

xxObject Oriented Programming Concepts

1.Data Hiding

It just provides a way to protect our data from the outside world. What it means is, let's say if we made our instance variable public, then anyone can change its state. But if we make our instance variable private/protected then actually we are restricting outside entities from making changes to it.

Data hiding is a concept in object-oriented programming which confirms the security of members of a class from unauthorized access. Data hiding is a technique of protecting the data members from being manipulated or hacked from any other source. Data is the most sensitive and volatile content of a program which if manipulated can result in an incorrect output and it also harms the integrity of the data. Data hiding is controlled in Java with the help of access modifiers (private, public and protected). The data which is public is accessible from outside the class hence if you want to hide your data or restrict it from being accessed from outside, declare your data private.

```
class DatHidingAndEncapsulationDemo{
    private int empId;
    private String empName;
    private int empAge;

    //Getter and Setter methods
    public int getEmpId(){
        return empId;
    }

    public String getEmpName(){
        return empName;
    }

    public int getEmpAge(){
        return empAge;
    }

    public void setEmpAge(int newValue){
        empAge = newValue;
    }

    public void setEmpName(String newValue){
        empName = newValue;
    }

    public void setEmpId(int newValue){
        empId = newValue;
    }
}
```

```
}  
}  
public class DataHidingAndEncapsulationTest{  
    public static void main(String args[]){  
        DatHidingAndEncapsulationDemo obj = new DatHidingAndEncapsulationDemo ();  
        obj.setEmpName("Mario");  
        obj.setEmpAge(32);  
        obj.setEmpid(112233);  
        System.out.println("Employee Name: " + obj.getEmpName());  
        System.out.println("Employee SSN: " + obj.getEmpid());  
        System.out.println("Employee Age: " + obj.getEmpAge());  
    }  
}
```

2. Abstraction

Abstraction is a methodology of hiding the implementation of internal details and showing the functionality to the users.

A layman who is using mobile phone doesn't know how it works internally but he can make phone calls.

Abstraction in Java is achieved using abstract classes and interfaces.

Abstraction in any programming language works in many ways. It can be seen from creating subroutines to defining interfaces for making low-level language calls.

Some abstractions try to limit the breadth of concepts a programmer needs, by completely hiding the abstractions they in turn are built on, e.g. design patterns.

Types of abstraction

Typically abstraction can be seen in two ways:

1. Data abstraction

Data abstraction is the way to create complex data types and exposing only meaningful operations to interact with the data type, whereas hiding all the implementation details from outside works.

The benefit of this approach involves capability of improving the implementation over time e.g. solving performance issues is any. The idea is that such changes are not supposed to have any impact on client code since they involve no difference in the abstract behavior.

2. Control abstraction

A software is essentially a collection of numerous statements written in any programming language. Most of the times, statement are similar and repeated over places multiple times.

Control abstraction is the process of identifying all such statements and expose them as a unit of work. We normally use this feature when we create a function to perform any work.

How to achieve abstraction in java?

As abstraction is one of the core principles of Object-oriented programming practices and Java following all OOPs principles, abstraction is one of the major building blocks of java language.

In java, abstraction is achieved by interfaces and abstract classes. Interfaces allows you to abstract the implemetation completely while abstract classes allow partial abstraction as well.

Data abstraction spans from creating simple data objects to complex collection implementations such as HashMap or HashSet.

Similarly, control abstraction can be seen from defining simple function calls to complete open source frameworks. control abstraction is main force behind structured programming.

3.Encapsulation

Encapsulation is a process of binding or wrapping the data and the codes that operates on the data into a single entity. This keeps the data safe from outside interface and misuse. One way to think about encapsulation is as a protective wrapper that prevents code and data from being arbitrarily accessed by other code defined outside the wrapper.

Encapsulation is a way to achieve "information hiding" so, following your example, you don't "need to know the internal working of the mobile phone to operate" with it. You have an interface to use the device behaviour without knowing implementation details.

Abstraction on the other side, can be explained as the capability to use the same interface for different objects. Different implementations of the same interface can exist. Details are hidden by encapsulation.

Abstraction : you'll never buy a "device", but always buy something more specific : iPhone, GSII, Nokia 3310... Here, iPhone, GSII and N3310 are concrete things, device is abstract.

Encapsulation : you've got several devices, all of them have got a USB port. You don't know what kind of printed circuit there's back, you just have to know you'll be able to plug a USB cable into it.

Abstraction is a concept, which is allowed by encapsulation. My example wasn't the best one (there's no real link between the two blocks).

You can do encapsulation without using abstraction, but if you wanna use some abstraction in your projects, you'll need encapsulation.

4. IS-A (Inheritance)

In Object oriented programming, IS-A relationship denotes “one object is type of another”.
IS-A relation denotes Inheritance methodology.

In Java, Inheritance can be implemented with **extends** (in case of class) and **implements** (in case of interface) keywords.

“Is-A” — A child is a part of parent

For example is :-

1. A Mammal is a Animal

2. A Dog is a Mammal

Hence: Dog IS-A Animal as well

```
public class Animal { }
```

```
public class Mammal extends Animal { }
```

```
public class Dog extends Mammal { }
```

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

Inheritance allows us to reuse of code, it improves reusability in your java application.

The biggest advantage of Inheritance is that the code that is already present in base class need not be rewritten in the child class.

Example :

```
class Teacher {  
    String designation = "Teacher";  
    String collegeName = "Prestige Point";  
    void does(){  
        System.out.println("Teaching");  
    }  
}  
  
public class ComputerTrainer extends Teacher  
{  
    String mainSubject = "java programming";
```

```
public static void main(String args[])
{
    ComputerTrainer obj = new ComputerTrainer ();
    System.out.println(obj.collegeName);
    System.out.println(obj.designation);
    System.out.println(obj.mainSubject);
    obj.does();
} }
```

The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

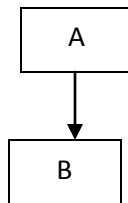
```
class Teacher {
    private String designation = "Teacher";
    private String collegeName = "Prestige Point";
    public String getDesignation() {
        return designation;
    }
    protected void setDesignation(String designation) {
        this.designation = designation;
    }
    protected String getCollegeName() {
        return collegeName;
    }
    protected void setCollegeName(String collegeName) {
        this.collegeName = collegeName;
    }
    void does(){
        System.out.println("Teaching");
    }
}

public class ComputerTrainer extends Teacher{
    String mainSubject = "JAVA";
    public static void main(String args[]){
        ComputerTrainer obj = new ComputerTrainer ();
        System.out.println(obj.getCollegeName());
        System.out.println(obj.getDesignation());
        System.out.println(obj.mainSubject);
        obj.does();
    }
}
```

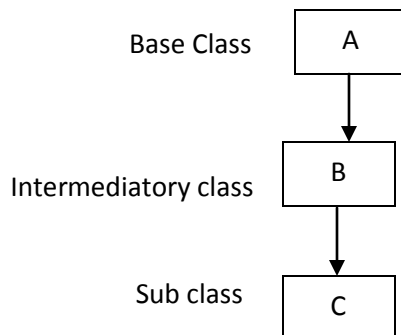
Types of Inheritance in Java

Below are the different types of inheritance which is supported by Java.

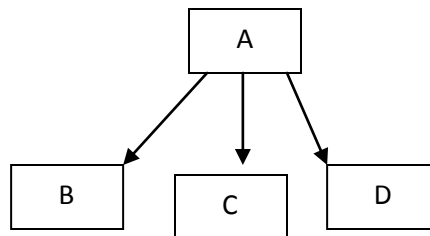
Single Inheritance : In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



Multilevel Inheritance : In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class **cannot directly access the grandparent's members**.



Hierarchical Inheritance : In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class. In below image, the class A serves as a base class for the derived class B, C and D.



Multiple Inheritance : In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. **Please note that Java does not support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces.**

Class C extends A,B
Is not allowed in java.

Why Multiple inheritance not supported by java in case of classes:

Java programming language does not permit you to extend more than one class is to avoid the issues of multiple inheritance of state, which is the ability to inherit fields from multiple classes

If multiple inheritance is allowed and When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses. It will cause two issues.

1. What if methods or constructors from different super classes instantiate the same field?
2. Which method or constructor will take precedence?

Consider the following class:

```
public class Abc{
    public void doSomething(){
    }
}

public class Xyz {
    public void doSomething(){
    }
}

public class Wxy extends Abc , Xyz {

    public static void main(String[] args)
    {
        Wxy x = new Wxy ();
        x.doSomething();
    } }
```

This type of syntax is not allowed in java to avoid ambiguity.

Which doSomething() method should be called here beccasue this method is available in both parent classes. To avoid this situation Java doesn't support multiple inheritance. This problem is called ambiguity.

Parent class reference variable may hold child class object :

Parent and Child classes having same data member in Java

The reference variable of the Parent class is capable to hold its object reference as well as its child object reference.

What about non-method members. For example :

// A Java program to demonstrate that non-method members are accessed according to reference
// type (Unlike methods which are accessed according to the referred object)

```
class Parent
{
    int value = 1000;
    Parent() { System.out.println("Parent Constructor"); }
    void helloFromParent() { System.out.println("Hello from Parent Class"); }
```

```
}

class Child extends Parent
{
    int value = 10;
    Child() { System.out.println("Child Constructor"); }
    void helloFromChild() { System.out.println("Hello from Child Class"); }
}

// Driver class
class Test
{
    public static void main(String[] args)
    {
        Child obj=new Child();
        System.out.println("Reference of Child Type : " + obj.value); // Reference of Child Type : 10
        obj. helloFromParent(); // no problem , this method is inherited in Child class
        obj. helloFromChild(); // child specific method, so we can invoke with child class reference variable.
        // Note that doing "Parent par = new Child()" would produce same result
        Parent par = obj;

        // Par holding obj will access the value variable of parent class
        System.out.println("Reference of Parent Type : " + par.value); // Reference of Parent Type : 1000
        obj. helloFromParent(); // no problem , this method is available in parent class , so with parent
                               //reference we can invoke this method
        obj. helloFromChild(); // it is child specific method, so we can not invoke this with parent class
                               //reference variable. You will get compile time error

    }
}
```

If a parent reference variable is holding the reference of the child class and we have the “value” variable in both the parent and child class, it will refer to the parent class “value” variable, whether it is holding child class object reference. The reference holding the child class object reference will not be able to access the members (functions or variables) of the child class. It is because compiler uses special run-time polymorphism mechanism only for methods.

OBSERVATIONS from above example:

1. Whatever the parent class has, is by default available to the child. Hence by using child reference, we can call both parent and child class methods.
2. Whatever the child class has, by default is not available to parent, hence on the parent class reference we can only call parent class methods but not child specific methods.
3. Parent class reference can be used to hold child class objects , but by using that reference we can call

- only parent class methods but not child specific methods.
4. We cant use child class reference to hold parent class objects

Entire java API is implemented based on inheritance. Every java class extends from Object class which has most common and basic methods required for all java classes. Hence we can say “Object” class is root class of all java methods.

Object class is available in **java.lang** package. All predefined classes and customized classes or user defined classes are by default child class of Object class.

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

If our class extend any other class, then our class is indirect child class of Object class.

Cyclic inheritance is not allowed in Java.

Class A extends A { }

Or

Class A extends B { }

Class B extends A { }

Is known as cyclic inheritance

5.HAS-A Relationship(Assoiciation)

Has-A means an instance of one class “has a” reference to an instance of another class or another instance of same class.

It is also known as “composition” or “aggregation”.

There is no specific keyword to implement HAS-A relationship but mostly we are depended upon “new” keyword.

Consider we are storing the information of the student, then we may create a class like below –

```
Class Student {  
    int roll;  
    String sname;  
    Address address;  
}
```

In the above class we have entity reference “Address” which stores again its own information like street,city,state,zip. like below –

```
Class Address {  
    String street;  
    String state;  
    String zip;  
    String city;  
}
```

so we can say that **Student HAS-A Address.**
Complete Program

```
public class Address {  
    String street;  
    String city;  
    String state;  
    String zip;  
  
    public Address(String street, String city,  
                    String state, String zip) {  
        this.street = street;  
        this.city = city;  
        this.state = state;  
        this.zip = zip;  
    }  
}
```

```
public class Student {  
    int roll;  
    Address address;  
    Student(int rollNo,Address addressDetail)  
    {  
        roll = rollNo;  
        address = addressDetail; }  
    void printStudentDetails(Address address1) {  
        System.out.println("Roll : " + roll);  
        System.out.println("Street : " + address1.street);  
        System.out.println("City : " + address1.city);  
        System.out.println("State : " + address1.state);  
        System.out.println("Zip : " + address1.zip);  
    }  
    public static void main(String[] args) {  
        Address address1;  
        address1 = new Address("1-ST","PN","Mah","41");  
        Student s1 = new Student(1,address1);  
        s1.printStudentDetails(address1);  
    }  
}
```

Composition :

Without existence of container object, if there is no chance of existence of contained objects then container and contained objects are said to be strongly associated and this strong association is known as composition.

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

- In composition, both the entities are dependent on each other.
- When there is a composition between two entities, the composed object cannot exist without the other entity.

Example of Composition:

```
public class Job {  
    private String role;  
    private long salary;  
    private int id;  
  
    public String getRole() {  
        return role;  
    }  
    public void setRole(String role) {  
        this.role = role;  
    }  
    public long getSalary() {  
        return salary;  
    }  
    public void setSalary(long salary) {  
        this.salary = salary;  
    }  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
}
```

```
public class Person {  
    //composition has-a relationship  
    private Job job;  
    public Person(){  
        this.job=new Job();  
        job.setSalary(1000L);  
    }  
    public long getSalary() {  
        return job.getSalary();  
    }  
}
```

```
public class TestPerson {  
  
    public static void main(String[] args) {  
  
        Person person = new Person();  
  
        long salary = person.getSalary();  
  
    }  
}
```

Aggregation

Without existence of container object, if there is a chance of existence of contained objects then container and contained objects are said to be loosely associated and this strong association is known as aggregation.

- It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity

```
class Author
{
    String authorName;
    int age;
    String place;
    Author(String name,int age,String
place)
    {
        this.authorName=name;
        this.age=age;
        this.place=place;
    }
    public String getAuthorName()
    {
        return authorName;
    }
    public int getAge()
    {
        return age;
    }
    public String getPlace()
    {
        return place;
    }
}
```

```
class Book
{
    String name;
    int price;
    Author auth;
    Book(String n,int p,Author at)
    {
        this.name=n;
        this.price=p;
        this.auth=at;
    }
    public void showDetail()
    {
        System.out.println("Book is"+name);
        System.out.println("price "+price);
        System.out.println("Author is +auth.getAuthorName());
    }
}
```

```
class TestAggregation
{
    public static void main(String args[])
    {
        Author ath=new Author("Me",22,"India");
        Book b=new Book("Java",550,ath);
        b.showDetail();
    }
}
```

6. Specialization and Generalization

Specialization is defined as the process of subclassing a superclass entity on the basis of some distinguishing characteristic of the entity in the superclass. The reason for creating such hierarchical relationship is that:

- Certain attributes of the superclass may apply to some but not all entities of the superclass. These extended subclasses then can be used to encompass the entities to which these attributes apply.
- Maybe one or more of the subclasses participates in a relationship with another classes that neither concerns its siblings or parent classes.

Generalization is the bottom-up process of abstraction, where we club the differences among entities according to the common feature and generalize them into a single superclass. The original entities are thus subclasses of it. In more simple terms, it is just the reverse of specialization, which is a top-down process whereas generalization is bottom up. That's it.

So, basically when we are referring only to specialization, it applies to both specialization and generalization as well, as one is the flip side of the other.

Example

```
class SuperClass{
    void displayOne(){
        System.out.println (" Class SuperClass's Method ");
    }
}

class SubClass extends SuperClass{
    void diaplayTwo(){
        System.out.println (" Class SubClass 's Method ");
    }
}

class Casting {
    public static void main(String args[]) {
        SuperClass sp = new SuperClass();
        sp.displayOne();           /* output : Class SuperClass's Method */
        SubClass sb = new SubClass();
        sb.displayOne();           /* output : Class SuperClass's Method */
        sb.diaplayTwo();           /* output : Class SubClass 's Method */

        // ***** Casting ***** //
        SuperClass sp1 = new new SubClass();
        sp1.displayOne();           /* output : Class SuperClass 's Method */
        sp1.diaplayTwo();           /* There is no possible to access */

        SubClass sb1 = new SuperClass();
        sb1 .displayOne();           /* There is no possible to access */
        sb1 .diaplayTwo();           /* There is no possible to access */
    }
}
```

7.Method Overloading

In Java, two or more methods can have same name if they differ in parameters (different number of parameters, different types of parameters, or both). These methods are called overloaded methods and this feature is called method overloading. For example:

```
void func() { ... }  
void func(int a) { ... }  
float func(double a) { ... }  
float func(int a, float b) { ... }
```

Here, func() method is overloaded. These methods have same name but accept different arguments.

Notice that, the return type of these methods are not same. Overloaded methods may or may not have different return type, but they must differ in parameters they accept.

Why method overloading?

Suppose, you have to perform addition of the given numbers but there can be any number of arguments (let's say either 2 or 3 arguments for simplicity).

In order to accomplish the task, you can create two methods **sumTwoNumber(int, int)** and **sumThreeNumber(int, int,int)** for two and three parameters respectively. However, other programmers as well as you in future may get confused as the behavior of both methods is same but they differ by name.

The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. **This helps to increase readability of the program.**

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

For example: This is a valid case of overloading

```
add(int, int)  
add(int, int, int)
```

2. Data type of parameters.

For example:

```
add(int, int)  
add(int, float)
```

3. Sequence of Data type of parameters.

For example:

add(int, float)

add(float, int)

Examples :

```
class MethodOverloading {
    public void methodOne() { System.out.println("no-argmethod"); }
    public void methodOne(int i) { System.out.println("int-argmethod"); } // overloaded methods
    public void methodOne(double d) { System.out.println("double-argmethod"); }

    public static void main(String[] args) {
        MethodOverloading t=new MethodOverloading ();
        t.methodOne();           //no-argmethod
        t.methodOne(10);         //int-argmethod
        t.methodOne(10.5);       //double-argmethod
    }
}
```

In overloading compiler is responsible to perform method resolution (decision) based on the reference type. Hence overloading is also considered as Compile time polymorphism or static binding or early binding

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.
- First 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 :

byte->short->int->long->float->double

char
↑

```
class AutomaticPromotionInOverloading
{
    public void methodOne(int i) { System.out.println("int-argmethod"); }
    public void methodOne(float f) // overloaded methods
    { System.out.println("float-argmethod"); }
    public static void main(String[] args)
```

```
{
AutomaticPromotionInOverloading t=new AutomaticPromotionInOverloading ();
t.methodOne('a'); //int-arg method
t.methodOne(10l); //float-argmethod
t.methodOne(10.5); //C.E:cannotfindsymbol
} }
```

Case 2 : In overloading Child will always get high priority then Parent :

While resolving overloaded methods compiler will always give precedence for a child type argument, when compared with child type argument.

class ChildWillAlwaysGetHighPriorityThenParent

```
{
public void methodOne(String s) { System.out.println("String version"); }
public void methodOne(Object o) { System.out.println("Object version"); }
public static void main(String[] args)
{
ChildWillAlwaysGetHighPriorityThenParent t=new ChildWillAlwaysGetHighPriorityThenParent ();
t.methodOne("Neeraj"); //String version
t.methodOne(new Object()); //Object version
t.methodOne(null); //String version
}}
```

Case 3: In overloading method resolution is always based on reference type and run time object never play any role in overloading.

```
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
}}
```


Example of method overloading :

```
public class Sum {  
    // Overloaded sum(). This sum takes two int parameters  
    public int sum(int x, int y)  
    {  
        return (x + y);  
    }  
    // Overloaded sum(). This sum takes three int parameters  
    public int sum(int x, int y, int z)  
    {  
        return (x + y + z);  
    }  
    // Overloaded sum(). This sum takes two double parameters  
    public double sum(double x, double y)  
    {  
        return (x + y);  
    }  
  
    public static void main(String args[])  
    {  
        Sum s = new Sum();  
        System.out.println(s.sum(10, 20));  
        System.out.println(s.sum(10, 20, 30));  
        System.out.println(s.sum(10.5, 20.5));  
    }  
}
```

8. Method Overriding

Declaring a method in child class which is already present in the parent class is called Method Overriding.

In simple words, overriding means to override the functionality of an existing method.

Note:

- What ever 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.
- The Parent class method which is overridden is called overridden method.
- The Child class method which is overriding is called overriding method.

When a method in a sub class has **same name, same number of arguments and same type signature as a method in its super class, then the method is known as overridden method**. Method overriding is also referred to as runtime polymorphism. The key benefit of **overriding is the ability to define method that's specific to a particular subclass type**.

```
class Animal
{
    public void eat()
    {
        System.out.println("Generic Animal eating");
    }
    void motility
    {
        System.out.println("Generic Animal motility behaviour");
    }
}

class Dog extends Animal
{
    public void eat() //eat() method overridden by Dog class.
    {
        System.out.println("Dog eat meat");
    }
    public void bark()
    {
        System.out.println("Dog is barking");
    }
}
```

Parent class reference variable may Hold Child class Object.

```
class TestAnimalAndDog
{
public static void main(String[] args)
{
Animal a = new Animal();
a.eat();          // Generic Animal eating
a.motility();    // Generic Animal motility behaviour

Dog d = new Dog();
d.eat(); // Dog eat meat (Overriding method eat() of child class invoked )
d.bark(); // Dog is barking (Child class Dog specific method invoked)
```

```
Animal an = new Dog(); // Parent class reference variable may Hold Child class Object
an.eat();          // Dog eat meat (Overriding method eat() of child class invoked )
an.motility(); // inherited method of parent class Animal is invoked - Generic Animal motility behaviour
an.bark(); // We will get Compile Time error. With Parent reference variable we can not invoke child
           //specific method.
```

```
Dog do = new Parent(); // we will get Compile Time error. Child reference variable can not hold child
                       //class Object.
}
}
```

Note :

- 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
- The process of overriding method resolution is also known as dynamic method dispatch.
- In overriding runtime object will play the role and reference type is dummy.

Rules for Overriding:

1. In java, a method can only be written in Subclass, not in same class.
2. The argument list should be exactly the same as that of the overridden method(In overriding method names and argument types must be matched i.e., method signatures must be Same). The return type should be the same or a subtype of the return type declared in the original overridden method in the super class. **(The return types must be same in overriding upto java 1.4V but from 1.5v co-variant return types are allowed, that means child class method return type need not be same as parent class return type, its child types are also allowed.)**

co-variant return types :

In object-oriented programming, a covariant return type of a method is one that can be replaced by a "narrower" type when the method is overridden in a subclass.

child class overridden method return type need not be same as parent class return type, its child types are also allowed.

```
class Parent
{
    Public Object methodOne()
    {
        return null;
    }
}
```

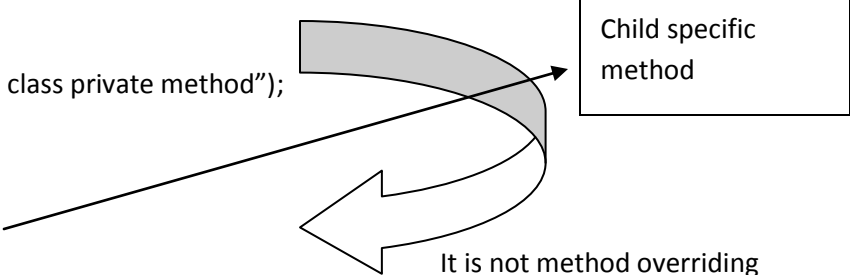
```
class Child extends Parent
{
    public String methodOne()           //String class is child class of Object class
    {
        return null;
    }
}
```

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

Important concept in context with method overriding:

1. Private methods are not visible in the Child classes hence overriding concept is not applicable on private methods applicable for private methods. Based on own requirement we can declare the same Parent class private method in Child class also, but it is not method overriding.

```
class Parent1
{
    private void methodOne()
    {
        System.out.println("Parent class private method");
    }
}
class Child1 extends Parent1
{
    private void methodOne()
    {
        System.out.println("Child class private method");
    }
}
```



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

```
class Parent2
{
    public final void methodOne() {}
}
class Child2 extends Parent2
{
    public void methodOne() {}
}
```

Output: Compile time error.

methodOne() in Child can not override methodOne() in Parent; overridden method is final.

3. Parent class non final methods we can override as final in child class.

```
class Parent3
{
    public void methodOne() {}
}
class Child3 extends Parent3
{
    public final void methodOne() {}
}
```

4. We can override native methods in the child classes.

```
class Parent4
{
    public native void methodOne();
}
class Child4 extends Parent4
{
    public void methodOne() {}
}
```

5. Parent class non native methods, we can override as native in child class.

```
class Parent5
{
    public void methodOne() { };
}
class Child5 extends Parent5
{
    public native void methodOne();
}
```

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

```

class Parent6
{
    public abstract void methodOne() ;
}
class Child6 extends Parent6
{
    public void methodOne()
    {
    }
}
  
```

7. We can override Parent class non abstract method as abstract to stop availability of Parent class method implementation to the Child classes.

```

class Parent7
{
    public void methodOne()
    {
    }
}
abstract class Child7 extends Parent7
{
    public abstract void methodOne() ;
}
  
```

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

Parent class method non access modifier	final	native	abstract	synchronized	strictfp
	x ↓	✓ ↓	✓ ↓	✓ ↓	✓ ↓
Child class Overridden method non access Modifier	non final	non native	non abstract	non synchronized	non strictfp

Parent class method non access modifier	non final	non native	non abstract	non synchronized	non strictfp
	✓ ↓	✓ ↓	✓ ↓	✓ ↓	✓ ↓
Child class Overridden method non access Modifier	final	native	abstract	synchronized	strictfp

9. static methods of parent class is not available for method overriding.

10. while overriding we can't reduce the scope of access modifier(default/no-modifier, public, protected)

Parent class method access modifier	public	protected	<default>	private
	√ ↓	√ ↓	√ ↓	↓
Child class Overridden method access Modifier	public	protected /public	<default>/ protected /public	Overriding concept not applicable