# Heading 1

# **Polymorphism in Java**

**Polymorphism in Java** is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism, also called method hiding. Here, we will focus on runtime polymorphism in java.

## **Runtime Polymorphism in Java**

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

### **Upcasting**

If the reference variable of Parent class refers to the object of Child class, it is known as Upcasting. For example:



1. **class** A{}
2. **class** B **extends** A{}
3. A a=**new** B();//upcasting

For upcasting, we can use the reference variable of class type or an interface type. For Example:

1. **interface** I{}
2. **class** A{}
3. **class** B **extends** A **implements** I{}

Here, the relationship of B class would be:

B IS-A A

B IS-A I

B IS-A Object

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

## **Java Runtime Polymorphism Example: Shape**

1. **class** Shape{
2. **void** draw(){System.out.println("drawing...");}  //non static method resolve on the basic of obj type.
3. }
4. **class** Rectangle **extends** Shape{
5. **void** draw(){System.out.println("drawing rectangle...");}
6. }
7. **class** Circle **extends** Shape{
8. **void** draw(){System.out.println("drawing circle...");}
9. }
10. **class** Triangle **extends** Shape{
11. **void** draw(){System.out.println("drawing triangle...");}
12. }
13. **class** TestPolymorphism2{
14. **public** **static** **void** main(String args[]){
15. Shape s;
16. s=**new** Rectangle();  //upcasting.
17. s.draw();
18. s=**new** Circle();
19. s.draw();
20. s=**new** Triangle();
21. s.draw();
22. }
23. }
24. **class** Bank{
25. **float** getRateOfInterest(){**return** 0;}
26. }
27. **class** SBI **extends** Bank{
28. **float** getRateOfInterest(){**return** 8.4f;}
29. }
30. **class** ICICI **extends** Bank{
31. **float** getRateOfInterest(){**return** 7.3f;}
32. }
33. **class** AXIS **extends** Bank{
34. **float** getRateOfInterest(){**return** 9.7f;}
35. }
36. **class** TestPolymorphism{
37. **public** **static** **void** main(String args[]){
38. Bank b;
39. b=**new** SBI();
40. System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
41. b=**new** ICICI();
42. System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
43. b=**new** AXIS();
44. System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
45. }
46. }

**static varible, static method, non static varible are resolve basic of reference type. non static mehtod are resolve on the bASIC OF OBJECT TYPE.**

  class A

{

void m1() //non static method resolve on the bascis of obj type.

{

System.out.println("Inside A's m1 method");

}

}

class B extends A

{

// overriding m1()

void m1()

{

System.out.println("Inside B's m1 method");

}

}

// Driver class

class Dispatch

{

public static void main(String args[])

{

// object of type A

A a = new A();

// object of type B

B b = new B();

// obtain a reference of type A

**A ref;**

// ref refers to an A object

ref = a;

// calling A's version of m1()

ref.m1(); m1 is NSM invoked based on object type.

// now ref refers to a B object

ref = b;

// calling B's version of m1()

ref.m1(); m1 is NSM invoked based on object type.

}

}

Inside A's m1 method

Inside B's m1 method

class A

{

static void m1() resolve on the basic of reference type.

{

System.out.println("Inside A's m1 method");

}

}

class B extends A

{

// overriding m1()

static void m1()

{

System.out.println("Inside B's m1 method");

}

}

Inside A's m1 method

Inside A's m1 method

// now ref refers to a B object

B rr; rr = b;

// calling B's version of m1()

rr.m1();

Inside A's m1 method

Inside B's m1 method

[**Static vs Dynamic binding**](https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/)

* Static binding is done during compile-time while dynamic binding is done during run-time.
* private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

# **Covariant Return Type**

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

#### **Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.**

### **Simple example of Covariant Return Type**

1. **class** A{
2. A get(){**return** **this**;}
3. }
5. **class** B1 **extends** A{
6. B1 get(){**return** **this**;}
7. **void** message(){System.out.println("welcome to covariant return type");}
9. **public** **static** **void** main(String args[]){
10. **new** B1().get().message();
11. }
12. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=B1)

Output:welcome to covariant return type

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

### **How is Covariant return types implemented?**

Java doesn't allow the return type based overloading but JVM always allows return type based overloading. JVM uses full signature of a method for lookup/resolution. Full signature means it includes return type in addition to argument types. i.e., a class can have two or more methods differing only by return type. javac uses this fact to implement covariant return types.