Learn OOPS Concepts

By

Stanley Stephen

Linkedin: https://www.linkedin.com/in/contactstanley/
Github: https://github.com/stanleymca/Learn_from_Stanley
Email: s.stanley.mca@gmail.com

OOPS Concepts

OOP (Object Oriented Programming)
It is a programming technique based on the concept of Class and Object.

Class
A class is a template which describe about data and behavior of an object.

Object
An object is an instance of class which can be related to a real life example with its attributes (data) and methods (behavior).

Core OOPS concepts are:

- ✓ Abstraction
- ✓ Encapsulation
- ✓ Polymorphism
- ✓ Inheritance
- ✓ Association
- ✓ Aggregation
- ✓ Composition
- ✓ Coupling
- ✓ Cohesion

1. Abstraction

It is a process of hiding the implementation details from the user, only functionality will be provided to the user. This can be achieved by Abstract class and Interface.

Interface

- ✓ It is an abstract type that is used to specify a behaviour that class must implement.
- ✓ By default the variables in an interface are public final static.
- ✓ By default the methods are public abstract but in Java 8 default, static concrete methods are also introduced.
- ✓ In Java 8, interface having one abstract method is denoted by annotation @FunctionalInterface.
- ✓ Interface can extend one or more interface.
- ✓ Interface can be nested inside another interface.

Abstract class

- ✓ An abstract class is a class with both abstract and concrete methods.
- ✓ An abstract class cannot be instantiated but they can be sub-classed.
- ✓ An abstract class can support non-static, non-final methods and attributes (protected, private in addition to public)
- ✓ An abstract class can hold state of the object.

- ✓ It can have constructor and member variables whereas interface default methods cannot hold state and it cannot have constructor and member variables as well.

2. Encapsulation

Encapsulation or data binding is about packaging of data (variable) and behavior (method) together in a class.

a) Declare the variables of a class as private.
b) Provide public setter and getter methods to modify and view the variable values.

```
public class CheckAccountBalance {
        private double balance=0;

        public void SetAccBalance(double bal) {
                this.balance=bal;
        }

        public double getAccBalance() {
                return balance;
        }
}
```

3. Polymorphism

It is a concept by which we can perform a single action by different ways.

1. Overloading (static / compile time)
2. Overriding (dynamic / runtime)

Method Overloading

- ✓ If a class has multiple methods having same name but different in parameters is known as Method Overloading.
- ✓ There are 2 types to overload a method
  a) by changing number of arguments
  b) by changing the datatype
- ✓ Method overloading is not possible by changing the return type of method only. One type is promoted to another implicitly if no matching datatype is found.
- ✓ There will be ambiguity in some cases when there are no matching type arguments, each method promotes similar number of arguments.

For example:

```
void sum(int a, long b) {
        System.out.println ("a") ;
}

void sum(long a, long b) {
        System.out.println ("b") ;
}

obj.sum(20, 20) ; // ambiguity // compile time error
```

One type is not de-promoted implicitly for example double cannot be de-promoted to any type implicitly.

- ✓ Thrown exceptions from methods are not considered.

Normally, we are developing our application interface base approach. For example, service and serviceImpl. So, my controller needs to inject serviceImpl to get business functionality. In this case, in controller class we are injecting service bean by taking service interface reference as below.

```
public interface BankService {
        public void doTransaction();
}
@Service
public class BankServiceImpl implements BankService {
        @Override
        public void doTransaction() {
                // LOGIC
        }
}
@Controller
public class BankController {

        @Autowired(required = true)
        private BankService service;
}
```

Here, internally IOC container instantiates my service bean as shown in below approach.

BankService service = new BankServiceImpl(); //Runtime polymorphism

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.

Usually, method overloading happens inside a single class, but a method can also be treated as overloaded in the subclass of that class — because the subclass inherits one version of the method from the parent class and then can have another overloaded version in its class definition.

Method Overriding

Method overriding means defining a method in a child class that is already defined in the parent class with the same method signature — same name, arguments, and return type (after Java 5, you can also use a covariant type as the return type).

Whenever we extend a super class in a child class, the child class automatically gets all the methods defined in the super. We call them derived methods. But in some cases, we do not want some derived methods to work in the manner that they do in the parent. We can override those methods in the child class. For example, we always override **equals**, **hashCode**, and **toString** from the Object class. If you're interested, you can read more on Why can't we override **clone()** method from the Object class.

In the case of abstract methods, either from a parent abstract class or an interface, we do not have any option: We need implement or, in other words, override all the abstract methods.

Method overriding is also known as Runtime Polymorphism and Dynamic Method Dispatch because the method that is going to be called is decided at runtime by the JVM.

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.

Sample code:

```
package com.stanley.polymorphism;
public class MethodOverriding {
	public static void main(String[] args) {
		A a = new A();// In Class A
		a.Show();	// show A
		a.Print();	// print A
		//a.Display(); // compile time error Display method of A is not visible B

		b = new B();	// In Class A In Class B
		b.Show();	// show B
		b.Print();	// print B
		b.Display();	// display B
```

```
                A ab = new B();      // In Class A In Class B
                ab.Show();           // show B
                ab.Print();          // print A

                //ab.Display();      // compile time error Display method of A is not visible
                ab.Check();          // check A
        }
}

class A {
        A() {
                System.out.println("In Class A");
        }

        void Show() {
                System.out.println("show A");
        }

        static void Print()  {
                System.out.println("print A");
        }
        private void Display() {
                System.out.println("display A");
        }

        void Check() {
                System.out.println("check A");
        }
}

class B extends A {

        B() {
                System.out.println("In Class B");
        }

        void Show() {
                System.out.println("show B");
        }

        static void Print() {
                System.out.println("print B");
        }
```

```
        void Display() {
                System.out.println("display B");
        }
}
```

Output:

```
In Class A
show A
print A

In Class A
In Class B
show B
print B
display B

In Class A
In Class B
show B
print A
check A
```

4. Inheritance (IS-A relationship)

A subclass can inherit the states and behaviors of it super class is known as Inheritance. Multiple inheritance is not possible due to ambiguity problem and cyclic inheritance is not allowed on Java.

5. Association (HAS-A relationship)

Association is a relationship between two classes where one class utilizes feature of another class. It is HAS-A relationship.

For example

- ✓ Student and ContactInfo.
- ✓ All Student HAS A ContactInfo
- ✓ Student and StudentIdCard
- ✓ All Student HAS A StudentIdCard

6. Aggregation

It is a type of association where it is HAS-A relationship and existence of both the class are not dependent on each other.

For example

- ✓ a department has several professors.
- ✓ Student has a ContactInfo. They are dependent on each other as ContactInfo can be used anywhere else for example Employee class.

7. Composition

It is a type of association where it is HAS-A relationship. In this relationship Class B cannot exists without Class A whereas Class A can exist without Class B.

For example, A university HAS several departments.

Student and StudentIdCard.

StudentIDCard class is dependent on Student Class and it cannot be used anywhere. For example, it cannot be used in Employee class.

8. Coupling

Coupling indicates level of dependency between two classes.
In good software design, classes should be loosely coupled to each other.

For example, Spring Framework.

9. Cohesion

Cohesion indicates level of logical connection of methods and attributes used in a class.
In good software design, Classes should be highly cohesive.

Please check out my other e-books in
https://github.com/stanleymca/Learn_from_Stanley