

POLYMORPHISM

16

Polymorphism came from the two Greek words 'poly' meaning many and morphos meaning forms. The ability to exist in different forms is called 'polymorphism'. In Java, a variable, an object or a method can exist in different forms, thus performing various tasks depending on the context. Because same variable or method can perform different tasks, the programmer has the advantage of writing flexible code.

Polymorphism with Variables

When using variables, some times inherently the data type of the result is decided by the compiler and accordingly execution proceeds. For example, in the statement:

```
System.out.println(a+b);
```

Java compiler decides the data type of the result of the expression $a+b$ depending on the data types of a and b . If a and b are int type, then $a+b$ will also be taken as int type. If a and b are float type variables, then $a+b$ will be taken as float type. If a is int and b is float, then the compiler converts a also into float and then sum is found. Thus, the result $a+b$ is exhibiting polymorphic nature. It may exist as an int or as a float or as some other data type depending on the context. This is also called 'Coercion'.

Important Interview Question

What is coercion?

Coercion is the automatic conversion between different data types done by the compiler.

Let us take a different example. See the following code:

```
float a = 15.5f;  
int x = (int)a;
```

In the second line, the actual data type of the variable is changed by using a cast operator. Even if 'a' is float type, it is converted into int type. If 'a' is taken as it is, it becomes float type and because we converted, it can take the form of an int. This means 'a' exists in two different forms. This also comes under polymorphism, which is called 'Conversion'.

Important Interview Question

What is conversion?

Conversion is an explicit change in the data type specified by the cast operator.

Polymorphism using Methods

If the same method performs different tasks, then that method is said to exhibit polymorphism. This statement is tricky. Let us stop and think again, how is it possible for a method to perform different tasks? It is possible only when the method assumes different bodies. It is something like this, take two persons with the same name 'Ravi'. Because the name is same but the persons are different, they can perform different tasks. In the same way, there may be two methods with the same name and they can perform different tasks. When we call the methods, we use same name but the task will be different depending on which method (body) is called.

Now, the crucial thing is to decide which method is called in a particular context. This decision may happen either at compile time or at runtime. This will lead to two types of polymorphism, Static polymorphism and Dynamic polymorphism. The word 'static' represents at compile time and 'dynamic' represents at run time. Let us discuss dynamic polymorphism first.

Dynamic Polymorphism

The polymorphism exhibited at runtime is called dynamic polymorphism. This means when a method is called, the method call is bound to the method body at the time of running the program dynamically. In this case, Java compiler does not know which method is called at the time of compilation. Only JVM knows at runtime which method is to be executed. Hence, this is also called 'runtime polymorphism' or 'dynamic binding'.

Let us take a class 'Sample' with two instance methods having same name as:

```
void add(int a, int b)
{
    System.out.println("Sum of two= "+ (a+b));
}

void add(int a, int b, int c)
{
    System.out.println("Sum of three= "+ (a+b+c));
}
```

The bodies of these methods are different and hence they can perform different tasks. For example, the first method adds two integer numbers and the second one adds three integer numbers. However, to call these methods, we use the same method name. Suppose, we called the method as

❑ s. add(10, 15); //s is the object of Sample class.

Now, who will decide which method is to be executed? Is it Java compiler or JVM? Because methods are called by using an object, the Java compiler can not decide at the time of compilation which method is actually called by the user. It has to wait till the object is created for Sample class. And the creation of object takes place at runtime by JVM. Now, JVM should decide which method is actually called by the user at runtime (dynamically).

The question is how JVM recognizes which method is called, when both the methods have same name. For this, JVM observes the signature of the methods. Method signature consists of a method name and its parameters. Even if, two methods have same name, their signature may vary. For example, two human beings may have same name but their signatures will differ.

Important Interview Question

What is method signature?

Method signature represents the method name along with method parameters.

When there is a difference in the method signatures, then the JVM understands both the methods are different and can call the appropriate method. The difference in the method signatures will arise because of one of the following reasons:

- ❑ There may be a difference in the number of parameters passed to the methods.

For example,

```
void add(int a, int b)
void add(int a, int b, int c)
```

In this case, if we call add(10, 15) then JVM executes the first method and if we call add(10, 15, 22) then it runs the second method.

- ❑ Or, there may be a difference in the data types of parameters.

For example,

```
void add(int a, float b)
void add(double a, double b)
```

In this case, if we call the method add(10, 5.5f) then JVM goes for the first method and if it is add(10.5, 22.9) then JVM runs the second one.

- ❑ Or, there may be a difference in the sequence (orderliness) of the parameters.

For example,

```
void add(int a, float b)
void add(float a, int b)
```

In this case, if we call add(10, 5.55) then JVM executes the first method and if it is add(5.5, 20) then it goes for the second method.

JVM matches the values passed to the method at the time of method call with the method signature and picks up the appropriate method. In this way, difference in the method signatures helps JVM to identify the correct method and execute it. Please see Program 1 to understand this.

Program 1: Write a program to create Sample class which contains two methods with the same name but with different signatures.

```
//Dynamic polymorphism
class Sample
{
    //method to add two values
    void add(int a,int b)
    {
        System.out.println("Sum of two= "+ (a+b));
    }

    //method to add three values
    void add(int a, int b, int c)
    {
        System.out.println("Sum of three= "+ (a+b+c));
    }
}

class Poly
```

```

{
    public static void main(String args[ ])
    {
        //create Sample class object
        Sample s = new Sample();

        //call add() and pass two values
        s.add(10,15); //This call is bound with first method

        //call add() and pass three values
        s.add(10,15,20); //This call is bound with second method
    }
}

```

Output:

```

C:\> javac Poly.java
C:\> java Poly
Sum of two= 25
Sum of three= 45

```

In the preceding program, JVM can understand and bound the method call with the appropriate method by observing the difference in the method signatures. Two methods with the same name but with different signatures have been written in Sample class. Such methods are called overloaded methods and this concept is referred to as method overloading.

Important Interview Question

What is method overloading?

Writing two or more methods in the same class in such a way that each method has same name but with different method signatures – is called method overloading.

In Program 1, we did method overloading using instance methods. And hence, Java compiler does not know which method is called at compilation time. But JVM knows and binds the method call with the correct method at the time of running the program. So, this is an example for dynamic polymorphism.

We understood that in method overloading JVM recognizes methods separately by observing the difference in the method signatures that arises due to the difference either in the number of parameters or in the data types of parameters or in the sequence of parameters. It is not possible if the programmer wants to write two methods in the same class without any of these differences. Doing so will come under writing duplicate methods and Java compiler will reject that code.

But it is always possible for the programmer to write two or more methods with same name and same signature in two different classes. This can be done in super and sub classes also. See Program 2. In this program, the super class 'One' has calculate() method which calculate square value. The sub class 'Two' is derived from class One. But the programmer who is creating the sub class does not want to calculate square value. His requirement is to calculate square root value. So he writes another method with the same name and same signature in the sub class but with a different body i.e., to calculate the square root value.

Observe the methods in the super and sub classes, written as:

```

void calculate(double x) //in super class One
void calculate(double x) //in sub class Two

```

These two methods have same name and same signatures and there is no difference anywhere in the parameters. When calculate() method is called by using the sub class object 't' as:

```
t.calculate(25);
```

The sub class method is only executed by the JVM, but not the super class method. In other words, the sub class calculate() method 'overrides' the super class calculate() method. This concept is also called 'method overriding'.

Important Interview Question

What is method overriding?

Writing two or more methods in super and sub classes such that the methods have same name and same signature - is called method overriding.

In method overriding, the Java compiler does not decide which method is called by the user, since it has to wait till an object to sub class is created. After creating the object, JVM has to bind the method call to an appropriate method. But the methods in super and sub classes have same name and same method signatures. Then how JVM decides which method is called?

Here, JVM calls the method depending on the class name of the object which is used to call the method. For example, we are calling the method by using the sub class object 't' as:

```
t.calculate(25);
```

So, the sub class method is only executed by the JVM. In Inheritance, we always create an object to sub class and hence only sub class method is called. In this way, method overriding using instance methods is an example for dynamic polymorphism.

In Program 2, JVM calls the sub class calculate() method as the object used to call the method is of sub class type.

Program 2: Write a program where calculate() method of super class is overridden by the calculate() method of sub class. The behavior of the calculate() method is dynamically decided.

```
//Dynamic polymorphism
class One
{
    //method to calculate square value
    void calculate(double x)
    {
        System.out.println("Square value= " + (x*x));
    }
}
class Two extends One
{
    //method to calculate square root value
    void calculate(double x)
    {
        System.out.println("Square root= " + Math.sqrt(x));
    }
}
class Poly
{
    public static void main(String args[ ])
    {
        //create sub class object t.
        Two t = new Two();

        //call calculate() method using t.
    }
}
```

```

        t.calculate(25);
    }
}

```

Output:

```

C:\> javac Poly.java
C:\> java Poly
Square root= 5.0

```

Remember, when a super class method is overridden by the sub class method, JVM calls only the sub class method and never the super class method. In other words, we can say the sub class method is replacing the super class method.

Important Interview Question

What is the difference between method overloading and method overriding?

See the table 16.1.

Table 16.1

Method Overloading	Method Overriding
Writing two or more methods with the same name but with different signatures is called method overloading.	Writing two or more methods with the same name and same signatures is called method overriding.
Method overloading is done in the same class.	Method overriding is done in super and sub classes.
In method overloading, method return type can be same or different.	In method overriding, method return type should also be same.
JVM decides which method is called depending on the difference in the method signatures.	JVM decides which method is called depending on the data type (class) of the object used to call the method.
Method overloading is done when the programmer wants to extend the already available feature.	Method overriding is done when the programmer wants to provide a different implementation (body) for the same feature.
Method overloading is code refinement. Same method is refined to perform a different task.	Method overriding is code replacement. The sub class method overrides (replaces) the super class method.

Static Polymorphism

The polymorphism exhibited at compilation time is called static polymorphism. Here the compiler knows without any ambiguity which method is called at the time of compilation. Of course, JVM executes the method later, but the compiler knows and can bind the method call with method code (body) at the time of compilation. So, it is also called 'static binding' or 'compile time polymorphism'.

In the previous section, we discussed that the method overloading and the method overriding using instance methods are examples for dynamic polymorphism. Similarly, achieving method overloading and method overriding by using static methods, private methods, and final methods are examples for static polymorphism. The reason is that all of these methods maintain only one copy in memory that is available to the objects of the class. So the Java compiler knows which method is called.

the time of compilation and it needs not wait till the objects are created. Since at the time of compilation, the method call can be bound with actual method body, this comes under static polymorphism.

Polymorphism with Static Methods

A static method is a method whose single copy in memory is shared by all the objects of the class. Static methods belong to the class rather than to the objects. So they are also called class methods. When static methods are overloaded or overridden, since they do not depend on the objects, the Java compiler need not wait till the objects are created to understand which method is called.

Let us re-write the Program 2 using static methods to understand how to override them.

Program 3: Write a program to use super class reference to call the calculate() method.

```
//Static polymorphism
class One
{
    //method to calculate square value
    static void calculate(double x)
    {
        System.out.println("Square value= "+ (x*x));
    }
}
class Two extends One
{
    //method to calculate square root value
    static void calculate(double x)
    {
        System.out.println("Square root= "+ Math.sqrt(x));
    }
}
class Poly
{
    public static void main(String args[])
    {
        //Super class reference refers to sub class object
        One o = new Two();

        //call calculate() method using super class reference
        o.calculate(25);
    }
}
```

Output:

```
C:\> javac Poly.java
C:\> java Poly
Square value= 625.0
```

In preceding program, the super class method is called. If sub class reference is used to call the calculate() method, then sub class method is called. In class Poly, we created super class reference to sub class object, as:

```
One o = new Two();
```

Here 'o' is super class reference variable. Using this variable, we are calling the calculate() method as o.calculate(25). So the super class calculate() method is executed. Suppose, we created a sub class reference as,

```
Two t = new Two();
```

and called the `calculate()` method using this sub class reference as `t.calculate(25)`, then sub class `calculate()` is executed by the JVM. The point is, when static methods are overridden the JVM decides which method is to be executed depending on the reference type used to call method.

Polymorphism with Private Methods

Private methods are the methods which are declared by using the access specifier 'private'. This access specifier makes the method not to be available outside the class. So other programs cannot access the private methods. Even private methods are not available in the sub classes. This means, there is no possibility to override the private methods of the super class in its sub class. So only method overloading is possible in case of private methods.

What happens if a method is written in the sub class with the same name as that of the private method of the super class? It is possible to do so. But in this case, the method in the super class and the method in the sub class act as different methods. The super class will get its own copy and the sub class will have its own copy. It does not come under method overriding.

Important Interview Question

Can you override private methods?

No. Private methods are not available in the sub classes, so they cannot be overridden.

The only way to call the private methods of a class is by calling them within the class. For this purpose, we should create a public method and call the private method from within it. When the public method is called, it calls the private method.

Polymorphism with Final Methods

Methods which are declared as 'final' are called final methods. Final methods cannot be overridden because they are not available to the sub classes. Therefore, only method overloading is possible with final methods.

There are two uses of declaring a method as 'final' given here:

- ❑ When a method is declared as final, the performance will be better. For example, take the following classes. In class A, we got a final method, `method1()`. This method is being overridden from class B's `method2()`:

```
class A
{
    final void method1()
    {
        System.out.println("Hello");
    }
}
class B
{
    void method2()
    {
        A.method1(); //call the final method
    }
}
```

Now, JVM physically copies the code of `method1()` into `method2()` of class B, as:

```
class B
{
    void method2()
    {
        System.out.println("Hello"); //body of final method copied
    }
}
```

Copying the code like this is called 'inline operation'. Previously, when we call `method2()`, it had to call `method1()` again. But now, in this version, we have avoided that method call. Every method call takes some time of the JVM. This overhead is avoided here, and hence the time of execution improves. In this way, final methods improve performance. Remember, JVM performs inline operation only if it feels there would be improvement in the performance. Otherwise, JVM does not perform this inline operation, even if a final method is written.

- When the programmer does not want others to override his method, he should declare his method as 'final'.

Important Interview Question

Can we take private methods and final methods as same?

Yes. The Java compiler assigns the value for the private methods at the time of compilation. Also, private methods can not be modified at run time. This is the same case with final methods also. Neither the private methods nor the final methods can be overridden. So, private methods can be taken as final methods.

Final Class

A final class is a class which is declared as final. final keyword before a class prevents inheritance. This means sub classes cannot be created to a final class. For example,

```
final class A
class B extends A //invalid
```

Important Interview Question

What is final?

'final' keyword is used in two ways:

- It is used to declare constants, as:

```
final double PI = 3.14159; //PI is constant.
```

It is used to prevent inheritance, as:

- final class A //sub class to A cannot be created.

What is the difference between dynamic polymorphism and static polymorphism?

Dynamic polymorphism is the polymorphism exhibited at runtime. Here, Java compiler does not understand which method is called at compilation time. Only JVM decides which method is called at runtime. Method overloading and method overriding using instance methods are the examples for dynamic polymorphism.

Static polymorphism is the polymorphism exhibited at compile time. Here, Java compiler knows which method is called. Method overloading and method overriding using static methods; method overriding using private or final methods are examples for static polymorphism.

Let us write a program where we take a class with the name 'Commercial' that contains code for calculating electricity bill for a commercial user. In this class, setName() method is used for storing the customer name into the instance variable. getName() method is useful to return the name. If we write a 'Domestic' class for calculating the electricity bill for a domestic user, setName() and getName() methods of the Commercial class are also needed by the 'Domestic' class.

Commercial class contains another method calculateBill() which calculates bill amount at a charge of Rs. 5.00 per unit. This method should be overridden by the Domestic class because, the charge varies for a domestic customer at Rs. 2.50 per unit.

Program 4: Let us make a program to show how to override the calculateBill() method of Commercial class inside the Domestic class.

```
//Electricity bill for commercial connection
class Commercial
{
    //take customer name
    private String name;

    //store customer name into name
    void setName(String name)
    {
        this.name = name;
    }

    //retrieve the name
    String getName()
    {
        return name;
    }

    //to calculate bill taking Rs. 5.00 per unit
    void calculateBill(int units)
    {
        System.out.println("Customer: " + getName());
        System.out.println("Bill amount= " + units*5.00);
    }
}

//Electricity bill for domestic connection
class Domestic extends Commercial
{
    //override the calculateBill() of Commercial class, to calculate
    //bill at Rs. 2.50 per unit
    void calculateBill(int units)
    {
        System.out.println("Customer: " + getName());
        System.out.println("Bill amount= " + units*2.50);
    }
}

//calculate electricity bill for commercial and domestic users
class ElectricityBill
{
    public static void main(String args[])
    {
        //call calculateBill() using the Commercial object
        Commercial c = new Commercial();
        c.setName("Raj Kumar");
        c.calculateBill(100);

        //call calculateBill() using the Domestic object
        Domestic d = new Domestic();
        d.setName("Vijaya Laxmi");
    }
}
```

```
        d.calculateBill(100);  
    }  
}
```

Output:

```
C:\> javac ElectricityBill.java  
C:\> java ElectricityBill  
Customer: Raj Kumar  
Bill amount= 500.0  
Customer: Vijaya Laxmi  
Bill amount= 250.0
```

Conclusion

If the same variable or method or an object performs different tasks, it is said to exhibit polymorphism. In this chapter, we have seen how variables and methods can exhibit polymorphism. A programmer goes for method overloading when he wants to extend the available features of the class. A programmer does method overriding when he wants to perform a different task with the same method. Even the objects can also change their form at runtime and can perform different tasks depending on how they are referenced. This is possible by casting the object types. This concept is discussed in the next chapter.