Advanced Java

Object-Oriented Programming Concepts

(Review)

Object-oriented programming (OOP) is a programming paradigm that organizes code by creating objects that encapsulate data and behavior.

In this document we explain the main concepts of OOP and give examples of their implementations in Java:

1. Classes and Objects:

A class defines the **blueprint** or **template** for creating objects of the same type. It encapsulates data (attributes) and behavior (methods) that objects of the class can exhibit.

Example:

Let's consider a class called "**Car**" that represents car objects. It can have attributes like "**make**," "**model**," and "**color**," and methods like "startEngine()", "drive()", and setter and getter method for its attributes. We can create multiple car objects from this class, each with its specific attributes and behavior.

Java Code:

```
public class Car {
   private String make;
   private String model;
   private String color;

   public void startEngine() {
      // Code to start the car's engine
      System.out.println("Car starting....");

   }
```

```
    public void drive() {
        // Code to control the car's movement
      System.out.println("Driving the car....");
    }
    // Getters and setters for attributes
    public String getMake() {
        return make;
    }
    public void setMake(String make) {
        this.make = make;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
}
```

Test the class:

To test our Car class we need to write a client code as follows.

```
public class CarClient {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Camry", "Red");
        myCar.startEngine();
        myCar.drive();
        myCar.setColor("Blue");

        System.out.println("Car details:");
        System.out.println("Make: " + myCar.getMake());
        System.out.println("Model: " + myCar.getModel());
        System.out.println("Color: " + myCar.getColor());
    }
}
```

In the code above, we create an instance of the `Car` class using the constructor and provide the make, model, and color as arguments. We then invoke the `startEngine()` and `drive()` methods on the `myCar` object to

simulate starting the engine and driving the car.

We also use the `setColor()` method to change the color of the car to blue. Finally, we print out the car's details by calling the `getMake()`, `getModel()`, and `getColor()` methods.

When you run the `CarClient` class, you should see the following output:

```
Car starting...
Driving the car...
Car details:
Make: Toyota
Model: Camry
Color: Blue
```

This client code tests the functionality of the `Car` class by invoking its methods and retrieving its attributes, providing a way to verify that the class behaves as expected.

2. Encapsulation:

Encapsulation refers to the practice of hiding internal details of an object and exposing only the necessary information through well-defined interfaces.

Example:

In our "Car" class, we can make the attributes private and provide public getter and setter methods to access and modify them. For instance, we can have a public method called "getColor()" to retrieve the color of a car object and a method called "setColor(String color)" to set the color of the car object.

Java Code:

```
public class Car {
    private String make;
    private String model;
    private String color;

    // Constructor
    public Car(String make, String model, String color) {
        this.make = make;
        this.model = model;
        this.color = color;
    }
    // Getters and setters for attributes
    public String getMake() {
        return make;
    }
    public void setMake(String make) {
        this.make = make;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
```

```
    // Other methods
    public void startEngine() {
        // Code to start the car's engine
    }
    public void drive() {
        // Code to control the car's movement
    }
}
```

3. Inheritance:

Inheritance allows the creation of new classes (derived classes or subclasses) based on existing classes (base classes or superclasses), inheriting their attributes and methods. It promotes code reuse and supports the "is-a" relationship.

Example:

Consider a class hierarchy where we have a base class called "Vehicle," and "Car" is a subclass of "Vehicle." The "Car" class inherits attributes and methods from the "Vehicle" class, such as "numberOfWheels" and "drive()". Additionally, the "Car" class can define its own specific attributes and methods.

Java Code:

```
public class Vehicle {
    private int numberOfWheels;

    public Vehicle(int numberOfWheels) {
        this.numberOfWheels = numberOfWheels;
    }
    public int getNumberOfWheels() {
        return numberOfWheels;
    }
    public void setNumberOfWheels(int n) {
        this.numberOfWheels = n;
    }

}
```

```
public class Car extends Vehicle {
   private String make;
   private String model;

   public Car(String make, String model) {
      super(4); // Invoke the superclass constructor
      this.make = make;
      this.model = model;
   }
   // Getters and setters for make and model attributes

   // Additional methods specific to Car
   public void printInfo() {
     System.out.println("Car Details:");
     System.out.println("Make: " + getMake());
     System.out.println("Model: " + getModel);
     System.out.println("No of Wheels: " + getNumberOfWheels()); // inherited method

   }
}
```

4. Polymorphism:

Polymorphism allows objects of different classes to be used as objects of a common superclass. It enables methods to be overridden in subclasses, providing different implementations while using the same **method signature**.

Example:

Continuing with our "Vehicle" and "Car" classes, suppose we have a method called "printDetails()" in the "Vehicle" class. Each subclass, like "Car" or "Motorcycle," can override this method to provide its own implementation. When we call "printDetails()" on a "Car" object, the overridden method in the "Car" class will be executed.

Java Code:

```
public class Vehicle {
   public void printDetails() {
      System.out.println("This is a vehicle.");
   }
}
public class Car extends Vehicle {
   @Override
   public void printDetails() {
      System.out.println("This is a car.");
   }
}
public class Motorcycle extends Vehicle {
   @Override
   public void printDetails() {
      System.out.println("This is a motorcycle.");
   }
}
// Usage
Vehicle vehicle1 = new Car();
vehicle1.printDetails(); // Output: This is a car.
Vehicle vehicle2 = new Motorcycle();
vehicle2.printDetails(); // Output: This is a motorcycle.
```

Note:

The `@Override` annotation in Java is used to indicate that a method in a subclass is intended to override a method with the same signature in its superclass. It serves as a helpful tool for both developers and the Java compiler.

The main purposes of the `@Override` annotation are:

1. Compiler checks: When a method is marked with `@Override`, the Java compiler verifies that the method actually overrides a method in the superclass. If there is no matching method in the superclass, a compilation error occurs. This helps catch potential errors or typos when overriding methods.

2. Code readability: The `@Override` annotation improves code readability by making it clear to developers that a method is intended to override a method in the superclass. It serves as a visual indicator that the method is providing a specific implementation of a superclass method, making the code more understandable and maintainable.

3. Documentation: The `@Override` annotation also serves as a form of documentation, indicating to other developers (including yourself in the future) that the method is intended to override a superclass method. It helps in understanding the design and structure of the code, especially when working with complex class hierarchies.

Overall, the `@Override` annotation in Java is a helpful tool for ensuring method overriding correctness, improving code readability, and providing documentation about the intent of the code.

5. Abstraction:

Abstraction focuses on modeling real-world concepts by creating abstract classes or interfaces that define common behavior and characteristics while hiding implementation details.

Example:

Suppose we have an abstract class called "Shape" with a method called "calculateArea()". This class may have subclasses like "Rectangle" or "Circle" that implement their specific version of the "calculateArea()" method. The abstract class defines the common behavior, allowing us to work with shapes without worrying about their specific implementations.

Java Code:

```
abstract class Shape {
   public abstract double calculateArea();
}

class Rectangle extends Shape {
   private double length;
   private double width;

   public Rectangle(double length, double width) {
      this.length = length;
      this.width = width;
   }
   @Override
   public double calculateArea() {
      return length * width;
   }
}
class Circle extends Shape {
   private double radius;

   public Circle(double radius) {
      this.radius = radius;
   }
   @Override
   public double calculateArea() {
      return Math.PI * Math.pow(radius, 2);
   }
}
```

```
// Usage in the client code
Shape rectangle = new Rectangle(4, 5);
double rectangleArea = rectangle.calculateArea();

Shape circle = new Circle(3);
double circleArea = circle.calculateArea();
```

Abstract Class vs. Interfaces:

Abstract Class in Java:

An abstract class is a class that cannot be instantiated and is typically used as a base class for other classes. It serves as a blueprint for its subclasses, providing common attributes and methods. However, it may also contain abstract methods that are meant to be implemented by its subclasses.

Key points about abstract classes:

1. An abstract class is declared using the `abstract` keyword.
2. It can have both concrete (implemented) methods and abstract (unimplemented) methods.
3. Abstract methods are declared without a body and are meant to be overridden by the subclasses.
4. Abstract classes can have constructors, instance variables, and non-abstract methods.
5. Abstract classes cannot be instantiated, but they can be used as reference types.

Here's an example to illustrate the usage of an abstract class:

```
//java
abstract class Shape {
  protected String color;

  public Shape(String color) {
    this.color = color;
  }

  public abstract double calculateArea();

  public void display() {
    System.out.println("This is a shape of color: " + color);
  }
}
```

```
class Rectangle extends Shape {
  private double length;
  private double width;

  public Rectangle(String color, double length, double width) {
    super(color);
    this.length = length;
    this.width = width;
  }

  @Override
  public double calculateArea() {
    return length * width;
  }
}

class Circle extends Shape {
  private double radius;

  public Circle(String color, double radius) {
    super(color);
    this.radius = radius;
  }

  @Override
  public double calculateArea() {
    return Math.PI * radius * radius;
  }
}

public class Main {
  public static void main(String[] args) {
    Shape rectangle = new Rectangle("Red", 4, 5);
    System.out.println("Area of rectangle: " + rectangle.calculateArea());
    rectangle.display();

    Shape circle = new Circle("Blue", 3);
    System.out.println("Area of circle: " + circle.calculateArea());
    circle.display();
  }
}
```

In the example above, the `Shape` class is an abstract class that defines the common behavior and attributes for shapes. It has an abstract method called `calculateArea()` that must be implemented by its subclasses. The

`Rectangle` and `Circle` classes inherit from the `Shape` class and provide their specific implementations of the `calculateArea()` method. The `display()` method is a concrete method in the `Shape` class that is inherited by its subclasses.

Interface in Java:

An interface in Java is a collection of abstract methods that define a contract for classes to follow. It specifies the methods that a class implementing the interface must provide, without specifying their implementation details. An interface can also include constants and default methods.

Key points about interfaces:

1. An interface is declared using the `interface` keyword.
2. It can only contain abstract methods (by default) or static final fields (constants).
3. All methods in an interface are implicitly public and abstract (no need to use the `abstract` keyword).
4. A class can implement multiple interfaces, allowing for multiple inheritance of behavior.
5. Interfaces can be used as reference types.
6. Java 8 introduced default methods, which provide a default implementation for methods in an interface.

Here's an example to illustrate the usage of an interface:

```
interface Vehicle {
  void start();
  void stop();
}
class Car implements Vehicle {
  @Override
  public void start() {
    System.out.println("Car starting...");
  }
```

```
  @Override
  public void stop() {
    System.out.println("Car stopping...");
  }
}

class Motorcycle implements Vehicle {
  @Override
  public void start() {
    System.out.println("Motorcycle starting...");
  }

  @Override
  public void stop() {
    System.out.println("Motorcycle stopping...");
  }
}

public class Main {
  public static void main(String[] args) {
    Vehicle car = new Car();
    car.start();
    car.stop();

    Vehicle motorcycle = new Motorcycle();
    motorcycle.start();
    motorcycle.stop();
  }
}
```

In the example above, the `Vehicle` interface defines the contract for vehicles, specifying the `start()` and `stop()` methods. The `Car` and `Motorcycle` classes implement the `Vehicle` interface and provide their specific implementations of these methods. The `start()` and `stop()` methods are called on objects of the `Car` and `Motorcycle` classes, respectively.

Interfaces are useful for achieving abstraction and providing a common behavior contract for classes. They allow for loose coupling and flexibility in the implementation of different classes while ensuring adherence to a specific set of methods defined in the interface.