Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

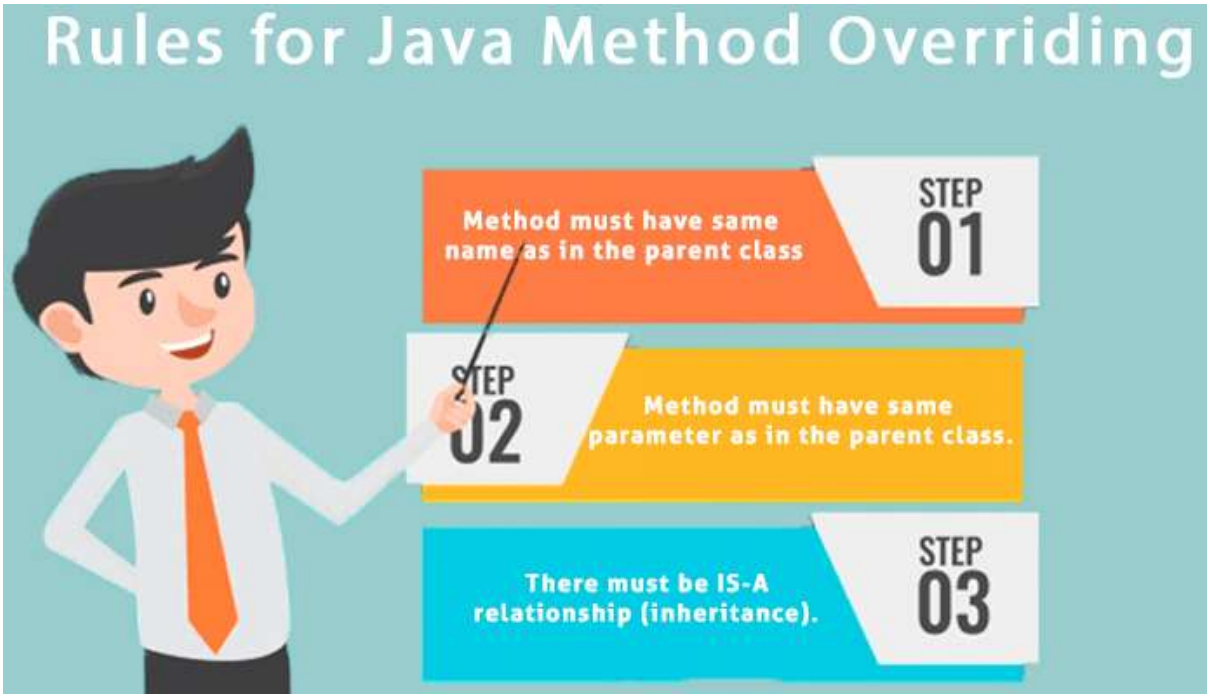
In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).



Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1. /Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. class Vehicle{
6.   void run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. class Bike extends Vehicle{
10.  public static void main(String args[]){
11.  //creating an instance of child class
12.  Bike obj = new Bike();
13.  //calling the method with child class instance
14.  obj.run();
15.  }
16. }
```

Output:

```
Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. class Vehicle{
4.   //defining a method
5.   void run(){System.out.println("Vehicle is running");}
6. }
7. //Creating a child class
8. class Bike2 extends Vehicle{
9.   //defining the same method as in the parent class
10.  void run(){System.out.println("Bike is running safely");}
```

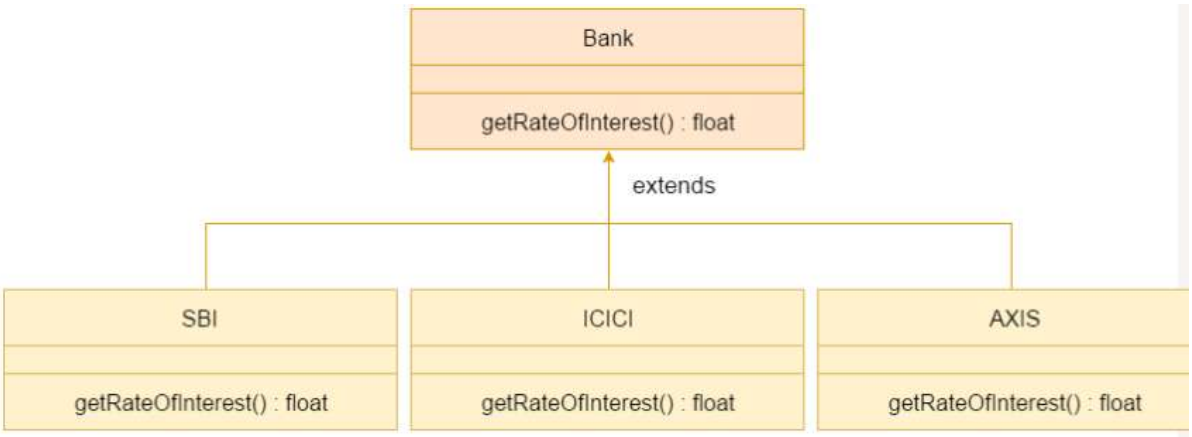
```
11.
12.  public static void main(String args[]){
13.  Bike2 obj = new Bike2();//creating object
14.  obj.run();//calling method
15.  }
16. }
```

Output:

```
Bike is running safely
```

A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



> method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

```
1.  //Java Program to demonstrate the real scenario of Java Method Overriding
2.  //where three classes are overriding the method of a parent class.
3.  //Creating a parent class.
4.  class Bank{
5.  int getRateOfInterest(){return 0;}
6.  }
7.  //Creating child classes.
8.  class SBI extends Bank{
9.  int getRateOfInterest(){return 8;}
10. }
11.
12. class ICICI extends Bank{
13. int getRateOfInterest(){return 7;}
```

```
14. }
15. class AXIS extends Bank{
16. int getRateOfInterest(){return 9;}
17. }
18. //Test class to create objects and call the methods
19. class Test2{
20. public static void main(String args[]){
21. SBI s=new SBI();
22. ICICI i=new ICICI();
23. AXIS a=new AXIS();
24. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
25. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
26. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
27. }
28. }
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Now considering the case of static methods, then static methods have following rules in terms of overloading and overriding.

- Can be overloaded by another static method.
- Cannot be overridden by another static method in sub-class. The reason behind this is that sub-class only hides the static method but not overrides it.

Following example demonstrates the same.

```
class SuperClass {
   public static void display() {
      System.out.println("SuperClass.display()");
   }

   //Method overloading of static method
   public static void display(int a) {
      System.out.println("SuperClass.display(int): " + a);
   }
```

```
}


class SubClass extends SuperClass {
   //Not method overriding but hiding
   public static void display() {
      System.out.println("SubClass.display()");
   }
}


public class Tester {
   public static void main(String[] args) {
      SuperClass object = new SubClass();

      //SuperClass display method is called
      //although object is of SubClass.
      object.display();
      object.display(1);
   }
}
```

Output

```
SuperClass.display()
SuperClass.display(int): 1
```

Notes

- The static method is resolved at compile time cannot be overridden by a subclass. An instance method is resolved at runtime can be overridden.
- A static method can be overloaded.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

Difference between method Overloading and Method Overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example

```
1. class OverloadingExample{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
```

Java Method Overriding example

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. }
```

Types of polymorphism in java- Runtime and Compile time polymorphism

In this guide we will see **types of polymorphism**. There are two types of polymorphism in java:
1) **Static Polymorphism** also known as compile time polymorphism
2) **Dynamic Polymorphism** also known as runtime polymorphism

Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compile time is known as static polymorphism. Method overloading is an example of compile time polymorphism.
Method Overloading: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

Example of static Polymorphism

Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

```
class SimpleCalculator
{
    int add(int a, int b)
    {
         return a+b;
    }
    int  add(int a, int b, int c)
    {
         return a+b+c;
    }
```

```
}
public class Demo
{
   public static void main(String args[])
   {
            SimpleCalculator obj = new SimpleCalculator();
       System.out.println(obj.add(10, 20));
       System.out.println(obj.add(10, 20, 30));
   }
}
```

Output:

```
30
60
```

Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, that is why it is called runtime polymorphism.

Example
In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

To understand the concept of overriding, you should have the basic knowledge of inheritance in Java.

```
class ABC{
   public void myMethod(){
        System.out.println("Overridden Method");
   }
}
public class XYZ extends ABC{

   public void myMethod(){
        System.out.println("Overriding Method");
   }
   public static void main(String args[]){
        ABC obj = new XYZ(); //liskov substitution principle
        obj.myMethod();
   }
}
```

Output:

```
Overriding Method
```

When an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time.
Since both the classes, child class and parent class have the same method. Which version of the method(child class or parent class) will be called is determined at runtime by JVM.

Few more overriding examples:

```
ABC obj = new ABC();
obj.myMethod();
// This would call the myMethod() of parent class ABC

XYZ obj = new XYZ();
obj.myMethod();
// This would call the myMethod() of child class XYZ

ABC obj = new XYZ();
obj.myMethod();
// This would call the myMethod() of child class XYZ
```

In the third case the method of child class is to be executed because which method is to be executed is determined by the type of object and since the object belongs to the child class, the child class version of myMethod() is called.

super keyword in java

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(super.color);//prints color of Animal class
9. }
10. }
11. class TestSuper1{
12. public static void main(String args[]){
13. Dog d=new Dog();
14. d.printColor();
15. }}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10. }
11. }
12. class TestSuper2{
13. public static void main(String args[]){
14. Dog d=new Dog();
15. d.work();
16. }}
```

Output:

```
eating...
barking...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

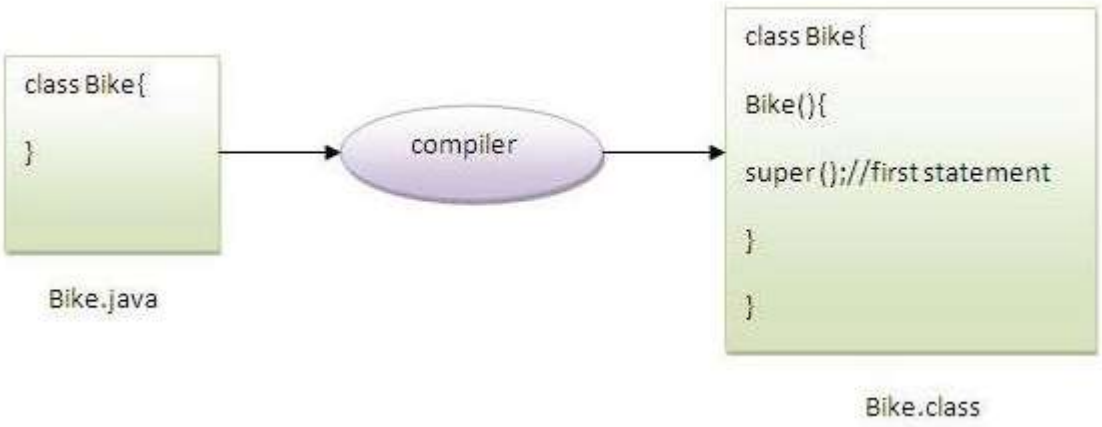
```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
```

```
6. super();
7. System.out.println("dog is created");
8. }
9. }
10. class TestSuper3{
11. public static void main(String args[]){
12. Dog d=new Dog();
13. }}
```

Output:

```
animal is created
dog is created
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. System.out.println("dog is created");
7. }
8. }
```

```
9. class TestSuper4{
10. public static void main(String args[]){
11. Dog d=new Dog();
12. }}
```

Output:

```
animal is created
dog is created
```

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
1. class Person{
2. int id;
3. String name;
4. Person(int id,String name){
5. this.id=id;
6. this.name=name;
7. }
8. }
9. class Emp extends Person{
10. float salary;
11. Emp(int id,String name,float salary){
12. super(id,name);//reusing parent constructor
13. this.salary=salary;
14. }
15. void display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. class TestSuper5{
18. public static void main(String[] args){
19. Emp e1=new Emp(1,"ankit",45000f);
20. e1.display();
21. }}
```

Output:

```
1 ankit 45000
```