



OBJECT ORIENTED CONCEPT IN JAVA - POLYMORPHISM

Lecture Notes

A COMPLETE GUIDE FOR
BEGINNERS

Softech Solutions Inc.

www.softechnosolutionsgroup.com

saney.alam@softechnosolutionsgroup.com

Object Oriented Concept in Java - Polymorphism

<i>Table of Contents</i>	<i>Page</i>
<i>POLYMORPHISM</i>	<i>02</i>
<i>VIRTUAL METHODS</i>	<i>03</i>

POLYMORPHISM

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one **IS-A** test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the **IS-A** test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

```
public interface Vegetarian{}  
public class Animal{}  
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritances. Following are true for the above examples –

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal –

Example

```
Deer d = new Deer();  
Animal a = d;  
Vegetarian v = d;  
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

VIRTUAL METHODS

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

Example

```
/* File name : Employee.java */  
public class Employee {  
    private String name;  
    private String address;  
    private int number;  
  
    public Employee(String name, String address, int number) {  
        System.out.println("Constructing an Employee");  
        this.name = name;  
        this.address = address;  
        this.number = number;  
    }  
  
    public void mailCheck() {  
        System.out.println("Mailing a check to " + this.name + " " +  
this.address);  
    }  
  
    public String toString() {  
        return name + " " + address + " " + number;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getAddress() {
```

```
        return address;
    }

    public void setAddress(String newAddress) {
        address = newAddress;
    }

    public int getNumber() {
        return number;
    }
}
```

Now suppose we extend Employee class as follows –

```
/* File name : Salary.java */
public class Salary extends Employee {
    private double salary; // Annual salary

    public Salary(String name, String address, int number, double salary) {
        super(name, address, number);
        setSalary(salary);
    }

    public void mailCheck() {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName()
            + " with salary " + salary);
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double newSalary) {
        if(newSalary >= 0.0) {
            salary = newSalary;
        }
    }

    public double computePay() {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}
```

Now, you study the following program carefully and try to determine its output –

```
/* File name : VirtualDemo.java */
public class VirtualDemo {

    public static void main(String [] args) {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
    }
}
```

```
System.out.println("Call mailCheck using Salary reference --");
s.mailCheck();
System.out.println("\n Call mailCheck using Employee reference--");
e.mailCheck();
}
}
```

This will produce the following result –

Output

```
Constructing an Employee
Constructing an Employee

Call mailCheck using Salary reference --
Within mailCheck of Salary class
ailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
ailing check to John Adams with salary 2400.0
```

Here, we instantiate two Salary objects. One using a Salary reference s, and the other using an Employee reference e.

While invoking s.mailCheck(), the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

mailCheck() one is quite different because e is an Employee reference. When the compiler sees e.mailCheck(), the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.