Inheritance, Polymorphism, Abstract Classes & Interfaces

Object-Oriented Programming in Java

State University of Zanzibar (SUZA)
BSc Computer Science

Contents

1 Inheritance

1.1 What is Inheritance?

Definition

Inheritance is a fundamental OOP concept where a new class (child/subclass) is created from an existing class (parent/superclass). The child class inherits all the attributes and methods of the parent class and can add its own unique features.

1.2 Why Use Inheritance?

- **Code Reusability:** Write code once in the parent class, use it in multiple child classes
- **Method Overriding:** Child classes can provide specific implementations
- **Hierarchical Classification:** Organize classes in a logical hierarchy
- **Extensibility:** Extend existing functionality without modifying original code

1.3 Inheritance Syntax in Java

```java
// Parent class (Superclass)
class Animal {
    String name;
    int age;

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

    public void sleep() {
        System.out.println(name + " is sleeping.");
    }
}

// Child class (Subclass)
class Dog extends Animal {
    String breed;

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

1.4 The `extends` Keyword

In Java, we use the `extends` keyword to create inheritance:

```
class ChildClass extends ParentClass {
    // Child class body
}
```

1.5 Types of Inheritance

1.5.1 1. Single Inheritance

One child class inherits from one parent class.

```
class Vehicle {
    String brand;
    int year;

    public void start() {
        System.out.println("Vehicle starting...");
    }
}

class Car extends Vehicle {
    int numberOfDoors;

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

1.5.2 2. Multilevel Inheritance

A class inherits from a class that also inherits from another class.

```
class Animal {
    public void breathe() {
        System.out.println("Breathing...");
    }
}

class Mammal extends Animal {
    public void feedMilk() {
        System.out.println("Feeding milk to young ones.");
    }
}

class Dog extends Mammal {
    public void bark() {
        System.out.println("Barking...");
    }
}

// Dog can use breathe(), feedMilk(), and bark()
```

1.5.3 3. Hierarchical Inheritance

Multiple child classes inherit from a single parent class.

```
class Shape {
    String color;

    public void draw() {
        System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {
    double radius;

    public double getArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double width, height;

    public double getArea() {
        return width * height;
    }
}
```

Important Note

Java does NOT support multiple inheritance with classes (a class cannot extend more than one class). This is to avoid the "Diamond Problem". However, Java supports multiple inheritance through interfaces.

1.6 The super Keyword

The `super` keyword refers to the parent class. It is used to:

1.6.1 1. Call Parent Constructor

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

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

class Student extends Person {
```

```
    String studentId;

    public Student(String name, int age, String studentId) {
        super(name, age);  // Call parent constructor
        this.studentId = studentId;
    }
}
```

1.6.2 2. Access Parent Methods

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

class Cat extends Animal {
    @Override
    public void makeSound() {
        super.makeSound();  // Call parent method first
        System.out.println("Meow!");
    }
}
```

1.6.3 3. Access Parent Variables

```
class Parent {
    String message = "Hello from Parent";
}

class Child extends Parent {
    String message = "Hello from Child";

    public void displayMessages() {
        System.out.println(message);        // Child's message
        System.out.println(super.message);  // Parent's message
    }
}
```

1.7 Complete Inheritance Example

```
// Base class
class Employee {
    private String name;
    private String employeeId;
    protected double baseSalary;

```

```
    public Employee(String name, String employeeId, double
       baseSalary) {
        this.name = name;
        this.employeeId = employeeId;
        this.baseSalary = baseSalary;
    }

    public String getName() {
        return name;
    }

    public double calculateSalary() {
        return baseSalary;
    }

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("ID: " + employeeId);
        System.out.println("Salary: $" + calculateSalary());
    }
}

// Derived class
class Manager extends Employee {
    private double bonus;
    private int teamSize;

    public Manager(String name, String id, double salary,
                   double bonus, int teamSize) {
        super(name, id, salary);
        this.bonus = bonus;
        this.teamSize = teamSize;
    }

    @Override
    public double calculateSalary() {
        return baseSalary + bonus;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Team Size: " + teamSize);
        System.out.println("Bonus: $" + bonus);
    }
}

// Main class
public class InheritanceDemo {
    public static void main(String[] args) {
        Employee emp = new Employee("John", "E001", 50000);
```

```
57          Manager mgr = new Manager("Alice", "M001", 70000, 15000,
                10);
58
59          System.out.println("=== Employee ===");
60          emp.displayInfo();
61
62          System.out.println("\n=== Manager ===");
63          mgr.displayInfo();
64      }
65  }
```

Output:

```
=== Employee ===
Name: John
ID: E001
Salary: $50000.0

=== Manager ===
Name: Alice
ID: M001
Salary: $85000.0
Team Size: 10
Bonus: $15000.0
```

2 Polymorphism

2.1 What is Polymorphism?

Definition

Polymorphism means "many forms". In OOP, it refers to the ability of objects to take on multiple forms. The same method name can behave differently based on which object calls it or what parameters are passed.

2.2 Types of Polymorphism

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

2.3 Method Overloading (Compile-time Polymorphism)

Method overloading allows multiple methods with the **same name** but **different parameters** in the same class.

```java
class Calculator {
    // Method with two int parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Method with three int parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with two double parameters
    public double add(double a, double b) {
        return a + b;
    }

    // Method with String parameters (concatenation)
    public String add(String a, String b) {
        return a + b;
    }
}

public class OverloadingDemo {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        System.out.println(calc.add(5, 10));           // 15
        System.out.println(calc.add(5, 10, 15));       // 30
        System.out.println(calc.add(5.5, 10.5));       // 16.0
```

```
30          System.out.println(calc.add("Hello ", "World")); // Hello
                World
31      }
32  }
```

Rules for Method Overloading

- Methods must have the **same name**
- Methods must have **different parameter lists** (number, type, or order)
- Return type alone is NOT sufficient to overload a method
- Access modifiers can be different

2.4 Method Overriding (Runtime Polymorphism)

Method overriding occurs when a child class provides a **specific implementation** of a method that is already defined in its parent class.

```
1   class Animal {
2       public void makeSound() {
3           System.out.println("Animal makes a sound");
4       }
5
6       public void move() {
7           System.out.println("Animal moves");
8       }
9   }
10
11  class Dog extends Animal {
12      @Override
13      public void makeSound() {
14          System.out.println("Dog barks: Woof! Woof!");
15      }
16
17      @Override
18      public void move() {
19          System.out.println("Dog runs on four legs");
20      }
21  }
22
23  class Bird extends Animal {
24      @Override
25      public void makeSound() {
26          System.out.println("Bird chirps: Tweet! Tweet!");
27      }
28
29      @Override
30      public void move() {
31          System.out.println("Bird flies with wings");
32      }
```

```
}

class Fish extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Fish doesn't make sound");
    }

    @Override
    public void move() {
        System.out.println("Fish swims in water");
    }
}
```

Rules for Method Overriding

- Method must have the **same name and parameters**
- Method must have the **same or covariant return type**
- Method cannot have a more restrictive access modifier
- Use `@Override` annotation (recommended)
- `static`, `final`, and `private` methods cannot be overridden

2.5 Polymorphism in Action

The power of polymorphism is shown when a parent reference holds a child object:

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        // Parent reference, Child object
        Animal animal1 = new Dog();
        Animal animal2 = new Bird();
        Animal animal3 = new Fish();

        // Same method call, different behavior
        animal1.makeSound();  // Dog barks: Woof! Woof!
        animal2.makeSound();  // Bird chirps: Tweet! Tweet!
        animal3.makeSound();  // Fish doesn't make sound

        System.out.println();

        animal1.move();  // Dog runs on four legs
        animal2.move();  // Bird flies with wings
        animal3.move();  // Fish swims in water

        // Using an array of Animals
        System.out.println("\n--- Animal Orchestra ---");
        Animal[] animals = {new Dog(), new Bird(), new Fish()};

```

```
        for (Animal a : animals) {
            a.makeSound();
        }
    }
}
```

2.6 Real-World Example: Payment System

```
class Payment {
    protected double amount;

    public Payment(double amount) {
        this.amount = amount;
    }

    public void processPayment() {
        System.out.println("Processing generic payment of $" +
            amount);
    }

    public void printReceipt() {
        System.out.println("Receipt: $" + amount + " paid.");
    }
}

class CreditCardPayment extends Payment {
    private String cardNumber;

    public CreditCardPayment(double amount, String cardNumber) {
        super(amount);
        this.cardNumber = cardNumber;
    }

    @Override
    public void processPayment() {
        System.out.println("Processing credit card payment...");
        System.out.println("Card: **** **** **** " +
                           cardNumber.substring(cardNumber.length()
                               - 4));
        System.out.println("Amount: $" + amount);
        System.out.println("Payment successful!");
    }
}

class MobilePayment extends Payment {
    private String phoneNumber;

    public MobilePayment(double amount, String phoneNumber) {
        super(amount);
        this.phoneNumber = phoneNumber;
```

```
41      }
42
43      @Override
44      public void processPayment() {
45          System.out.println("Processing mobile money payment...");
46          System.out.println("Phone: " + phoneNumber);
47          System.out.println("Amount: $" + amount);
48          System.out.println("Check your phone for confirmation.");
49      }
50  }
51
52  class BankTransfer extends Payment {
53      private String accountNumber;
54
55      public BankTransfer(double amount, String accountNumber) {
56          super(amount);
57          this.accountNumber = accountNumber;
58      }
59
60      @Override
61      public void processPayment() {
62          System.out.println("Processing bank transfer...");
63          System.out.println("Account: " + accountNumber);
64          System.out.println("Amount: $" + amount);
65          System.out.println("Transfer initiated. Allow 1-3 business
                days.");
66      }
67  }
68
69  public class PaymentDemo {
70      public static void main(String[] args) {
71          // Polymorphism: Different payment types
72          Payment[] payments = {
73              new CreditCardPayment(150.00, "1234567890123456"),
74              new MobilePayment(75.50, "+255-123-456-789"),
75              new BankTransfer(500.00, "ACC-001234567")
76          };
77
78          for (Payment p : payments) {
79              p.processPayment();
80              p.printReceipt();
81              System.out.println("--------------------");
82          }
83      }
84  }
```

3 Abstract Classes

3.1 What is an Abstract Class?

Definition

An **Abstract Class** is a class that cannot be instantiated (you cannot create objects from it). It serves as a blueprint for other classes and can contain both abstract methods (without implementation) and concrete methods (with implementation).

3.2 Abstract Class Syntax

```
abstract class ClassName {
    // Regular variables
    private String name;

    // Constructor
    public ClassName(String name) {
        this.name = name;
    }

    // Abstract method (no body)
    public abstract void abstractMethod();

    // Concrete method (has body)
    public void concreteMethod() {
        System.out.println("This is a concrete method");
    }
}
```

3.3 Key Characteristics of Abstract Classes

- Declared with the `abstract` keyword
- **Cannot be instantiated** directly
- Can have abstract methods (must be overridden by subclasses)
- Can have concrete methods (inherited by subclasses)
- Can have constructors, variables, and static methods
- A class that extends an abstract class must implement all abstract methods OR be declared abstract itself

3.4 Abstract Class Example: Shape

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

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

    // Abstract methods - must be implemented by subclasses
    public abstract double calculateArea();
    public abstract double calculatePerimeter();

    // Concrete method - inherited by subclasses
    public void displayColor() {
        System.out.println("Color: " + color);
    }

    public void displayInfo() {
        displayColor();
        System.out.println("Area: " + calculateArea());
        System.out.println("Perimeter: " + calculatePerimeter());
    }
}

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;
    }

    @Override
    public double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
}

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

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

```
52
53      @Override
54      public double calculateArea() {
55          return width * height;
56      }
57
58      @Override
59      public double calculatePerimeter() {
60          return 2 * (width + height);
61      }
62  }
63
64  class Triangle extends Shape {
65      private double base;
66      private double height;
67      private double side1, side2, side3;
68
69      public Triangle(String color, double base, double height,
70                      double s1, double s2, double s3) {
71          super(color);
72          this.base = base;
73          this.height = height;
74          this.side1 = s1;
75          this.side2 = s2;
76          this.side3 = s3;
77      }
78
79      @Override
80      public double calculateArea() {
81          return 0.5 * base * height;
82      }
83
84      @Override
85      public double calculatePerimeter() {
86          return side1 + side2 + side3;
87      }
88  }
89
90  public class AbstractShapeDemo {
91      public static void main(String[] args) {
92          // Cannot do: Shape s = new Shape("Red"); // Error!
93
94          Shape circle = new Circle("Red", 5.0);
95          Shape rectangle = new Rectangle("Blue", 4.0, 6.0);
96          Shape triangle = new Triangle("Green", 3.0, 4.0, 3.0, 4.0,
                5.0);
97
98          System.out.println("=== Circle ===");
99          circle.displayInfo();
100
101         System.out.println("\n=== Rectangle ===");
```

```
102         rectangle.displayInfo();
103
104         System.out.println("\n=== Triangle ===");
105         triangle.displayInfo();
106     }
107 }
```

Output:

```
=== Circle ===
Color: Red
Area: 78.53981633974483
Perimeter: 31.41592653589793

=== Rectangle ===
Color: Blue
Area: 24.0
Perimeter: 20.0

=== Triangle ===
Color: Green
Area: 6.0
Perimeter: 12.0
```

4 Interfaces

4.1 What is an Interface?

Definition

An **Interface** is a completely abstract type that defines a contract of methods that implementing classes must follow. It specifies WHAT a class must do, but not HOW it does it.

4.2 Interface Syntax

```java
interface InterfaceName {
    // Constants (implicitly public static final)
    int MAX_VALUE = 100;

    // Abstract methods (implicitly public abstract)
    void method1();
    int method2(String param);

    // Default method (Java 8+)
    default void defaultMethod() {
        System.out.println("Default implementation");
    }

    // Static method (Java 8+)
    static void staticMethod() {
        System.out.println("Static method in interface");
    }
}
```

4.3 Implementing an Interface

```java
interface Drawable {
    void draw();
    void resize(int percentage);
}

class Circle implements Drawable {
    private double radius;

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

    @Override
    public void draw() {
        System.out.println("Drawing a circle with radius " + radius
            );
```

```
    }

    @Override
    public void resize(int percentage) {
        radius = radius * percentage / 100;
        System.out.println("Circle resized. New radius: " + radius)
            ;
    }
}
```

4.4 Multiple Interface Implementation

A class can implement multiple interfaces (Java's way of achieving multiple inheritance):

```
interface Playable {
    void play();
    void stop();
}

interface Recordable {
    void record();
    void stopRecording();
}

interface Streamable {
    void stream(String url);
}

// A class implementing multiple interfaces
class MediaPlayer implements Playable, Recordable, Streamable {
    private String currentMedia;
    private boolean isPlaying = false;
    private boolean isRecording = false;

    @Override
    public void play() {
        isPlaying = true;
        System.out.println("Playing media...");
    }

    @Override
    public void stop() {
        isPlaying = false;
        System.out.println("Stopped playing.");
    }

    @Override
    public void record() {
        isRecording = true;
        System.out.println("Recording started...");
    }
```

```
38
39      @Override
40      public void stopRecording() {
41          isRecording = false;
42          System.out.println("Recording stopped.");
43      }
44
45      @Override
46      public void stream(String url) {
47          System.out.println("Streaming from: " + url);
48      }
49  }
```

4.5 Interface vs Abstract Class

Abstract Class	Interface
Can have abstract and concrete methods	All methods are abstract (except default/static)
Can have instance variables	Can only have constants (static final)
Can have constructors	Cannot have constructors
Single inheritance only	Multiple implementation allowed
Use `extends` keyword	Use `implements` keyword
Can have any access modifier	Methods are implicitly public
Represents "is-a" relationship	Represents "can-do" capability

4.6 Real-World Example: E-commerce System

```
1  // Interfaces defining capabilities
2  interface Sellable {
3      double getPrice();
4      void applyDiscount(double percentage);
5  }
6
7  interface Shippable {
8      double getWeight();
9      String getShippingDimensions();
10     double calculateShippingCost(String destination);
11 }
12
13 interface Reviewable {
14     void addReview(String review, int rating);
15     double getAverageRating();
16 }
17
18 // Product class implementing multiple interfaces
19 class Product implements Sellable, Shippable, Reviewable {
20     private String name;
21     private double price;
```

```
22      private double weight;
23      private String dimensions;
24      private java.util.List<Integer> ratings = new java.util.
            ArrayList<>();
25
26      public Product(String name, double price, double weight, String
            dims) {
27          this.name = name;
28          this.price = price;
29          this.weight = weight;
30          this.dimensions = dims;
31      }
32
33      // Sellable implementation
34      @Override
35      public double getPrice() {
36          return price;
37      }
38
39      @Override
40      public void applyDiscount(double percentage) {
41          price = price * (1 - percentage / 100);
42          System.out.println(name + " new price: $" + price);
43      }
44
45      // Shippable implementation
46      @Override
47      public double getWeight() {
48          return weight;
49      }
50
51      @Override
52      public String getShippingDimensions() {
53          return dimensions;
54      }
55
56      @Override
57      public double calculateShippingCost(String destination) {
58          double baseCost = weight * 0.5;
59          if (destination.equals("international")) {
60              baseCost *= 3;
61          }
62          return baseCost;
63      }
64
65      // Reviewable implementation
66      @Override
67      public void addReview(String review, int rating) {
68          ratings.add(rating);
69          System.out.println("Review added: " + review + " (" +
                rating + "/5)");
```

```
70      }
71
72      @Override
73      public double getAverageRating() {
74          if (ratings.isEmpty()) return 0;
75          int sum = 0;
76          for (int r : ratings) sum += r;
77          return (double) sum / ratings.size();
78      }
79
80      public void displayInfo() {
81          System.out.println("Product: " + name);
82          System.out.println("Price: $" + price);
83          System.out.println("Weight: " + weight + " kg");
84          System.out.println("Rating: " + getAverageRating() + "/5");
85      }
86  }
87
88  public class InterfaceDemo {
89      public static void main(String[] args) {
90          Product laptop = new Product("Laptop Pro", 1200.00, 2.5, "
                35x25x2 cm");
91
92          laptop.displayInfo();
93          System.out.println();
94
95          laptop.addReview("Great performance!", 5);
96          laptop.addReview("Good value", 4);
97          laptop.addReview("Battery could be better", 3);
98
99          System.out.println("\nAverage Rating: " + laptop.
                getAverageRating());
100
101         laptop.applyDiscount(10);
102
103         System.out.println("\nShipping cost (local): $" +
104                                 laptop.calculateShippingCost("local"));
105         System.out.println("Shipping cost (international): $" +
106                                 laptop.calculateShippingCost("
                                    international"));
107     }
108 }
```

5 Combining All Concepts

5.1 Comprehensive Example: Animal Shelter System

```
// Interface for adoptable animals
interface Adoptable {
    boolean isAvailableForAdoption();
    void adopt(String ownerName);
    double getAdoptionFee();
}

// Interface for animals that can be trained
interface Trainable {
    void train(String command);
    String[] getLearnedCommands();
}

// Abstract base class
abstract class Animal {
    protected String name;
    protected int age;
    protected String species;

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

    public abstract void makeSound();
    public abstract void eat();

    public void sleep() {
        System.out.println(name + " is sleeping. Zzz...");
    }

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

// Dog class - extends Animal, implements interfaces
class Dog extends Animal implements Adoptable, Trainable {
    private String breed;
    private boolean adopted = false;
    private String ownerName;
    private java.util.List<String> commands = new java.util.
       ArrayList<>();

```

```
47      public Dog(String name, int age, String breed) {
48          super(name, age, "Dog");
49          this.breed = breed;
50      }
51
52      @Override
53      public void makeSound() {
54          System.out.println(name + " barks: Woof! Woof!");
55      }
56
57      @Override
58      public void eat() {
59          System.out.println(name + " is eating dog food.");
60      }
61
62      // Adoptable implementation
63      @Override
64      public boolean isAvailableForAdoption() {
65          return !adopted;
66      }
67
68      @Override
69      public void adopt(String ownerName) {
70          this.adopted = true;
71          this.ownerName = ownerName;
72          System.out.println(name + " has been adopted by " +
                ownerName + "!");
73      }
74
75      @Override
76      public double getAdoptionFee() {
77          return 150.00;
78      }
79
80      // Trainable implementation
81      @Override
82      public void train(String command) {
83          commands.add(command);
84          System.out.println(name + " learned: " + command);
85      }
86
87      @Override
88      public String[] getLearnedCommands() {
89          return commands.toArray(new String[0]);
90      }
91
92      @Override
93      public void displayInfo() {
94          super.displayInfo();
95          System.out.println("Breed: " + breed);
```

```
96          System.out.println("Adopted: " + (adopted ? "Yes by " +
                ownerName : "No"));
97      }
98  }
99
100 // Cat class - extends Animal, implements Adoptable only
101 class Cat extends Animal implements Adoptable {
102     private boolean isIndoor;
103     private boolean adopted = false;
104
105     public Cat(String name, int age, boolean isIndoor) {
106         super(name, age, "Cat");
107         this.isIndoor = isIndoor;
108     }
109
110     @Override
111     public void makeSound() {
112         System.out.println(name + " meows: Meow!");
113     }
114
115     @Override
116     public void eat() {
117         System.out.println(name + " is eating cat food.");
118     }
119
120     @Override
121     public boolean isAvailableForAdoption() {
122         return !adopted;
123     }
124
125     @Override
126     public void adopt(String ownerName) {
127         this.adopted = true;
128         System.out.println(name + " has been adopted by " +
                ownerName + "!");
129     }
130
131     @Override
132     public double getAdoptionFee() {
133         return 100.00;
134     }
135 }
136
137 // Main demonstration
138 public class AnimalShelterDemo {
139     public static void main(String[] args) {
140         // Create animals
141         Dog buddy = new Dog("Buddy", 3, "Golden Retriever");
142         Dog max = new Dog("Max", 2, "German Shepherd");
143         Cat whiskers = new Cat("Whiskers", 4, true);
144
```

```
145         System.out.println("=== Animal Shelter System ===\n");
146
147         // Polymorphism with Animal reference
148         Animal[] animals = {buddy, max, whiskers};
149
150         System.out.println("--- All Animals ---");
151         for (Animal a : animals) {
152             a.displayInfo();
153             a.makeSound();
154             System.out.println();
155         }
156
157         // Polymorphism with interface reference
158         System.out.println("--- Adoptable Animals ---");
159         Adoptable[] adoptables = {buddy, max, whiskers};
160
161         for (Adoptable a : adoptables) {
162             if (a.isAvailableForAdoption()) {
163                 System.out.println("Adoption fee: $" + a.
                        getAdoptionFee());
164             }
165         }
166
167         System.out.println("\n--- Training Session ---");
168         buddy.train("Sit");
169         buddy.train("Stay");
170         buddy.train("Fetch");
171
172         System.out.println("\n" + buddy.name + "'s commands:");
173         for (String cmd : buddy.getLearnedCommands()) {
174             System.out.println("  - " + cmd);
175         }
176
177         System.out.println("\n--- Adoption Process ---");
178         buddy.adopt("John Smith");
179         whiskers.adopt("Jane Doe");
180
181         System.out.println("\n--- Updated Status ---");
182         buddy.displayInfo();
183     }
184 }
```

6 Exercises

Exercise 1: Basic Inheritance

Create a class hierarchy for a university system:

1. Create a base class `Person` with attributes: `name`, `age`, `email`
2. Create a derived class `Student` that adds: `studentId`, `major`, `gpa`
3. Create a derived class `Professor` that adds: `employeeId`, `department`, `salary`
4. Implement appropriate constructors using `super`
5. Override `toString()` method in each class
6. Create a main method to test your classes

Exercise 2: Method Overloading

Create a `MathOperations` class with overloaded methods:

1. `multiply(int, int)` - returns product of two integers
2. `multiply(int, int, int)` - returns product of three integers
3. `multiply(double, double)` - returns product of two doubles
4. `multiply(int[], int)` - multiplies each array element by the integer
5. Test all methods in main

Exercise 3: Method Overriding and Polymorphism

Create a vehicle rental system:

1. Base class `Vehicle` with: `brand`, `model`, `dailyRate`
2. Method `calculateRentalCost(int days)`
3. Derived classes: `Car`, `Motorcycle`, `Truck`
4. Override `calculateRentalCost()` with different rates:
   - Car: base rate
   - Motorcycle: 20% discount
   - Truck: 50% surcharge
5. Use polymorphism to process an array of vehicles

Exercise 4: Abstract Class

Create an abstract class for a banking system:

1. Abstract class `BankAccount` with:
   - Attributes: `accountNumber`, `balance`, `ownerName`
   - Abstract methods: `withdraw()`, `calculateInterest()`
   - Concrete methods: `deposit()`, `getBalance()`
2. Concrete class `SavingsAccount`:
   - Interest rate: 4% annually
   - Minimum balance: $100
3. Concrete class `CheckingAccount`:
   - No interest
   - Overdraft limit: $500

Exercise 5: Interfaces

Design a smart home system using interfaces:

1. Interface `Switchable`: `turnOn()`, `turnOff()`, `isOn()`
2. Interface `Adjustable`: `increaseLevel()`, `decreaseLevel()`, `getLevel()`
3. Interface `Programmable`: `setTimer(int minutes)`, `cancelTimer()`
4. Create classes implementing these interfaces:
   - `SmartLight` - implements all three
   - `SmartFan` - implements Switchable and Adjustable
   - `SmartTV` - implements Switchable and Programmable
5. Demonstrate polymorphism using interface references

Exercise 6: Comprehensive Challenge

Create a complete Library Management System:

1. Abstract class `LibraryItem` with: `title`, `itemId`, `isAvailable`
2. Interface `Borrowable`: `borrow()`, `returnItem()`, `getDueDate()`
3. Interface `Reservable`: `reserve()`, `cancelReservation()`
4. Classes: `Book`, `DVD`, `Magazine`
   - Book: implements Borrowable and Reservable
   - DVD: implements Borrowable only
   - Magazine: implements neither (reference only)
5. Each class should override appropriate methods
6. Create a `Library` class that manages all items
7. Demonstrate all concepts learned in this lecture

7 Summary

7.1 Key Takeaways

- **Inheritance** allows code reuse through parent-child relationships using `extends`
- **Polymorphism** enables objects to take multiple forms through overloading and overriding
- **Abstract Classes** provide partial implementation and define contracts for subclasses
- **Interfaces** define pure contracts and enable multiple inheritance of behavior

7.2 When to Use What

Concept	Use When
Inheritance	Classes share common attributes and have "is-a" relationship
Method Overloading	Same operation needs to work with different parameters
Method Overriding	Subclass needs to provide specific implementation
Abstract Class	Want to share code among related classes with some abstract behavior
Interface	Want to define a contract that unrelated classes can implement

7.3 Best Practices

1. Always use `@Override` annotation when overriding methods
2. Prefer composition over inheritance when appropriate
3. Program to interfaces, not implementations
4. Keep inheritance hierarchies shallow (2-3 levels max)
5. Use abstract classes for related classes; interfaces for unrelated classes
6. Follow the Liskov Substitution Principle: subclasses should be substitutable for their base classes

Practice these concepts by completing the exercises. Understanding inheritance, polymorphism, abstract classes, and interfaces is fundamental to mastering object-oriented programming.