Menu

Method Overriding in Java | Example Program

Method overriding in Java means redefining a method in a subclass to replace the functionality of superclass method.

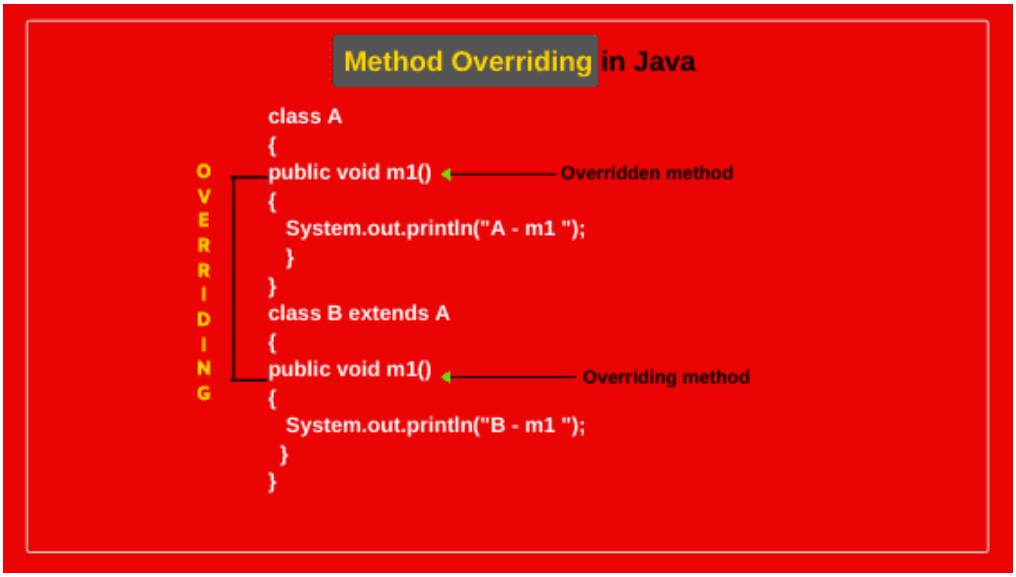
When the method of superclass is overridden in the subclass to provide more specific implementation, it is called method overriding.

In other words, when the superclass method is available to subclass by default through inheritance and subclass does not satisfy with superclass implementation, then the subclass is allowed to redefine that method on its requirement. This feature is called method overriding in Java.

By default, a subclass inherits methods from the superclass via inheritance. If you are not satisfied with the implementation (or functionality or behavior) of an inherited method, you do not need to modify that method in superclass.

Because changing the coding inside method is not a good idea. You should extend that class and override it by specifying a new implementation in the subclass.

To override a method in a subclass, the method must be defined in the subclass using the same signature and same return type as in its superclass as shown in the below figure.



The superclass method which is overridden is called **overridden method**.

The subclass method which is overriding the superclass method is called **overriding method in java**.

When to need Method overriding?

Let's take a simple example program to understand the need for method overriding in Java.

Assume that a young couple wants to marry. They have fixed the date of engagement and marriage. Let's write code for it.

Program source code 1:

```
public class Marry
{
  void engagementDate()
  {
    System.out.println("Engagement will be done on 23 Dec.");
  }
// Overridden method.
   void marryDate()
   {
      System.out.println("Marry will be on 25 Dec");
   }
}
```

But due to Christmas day on 25 Dec, the young couple wants to change date of marrying. So, what will you do?

You will open class and change the date of marrying. It is the worst practice in Java because as per the object-oriented programming concept, the best practice is that class should not open for modification.

So, If you want to add new functionality to the existing class or if you want to modify the existing functionality of the class, you should not disturb the existing class.

You should always write subclass of existing class and add new functionality in subclass like this:

```
public class Change extends Marry
{
 // Overriding method.
   void marrydate()
   {
     System.out.println("Marry will be on 27 Dec");
    }
 }
public class MyClass
{
  public static void main(String[] args)
  {
     Change obj = new Change();
      obj.engagementDate();
      obj.marrydate();
   }
 }
```

```
Output:
    Engagement will be done on 23 Dec.
    Marry will be on 27 Dec
```

From the above program, it is clear that subclass Change is implementing marryDate() method with the same signature as in the superclass marryDate() method. The class Change is overriding marryDate() method of class Marry.

Hope that you will have understood the need or purpose of using method overriding in java with the help of this example program.

Why do we need to create Subclass in Java?

There are mainly three reasons for which we need to create subclass of superclass in Java. They are as follows:

1. To add a new feature or properties. For example, a student has properties like age and location.

But in the future, if we get a new requirement to add one more property “address” for that student, we should make a subclass of that class and add a new property address in the subclass.

2. To override or change the existing functionality of superclass method.

3. To inherit the existing functionality of superclass method.

Features of Method overriding

There are the following features of method overriding in Java. They are as follows:

- Method overriding technique supports the runtime polymorphism.
- It allows a subclass to provide its own implementation of the method which is already provided by the superclass.
- Only the instance method can be overridden in Java.
- An instance variable can never be overridden in Java.
- The overriding method can not be more restrictive access modifiers than overridden method of the superclass.
- Overriding concept is not applicable for private, final, static, and main method in Java.
- From Java 5 onwards, method overriding can also be done by changing the covariant return type only.
- Overriding method cannot throw any checked exception.

Method Overriding rules in Java

When you are overriding superclass method in a subclass, you need to follow certain rules. They are as follows.

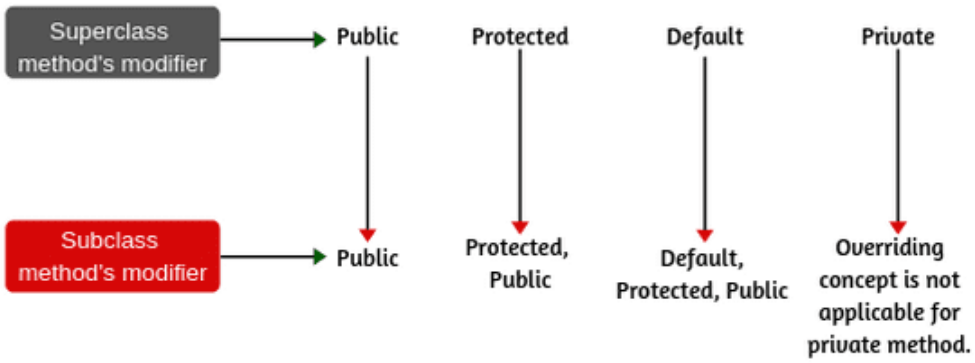
1. Subclass method name must be the same as superclass method name.

2. The parameters of subclass method must be the same as superclass method parameters. i.e. In overriding, method name and argument types must be matched. In other words, the method signature must be the same or matched.

3. Must be Is-A relationship (Inheritance).

4. Subclass method's return type must be the same as superclass method return type. But this rule is applicable until Java 1.4 version only. From Java 1.5 version onwards, covariant return types are also allowed.

5. Subclass method's access modifier must be the same or less than the superclass method access modifier. Look at the below figure to understand better.



The access modifier of the overriding method cannot be more restrictive than overridden method of superclass.

Private > Default > Protected > Public
More restrictive → Less restrictive

Fig: Applicable access modifiers to the overriding method

6. Overriding method cannot throw new or broader checked exceptions. But it can throw fewer or narrower checked exceptions or any unchecked exception.

Key Points:

1. While the overriding method, we can increase the visibility of the overriding method but cannot reduce it. For example, if superclass method is protected, we can override as a public method in the subclass.

2. Similarly, the default method of superclass can be overridden by default, protected, or public.

3. We cannot override a method if we do not inherit it. A private method cannot be overridden because it cannot be inherited in the subclass.

What is @Override annotation in Java?

@Override is an annotation that was introduced in Java 5. It is used by the compiler to check all the rules of overriding. If we do not write this annotation, the compiler will apply only three overriding rules such as

- Superclass and subclass relation.
- Method name same.
- Parameters are the same.

But if we have written as @Override on subclass method, the compiler will apply all rules irrespective of the above three points.

Let's understand it better with the help of examples.

a. Without annotation

```
In superclass
private void msg()
{
  System.out.println("Hello");
}
In subclass
private void msg()
{
  System.out.println("Hi")
}
```

Compile-time: OK

Runtime: OK

b. With annotation

```
In superclass
private void msg()
{
  System.out.println("Hello");
}
In subclass
@Override
private void msg()
{
```

```
    System.out.println("Hi");
}
```

Compile-time: Error- private method cannot be overridden.
Runtime: OK

When to use @Override Annotation in Java?

@Override annotation is used just for readability and understanding. This means that if a developer left a company and a new developer joins the company, he/she will easily understand that this method is overridden.

So, he can modify it easily in the future. At least he/she will understand that there is some relationship between the two methods.

Who (Java compiler or JVM) decide which method is to be executed?

In method overriding, Java compiler does not decide which method is to be executed. Because it has to wait till an object of subclass is created. After the creation of subclass object, JVM binds the method call to suitable method.

But the method in superclass and subclasses have the same signatures. Then how JVM decides which method is called by the programmer?

How JVM decides which method is to be called?

In method overriding, JVM decides method call depending on the runtime object of the class. That is, method resolution is always resolved by JVM based on the runtime object. During runtime, the actual object is used for calling the method.

In other words, you always check the reference variable is pointing to the object of which class?

Java Method Overriding Example Program

Now, let's take some various example programs based on java method overriding

Program source code 2:

```
package overridingProgram;
public class A
{
 void m1()
 {
  System.out.println("A-m1");
 }
// Overridden method.
  void m2()
  {
   System.out.println("A-m2");
  }
 }
public class B extends A
{
 // Overriding method.
   void m2()
   {
     System.out.println("B-m2");
   }
 // Newly defined method in class B.
   void m3()
   {
     System.out.println("B-m3");
   }
}
 public class MyTest
 {
 public static void main(String[] args)
 {
  A a = new A();
   a.m1();
   a.m2();
  B b = new B();
   b.m1();
   b.m2();
   b.m3();
  A a1 = new B();
   a1.m1();
```

```
    a1.m2();
  }
}
```

```
Output:
        A-m1
        A-m2
        A-m1
        B-m2
        B-m3
        A-m1
        B-m2
```

Explanation:

1. a.m1() will call m1() method of class A because the reference variable 'a' is pointing to the object of class A. Similarly, a.m2() will also call m2() method of class A.

2. b.m1() will call m1() method of class B because m1() of class A is available by default to class B due to inheritance and the reference variable 'b' is pointing to the objects of class B.

Similarly, b.m2() will call m2() method of class B because 'b' is pointing to the objects of the class. b.m3() will call the method of class B.

3. A a = new B() tells that the reference variable 'a1' of class A is pointing to the objects of class B. That is, the parent reference variable can hold child class object. a1.m1() will call the m1() method of class B because it is available by default.

a1.m2() will call m2() method of class B because the reference variable a1 is pointing to objects of class B.

Role of Java Compiler & JVM in Overriding

Role of Java compiler in overriding:

During the compile-time in the above program (A a1), a1 is a reference variable of parent class A. i.e. a1 is of parent type. So, java compiler will be gone to check that m2() method is available or not.

If it is available in the parent class, no problem. The code will be happily compiled.

Role of JVM in overriding:

At runtime, JVM will check that the reference variable is pointing to the which class object? parent class object or child class object.

If it is pointing to the parent class object and m2() method is available in the parent class, it will execute the m2() method of the parent class.

But if it is pointing to the child class object, JVM will immediately check that in the child class, the m2 method is overriding or not.

If it is not overriding in the child class, JVM will call the default parent m2 method available in the child class.

Thus, JVM will execute the parent method only. If the m2 is overriding in the child class, at the runtime, JVM will execute the child class method based on the runtime object (new B()).

In the overriding method, the resolution is always based on the runtime object by JVM.

Let's understand some more example programs based on method overriding concepts.

Program source code 3:

```
package overridingProgram;
public class College
{
 public void collegeName()
 {
   System.out.println("Name of my college is PIET ");
 }
```

```
// Overridden method.
   void estYear()
   {
      System.out.println("It was established in 1999");
   }
 }
public class MyCollege extends College
{
// Here, we are increasing visibility the overriding method.
   protected void estYear()
   {
     System.out.println("It was established in 2001");
   }
 }
public class Test
{
  public static void main(String[] args)
  {
    MyCollege mc = new MyCollege();
     mc.collegeName();
     mc.estYear();

   College c = new MyCollege();
    c.estYear();
   }
 }
```

```
Output:
       Name of my college is PIET
       It was established in 2001
       It was established in 2001
```

Program source code 4:

```
package overridingProgram;
public class X
{
  public void m1()
```

```
  {
    System.out.println("m1-X");
  }
}
public class Y extends X
{
// Here, we are reducing the visibility of overriding method. So, it will generate compile time error.
   void m1() // Cannot reduce the visibility of inherited method from X.
   {
     System.out.println("m1-Y");
   }
 }
public class XY
{
  public static void main(String[] args)
  {
    Y y = new Y();
     y.m1();
   }
 }
```

```
Output:
      Error: Unresolved compilation problem: Cannot reduce the visibility of the inherited method from X
```

Why Method overriding is called Runtime polymorphism?

Let's consider program source code 2.

1. We have class A with m2(): "A-m2" prints.
2. We have subclass B with m2(): "B-m2" prints.
3. Client wrote the main method and in the main,

    a. A a1 = new B();

    b. a1.m2();

    c. The above line is compiled as class A has m2().

    d. At runtime, we will get the output of m1 which is there at class B.

e. As at compile time, we think m2 of class A will get called but at runtime m2 of class B got executed because, in method overriding, method resolution is always based on runtime object (new B()).

Therefore, it is also called runtime polymorphism or dynamic polymorphism, or late binding in java.

Why cannot private method be overridden?

We cannot override private method because when superclass method is private, it will not be visible to the subclass. Whatever methods we writing in the subclass will be treated as a new method but not an overridden method.

Let's take an example program where we will declare overridden method as private in superclass and see that it is visible in subclass or not.

Program source code 5:

```
package overridingProgram;
public class X
{
  private void m1()
  {
    System.out.println("m1-X");
  }
 }
public class Y extends X
{

 }
public class XY
{
 public static void main(String[] args)
 {
    Y y = new Y();
     y.m1(); // Compile time error because method m1() from type X is not visible in class Y.
  }
}
```

Can we override static method in Java?

No, we cannot override a static method. We cannot also override a static method with an instance method because static method is bound with class whereas an instance method is bound with an object.

For more detail, go to this tutorial: Can we override static method in Java in 3 cases?

Program source code 6:

```
package overridingProgram;
public class X
{
  static void m1()
  {
    System.out.println("m1-X");
  }
}
public class Y extends X
{
 @Override
 static void m1() // Compile time error because the method m1() of type Y must override or implement a supertype method.
 {
   System.out.println("m1-Y");
 }
}
public class XY
{
  public static void main(String[] args)
  {
     Y y = new Y();
      y.m1();
   }
}
```

Can we stop method overriding in Java?

Yes, we can stop method overriding by declaring method as final.

Let's make a program where we will declare the overriding method as final in the subclass and will try to override method in sub – sub class.

Program source code 7:

```
package overridingProgram;
public class X
{
// Overridden method.
  void m1()
  {
    System.out.println("m1-X");
  }
}
public class Y extends X
{
  @Override
 final protected void m1() // By declaring method as final, we can stop overriding.
 {
    System.out.println("m1-Y");
  }
 }
public class XY extends Y
{
   @Override
  public void m1() // Cannot override the final method from Y.
  {
     System.out.println("m1-XY");
  }
public static void main(String[] args)
{
  Y y = new Y();
   y.m1();
  }
}
```

Use of Method overriding in Java

1. Method overriding is used to achieve runtime polymorphism in Java.
2. It is used to change the existing functionality of the superclass method.

Note: Method overriding occurs only when the signatures of the super and subclasses methods are identical. If they are not, then both methods are simply overloaded methods.

Hope that this tutorial has covered almost all important concepts of **method overriding in Java** with example programs. I hope that you will have understood method overriding rules, features, and uses.
Thanks for reading!!!

> ***Next ⇒ Covariant return type in Java***

⇐ Prev Next ⇒

Core Java

Method Overriding Rules in Java, What is @Override Annotation in Java, What is Method Overriding in Java

Leave a Comment

You must be logged in to post a comment.

Java Tutorials