# Java Programming Paradigms

## Object-Oriented Programming (OOP) in Java

### 1. Fundamental Concepts of OOP

#### 1.1 Classes and Objects

```java
// Class definition
public class Person {
    // Instance variables (attributes)
    private String name;
    private int age;

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

    // Methods
    public void introduce() {
        System.out.println("Hi, I'm " + name + " and I'm " + age + " years old.");
    }

    // Getters and Setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

// Object creation and usage
public class OOPDemo {
    public static void main(String[] args) {
        Person person1 = new Person("Alice", 25);
        person1.introduce();
    }
}
```

#### 1.2 Four Pillars of OOP

##### 1.2.1 Encapsulation

```
public class BankAccount {
    // Private variables - encapsulation
    private double balance;

    // Public methods to interact with private data
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

#### 1.2.2 Inheritance

```
// Parent class
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat() {
        System.out.println(name + " is eating");
    }
}

// Child class
public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    // Method overriding
    @Override
    public void eat() {
        System.out.println(name + " is eating dog food");
    }

    // Additional method
```

```
    public void bark() {
        System.out.println(name + " is barking");
    }
}
```

#### 1.2.3 Polymorphism

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        // Method Overloading
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 3));
        System.out.println(calc.add(5.5, 3.2));

        // Runtime Polymorphism
        Animal myDog = new Dog("Buddy");
        myDog.eat();  // Calls Dog's eat method
    }
}

class Calculator {
    // Method Overloading
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}
```

#### 1.2.4 Abstraction

```
// Abstract class
abstract class Shape {
    // Abstract method
    public abstract double calculateArea();

    // Concrete method
    public void display() {
        System.out.println("This is a shape");
    }
}

// Concrete class
class Circle extends Shape {
    private double radius;
```

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

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

# Functional Programming in Java

## 2. Functional Programming Concepts

### 2.1 Lambda Expressions

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class LambdaDemo {
    public static void main(String[] args) {
        // Traditional approach
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Lambda expression for filtering
        List<Integer> evenNumbers = numbers.stream()
            .filter(n -> n % 2 == 0)
            .collect(Collectors.toList());

        // Lambda expression for transformation
        List<Integer> squaredNumbers = numbers.stream()
            .map(n -> n * n)
            .collect(Collectors.toList());

        // Lambda with functional interface
        MathOperation addition = (a, b) -> a + b;
        System.out.println("Addition: " + addition.operate(5, 3));
    }

    // Functional interface
    @FunctionalInterface
    interface MathOperation {
        int operate(int a, int b);
    }
}
```

### 2.2 Stream API

```
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class StreamAPIDemo {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        // Filtering
        List<String> longNames = names.stream()
            .filter(name -> name.length() > 4)
            .collect(Collectors.toList());

        // Mapping
        List<Integer> nameLengths = names.stream()
            .map(String::length)
            .collect(Collectors.toList());

        // Reducing
        Optional<String> longestName = names.stream()
            .reduce((name1, name2) ->
                name1.length() > name2.length() ? name1 : name2);

        // Sorting
        List<String> sortedNames = names.stream()
            .sorted()
            .collect(Collectors.toList());
    }
}
```

### 2.3 Method References

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class MethodReferenceDemo {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("alice", "bob", "charlie");

        // Method reference to static method
        List<String> upperNames = names.stream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        // Method reference to instance method
        names.forEach(System.out::println);

        // Constructor method reference
```

```
        List<Integer> lengths = names.stream()
            .map(String::length)
            .collect(Collectors.toList());
    }
}
```

# Comparison of OOP and Functional Programming

## Key Differences

| Aspect | Object-Oriented Programming | Functional Programming |
|---|---|---|
| Core Concept | Objects and Classes | Functions and Immutability |
| State Management | Mutable state | Immutable state |
| Primary Focus | Data and Behavior | Computation and Transformation |
| Inheritance | Class-based inheritance | Composition and Higher-Order Functions |
| Side Effects | Often present | Minimized |
| Code Reusability | Through Inheritance and Polymorphism | Through Function Composition |

# Best Practices

## OOP Best Practices

1. Follow SOLID principles
2. Keep classes focused and cohesive
3. Use interfaces for abstraction
4. Prefer composition over inheritance
5. Make classes immutable when possible

## Functional Programming Best Practices

1. Use pure functions
2. Avoid side effects
3. Prefer immutability
4. Use higher-order functions
5. Leverage stream API for complex operations

# Practical Exercises

## OOP Exercise: Banking System

```
public class BankingSystem {
    public static void main(String[] args) {
        // Implement a complete banking system with:
        // 1. Account creation
        // 2. Deposit and withdrawal
```

```
        // 3. Interest calculation
        // 4. Transaction history
    }
}
```

### Functional Programming Exercise: Data Processing

```
public class DataProcessingDemo {
    public static void main(String[] args) {
        // Create a list of employees
        // 1. Filter employees by department
        // 2. Calculate average salary
        // 3. Group employees by age range
        // 4. Find highest-paid employee
    }
}
```

## Conclusion

Both Object-Oriented and Functional Programming paradigms have their strengths. Modern Java encourages a multi-paradigm approach, allowing developers to use the best approach for each specific problem.

## Recommended Learning Path

1. Master OOP fundamentals
2. Understand functional programming concepts
3. Practice combining both paradigms
4. Study design patterns
5. Solve complex problems using both approaches