Object-Oriented Programming with Java (CMP 175)

Comprehensive Guide

Contents

1 Introduction to Object-Oriented Programming

1.1 Problems in Procedure-Oriented Programming

Procedure-oriented programming (POP) relies on procedures or functions to perform tasks, with data and functions often separated. This approach becomes problematic in large-scale projects due to several limitations. POP lacks modularity, making code organization difficult. It struggles with code reuse, as functions are often tied to specific data structures. Additionally, maintaining and debugging large procedural codebases is challenging due to global data accessibility, which can lead to unintended side effects. For example, in languages like C, global variables can be modified by any function, increasing the risk of errors.

1.1.1 Limitations of Scalability

As programs grow, POPs lack of structure makes it hard to manage complexity. Without clear boundaries between data and operations, adding new features often requires modifying existing code, leading to potential errors. For instance, a payroll system written in POP might require changes across multiple functions to add a new employee type, increasing development time and error risks.

1.1.2 Code Maintenance Challenges

Debugging in POP is cumbersome because functions can access and modify global data unpredictably. A single change in a global variable can cascade through unrelated parts of the program, causing bugs that are difficult to trace. This section includes examples of procedural code to illustrate maintenance issues compared to OOP approaches.

1.2 Advantages and Disadvantages of OOP

Object-Oriented Programming (OOP) addresses many POP limitations by organizing code into objects that combine data and behavior. Advantages include modularity, reusability, scalability, and maintainability. Encapsulation hides data, reducing errors, while inheritance and polymorphism enable code reuse and flexibility. However, OOP introduces complexity, requiring a learning curve, and may incur performance overhead due to object creation and management.

1.2.1 Advantages

OOPs modularity allows developers to work on independent objects, simplifying team collaboration. Reusability through inheritance reduces redundant code, and polymorphism allows flexible implementations. For example, a banking system can use a base `Account` class with derived `SavingsAccount` and `CheckingAccount` classes, reusing common functionality.

1.2.2 Disadvantages

OOPs complexity can overwhelm beginners, and its abstraction layers may lead to slower execution compared to POP for simple tasks. This section discusses trade-offs, such as the overhead of creating objects versus the simplicity of procedural functions.

1.3 Features and Concepts of OOP

OOP is built on four core concepts: encapsulation, inheritance, polymorphism, and abstraction. Objects represent real-world entities with attributes (data) and methods (behavior), while classes define blueprints for objects. Encapsulation protects data, inheritance enables code reuse, polymorphism allows flexibility, and abstraction simplifies complex systems.

1.3.1 Objects and Classes

A class defines properties and behaviors, while an object is an instance of a class. For example, a `Car` class may define attributes like `color` and methods like `drive()`, with objects representing specific cars.

1.3.2 Key Concepts

This subsection explains:

- **Abstraction**: Hiding complex details and exposing only necessary parts (e.g., a `Car` class hides engine details).
- **Encapsulation**: Bundling data and methods, restricting access (e.g., private fields with public getters/setters).
- **Inheritance**: Creating new classes from existing ones (e.g., `ElectricCar` extends `Car`).
- **Polymorphism**: Allowing objects to be treated as instances of their parent class (e.g., method overriding).

1.4 Basic OOP Principles

This section reinforces the four OOP principles with Java-specific examples, showing how they improve code design. For instance, encapsulation is demonstrated with private fields and public methods, while inheritance is shown with a class hierarchy.

1.4.1 Abstraction

Abstraction reduces complexity by hiding implementation details. In Java, abstract classes and interfaces provide abstraction mechanisms. Examples include defining an abstract `Shape` class with a method `calculateArea()`.

1.4.2 Encapsulation

Encapsulation protects data by restricting access to fields using access specifiers. A Java example shows a `BankAccount` class with private balance and public deposit/withdraw methods.

2 Basic Java Programming

2.1 Features of Java and JVM

Java is a versatile, platform-independent language due to its write once, run anywhere philosophy. The Java Virtual Machine (JVM) executes Java bytecode, enabling portability across platforms. Key features include simplicity, robustness (via exception handling), security, and multithreading support.

2.1.1 Platform Independence

Java compiles source code into bytecode, which the JVM interprets on any platform. This section explains the compilation and execution process, including the role of the Java Runtime Environment (JRE).

2.1.2 JVM Architecture

The JVM comprises the class loader, runtime data areas (e.g., heap, stack), and execution engine. This subsection details how the JVM manages memory and executes bytecode, with diagrams illustrating its components.

2.2 Java Program Structure and Naming Conventions

A Java program consists of classes, with one containing the `main` method as the entry point. Naming conventions ensure readability: classes use PascalCase (e.g., `MyClass`), methods and variables use camelCase (e.g., `myMethod`), and constants use UPPER_CASE (e.g., `MAX_VALUE`).

2.2.1 Program Structure

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

This example shows a basic Java program, explaining the roles of `public`, `static`, and `void`.

2.2.2 Naming Conventions

This subsection provides detailed guidelines and examples for naming classes, methods, variables, and packages to ensure consistency and clarity.

2.3 Variables, Constants, and Data Types

Variables store data, constants are immutable, and Java supports primitive (e.g., `int`, `double`) and reference (e.g., `String`, arrays) data types. This section explains variable declaration, initialization, and scope.

2.3.1 Primitive Data Types

Covers `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`, with examples of their ranges and uses.

2.3.2 Reference Data Types

Explains how reference types point to objects, including `String`, arrays, and custom classes, with examples of object creation and memory allocation.

2.4 Operators and I/O Operations

Java supports arithmetic (+, -, *, /), relational (==, !=, >), logical (, ||), and bitwise operators. Input/output operations use classes like `Scanner` for input and `System.out` for output.

2.4.1 Operators

Examples include arithmetic operations (e.g., `5 + 3`) and logical operations (e.g., `a && b`).

2.4.2 I/O Operations

```java
import java.util.Scanner;
public class InputOutput {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name);
        scanner.close();
    }
}
```

This example demonstrates reading input and printing output.

2.5 Control Statements and Arrays

Control statements include `if-else`, `switch`, `for`, `while`, and `do-while`. Arrays store multiple values of the same type.

2.5.1 Control Statements

Examples include a `for` loop to sum numbers and a `switch` statement for menu selection.

2.5.2 Arrays

```java
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println(num);
}
```

This example shows array declaration and iteration using enhanced for loops.

2.6 Command Line Arguments

Command-line arguments are passed to the `main` methods `String[] args` parameter, allowing runtime input.

2.6.1 Using args in main

```java
public class CommandLine {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println("Argument: " + arg);
        }
    }
}
```

This example demonstrates accessing and processing command-line arguments.

3 Classes and Objects

3.1 Object Creation and Instance Variables

Objects are instances of classes, created using the `new` keyword. Instance variables store object-specific data.

3.1.1 Object Creation

```java
class Car {
    String color;
    void drive() {
        System.out.println("Driving a " + color + " car");
    }
}
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.color = "Red";
        myCar.drive();
    }
}
```

This example shows object creation and accessing instance variables.

3.2 Access Specifiers and Encapsulation

Access specifiers (`public`, `private`, `protected`, `default`) control access to class members. Encapsulation hides data using private fields and public methods.

3.2.1 Encapsulation Example

```java
class BankAccount {
    private double balance;
    public void deposit(double amount) {
        if (amount > 0) balance += amount;
    }
    public double getBalance() {
        return balance;
    }
}
```

This example demonstrates encapsulation with private fields and public methods.

3.3 Constructors and Garbage Collection

Constructors initialize objects, and Javas garbage collector automatically reclaims memory from unused objects.

3.3.1 Constructors

```java
class Student {
    String name;
    Student(String name) {
        this.name = name;
    }
}
```

This example shows a parameterized constructor.

3.3.2 Garbage Collection

Explains how the JVMs garbage collector identifies and frees memory, with examples of object dereferencing.

3.4 this Pointer and Static Members

The `this` keyword refers to the current object, resolving naming conflicts. Static members belong to the class, not instances.

3.4.1 this Keyword

```java
class Person {
    String name;
    void setName(String name) {
        this.name = name;
    }
}
```

This example shows `this` resolving ambiguity.

3.4.2 Static Members

```java
class Counter {
    static int count = 0;
    Counter() {
        count++;
    }
}
```

This example demonstrates static variables shared across instances.

4 Inheritance and Polymorphism

4.1 Inheritance and Code Reuse

Inheritance allows a class to inherit properties and methods from another, promoting code reuse.

4.1.1 Extending Classes

```java
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}
```

This example shows a `Dog` class inheriting from `Animal`.

4.2 Types of Inheritance

Java supports single, multilevel, and hierarchical inheritance (multiple inheritance is achieved via interfaces).

4.2.1 Types of Inheritance

Examples include single inheritance (`Dog` extends `Animal`), multilevel (`Puppy` extends `Dog` extends `Animal`), and hierarchical (`Cat` and `Dog` extend `Animal`).

4.3 Polymorphism (Overriding, Overloading)

Polymorphism allows methods to take different forms. Method overriding redefines a parent class method, while overloading defines multiple methods with different parameters.

4.3.1 Method Overriding

```java
class Animal {
    void sound() {
        System.out.println("Some sound...");
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}
```

This example shows overriding the `sound` method.

4.3.2 Method Overloading

```java
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}
```

This example demonstrates method overloading.

4.4 Abstract Classes and Methods

Abstract classes cannot be instantiated and may contain abstract methods, which subclasses must implement.

4.4.1 Abstract Class Example

```java
abstract class Shape {
    abstract double calculateArea();
}
class Circle extends Shape {
    double radius;
    Circle(double radius) {
        this.radius = radius;
    }
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

This example shows an abstract `Shape` class and a concrete `Circle` class.

4.5 Packages and Interfaces

Packages organize classes, while interfaces define contracts for classes to implement.

4.5.1 Packages

```java
package com.example;
public class MyClass {
    // Class code
}
```

This example shows package declaration and usage.

4.5.2 Interfaces

```java
interface Drawable {
    void draw();
}
class Circle implements Drawable {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

This example demonstrates implementing an interface.

4.6 Lambda Functions

Lambda expressions enable functional programming by treating functions as objects.

4.6.1 Lambda Example

```java
interface Operation {
    int operate(int a, int b);
}
public class Main {
    public static void main(String[] args) {
        Operation add = (a, b) -> a + b;
        System.out.println(add.operate(5, 3));
    }
}
```

This example shows a lambda expression for addition.

5 Exception Handling

5.1 Introduction to Exceptions

Exceptions are events that disrupt normal program flow, such as division by zero. Java uses exceptions to handle errors gracefully.

5.1.1 Exception Basics

This subsection explains errors (unrecoverable) versus exceptions (recoverable), with examples like `NullPointerException`.

5.2 Types of Exceptions

Java has checked exceptions (e.g., `IOException`, checked at compile-time) and unchecked exceptions (e.g., `RuntimeException`, detected at runtime).

5.2.1 Checked vs Unchecked

Examples include catching an `IOException` (checked) versus handling an `ArrayIndexOutOfBoundsExce`
(unchecked).

5.3 Exception Handling Mechanisms

Java uses `try`, `catch`, `throw`, `throws`, and `finally` for exception handling.

5.3.1 Try-Catch Example

```java
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
} finally {
    System.out.println("Cleanup");
}
```

This example shows handling a division-by-zero exception.

5.4 Built-in and User-Defined Exceptions

Java provides built-in exceptions like `NullPointerException`. User-defined exceptions extend `Exception`.

5.4.1 Custom Exceptions

```java
class InvalidAgeException extends Exception {
    InvalidAgeException(String message) {
        super(message);
    }
}
```

This example defines a custom exception.

6 Streams and File I/O

6.1 Byte and Character Streams

Byte streams (e.g., `FileInputStream`) handle raw bytes, while character streams (e.g., `FileReader`) handle text.

6.1.1 Stream Types

Examples compare reading a file using `FileInputStream` versus `FileReader`.

6.2 I/O Class Hierarchy

The `java.io` package includes classes like `InputStream`, `OutputStream`, `Reader`, and `Writer`.

6.2.1 Class Hierarchy

This subsection details the hierarchy and key classes, with diagrams.

6.3 File Manipulation (Input/Output Streams)

Reading and writing files using streams, including buffered streams for efficiency.

6.3.1 File I/O Example

```java
import java.io.*;
public class FileExample {
    public static void main(String[] args) throws IOException {
        FileWriter writer = new FileWriter("output.txt");
        writer.write("Hello, File!");
        writer.close();
    }
}
```

This example shows writing to a file.

7 GUI Programming with Swing

7.1 Introduction to GUI in Java

Swing provides lightweight components for building graphical user interfaces (GUIs) in Java, replacing older AWT components.

7.1.1 Swing Overview

This subsection explains Swings architecture and advantages, such as platform independence.

7.2 Swing Components (Buttons, Labels, Text Fields)

Swing includes components like `JButton`, `JLabel`, and `JTextField` for building GUIs.

7.2.1 Component Examples

```java
import javax.swing.*;
public class SimpleGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Simple GUI");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel("Hello, Swing!");
        frame.add(label);
        frame.setVisible(true);
    }
}
```

This example creates a simple Swing window.

7.3 Event-Driven Programming (Mouse/Key Events)

Swing uses event listeners to handle user interactions like mouse clicks or key presses.

7.3.1 Event Handling

```java
import javax.swing.*;
import java.awt.event.*;
public class ButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Example");
        JButton button = new JButton("Click Me");
        button.addActionListener(e -> JOptionPane.
           showMessageDialog(null, "Button Clicked!"));
        frame.add(button);
        frame.setSize(200, 100);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

This example handles a button click event.

7.4 Building Interactive Applications

This section walks through building a complete Swing application, such as a simple calculator.

7.4.1 Sample Application

A calculator GUI with buttons for digits and operations, demonstrating layout managers and event handling.

8 Generics

8.1 Advantages of Generics

Generics provide type safety, eliminate casts, and improve code readability by allowing parameterized types.

8.1.1 Type Safety

Generics prevent runtime errors, e.g., adding a `String` to an `ArrayList<Integer>`.

8.2 Generic Classes, Methods, and Constructors

Generic classes and methods allow reusable code with type parameters.

8.2.1 Generic Class Example

```java
class Box<T> {
    T item;
    void setItem(T item) {
        this.item = item;
    }
    T getItem() {
        return item;
    }
}
```

This example defines a generic `Box` class.

8.3 Polymorphism in Generics

Bounded types restrict generics to specific types, enabling polymorphic behavior.

8.3.1 Bounded Types

```java
class GenericBound<T extends Number> {
    T value;
    T getValue() {
        return value;
    }
}
```

This example restricts `T` to subclasses of `Number`.