Object-Oriented Programming in Java Training Module

A Comprehensive Guide for Beginners

Prepared by xAI Training Team

May 07, 2025

Contents

1 Introduction to OOP in Java

Object-Oriented Programming (OOP) is a paradigm that organizes code into objects, promoting modularity and reusability. Java, a fully object-oriented language, implements OOP through classes and objects. This module covers core OOP concepts—encapsulation, inheritance, polymorphism, and abstraction—with practical Java examples.

1.1 Why Use OOP in Java?

- **Modularity**: Encapsulates data and behavior in objects.
- **Reusability**: Inheritance and polymorphism reduce code duplication.
- **Scalability**: Facilitates large-scale application development.

1.2 Setting Up the Environment

Install the Java Development Kit (JDK) from Oracle's website. Use an IDE like IntelliJ IDEA or Eclipse. Verify installation with:

```
java -version
```

2 Classes and Objects

Classes are blueprints for objects, which are instances of classes.

2.1 Creating a Class

Below is a simple class representing a `Person`.

```
public class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    public static void main(String[] args) {
        Person person = new Person("Alice", 25);
        person.displayInfo();
    }
}
```

Explanation:

- `class Person`: Defines the class.
- `this.name`: Refers to the instance variable.
- `new Person`: Creates an object.

3 Encapsulation

Encapsulation hides data and exposes it through methods, ensuring data integrity.

3.1 Using Getters and Setters

```java
public class BankAccount {
    private double balance;
    private String accountHolder;

    public BankAccount(String accountHolder, double initialBalance)
       {
        this.accountHolder = accountHolder;
        this.balance = initialBalance;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        if (balance >= 0) {
            this.balance = balance;
        }
    }

    public String getAccountHolder() {
        return accountHolder;
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount("Bob", 1000.0);
        account.setBalance(1500.0);
        System.out.println("Account Holder: " + account.
           getAccountHolder());
        System.out.println("Balance: $" + account.getBalance());
    }
}
```

4 Inheritance

Inheritance allows a class to inherit properties and methods from another.

4.1 Single Inheritance Example

```
class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
    }

    void honk() {
        System.out.println("Beep!");
    }
}

class Car extends Vehicle {
    int wheels;

    Car(String brand, int wheels) {
        super(brand);
        this.wheels = wheels;
    }

    void displayInfo() {
        System.out.println("Brand: " + brand + ", Wheels: " + wheels
            );
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        Car car = new Car("Toyota", 4);
        car.honk();
        car.displayInfo();
    }
}
```

5 Polymorphism

Polymorphism allows objects to be treated as instances of their parent class, with method overriding for specific behavior.

5.1 Method Overriding

```java
class Animal {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow!");
    }
}

public class PolymorphismExample {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();
        dog.makeSound(); // Woof!
        cat.makeSound(); // Meow!
    }
}
```

6 Abstraction

Abstraction hides implementation details, exposing only essential features.

6.1 Using Abstract Classes

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

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double calculateArea() {
```

```
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double width, height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

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

public class AbstractionExample {
    public static void main(String[] args) {
        Shape circle = new Circle(5);
        Shape rectangle = new Rectangle(4, 6);
        System.out.println("Circle Area: " + circle.calculateArea())
            ;
        System.out.println("Rectangle Area: " + rectangle.
            calculateArea());
    }
}
```

7 Interfaces

Interfaces define contracts for classes to implement.

7.1 Interface Example

```
interface Printable {
    void print();
}

class Document implements Printable {
    String content;

    Document(String content) {
        this.content = content;
    }

    @Override
    public void print() {
        System.out.println("Printing: " + content);
```

```
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Printable doc = new Document("Sample Document");
        doc.print();
    }
}
```

8 Exception Handling in OOP

Handle errors in OOP applications using `try-catch`.

8.1 Custom Exception Example

```
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}

class Bank {
    private double balance;

    public Bank(double balance) {
        this.balance = balance;
    }

    public void withdraw(double amount) throws
       InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Insufficient funds
               !");
        }
        balance -= amount;
        System.out.println("Withdrawal successful. New balance: $" +
            balance);
    }
}

public class ExceptionExample {
    public static void main(String[] args) {
        Bank bank = new Bank(1000);
        try {
            bank.withdraw(1500);
        } catch (InsufficientFundsException e) {
            System.out.println("Error: " + e.getMessage());
        }
```

```
    }
}
```

9 Conclusion

This module covers OOP principles in Java—classes, encapsulation, inheritance, polymorphism, abstraction, and interfaces. Practice these examples and explore advanced topics like design patterns and Java frameworks.

10 References

- Oracle Java Documentation: `https://docs.oracle.com/javase`
- Java Tutorials by Oracle: `https://docs.oracle.com/javase/tutorial`