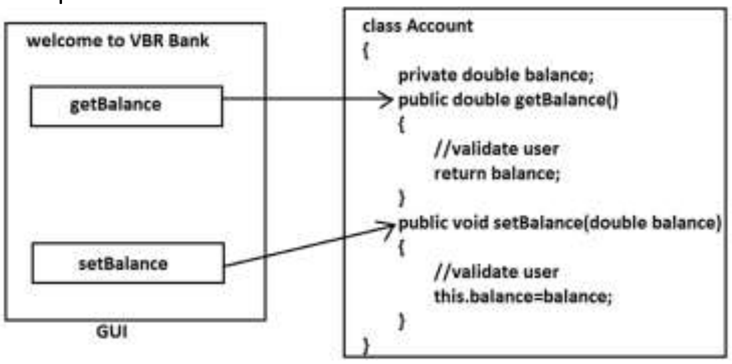
**1. We can achieve security as we are not highlighting our internal implementation.(i.e., outside person doesn't aware our internal implementation.)**
**2. Enhancement will become very easy because without effecting end user we can able to perform any type of changes in our internal system.**
**3. It 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;
    }
}
```

**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 {
int x=10; //not
}
class B extends A {
private int y=20; //not
}
class C extends B {
private int z=30; //not
}
```

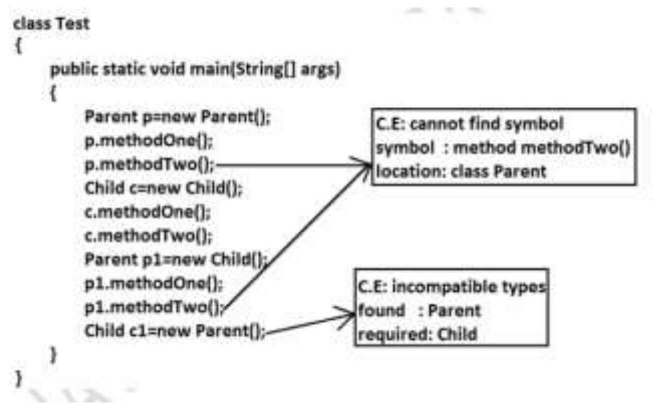
Note: if the parent class is not tightly encapsulated then no child class is tightly encapsulated.

## 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 :

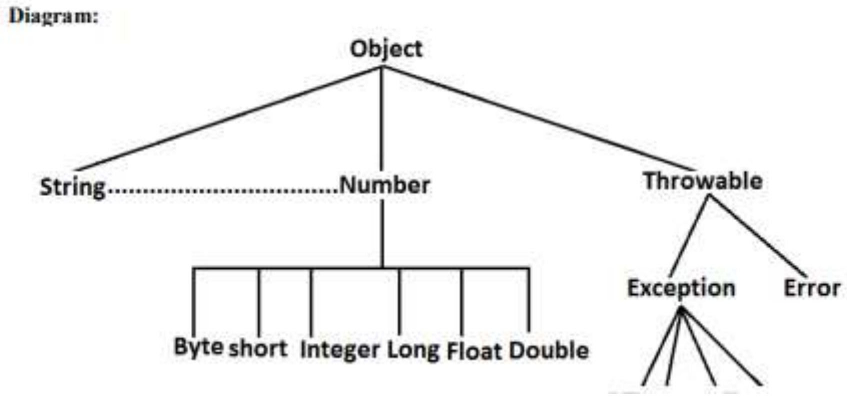
1. Whatever the parent has by default available to the child but whatever the child has by default not available to the parent. Hence on the child reference we can call both parent and child class methods. But on the parent reference we can call only methods available in the parent class and we can't 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:**

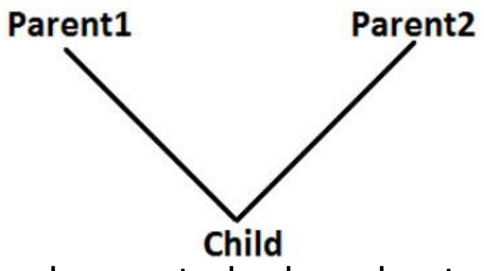
```
class Loan {
//common methods which are required for any type of loan.
}
class HousingLoan extends Loan {
//Housing loan specific methods.
}
class EducationLoan extends Loan {
//Education Loan specific methods.
}
```

**� 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.**

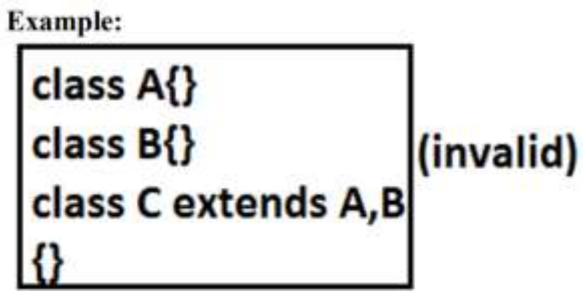


## Multiple inheritance :

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



**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:

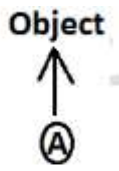
```
class A{}
class B{}
class C extends A,B
{}
```

(invalid)

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

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

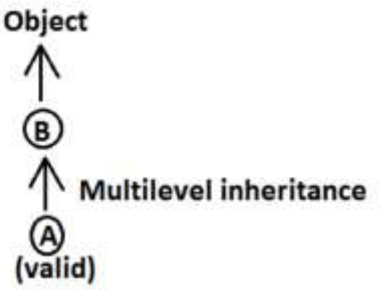
```
class A{}
```



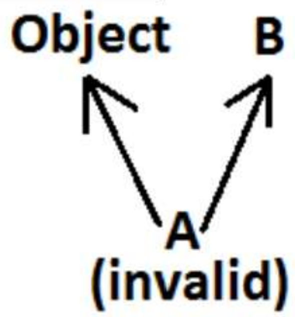
**Example:**
**If our class extends any other class then our class is not direct child class of object, It is indirect child class of object , which forms multilevel inheritance**

Example 1:

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

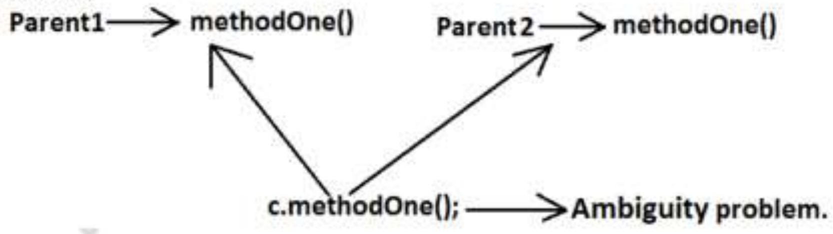


**Example 2:**



**Why java won't provide support for multiple inheritance?**
**There may be a chance of raising ambiguity problems**

Example:



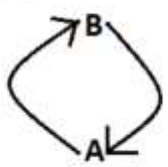
**Why ambiguity problem won't be there in interfaces?**
**Interfaces having dummy declarations and they won't have implementations hence no ambiguity problem.**

## 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{} $\xrightarrow{C.E}$ cyclic inheritance involving A

# HAS-A relationship:

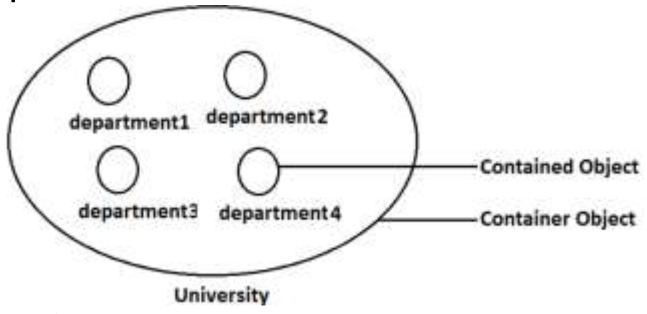
**1. HAS-A relationship is also known as composition (or) aggregation.**
**2. There is no specific keyword to implement HAS-A relationship but mostly we**

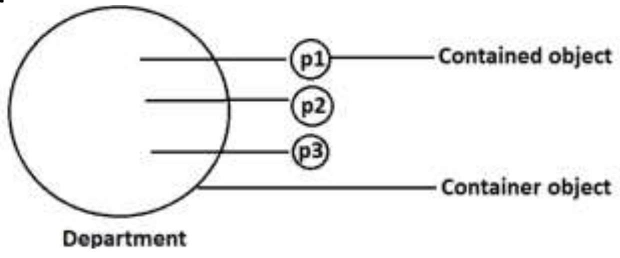
can use new operator.
3. The main advantage of HAS-A relationship is reusability.
Example:

```
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:

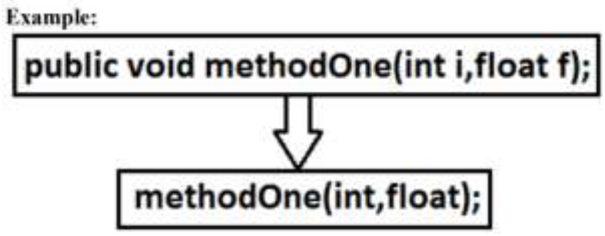
Composition:
Without existing container object if there is no chance of existing contained objects then the relationship between container object and contained object is called composition which is a strong association.
Example:
University consists of several departments whenever university object destroys 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:


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.



 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:

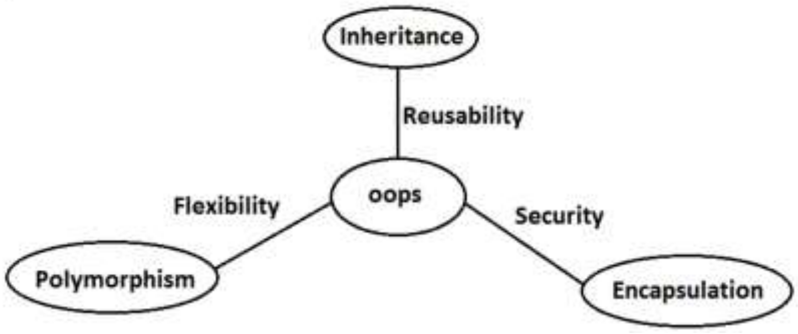
Same name with different forms is the concept of polymorphism.

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)

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

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

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.

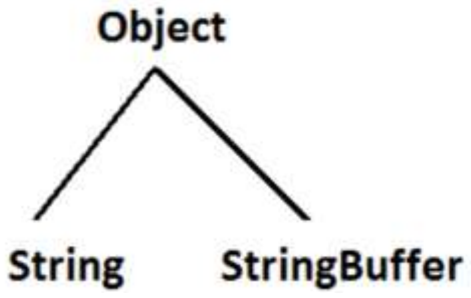
 In overloading if compiler is unable to find the method with exact match we won't get any compile time error immediately.
 1st compiler promotes the argument to the next level and checks whether the matched method is available or not if it is available then that method will be considered if it is not 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 :

$$\begin{tikzcd}
\text{byte} \arrow[r] & \text{short} \arrow[dr] & & & & \\
& & \text{int} \arrow[r] & \text{long} \arrow[r] & \text{float} \arrow[r] & \text{double} \\
& \text{char} \arrow[ur] & & & &
\end{tikzcd}$$

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
}
}
```



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.

Example 1:
```

class Parent {
public void property(){
System.out.println("cash+land+gold");
}
public void marry() {
System.out.println("subbalakshmi"); //overridden
method
}
}
class Child extends Parent{ //overriding
public void marry() {
System.out.println("3sha/4me/9tara/anushka");
//overriding method
}
}
class Test {
public static void main(String[] args){
Parent p=new Parent();
p.marry();//subbalakshmi(parent method)
Child c=new Child();
c.marry();//Trisha/nayanatara/anushka(child method)
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**

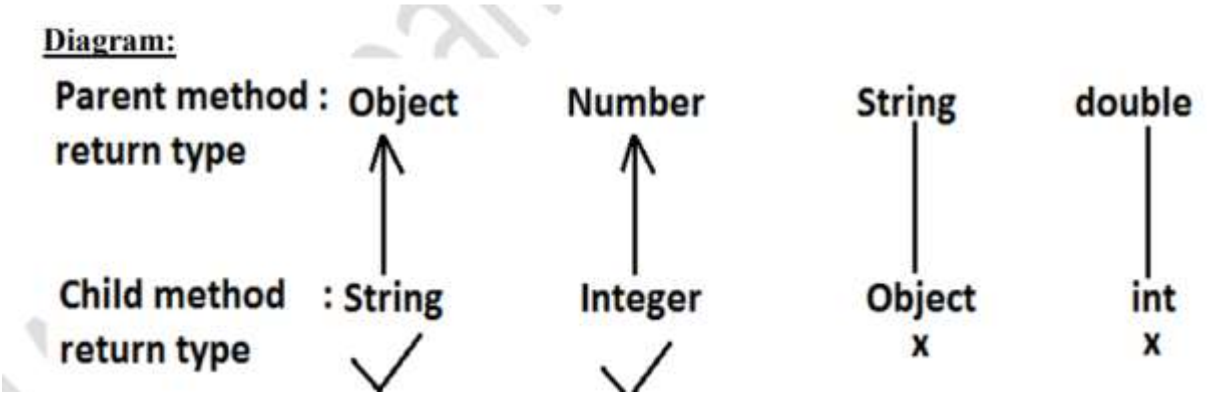
**Note: In overriding runtime object will play the role and reference type is dummy.**

## Rules for overriding :

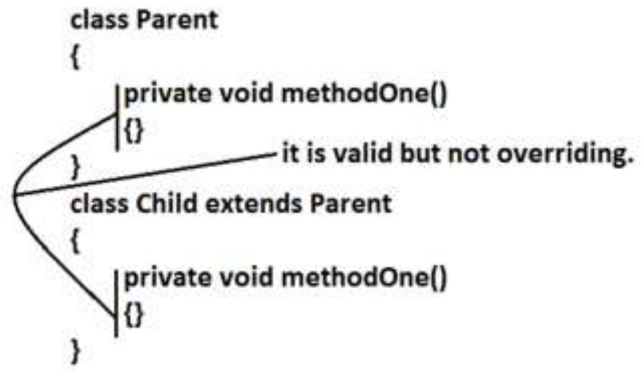
**1. In overriding method names and arguments must be same. That is method signature must be same.**
**2. Until 1.4 version the return types must be same but from 1.5 version onwards covariant return types are allowed.**
**3. According to this Child class method return type need not be same as Parent 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".**



**Co-variant return type concept is applicable only for object types but not for primitives.**
**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()
    {}
}
class Child extends Parent
{
    private void methodOne()
    {}
}
```

it is valid but not overriding.

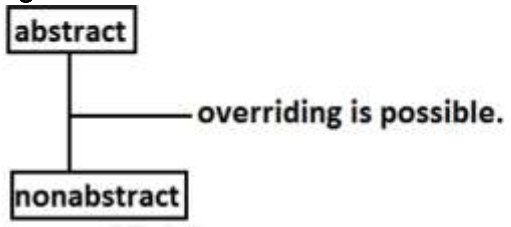
**Parent class final methods we can't override in the Child class.**

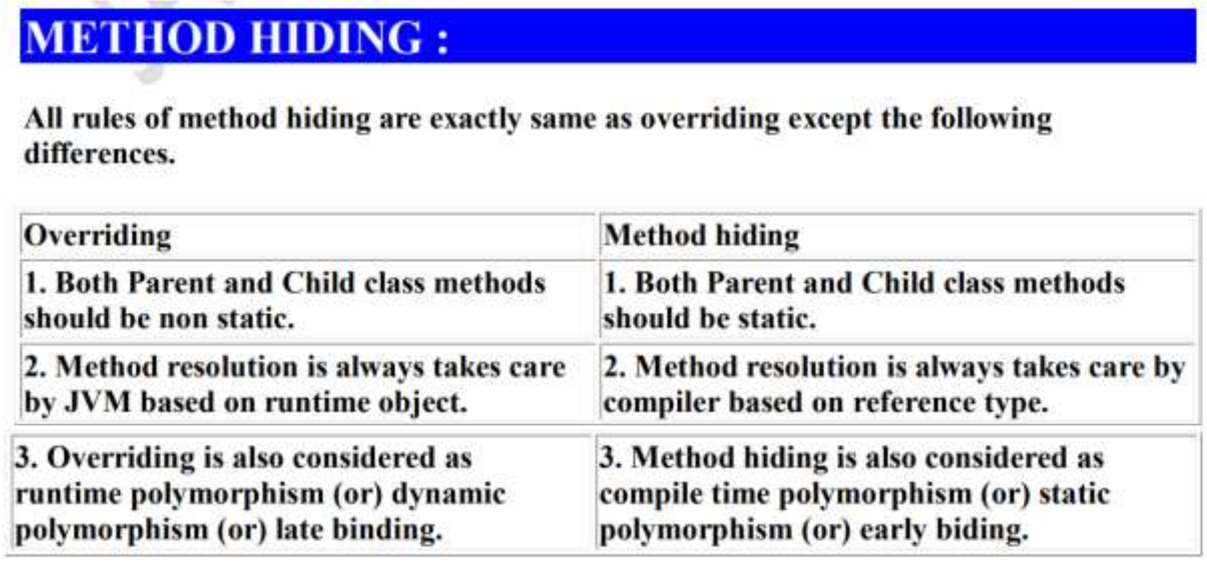
Example:
```
class Parent {
public final void methodOne() {}
}
class Child extends Parent{
public void methodOne(){}
}
```
Output:
```
Compile time error:
methodOne() in Child cannot override methodOne()
in Parent; overridden method is final
```

Parent class non final methods we can override as final in child class. We can override native methods in the child classes.

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

Example:
```
abstract class Parent {
public abstract void methodOne();
}
class Child extends Parent {
public void methodOne() { }
}
```
Diagram:





## 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. |

## 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.

## 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.

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

The only applicable modifiers for the constructors are public, default, private, protected.

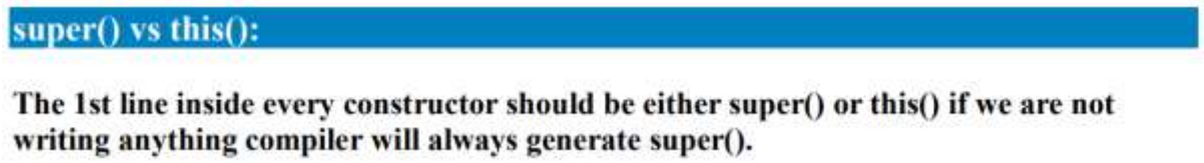
If we are using any other modifier we will get compile time error.

## 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.



**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().

<u>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.</u>

Example:

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

Output:

```
Compile time error.
```

**Call to super must be first statement in constructor**

**<u>Case 2: We can use either super() (or) this() but not both simultaneously.</u>**

**Example:**

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

**Output:**

**Compile time error.**

**Call to this must be first statement in constructor**

**<u>Case 3: We can use super() (or) this() only inside constructor. If we are using anywhere</u> else we will get compile time error.**

**Example:**

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

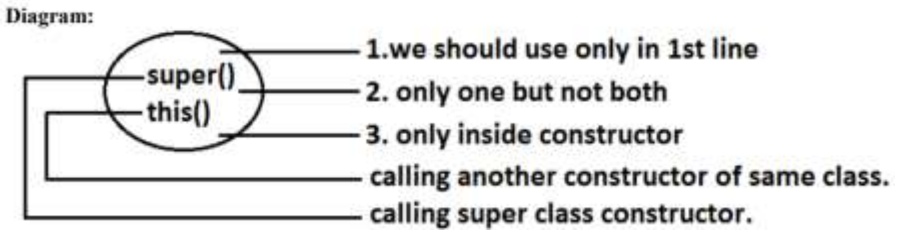
**Output:**

**Compile time error.**

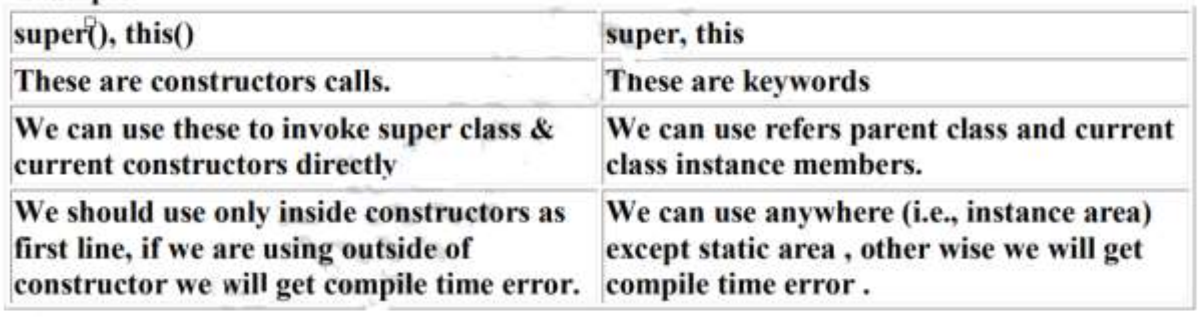
**Call to super must be first statement in constructor**

## That is we can call a constructor directly from another constructor only.

**Diagram:**


super()
this()
1.we should use only in 1st line
2. only one but not both
3. only inside constructor
calling another constructor of same class.
calling super class constructor.

**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 . |

## <u>Overloaded constructors :</u>

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