Java – OOPS Concepts

Polymorphism

Now that we have established the basics of Inheritance in Java, the whole wide world of Polymorphism is at our helm for exploring. Let's get to it…

The Merriam Webster dictionary defines polymorphism as –

- the quality or state of existing in or assuming different forms: such as
  - : existence of a species in several forms independent of the variations of sex
  - : existence of a gene in several allelic forms
  - also : a variation in a specific DNA sequence
  - : existence of a molecule (such as an enzyme) in several forms in a single species
- the property of crystallizing in two or more forms with distinct structure

Put simply, polymorphism allows one entity to exist in multiple forms. Java Polymorphism is no different. Let's continue with the same Parent-Child example from the Inheritance discussion and enhance both classes to include a definition of whoami method –

Parent.java

```
package io.budbak.oops.polymorphism;

public class Parent {

    private String name = "Parent";

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String whoami() {
        return this.getName();
    }
}
```

Child.java

```
package io.budbak.oops.polymorphism;

public class Child extends Parent {

    private String name = "Child";

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String whoami() {
        return "Mr./Ms." + this.getName();
    }

}
```

Both classes now provide their own version of whoami method – Parent prints the name attribute directly while Child decorates the name attribute with a generic salutation.

Since Child extends Parent, whoami method exists in both the forms for the Child entity – thus implementing polymorphism. But, question is – which method will get invoked if we call whoami?

Answer lies in the contracts of polymorphism – as outlined below –

- Child class is said to be overriding a method from parent class if it defines a method already defined in the parent class with the same method signature i.e. same name, arguments and return type (covariant returns supported since Java 5).
- The overriding method's visibility cannot be broader than overridden method's visibility.
- E.g. If Parent class method is public, Child class method cannot be private, protected or default.

- The overriding method's return type cannot be super than overridden method's return type.
- E.g. If Parent class returns a Parent, Child class cannot return Object (super type of Parent) but can return Child (as Child extends Parent)
- The overriding method cannot throw an exception higher than overridden method's exception.
- If the Parent class method were to throw an Exception e.g. public String whoami() throws Exception, then the Child class method cannot throw Throwable/Error but can throw any sub-class of Exception e.g. public String whoami() throws RuntimeException/NumericException

This type of polymorphism is defined as Runtime Polymorphism because which method to invoke is decided on runtime depending on the type of object on which the method is invoked. Let's see a demo –

Case I:

- Reference – Parent
- Object – Parent
- Result – Parent

```
public class Main {

    public static void main(String[] args) {

        Parent parent = new Parent();
        System.out.println(parent.whoami());

    }

}
```

Output –

```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (May 20, 2019 1:13:13 PM)
Parent
```

Why Parent – This is a straight-forward case. You only have one class, one type, one object and you are invoking method on it. No doubts here, Parent class' whoami method will be invoked.

Case II:

- Reference – Child
- Object – Child
- Result – Mr./Ms.Child

```
public class Main {

    public static void main(String[] args) {

        Child child = new Child();
        System.out.println(child.whoami());

    }

}
```

Output –

```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (May 20, 2019 1:19:27 PM)
Mr./Ms.Child
```

This is another straight-forward case. You only have one class, one type, one object and you are invoking method on it. No doubts here, Child class' whoami method will be invoked.

So far we have not leveraged the power of polymorphism. Let's do that now with Case III.

Case III:

- Reference – Parent
- Object – Child
- Output – Mr./Ms.Child

```
public class Main {

    public static void main(String[] args) {

        Parent object = new Child();
        System.out.println(object.whoami());

    }

}
```

Output –

```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (May 20, 2019 1:21:07 PM)
Mr./Ms.Child
```

Java supports two types of Polymorphism – Runtime (via Method Overriding) and Compile-time (via Method Overloading). We will cover method overloading later in this article. Let's focus on Runtime Polymorphism for now.

When we initialize a Child object and make a Parent type reference point to it and call the whoami method, the following happens –

Compile time activities –

1. The Java compiler checks whether there is a relationship between the two types being linked – Parent and Child. Since in our case, Child extends Parent, we are good to go to the next step.

2. The Java compiler then checks whether there is a "whoami" method definition available in the Parent class. If yes, it gives the go ahead.

    a. If Parent class did not have a method definition for "whoami", the Java compiler would scream at the top of its voice –

```
public class Main {

    public static void main(String[] args) {

        Parent object = new Child();
        System.out.println(object.whoami());

    }

}
```

The method whoami() is undefined for the type Parent

2 quick fixes available:

- Create method 'whoami()' in type 'Parent'
- Add cast to 'object'

Press 'F2' for focus

Runtime –

1. The method invocation by Java Runtime Environment is done based on the type of object on which the method is invoked. In our example, since the object is of type Child, the Child class' whoami method is invoked.
    a. If Child class did not have a method definition for "whoami", the parent class' whoami method is invoked. However, since the method is using this.name attribute, and this is pointing to the Child class, the output is –

```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (May 20, 2019 1:58:24 PM)
Child
```

As you can see, polymorphism is an important tool in every Java Developer's toolkit and the entire code universe is overloaded with the examples of polymorphism. Even within Java, you will find examples of polymorphism e.g. Collection classes, Exceptions, etc.

However, if you really do not want the child classes to override your method, you can mark the method(s) in parent class as "final". That way, the child classes will not be able to override your method(s) and they will get your version of the method functionality. Period.

Ah, we almost forgot – we are yet to discuss overloading. Method Overloading is referred to as "compile-time polymorphism" however it is almost completely different from the usual Polymorphism activities which mandate an inheritance relationship between the participating classes. Inheritance is optional in case of Method Overloading.

Method Overloading means, a class can have two methods with same name, same or different return type but necessarily different method parameters. Which method to invoke is decided at compile-time only since the calling type and the method parameters are pre-defined and that's why this type of polymorphism is also referred to as Compile-time polymorphism in Java.

That's it for now on Polymorphism in Java, but before we go, we will leave you with some Do's and Don'ts for Method Overriding and Method Overloading (very well collated at DZone) –

Method Overriding Rules

Similar to method overloading, we also have some mandatory and optional rules we need to follow to override a method.

With respect to the method it overrides, the overriding method must follow following mandatory rules:

- It must have the same method name.
- It must have the same arguments.
- It must have the same return type. From Java 5 onward, the return type can also be a subclass (subclasses are a covariant type to their parents).
- It must not have a more restrictive access modifier (if parent --> protected then child --> private is not allowed).
- It must not throw new or broader checked exceptions.

And if both overriding methods follow the above mandatory rules, then they:

- May have a less restrictive access modifier (if parent --> protected then child --> public is allowed).
- May throw fewer or narrower checked exceptions or any unchecked exception.

Apart from the above rules, there are also some facts to keep in mind:

- Only inherited methods can be overridden. That means methods can be overridden only in child classes.
- Constructors and private methods are not inherited, so they cannot be overridden.
- Abstract methods must be overridden by the first concrete (non-abstract) subclass.
- final methods cannot be overridden.
- A subclass can use super.overridden_method() to call the superclass version of an overridden method.

Method Overloading Rules

There are some rules we need to follow to overload a method. Some of them are mandatory while some are optional.

Two methods will be treated as overloaded if both follow the mandatory rules below:

- Both must have the same method name.
- Both must have different argument lists.

And if both methods follow the above mandatory rules, then they may or may not:

- Have different return types.
- Have different access modifiers.
- Throw different checked or unchecked exceptions.