Mastering OOP Concepts in Java with Examples

Learn Object-Oriented Programming (OOP) in Java effortlessly! This guide covers key topics like classes, methods, inheritance, and abstraction with easy-to-follow examples for real-world applications.

By Great Learning Editorial Team / Updated on Jan 6, 2025 / Published on May 15, 2023

Table of contents

OOP is all about designing software using objects, where data and behaviour come together. Unlike traditional procedural programming that focuses on sequences of tasks, OOP revolves around the interaction of objects to solve problems.

Its four key principles, **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction,** form the building blocks of this approach.

Learning OOP in Java will help you write efficient, easily maintainable code that can be reused in other projects when developing large applications.

Let's dive deep into these concepts and see how they work in Java!

Understanding the Basics of OOP

What is a Class?

A class is a blueprint for creating objects. It defines the attributes (data) and methods (behaviours) of the objects. Think of a class as a template.

Example:

```
class Car {

    String brand;

    int speed;

    void drive() {

        System.out.println(brand + " is driving at " + speed +

    }

}

Here, Car is a class with attributes brand and speed, and a be
```

What is an Object?

An object is an instance of a class. It represents a specific entity with its own values for the defined attributes.

Example:

```
Car myCar = new Car();

myCar.brand = "Toyota";

myCar.speed = 120;

myCar.drive();  // Output: Toyota is driving at 120 km/h.
```

Attributes and Methods

Attributes (fields) store data, while methods define actions an object can perform. The brand and speed in the example are attributes, and drive() is a method.

First Pillar of OOP: Encapsulation

Definition:

Encapsulation bundles all the data (attributes) and method (behaviours) which are related in their functionality and role are grouped together or packaged in a single entity or class. It also limits the direct input and output of object data and allows input and output only by means of **getters** and **setters**.

Benefits of Encapsulation:

- **Data Security**: Protects information from being accessed by unauthorized personnel.
- **Flexibility**: Enables change in the implementation aspect without any effect on the other part of the code.
- **Improved Maintenance**: Facilitates first and foremost debugging and secondly, makes the code more comprehensible.

Example:

```
class Person {

    private String name;  // Private attribute

    private int age;

```

```
    // Getter for name

    public String getName() {

        return name;

    }

    // Setter for name

    public void setName(String name) {

        this.name = name;

    }

    // Getter for age

    public int getAge() {

        return age;

    }

    // Setter for age

    public void setAge(int age) {

        if (age > 0) {  // Validation

            this.age = age;

        } else {

            System.out.println("Age must be positive.");

        }

    }

}
```

Usage:

```
Person p = new Person();

p.setName("John");

p.setAge(25);

System.out.println(p.getName() + " is " + p.getAge() + " years
```

Second Pillar of OOP: Inheritance

Definition:

Inheritance allows a class (child) to obtain its attributes and actions from another class (parent). It supports the **reuse of code** and also gives classes a well-defined relationship where one class is a specialized version of the other.

Benefits of Inheritance:

- Eliminates code redundancy.
- Enables hierarchical classifications.
- Facilitates the use of polymorphism.

Example:

```
// Parent class

class Animal {

    void eat() {

        System.out.println("This animal eats food.");

    }

}

// Child class

class Dog extends Animal {

    void bark() {

        System.out.println("The dog barks.");
```

```
20
21      }
22
23  }
```

Usage:

```
1  Dog myDog = new Dog();
2
3  myDog.eat();  // Output: This animal eats food.
4
5  myDog.bark(); // Output: The dog barks.
```

To learn more about this concept, check the free inheritance in the Java course.

Third Pillar of OOP: Polymorphism

Definition:

Polymorphism allows objects to take multiple forms, enabling the same method to behave differently based on context. It is achieved through:

1. **Compile-time Polymorphism (Method Overloading)**
2. **Runtime Polymorphism (Method Overriding)**

1. Method Overloading (Compile-time Polymorphism):

Allows methods with the same name but different parameter lists.

Example:

```
1  class Calculator {
2
3      int add(int a, int b) {
4
5          return a + b;
6
7      }
8
9      double add(double a, double b) {
10
```

```
        return a + b;

    }

}
```

Usage:

```
Calculator calc = new Calculator();

System.out.println(calc.add(2, 3));       // Output: 5

System.out.println(calc.add(2.5, 3.5));   // Output: 6.0
```

2. Method Overriding (Runtime Polymorphism):

Allows a subclass to provide a specific implementation of a method already defined in its superclass.

Example:

```
class Animal {

    void sound() {

        System.out.println("Some generic animal sound");

    }

}

class Cat extends Animal {

    @Override

    void sound() {

        System.out.println("Meow");

    }

}
```

Usage:

```
Animal myAnimal = new Cat();

myAnimal.sound();  // Output: Meow
```

Explore more about **Polymorphism in Java** with this in-depth guide.

Fourth Pillar of OOP: Abstraction

Definition:

Abstraction focuses on **what an object does** rather than **how it does it.** It hides implementation details and exposes only essential features.

Benefits of Abstraction:

- Reduces complexity.
- Enhances maintainability.

Example Using Abstract Classes:

```java
abstract class Shape {

    abstract void draw();  // Abstract method

}

class Circle extends Shape {

    void draw() {

        System.out.println("Drawing a Circle");

    }

}

class Rectangle extends Shape {

    void draw() {

        System.out.println("Drawing a Rectangle");

    }

}
```

Usage:

```java
Shape s1 = new Circle();

Shape s2 = new Rectangle();

s1.draw();  // Output: Drawing a Circle

s2.draw();  // Output: Drawing a Rectangle
```

Building Blocks in Java OOP

Java's OOP features rely on several key building blocks that enhance the creation, organization, and visibility of objects and their behaviour.

1. Constructors: Special Methods to Initialize Objects

A constructor is a special method invoked when an object is created. It initializes the object and sets its state.

- **Features of Constructors:**
  - The name of the constructor is the same as the class name.
  - It does not have a return type.
  - Constructors can be overloaded.

Example:

```
class Car {

    String brand;

    int speed;

    // Constructor

    Car(String brand, int speed) {

        this.brand = brand;

        this.speed = speed;

    }

    void display() {

        System.out.println("Brand: " + brand + ", Speed: " + s

    }

}

public class Main {

    public static void main(String[] args) {

        Car myCar = new Car("Toyota", 120); // Constructor cal

        myCar.display(); // Output: Brand: Toyota, Speed: 120

    }

}
```

Here, the constructor initializes brand and speed when the object myCar is created.

2. this Keyword: Refers to the Current Object

The this keyword is a reference to the current object and is used for:

- Resolving naming conflicts between class attributes and method parameters.
- Calling another constructor in the same class.
- Referring to the current object in a method or constructor.

Example:

```
class Employee {

    String name;

    Employee(String name) {

        this.name = name; // Refers to the class attribute

    }

    void display() {

        System.out.println("Employee Name: " + this.name);

    }

}
```

3. Access Modifiers: Control the Visibility of Attributes and Methods

Access modifiers define the scope of variables, methods, and classes. Java provides four types of access modifiers:

- **Private:** Accessible only within the class.
- **Default:** Accessible within the same package.
- **Protected:** Accessible within the package and by subclasses.
- **Public:** Accessible from anywhere.

Example:

```
class Example {

    private int privateValue = 10; // Private: restricted to t

    public int publicValue = 20;  // Public: accessible anywhe

    public int getPrivateValue() {

        return privateValue; // Controlled access via a public

    }

}
```

Learn more about **Access Modifiers in Java** with this insightful guide.

4. Static Members: Define Shared Variables and Methods

The static keyword denotes that a field or method belongs to the class, not any specific instance. These are shared across all instances of the class.

Example:

```
class Counter {

    static int count = 0; // Static variable shared by all obj

    Counter() {

        count++;

    }

    static void displayCount() {

        System.out.println("Count: " + count); // Static metho

    }

}

public class Main {

    public static void main(String[] args) {

        new Counter();
```

```
        new Counter();

        Counter.displayCount(); // Output: Count: 2

    }
}
```

Advanced OOP Features

1. Interfaces: Define Contracts for Implementing Classes

An interface is a blueprint of a class that contains abstract methods. Classes that implement an interface must provide implementations for its methods.

Example:

```
interface Printable {

    void print(); // Abstract method

}

class Document implements Printable {

    public void print() {

        System.out.println("Printing Document...");

    }

}

public class Main {

    public static void main(String[] args) {

        Printable doc = new Document();

        doc.print(); // Output: Printing Document...

    }

}
```

2. Inner Classes: Enable Modular Organization

Inner classes are defined within another class and can access its private members. They are used for better encapsulation and logical grouping.

Types of Inner Classes:

1. **Member Inner Class**
2. **Static Nested Class**
3. **Local Inner Class**
4. **Anonymous Inner Class**

Example of Member Inner Class:

```
class Outer {

    private String message = "Hello from Outer class";

    class Inner {

        void display() {

            System.out.println(message); // Accessing private

        }

    }

}

public class Main {

    public static void main(String[] args) {

        Outer outer = new Outer();

        Outer.Inner inner = outer.new Inner();

        inner.display(); // Output: Hello from Outer class

    }

}
```

3. Anonymous Classes: Simplify Short-lived Object Creation

Anonymous classes are unnamed inner classes, often used to override methods or provide specific functionality.

Example:

```
interface Greeting {

    void sayHello();

}

public class Main {

    public static void main(String[] args) {

        Greeting greet = new Greeting() { // Anonymous class

            public void sayHello() {

                System.out.println("Hello, Anonymous World!");

            }

        };

        greet.sayHello(); // Output: Hello, Anonymous World!

    }

}
```

Conclusion

Java supports Object Oriented Programming (OOP) which gives developers the chance to build scalable, reusable and maintainable software. Being able to use classes, objects, inheritance, and polymorphism are basic, and advanced features such as interfaces and design patterns will enable you to make robust, efficient applications.

OOP principles like encapsulation, abstraction and modularity help us handle complex problems by simplifying code and allowing us to manage and extend code.

Experience of SOLID principles and design patterns will help you gain more experience.

To enhance your skills further, check out the free programming courses that can help you build a solid foundation for your development journey.

FAQs on OOP Concepts in Java

1. What is the difference between a class and an object in Java?
A class is a blueprint or template that defines the properties and behaviors of a type. An object is an instance of a class that holds specific values for the properties defined by the class.

2. Why is the static keyword important in Java OOP?
The static keyword signifies that a field or method belongs to the class rather than any individual object. This allows shared access to the field or method across all instances of the class.

3. What is the main difference between method overloading and method overriding?
Method Overloading: Occurs within the same class and involves methods with the same name but different parameter lists.
Method Overriding: Happens between a superclass and a subclass where a method in the subclass has the same signature as a method in the superclass.

4. How does Java implement abstraction?
Java achieves abstraction using abstract classes and interfaces. Abstract classes can have both abstract methods and implemented methods, while interfaces define a contract that implementing classes must follow.

5. What is encapsulation, and why is it important?
Encapsulation is the practice of restricting direct access to a class's fields and methods, usually by using private access modifiers and providing public getter and