Four Main Object Oriented Programming Concepts of Java

Difficulty Level : Easy Last Updated : 22 Nov, 2021

Object-oriented programming generally referred to as OOPS is the backbone of java as java being a completely object-oriented language. Java organizes a program around the various objects and well-defined interfaces. There are four pillars been here in OOPS which are listed below. These **concepts** aim to implement real-world entities in programs.

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction is a process of hiding implementation details and exposes only the functionality to the user. In abstraction, we deal with ideas and not events. This means the user will only know “what it does” rather than “how it does”.

There are two ways to achieve abstraction in Java

1. Abstract class (0 to 100%)
2. Interface (100%)

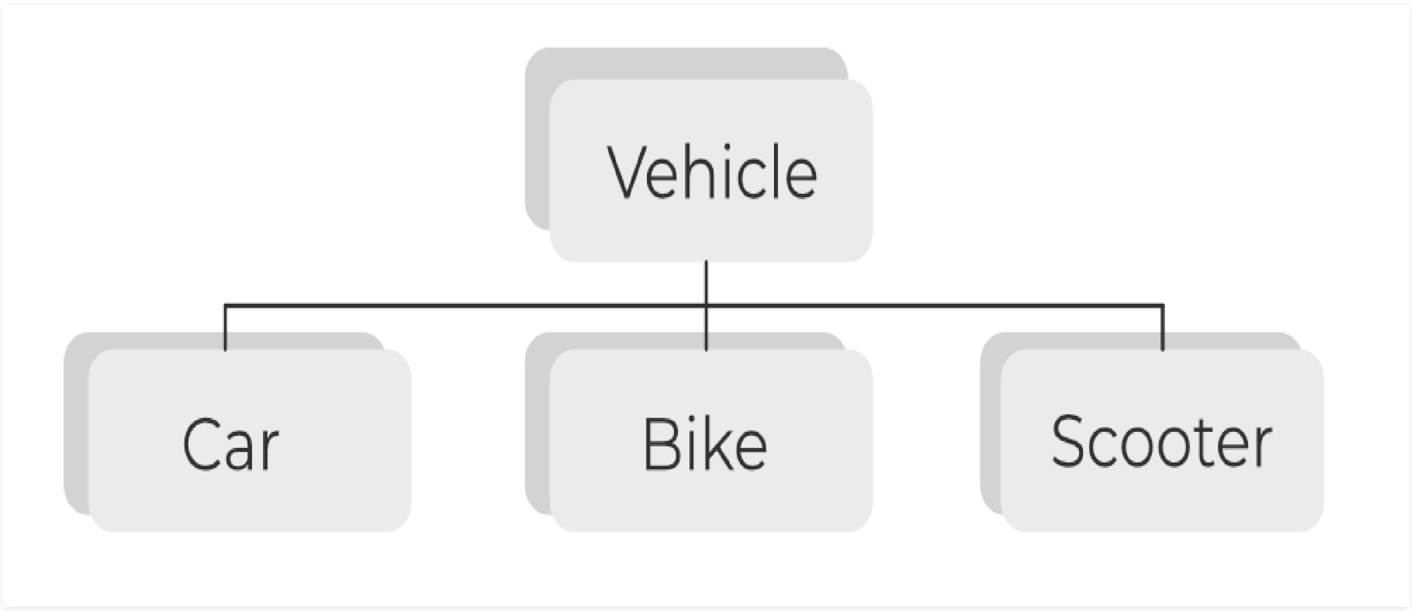
> ***Real-Life Example:*** *A driver will focus on the car functionality (Start/Stop -> Acceler ate/ Break), he/she does not bother about how the Accelerate/ brake mechanism w orks internally. And this is how the abstraction works.*

Certain key points should be remembered regarding this pillar of OOPS as follows:

- The class should be abstract if a class has one or many abstract methods
- An abstract class can have constructors, concrete methods, static method, and final method
- Abstract class can’t be instantiated directly with the ***new*** operator. It can be possible as shown in pre tag below:

```
A b = new B();
```

- The child class should override all the abstract methods of parent else the child class should be declared with abstract keyword



Example:

```java
// Abstract class
public abstract class Car {
    public abstract void stop();
}

// Concrete class
public class Honda extends Car {
    // Hiding implementation details
    @Override public void stop()
    {
        System.out.println("Honda::Stop");
        System.out.println(
            "Mechanism to stop the car using break");
    }
}

public class Main {
    public static void main(String args[])
    {
        Car obj
            = new Honda(); // Car object =>contents of Honda
        obj.stop(); // call the method
    }
}
```

Pillar 2: Encapsulation

Encapsulation is the process of wrapping code and data together into a single unit.

> ***Real-Life Example:***
>
> *A capsule which is mixed of several medicines. The medicines are hidden data to th e end user.*

In order to achieve encapsulation in java follow certain steps as proposed below:

- Declare the variables as private
- Declare the setters and getters to set and get the variable values

> ***Note:*** *There are few disadvantages of encapsulation in java as follows:*
>
> 1. ***Control Over Data:*** *We can write the logic in the setter method to not store the n egative values for an Integer. So by this way we can control the data.*
> 2. ***Data Hiding:*** *The data members are private so other class can't access the data members.*
> 3. ***Easy to test:*** *Unit testing is easy for encapsulated classes*

Example:

```
// A Java class which is a fully encapsulated class.
public class Car
{

        // private variable
        private String name;

        // getter method for name
        public String getName()
    {
           return name;

    }

        // setter method for name
        public void setName(String name)
    {
           this.name = name
    }

}


// Java class to test the encapsulated class.
public class Test
{
       public static void main(String[] args)
    {

            // creating instance of the encapsulated class
            Car car
            = new Car();

            // setting value in the name member
            car.setName("Honda");

            // getting value of the name member
            System.out.println(car.getName());

    }

}
```

Pillar 3: Inheritance

Inheritance is the process of one class inheriting properties and methods from another class in Java. Inheritance is used when we have **is-a** relationship between objects. Inheritance in Java is implemented using **extends** keyword.

Real-life Example:

The planet Earth and Mars inherits the super class Solar System and Solar system inherits the Milky Way Galaxy. So Milky Way Galaxy is the top super class for Class Solar System, Earth and Mars.

Let us do discuss the usage of inheritance in java applications with a generic example before proposing the code. So consider an example extending the Exception class to create an application-specific Exception class that contains more information like error codes. For example NullPointerException.

There are 5 different types of inheritance in java as follows:

1. **Single Inheritance:** Class B inherits Class B using extends keyword
2. **Multilevel Inheritance:** Class C inherits class B and B inherits class A using extends keyword
3. **Hierarchy Inheritance:** Class B and C inherits class A in hierarchy order using extends keyword
4. **Multiple Inheritance:** Class C inherits Class A and B. Here A and B both are superclass and C is only one child class. Java is not supporting Multiple Inheritance, but we can implement using Interfaces.
5. **Hybrid Inheritance:** Class D inherits class B and class C. Class B and C inherits A. Here same again Class D inherits two superclass, so Java is not supporting Hybrid Inheritance as well.

Example:

```java
// super class
class Car {
    // the Car class have one field
    public String wheelStatus;
    public int noOfWheels;

    // the Car class has one constructor
    public Car(String wheelStatus, int noOfWheels)
    {
        this.wheelStatus = wheelStatus;
        this.noOfWheels = noOfWheels;
    }

    // the Car class has three methods
    public void applyBrake()
    {
        wheelStatus = "Stop" System.out.println(
            "Stop the car using break");
    }

    // toString() method to print info of Car
    public String toString()
    {
        return ("No of wheels in car " + noOfWheels + "\n"
                + "status of the wheels " + wheelStatus);
    }
}

// sub class
class Honda extends Car {

    // the Honda subclass adds one more field
    public Boolean alloyWheel;

    // the Honda subclass has one constructor
    public Honda(String wheelStatus, int noOfWheels,
                 Boolean alloyWheel)
    {
        // invoking super-class(Car) constructor
        super(wheelStatus, noOfWheels);
        alloyWheel = alloyWheel;
    }

    // the Honda subclass adds one more method
    public void setAlloyWheel(Boolean alloyWheel)
    {
        alloyWheel = alloyWheel;
    }

    // overriding toString() method of Car to print more
    // info
    @Override public String toString()
    {
        return (super.toString() + "\nCar alloy wheel "
                + alloyWheel);
    }
}

// driver class
public class Main {
    public static void main(String args[])
    {

        Honda honda = new Honda(3, 100, 25);
        System.out.println(honda.toString());
    }
}
```

Pillar 4: Polymorphism in java

Polymorphism is the ability to perform many things in many ways. The word Polymorphism is from two different Greek words- poly and morphs. “Poly” means many, and “Morphs” means forms. So polymorphism means many forms. The polymorphism can be present in the case of inheritance also. The functions behave differently based on the actual implementation.

> ***Real-life Example:***
>
> *A delivery person delivers items to the user. If it’s a postman he will deliver the letter s. If it’s a food delivery boy he will deliver the foods to the user. Like this polymorphi sm implemented different ways for the delivery function.*

There are two types of polymorphism as listed below:

1. Static or Compile-time Polymorphism
2. Dynamic or Run-time Polymorphism

Static or Compile-time Polymorphism when the compiler is able to determine the actual function, it’s called **compile-time** polymorphism. Compile-time polymorphism can be achieved by **method overloading** in java. When different functions in a class have the same name but different signatures, it’s called method overloading. A method signature contains the name and method arguments. So, overloaded methods have different arguments. The arguments might differ in the numbers or the type of arguments.

Example 1: Static Polymorphism

```
public class Car{

    public void speed() {
    }

    public void speed(String accelerator) {
    }

    public int speed(String accelerator, int speedUp) {
        return carSpeed;
    }
}
```

Dynamic or Run-time Polymorphism occurs when the compiler is not able to determine whether it's superclass method or sub-class method it's called **run-time** polymorphism. The run-time polymorphism is achieved by **method overriding**. When the superclass method is overridden in the subclass, it's called method overriding.

Example 2: Dynamic Polymorphism

```java
import java.util.Random;

class DeliveryBoy {

    public void deliver() {
        System.out.println("Delivering Item");
    }

    public static void main(String[] args) {
        DeliveryBoy deliveryBoy = getDeliveryBoy();
        deliveryBoy.deliver();
    }

    private static DeliveryBoy getDeliveryBoy() {
        Random random = new Random();
        int number = random.nextInt(5);
        return number % 2 == 0 ? new Postman() : new FoodDeliveryBoy();
    }
}

class Postman extends DeliveryBoy {
    @Override
    public void deliver() {
        System.out.println("Delivering Letters");
    }
}

class FoodDeliveryBoy extends DeliveryBoy {
    @Override
    public void deliver() {
        System.out.println("Delivering Food");
    }
}
```

Output

```
Delivering Letters
```