Introduction to Java and Core OOP Concepts

Dr. Ratnesh Prasad Srivastava
Department of CSIT, GGV, Bilaspur (C.G.)

Academic Year: 2026-27

Course Information

Course Code	CIUDMJT1
Course Title	Object-Oriented Programming with Java
Credit Hours	3-0-3 (3 Lecture, 0 Tutorial, 3 Practical)
Prerequisites	Programming Fundamentals
Textbook	"Java: The Complete Reference" by Herbert Schildt
Reference	"Head First Java" by Kathy Sierra and Bert Bates

Contents

1 Unit II: Advanced OOP Principles in Java

1.1 Learning Objectives

- Master inheritance concepts and implement different types of inheritance hierarchies
- Understand and implement polymorphism through method overloading and overriding
- Design and implement abstract classes and interfaces effectively
- Create and use packages for modular code organization
- Utilize static members for class-level operations
- Apply final keyword for creating constants and preventing inheritance/method overriding
- Design robust object-oriented systems using advanced OOP principles

2 Inheritance

2.1 Introduction to Inheritance

Inheritance Concept

Inheritance is an IS-A relationship where a new class (child/subclass) acquires properties and behaviors of an existing class (parent/superclass). It promotes **code reusability** and establishes a **hierarchical relationship** between classes.

2.2 The 'extends' Keyword

```java
// ==================== PARENT CLASS ====================
class Vehicle {
    // Protected members are accessible in child classes
    protected String brand;
    protected int year;
    protected double price;

    public Vehicle(String brand, int year, double price) {
        this.brand = brand;
        this.year = year;
        this.price = price;
        System.out.println("Vehicle constructor called");
    }

    public void displayInfo() {
        System.out.println("Brand: " + brand + ", Year: " + year +
                           ", Price: $" + price);
    }

    public void startEngine() {
        System.out.println("Vehicle engine starting...");
    }
```

```
23
24      public double calculateDepreciation(int currentYear) {
25          int age = currentYear - year;
26          return price * (0.1 * age);  // 10% depreciation per year
27      }
28  }
29
30  // ==================== CHILD CLASS ====================
31  // Line 33: Car class EXTENDS Vehicle class
32  // 'extends' establishes inheritance relationship
33  // Car IS-A Vehicle
34  class Car extends Vehicle {
35      // Additional attributes specific to Car
36      private int numberOfDoors;
37      private String fuelType;
38
39      // Line 40: Constructor with super() call
40      public Car(String brand, int year, double price,
41                 int doors, String fuel) {
42          // Line 43: super() MUST be first statement in constructor
43          // Calls parent class constructor
44          super(brand, year, price);
45
46          this.numberOfDoors = doors;
47          this.fuelType = fuel;
48          System.out.println("Car constructor called");
49      }
50
51      // Additional methods specific to Car
52      public void openTrunk() {
53          System.out.println("Car trunk opened");
54      }
55
56      // Line 54: Method overriding - same signature as parent
57      @Override
58      public void displayInfo() {
59          // Line 56: Call parent class method using super
60          super.displayInfo();  // Reuse parent's implementation
61          System.out.println("Doors: " + numberOfDoors +
62                             ", Fuel: " + fuelType);
63      }
64
65      // Line 61: Additional method not in parent
66      public void checkFuel() {
67          System.out.println("Checking " + fuelType + " fuel level...");
68      }
69  }
70
71  // ==================== MAIN CLASS ====================
72  public class InheritanceDemo {
73      public static void main(String[] args) {
74          System.out.println("=== BASIC INHERITANCE DEMO ===\n");
75
76          // Line 71: Creating Car object
77          // Car inherits all non-private members from Vehicle
78          Car myCar = new Car("Toyota", 2020, 25000.0, 4, "Petrol");
79
80          System.out.println("\n--- Accessing Inherited Members ---");
```

```
81          // Line 75: Accessing inherited method
82          myCar.startEngine();  // Inherited from Vehicle
83
84          // Line 77: Accessing inherited protected member (through
                public method)
85          System.out.println("Brand (via inherited method): " + myCar.
                brand);
86
87          // Line 80: Calling overridden method
88          myCar.displayInfo();  // Calls Car's version
89
90          // Line 83: Calling child-specific method
91          myCar.openTrunk();
92          myCar.checkFuel();
93
94          // Line 87: Using inherited calculation method
95          double depreciation = myCar.calculateDepreciation(2024);
96          System.out.printf("\nDepreciation after 4 years: $%.2f\n",
                depreciation);
97
98          // Line 91: Demonstrating IS-A relationship
99          System.out.println("\n--- IS-A Relationship Test ---");
100         System.out.println("myCar instanceof Car: " + (myCar instanceof
                Car));
101         System.out.println("myCar instanceof Vehicle: " + (myCar
                instanceof Vehicle));
102         System.out.println("myCar instanceof Object: " + (myCar
                instanceof Object));
103     }
104 }
```

Listing 1: Basic Inheritance with 'extends' Keyword

InheritanceDemo Program Output

```
=== BASIC INHERITANCE DEMO ===

Vehicle constructor called
Car constructor called

--- Accessing Inherited Members ---
Vehicle engine starting...
Brand (via inherited method): Toyota
Brand: Toyota, Year: 2020, Price: $25000.0
Doors: 4, Fuel: Petrol
Car trunk opened
Checking Petrol fuel level...

Depreciation after 4 years: $10000.00

--- IS-A Relationship Test ---
myCar instanceof Car: true
myCar instanceof Vehicle: true
myCar instanceof Object: true
```

2.3 Types of Inheritance

```java
// ============= SINGLE INHERITANCE =============
class Animal {
    protected String name;
    protected int age;

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

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

// Single Inheritance: Dog      Animal
class Dog extends Animal {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }

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

```

```java
// ============= MULTILEVEL INHERITANCE =============
class Person {
    protected String name;
    protected int id;

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

    public void displayPerson() {
        System.out.println("ID: " + id + ", Name: " + name);
    }
}

// Employee IS-A Person (First level)
class Employee extends Person {
    protected String department;
    protected double salary;

    public Employee(String name, int id, String dept, double salary) {
        super(name, id);
        this.department = dept;
        this.salary = salary;
    }

    public void displayEmployee() {
        displayPerson();
        System.out.println("Dept: " + department + ", Salary: $" +
            salary);
    }
}

// Manager IS-A Employee (Second level) IS-A Person (Third level)
class Manager extends Employee {
    private int teamSize;

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

    public void conductMeeting() {
        System.out.println(name + " is conducting meeting with " +
                           teamSize + " team members");
    }

    @Override
    public void displayEmployee() {
        super.displayEmployee();
        System.out.println("Team Size: " + teamSize + ", Role: Manager"
            );
    }
}

// ============= HIERARCHICAL INHERITANCE =============
class Shape {
```

```
85      protected String color;
86
87      public Shape(String color) {
88          this.color = color;
89      }
90
91      public double calculateArea() {
92          return 0;  // Default implementation
93      }
94
95      public void display() {
96          System.out.println("Color: " + color);
97      }
98  }
99
100 // Multiple classes inheriting from Shape (Hierarchical)
101 class Circle extends Shape {
102     private double radius;
103
104     public Circle(String color, double radius) {
105         super(color);
106         this.radius = radius;
107     }
108
109     @Override
110     public double calculateArea() {
111         return Math.PI * radius * radius;
112     }
113
114     @Override
115     public void display() {
116         super.display();
117         System.out.println("Shape: Circle, Radius: " + radius +
118                            ", Area: " + calculateArea());
119     }
120 }
121
122 class Rectangle extends Shape {
123     private double length;
124     private double width;
125
126     public Rectangle(String color, double length, double width) {
127         super(color);
128         this.length = length;
129         this.width = width;
130     }
131
132     @Override
133     public double calculateArea() {
134         return length * width;
135     }
136
137     @Override
138     public void display() {
139         super.display();
140         System.out.println("Shape: Rectangle, Length: " + length +
141                            ", Width: " + width + ", Area: " +
                                calculateArea());
```

```
    }
}

class Triangle extends Shape {
    private double base;
    private double height;

    public Triangle(String color, double base, double height) {
        super(color);
        this.base = base;
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return 0.5 * base * height;
    }

    @Override
    public void display() {
        super.display();
        System.out.println("Shape: Triangle, Base: " + base +
                           ", Height: " + height + ", Area: " +
                              calculateArea());
    }
}

// ============= HYBRID INHERITANCE (using interfaces) =============
// Note: Java doesn't support multiple inheritance with classes
// We achieve it through interfaces (shown in interfaces section)

// ==================== MAIN CLASS ====================
public class InheritanceTypesDemo {
    public static void main(String[] args) {
        System.out.println("=== SINGLE INHERITANCE ===");
        Dog dog = new Dog("Buddy", 3, "Golden Retriever");
        dog.eat();   // Inherited from Animal
        dog.bark();  // Dog's own method
        System.out.println();

        System.out.println("=== MULTILEVEL INHERITANCE ===");
        Manager mgr = new Manager("Alice", 101, "IT", 85000, 10);
        mgr.displayPerson();     // From Person class
        mgr.displayEmployee();   // From Employee class (overridden)
        mgr.conductMeeting();    // Manager's own method

        // Demonstrating upcasting
        Person p = mgr;  // Manager IS-A Person
        System.out.println("\nUpcasting Manager to Person:");
        p.displayPerson();

        // Demonstrating downcasting
        if (p instanceof Manager) {
            Manager m = (Manager) p;  // Downcasting
            m.conductMeeting();
        }
        System.out.println();

```

```
        System.out.println("=== HIERARCHICAL INHERITANCE ===");
        Shape[] shapes = new Shape[3];
        shapes[0] = new Circle("Red", 5.0);
        shapes[1] = new Rectangle("Blue", 4.0, 6.0);
        shapes[2] = new Triangle("Green", 3.0, 4.0);

        double totalArea = 0;
        for (Shape shape : shapes) {
            shape.display();
            totalArea += shape.calculateArea();
            System.out.println();
        }
        System.out.println("Total Area of all shapes: " + totalArea);

        // Demonstrating polymorphism
        System.out.println("\n=== POLYMORPHIC BEHAVIOR ===");
        Shape shape1 = new Circle("Yellow", 7.0);
        Shape shape2 = new Rectangle("Purple", 5.0, 3.0);

        // Same method call, different behaviors
        System.out.println("Circle Area: " + shape1.calculateArea());
        System.out.println("Rectangle Area: " + shape2.calculateArea())
            ;
    }
}
```

Listing 2: All Types of Inheritance Implementation

InheritanceTypesDemo Program Output

```
=== SINGLE INHERITANCE ===
Buddy is eating
Buddy (a Golden Retriever) is barking

=== MULTILEVEL INHERITANCE ===
ID: 101, Name: Alice
ID: 101, Name: Alice
Dept: IT, Salary: $85000.0
Team Size: 10, Role: Manager
Alice is conducting meeting with 10 team members

Upcasting Manager to Person:
ID: 101, Name: Alice
Alice is conducting meeting with 10 team members

=== HIERARCHICAL INHERITANCE ===
Color: Red
Shape: Circle, Radius: 5.0, Area: 78.53981633974483

Color: Blue
Shape: Rectangle, Length: 4.0, Width: 6.0, Area: 24.0

Color: Green
Shape: Triangle, Base: 3.0, Height: 4.0, Area: 6.0

Total Area of all shapes: 108.53981633974483

=== POLYMORPHIC BEHAVIOR ===
Circle Area: 153.93804002589985
Rectangle Area: 15.0
```

2.4 The 'super' Keyword

```java
// ==================== PARENT CLASS ====================
class Employee {
    protected int id;
    protected String name;
    protected double baseSalary;

    public Employee(int id, String name, double baseSalary) {
        System.out.println("Employee constructor called");
        this.id = id;
        this.name = name;
        this.baseSalary = baseSalary;
    }

    public double calculateSalary() {
        return baseSalary;
```

```
    }

    public void displayInfo() {
        System.out.println("ID: " + id + ", Name: " + name +
                           ", Base Salary: $" + baseSalary);
    }

    public final void companyPolicy() {
        System.out.println("All employees must follow company policy");
    }
}

// ==================== CHILD CLASS ====================
class Manager extends Employee {
    private double bonusPercentage;
    private int teamSize;

    // Line 35: Constructor using super() to call parent constructor
    public Manager(int id, String name, double baseSalary,
                   double bonusPercentage, int teamSize) {
        // Line 38: super() MUST be first statement
        super(id, name, baseSalary);  // Calls Employee constructor
        this.bonusPercentage = bonusPercentage;
        this.teamSize = teamSize;
        System.out.println("Manager constructor called");
    }

    // Line 44: Method overriding with super.methodName()
    @Override
    public double calculateSalary() {
        // Line 46: Access parent class method using super
        double base = super.calculateSalary();
        double bonus = base * (bonusPercentage / 100);
        return base + bonus;
    }

    @Override
    public void displayInfo() {
        // Line 52: Call parent class method
        super.displayInfo();
        System.out.println("Bonus %: " + bonusPercentage +
                           "%, Team Size: " + teamSize +
                           ", Total Salary: $" + calculateSalary());
    }

    // Line 58: Accessing parent class fields using super
    public void updateBaseSalary(double newSalary) {
        // super.baseSalary = newSalary;  // Alternative way
        this.baseSalary = newSalary;  // Inherited, so accessible
        System.out.println(name + "'s base salary updated to: $" +
            newSalary);
    }

    // Cannot override final method
    // public void companyPolicy() { }  // Compilation error

    // Line 67: Additional method using super in different context
    public void showHierarchy() {
```

```
        System.out.println("\nManager Hierarchy:");
        System.out.println("1. Manager -> Employee");
        System.out.println("2. Employee -> Object");
        System.out.println("Super class of Manager: " +
                           super.getClass().getSuperclass().getName());
    }
}

// ==================== GRANDCHILD CLASS ====================
class SeniorManager extends Manager {
    private String region;

    public SeniorManager(int id, String name, double baseSalary,
                         double bonusPercentage, int teamSize, String
                            region) {
        // Line 80: super() calls Manager constructor, which calls
            Employee constructor
        super(id, name, baseSalary, bonusPercentage, teamSize);
        this.region = region;
        System.out.println("SeniorManager constructor called");
    }

    @Override
    public double calculateSalary() {
        // Line 86: super.calculateSalary() calls Manager's
            calculateSalary()
        double managerSalary = super.calculateSalary();
        double regionalAllowance = managerSalary * 0.1;  // 10% extra
        return managerSalary + regionalAllowance;
    }

    @Override
    public void displayInfo() {
        // Line 92: super.displayInfo() calls Manager's displayInfo()
        super.displayInfo();
        System.out.println("Region: " + region +
                           ", Senior Manager Salary: $" + calculateSalary
                              ());
    }

    // Line 97: Demonstrating super chain
    public void showAllSuperClasses() {
        System.out.println("\nSeniorManager's Class Hierarchy:");
        Class<?> currentClass = this.getClass();

        while (currentClass.getSuperclass() != null) {
            System.out.println("Superclass: " + currentClass.
                getSuperclass().getName());
            currentClass = currentClass.getSuperclass();
        }
    }
}

// ==================== MAIN CLASS ====================
public class SuperKeywordDemo {
    public static void main(String[] args) {
        System.out.println("=== 'super' KEYWORD DEMONSTRATION ===\n");

```

```
126         // Line 113: Creating Manager object
127         System.out.println("--- Creating Manager ---");
128         Manager manager = new Manager(101, "John Doe", 50000, 20, 8);
129
130         System.out.println("\n--- Manager Operations ---");
131         manager.displayInfo();  // Calls overridden method
132         manager.updateBaseSalary(55000);
133         manager.displayInfo();
134         manager.companyPolicy();  // Inherited final method
135         manager.showHierarchy();
136
137         System.out.println("\n" + "=".repeat(50));
138
139         // Line 125: Creating SeniorManager object
140         System.out.println("\n--- Creating SeniorManager ---");
141         SeniorManager seniorManager = new SeniorManager(
142             201, "Alice Smith", 70000, 25, 15, "North America");
143
144         System.out.println("\n--- SeniorManager Operations ---");
145         seniorManager.displayInfo();  // Calls chain of overridden
                methods
146         seniorManager.showAllSuperClasses();
147
148         // Line 134: Demonstrating super in constructor chain
149         System.out.println("\n--- Constructor Chain Demonstration ---")
                ;
150         System.out.println("Order of constructor calls:");
151         System.out.println("1. Employee constructor");
152         System.out.println("2. Manager constructor");
153         System.out.println("3. SeniorManager constructor");
154
155         // Line 140: Testing inheritance chain
156         System.out.println("\n--- Inheritance Chain Test ---");
157         System.out.println("seniorManager instanceof SeniorManager: " +
158                            (seniorManager instanceof SeniorManager));
159         System.out.println("seniorManager instanceof Manager: " +
160                            (seniorManager instanceof Manager));
161         System.out.println("seniorManager instanceof Employee: " +
162                            (seniorManager instanceof Employee));
163         System.out.println("seniorManager instanceof Object: " +
164                            (seniorManager instanceof Object));
165
166         // Line 148: Polymorphic behavior with super
167         System.out.println("\n--- Polymorphic Salary Calculation ---");
168         Employee emp = seniorManager;  // Upcasting
169         System.out.println("Salary via Employee reference: $" +
170                            emp.calculateSalary());  // Calls
                               SeniorManager's version
171     }
172 }
```

Listing 3: Comprehensive 'super' Keyword Usage

SuperKeywordDemo Program Output

```
=== 'super' KEYWORD DEMONSTRATION ===

--- Creating Manager ---
Employee constructor called
Manager constructor called

--- Manager Operations ---
ID: 101, Name: John Doe, Base Salary: $50000.0
Bonus %: 20.0%, Team Size: 8, Total Salary: $60000.0
John Doe's base salary updated to: $55000.0
ID: 101, Name: John Doe, Base Salary: $55000.0
Bonus %: 20.0%, Team Size: 8, Total Salary: $66000.0
All employees must follow company policy

Manager Hierarchy:
1. Manager -> Employee
2. Employee -> Object
Super class of Manager: Employee

================================================

--- Creating SeniorManager ---
Employee constructor called
Manager constructor called
SeniorManager constructor called

--- SeniorManager Operations ---
ID: 201, Name: Alice Smith, Base Salary: $70000.0
Bonus %: 25.0%, Team Size: 15, Total Salary: $87500.0
Region: North America, Senior Manager Salary: $96250.0

SeniorManager's Class Hierarchy:
Superclass: Manager
Superclass: Employee
Superclass: java.lang.Object

--- Constructor Chain Demonstration ---
Order of constructor calls:
1. Employee constructor
2. Manager constructor
3. SeniorManager constructor

--- Inheritance Chain Test ---
seniorManager instanceof SeniorManager: true
seniorManager instanceof Manager: true
seniorManager instanceof Employee: true
seniorManager instanceof Object: true

--- Polymorphic Salary Calculation---
Salary via Employee reference: $96250.0
```

2.5 Method Overriding

```
// ==================== PARENT CLASS ====================
class BankAccount {
    protected String accountNumber;
    protected String accountHolder;
    protected double balance;

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

    // Method to be overridden by child classes
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: $" + amount);
        } else {
            System.out.println("Invalid deposit amount");
        }
    }

    // Method to be overridden
    public boolean withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: $" + amount);
            return true;
        }
        System.out.println("Withdrawal failed. Insufficient funds.");
        return false;
    }

    // Method to be overridden
    public double calculateInterest() {
        return 0;  // Default: no interest
    }

    // Final method - cannot be overridden
    public final void displayAccountInfo() {
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Account Holder: " + accountHolder);
        System.out.println("Balance: $" + balance);
    }

    // This method uses calculateInterest() - demonstrates polymorphism
    public void applyInterest() {
        double interest = calculateInterest();
        balance += interest;
        System.out.println("Interest applied: $" + interest);
    }
}

// ==================== SAVINGS ACCOUNT ====================
class SavingsAccount extends BankAccount {
```

```
56      private static final double INTEREST_RATE = 4.5;  // 4.5% annual
57
58      public SavingsAccount(String accountNumber, String accountHolder,
            double balance) {
59          super(accountNumber, accountHolder, balance);
60      }
61
62      // Line 58: Overriding deposit method with additional functionality
63      @Override
64      public void deposit(double amount) {
65          if (amount > 0) {
66              // Add parent's functionality plus extra
67              super.deposit(amount);  // Call parent's deposit
68              System.out.println("Savings account deposit successful");
69
70              // Additional functionality
71              if (amount >= 1000) {
72                  System.out.println("Bonus: You deposited $1000 or more!
                        ");
73              }
74          }
75      }
76
77      // Line 71: Overriding withdraw with restrictions
78      @Override
79      public boolean withdraw(double amount) {
80          if (amount > 0 && amount <= balance && amount <= 10000) {
81              balance -= amount;
82              System.out.println("Savings account withdrawal: $" + amount
                    );
83              System.out.println("Note: Maximum $10,000 withdrawal limit
                    per transaction");
84              return true;
85          } else if (amount > 10000) {
86              System.out.println("Withdrawal failed. Maximum limit is $10
                    ,000 for savings account.");
87              return false;
88          }
89          return super.withdraw(amount);  // Use parent's logic for other
                cases
90      }
91
92      // Line 84: Overriding calculateInterest with savings-specific
            logic
93      @Override
94      public double calculateInterest() {
95          return balance * (INTEREST_RATE / 100);
96      }
97
98      // Line 89: Additional method specific to SavingsAccount
99      public void printPassbook() {
100         System.out.println("Printing passbook for savings account: " +
                accountNumber);
101     }
102 }
103
104 // ==================== CURRENT ACCOUNT ====================
105 class CurrentAccount extends BankAccount {
```

```
106     private static final double OVERDRAFT_LIMIT = 5000;
107
108     public CurrentAccount(String accountNumber, String accountHolder,
            double balance) {
109         super(accountNumber, accountHolder, balance);
110     }
111
112     // Line 101: Overriding withdraw to allow overdraft
113     @Override
114     public boolean withdraw(double amount) {
115         if (amount > 0 && (balance + OVERDRAFT_LIMIT) >= amount) {
116             balance -= amount;
117             System.out.println("Current account withdrawal: $" + amount
                    );
118
119             if (balance < 0) {
120                 System.out.println("Overdraft used. Current balance: $"
                        + balance);
121                 System.out.println("Remaining overdraft limit: $" +
122                                    (OVERDRAFT_LIMIT + balance));
123             }
124             return true;
125         }
126         System.out.println("Withdrawal failed. Exceeds overdraft limit.
                ");
127         return false;
128     }
129
130     // Line 117: Overriding calculateInterest (no interest for current
            accounts)
131     @Override
132     public double calculateInterest() {
133         System.out.println("No interest for current accounts");
134         return 0;
135     }
136
137     // Line 123: Overriding deposit with transaction fee
138     @Override
139     public void deposit(double amount) {
140         if (amount > 0) {
141             double transactionFee = Math.min(10, amount * 0.01);  // 1%
                    fee, max $10
142             double netAmount = amount - transactionFee;
143             balance += netAmount;
144             System.out.println("Deposited: $" + amount);
145             System.out.println("Transaction fee: $" + transactionFee);
146             System.out.println("Net amount added: $" + netAmount);
147         }
148     }
149
150     // Line 135: Additional method specific to CurrentAccount
151     public void issueCheque(int chequeNumber, double amount) {
152         System.out.println("Cheque #" + chequeNumber + " issued for $"
                + amount);
153         withdraw(amount);
154     }
155 }
156
```

```java
// ==================== FIXED DEPOSIT ACCOUNT ====================
class FixedDepositAccount extends BankAccount {
    private double interestRate;
    private int tenureMonths;

    public FixedDepositAccount(String accountNumber, String accountHolder,
                               double balance, double interestRate, int tenureMonths) {
        super(accountNumber, accountHolder, balance);
        this.interestRate = interestRate;
        this.tenureMonths = tenureMonths;
    }

    // Line 150: Overriding withdraw with FD restrictions
    @Override
    public boolean withdraw(double amount) {
        System.out.println("Cannot withdraw from Fixed Deposit before maturity!");
        System.out.println("Tenure: " + tenureMonths + " months remaining");
        return false;
    }

    // Line 157: Overriding calculateInterest with FD calculation
    @Override
    public double calculateInterest() {
        return balance * (interestRate / 100) * (tenureMonths / 12.0);
    }

    // Line 162: Overriding deposit (cannot deposit more in FD)
    @Override
    public void deposit(double amount) {
        System.out.println("Cannot add more funds to existing Fixed Deposit!");
        System.out.println("Create a new Fixed Deposit for additional investment.");
    }

    // Line 168: Additional method
    public void displayMaturityDetails() {
        double maturityAmount = balance + calculateInterest();
        System.out.println("Fixed Deposit Maturity Details:");
        System.out.println("Principal: $" + balance);
        System.out.println("Interest Rate: " + interestRate + "%");
        System.out.println("Tenure: " + tenureMonths + " months");
        System.out.println("Maturity Amount: $" + maturityAmount);
    }
}

// ==================== MAIN CLASS ====================
public class MethodOverridingDemo {
    public static void main(String[] args) {
        System.out.println("=== METHOD OVERRIDING DEMONSTRATION ===\n");

        // Line 183: Create array of BankAccount references
        // This demonstrates polymorphism
```

```
208         BankAccount[] accounts = new BankAccount[3];
209
210         accounts[0] = new SavingsAccount("SAV001", "Alice", 5000);
211         accounts[1] = new CurrentAccount("CUR001", "Bob", 3000);
212         accounts[2] = new FixedDepositAccount("FD001", "Charlie",
                10000, 7.5, 12);
213
214         System.out.println("=== PROCESSING ACCOUNTS POLYMORPHICALLY
                ===\n");
215
216         // Line 192: Process each account using same interface but
                different behaviors
217         for (BankAccount account : accounts) {
218             System.out.println("\n--- Processing " +
219                                account.getClass().getSimpleName() + " ---
                                    ");
220
221             account.displayAccountInfo();  // Same for all (final
                    method)
222
223             // Line 198: Polymorphic method calls
224             account.deposit(1500);         // Different behavior for
                    each
225             account.withdraw(2000);        // Different behavior for
                    each
226
227             // Line 201: calculateInterest() behaves differently for
                    each account
228             double interest = account.calculateInterest();
229             System.out.printf("Annual Interest: $%.2f\n", interest);
230
231             account.applyInterest();       // Uses overridden
                    calculateInterest()
232             account.displayAccountInfo();
233             System.out.println();
234         }
235
236         // Line 209: Type-specific operations (requires downcasting)
237         System.out.println("=== TYPE-SPECIFIC OPERATIONS ===");
238
239         // Line 212: Downcasting SavingsAccount
240         if (accounts[0] instanceof SavingsAccount) {
241             SavingsAccount savings = (SavingsAccount) accounts[0];
242             savings.printPassbook();
243         }
244
245         // Line 218: Downcasting CurrentAccount
246         if (accounts[1] instanceof CurrentAccount) {
247             CurrentAccount current = (CurrentAccount) accounts[1];
248             current.issueCheque(1001, 500);
249         }
250
251         // Line 224: Downcasting FixedDepositAccount
252         if (accounts[2] instanceof FixedDepositAccount) {
253             FixedDepositAccount fd = (FixedDepositAccount) accounts[2];
254             fd.displayMaturityDetails();
255         }
256
```

```
257         // Line 230: Demonstrating @Override annotation importance
258         System.out.println("\n=== @OVERRIDE ANNOTATION BENEFITS ===");
259         System.out.println("1. Compile-time checking: Ensures method
                exists in parent");
260         System.out.println("2. Readability: Clearly indicates
                overriding intent");
261         System.out.println("3. Prevents accidental overloading instead
                of overriding");
262
263         // Line 236: Testing edge cases
264         System.out.println("\n=== EDGE CASE TESTING ===");
265         accounts[0].withdraw(15000);  // Should fail for savings (limit
                $10,000)
266         accounts[1].withdraw(10000);  // Should succeed for current (
                with overdraft)
267         accounts[2].withdraw(1000);   // Should fail for FD
268     }
269 }
```

Listing 4: Complete Method Overriding Implementation

MethodOverridingDemo Program Output

```
=== METHOD OVERRIDING DEMONSTRATION ===
=== PROCESSING ACCOUNTS POLYMORPHICALLY ===
--- Processing SavingsAccount ---
Account Number: SAV001
Account Holder: Alice
Balance: $5000.0
Deposited: $1500.0
Savings account deposit successful
Bonus: You deposited $1000 or more!
Savings account withdrawal: $2000.0
Note: Maximum $10,000 withdrawal limit per transaction
Annual Interest: $202.50,Interest applied: $202.50,
Account Number: SAV001,Account Holder: Alice
Balance: $4702.50
--- Processing CurrentAccount ---
Account Number: CUR001,Account Holder: Bob,Balance: $3000.0,Deposited: $1500.0,Tran
Current account withdrawal: $2000.0
Annual Interest: $0.00
No interest for current accounts
Interest applied: $0.00,Account Number: CUR001,
Account Holder: Bob,Balance: $2490.0
--- Processing FixedDepositAccount ---
Account Number: FD001,Account Holder: Charlie,
Balance: $10000.0
Cannot add more funds to existing Fixed Deposit!
Create a new Fixed Deposit for additional investment.
Cannot withdraw from Fixed Deposit before maturity!
Tenure: 12 months remaining,Annual Interest: $750.00,Interest applied: $750.00,Acco
=== TYPE-SPECIFIC OPERATIONS ===
Printing passbook for savings account: SAV001
Cheque #1001 issued for $500.0
Current account withdrawal: $500.0
Fixed Deposit Maturity Details:
Principal: $10000.0
Interest Rate: 7.5%
Tenure: 12 months
Maturity Amount: $10750.0
=== @OVERRIDE ANNOTATION BENEFITS ===
1. Compile-time checking: Ensures method exists in parent
2. Readability: Clearly indicates overriding intent
3. Prevents accidental overloading instead of overriding
=== EDGE CASE TESTING ===
Withdrawal failed. Maximum limit is $10,000 for savings account.
Current account withdrawal: $10000.0
Overdraft used. Current balance: $-7510.0
Remaining overdraft limit: $-2510.0
Cannot withdraw from Fixed Deposit before maturity!
Tenure: 12 months remaining
```

3 Polymorphism

3.1 Compile-time Polymorphism (Method Overloading)

```java
// ==================== CALCULATOR CLASS ====================
class Calculator {

    // ============= METHOD OVERLOADING EXAMPLES =============

    // Line 8: Overloaded add() methods - different parameter types

    // Version 1: Add two integers
    public int add(int a, int b) {
        System.out.println("Adding two integers: " + a + " + " + b);
        return a + b;
    }

    // Version 2: Add three integers (different number of parameters)
    public int add(int a, int b, int c) {
        System.out.println("Adding three integers: " + a + " + " + b +
            " + " + c);
        return a + b + c;
    }

    // Version 3: Add two doubles (different parameter types)
    public double add(double a, double b) {
        System.out.println("Adding two doubles: " + a + " + " + b);
        return a + b;
    }

    // Version 4: Add array of integers
    public int add(int[] numbers) {
        System.out.print("Adding array of " + numbers.length + "
            integers: ");
        int sum = 0;
        for (int num : numbers) {
            System.out.print(num + " ");
            sum += num;
        }
        System.out.println();
        return sum;
    }

    // Version 5: Variable arguments (varargs) - most flexible
    public int add(int... numbers) {
        System.out.print("Adding variable arguments: ");
        int sum = 0;
        for (int num : numbers) {
            System.out.print(num + " ");
            sum += num;
        }
        System.out.println();
        return sum;
    }

    // Version 6: Add strings (concatenation)
    public String add(String a, String b) {
```

```
52          System.out.println("Concatenating strings: \"" + a + "\" + \""
                + b + "\"");
53          return a + b;
54      }
55
56      // ============= OVERLOADED CONSTRUCTORS =============
57
58      // Constructor 1: Default
59      public Calculator() {
60          System.out.println("Default Calculator created");
61      }
62
63      // Constructor 2: With model parameter
64      public Calculator(String model) {
65          System.out.println("Calculator model " + model + " created");
66      }
67
68      // Constructor 3: With model and version
69      public Calculator(String model, String version) {
70          System.out.println("Calculator " + model + " v" + version + "
                created");
71      }
72
73      // ============= OVERLOADED WITH DIFFERENT RETURN TYPES
            =============
74      // Note: Return type alone doesn't differentiate overloaded methods
75
76      public double multiply(double a, double b) {
77          return a * b;
78      }
79
80      public int multiply(int a, int b) {
81          return a * b;
82      }
83
84      // This would cause compilation error - same signature
85      // public double multiply(int a, int b) { return a * b; }
86
87      // ============= OVERLOADED WITH DIFFERENT ORDER =============
88
89      public void display(int a, double b) {
90          System.out.println("int: " + a + ", double: " + b);
91      }
92
93      public void display(double a, int b) {
94          System.out.println("double: " + a + ", int: " + b);
95      }
96
97      // ============= OVERLOADED STATIC METHODS =============
98
99      public static int square(int x) {
100         return x * x;
101     }
102
103     public static double square(double x) {
104         return x * x;
105     }
106 }
```

```java

// ==================== MATH OPERATIONS CLASS ====================
class MathOperations {

    // Overloaded methods with automatic type promotion
    public void compute(int x) {
        System.out.println("Integer computation: " + x);
    }

    public void compute(long x) {
        System.out.println("Long computation: " + x);
    }

    public void compute(double x) {
        System.out.println("Double computation: " + x);
    }

    // Ambiguity demonstration
    public void test(int a, double b) {
        System.out.println("int, double version");
    }

    public void test(double a, int b) {
        System.out.println("double, int version");
    }
}

// ==================== MAIN CLASS ====================
public class MethodOverloadingDemo {
    public static void main(String[] args) {
        System.out.println("=== COMPILE-TIME POLYMORPHISM: METHOD
            OVERLOADING ===\n");

        Calculator calc = new Calculator();

        System.out.println("=== DIFFERENT NUMBER OF PARAMETERS ===");
        System.out.println("Result 1: " + calc.add(10, 20));
        System.out.println("Result 2: " + calc.add(10, 20, 30));
        System.out.println();

        System.out.println("=== DIFFERENT TYPES OF PARAMETERS ===");
        System.out.println("Result 3: " + calc.add(5.5, 3.2));
        System.out.println("Result 4: " + calc.add("Hello", " World!"))
            ;
        System.out.println();

        System.out.println("=== ARRAY PARAMETER ===");
        int[] numbers = {1, 2, 3, 4, 5};
        System.out.println("Result 5: " + calc.add(numbers));
        System.out.println();

        System.out.println("=== VARIABLE ARGUMENTS (VARARGS) ===");
        System.out.println("Result 6: " + calc.add(1, 2, 3, 4, 5, 6));
        System.out.println("Result 7: " + calc.add(10, 20));  // Uses
            varargs
        System.out.println();

        System.out.println("=== DIFFERENT RETURN TYPES ===");
```

```
        System.out.println("Multiply ints: " + calc.multiply(5, 4));
        System.out.println("Multiply doubles: " + calc.multiply(5.5,
            2.0));
        System.out.println();

        System.out.println("=== DIFFERENT PARAMETER ORDER ===");
        calc.display(10, 5.5);    // Calls int, double version
        calc.display(5.5, 10);    // Calls double, int version
        System.out.println();

        System.out.println("=== STATIC METHOD OVERLOADING ===");
        System.out.println("Square of 5: " + Calculator.square(5));
        System.out.println("Square of 5.5: " + Calculator.square(5.5));
        System.out.println();

        System.out.println("=== CONSTRUCTOR OVERLOADING ===");
        Calculator calc1 = new Calculator();
        Calculator calc2 = new Calculator("Scientific");
        Calculator calc3 = new Calculator("Graphing", "2.0");
        System.out.println();

        System.out.println("=== TYPE PROMOTION IN OVERLOADING ===");
        MathOperations math = new MathOperations();

        math.compute(10);      // Calls int version
        math.compute(10L);     // Calls long version
        math.compute(10.0);    // Calls double version
        math.compute(10.0f);   // Float promoted to double

        System.out.println("\n=== AMBIGUITY IN OVERLOADING ===");
        // math.test(10, 10);  // Ambiguous - compilation error
        math.test(10, 10.0);    // OK - int, double
        math.test(10.0, 10);    // OK - double, int
        System.out.println();

        System.out.println("=== METHOD RESOLUTION AT COMPILE TIME ===")
            ;
        System.out.println("Overloaded method resolution happens at
            COMPILE TIME");
        System.out.println("based on:");
        System.out.println("1. Method name");
        System.out.println("2. Number of parameters");
        System.out.println("3. Type of parameters");
        System.out.println("4. Order of parameters");
        System.out.println("\nCompiler determines which version to call
             during compilation.");

        // Demonstration of compile-time decision
        System.out.println("\n=== COMPILE-TIME DECISION DEMO ===");
        int result1 = calc.add(5, 10);         // Decided at compile
            time
        double result2 = calc.add(5.5, 2.5);  // Decided at compile
            time
        System.out.println("Integer addition result: " + result1);
        System.out.println("Double addition result: " + result2);
    }
}
```

Listing 5: Complete Method Overloading Implementation

MethodOverloadingDemo Program Output

```
=== COMPILE-TIME POLYMORPHISM: METHOD OVERLOADING
=== DIFFERENT NUMBER OF PARAMETERS ===
Adding two integers: 10 + 20,Result 1: 30,Adding three integers: 10 + 20 + 30,Resul
=== DIFFERENT TYPES OF PARAMETERS ===
Adding two doubles: 5.5 + 3.2
Result 3: 8.7
Concatenating strings: "Hello" + " World!"
Result 4: Hello World!
=== ARRAY PARAMETER ===
Adding array of 5 integers: 1 2 3 4 5
Result 5: 15
=== VARIABLE ARGUMENTS (VARARGS) ===
Adding variable arguments: 1 2 3 4 5 6
Result 6: 21
Adding variable arguments: 10 20
Result 7: 30
=== DIFFERENT RETURN TYPES ===
Multiply ints: 20
Multiply doubles: 11.0
=== DIFFERENT PARAMETER ORDER ===
int: 10, double: 5.5
double: 5.5, int: 10
=== STATIC METHOD OVERLOADING ===
Square of 5: 25
Square of 5.5: 30.25
=== CONSTRUCTOR OVERLOADING ===
Default Calculator created
Calculator model Scientific created
Calculator Graphing v2.0 created
=== TYPE PROMOTION IN OVERLOADING ===
Integer computation: 10
Long computation: 10
Double computation: 10.0
Double computation: 10.0
=== AMBIGUITY IN OVERLOADING ===
int, double version
double, int version
=== METHOD RESOLUTION AT COMPILE TIME ===
Overloaded method resolution happens at COMPILE TIME
based on:
1. Method name
2. Number of parameters
3. Type of parameters
4. Order of parameters
Compiler determines which version to call during compilation.
=== COMPILE-TIME DECISION DEMO ===
Adding two integers: 5 + 10
Integer addition result: 15
Adding two doubles: 5.5 + 2.5
Double addition result: 8.0
```

3.2 Runtime Polymorphism (Method Overriding and Dynamic Method Dispatch)

```
// ==================== BASE CLASS ====================
class Payment {
    protected double amount;
    protected String transactionId;

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

    // This method will be overridden - runtime polymorphism
    public void processPayment() {
        System.out.println("Processing generic payment of $" + amount);
    }

    public double calculateFee() {
        return amount * 0.02;  // 2% fee by default
    }

    public void displayDetails() {
        System.out.println("Transaction: " + transactionId +
                           ", Amount: $" + amount);
    }

    // Static method - not polymorphic
    public static void paymentPolicy() {
        System.out.println("All payments must follow security
            guidelines");
    }
}

// ==================== CREDIT CARD PAYMENT ====================
class CreditCardPayment extends Payment {
    private String cardNumber;
    private String cardHolder;

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

    // Line 40: Method overriding - runtime polymorphism
    @Override
    public void processPayment() {
        System.out.println("Processing Credit Card Payment:");
        System.out.println("Card Holder: " + cardHolder);
        System.out.println("Card Number: " + cardNumber);
        System.out.println("Amount: $" + amount);
        System.out.println("Verifying with bank...");
        System.out.println("Payment authorized!");
    }

```

```
54      // Line 50: Override with different fee calculation
55      @Override
56      public double calculateFee() {
57          double baseFee = super.calculateFee();
58          double additionalFee = amount * 0.015;  // 1.5% extra for
                credit card
59          return baseFee + additionalFee;
60      }
61
62      // Additional method
63      public void generateReceipt() {
64          System.out.println("Credit Card Receipt Generated");
65          System.out.println("Total with fees: $" + (amount +
                calculateFee()));
66      }
67
68      private String maskCardNumber(String cardNumber) {
69          return "****-****-****-" + cardNumber.substring(cardNumber.
                length() - 4);
70      }
71  }
72
73  // ==================== PAYPAL PAYMENT ====================
74  class PayPalPayment extends Payment {
75      private String email;
76
77      public PayPalPayment(double amount, String transactionId, String
            email) {
78          super(amount, transactionId);
79          this.email = email;
80      }
81
82      // Line 76: Different implementation for PayPal
83      @Override
84      public void processPayment() {
85          System.out.println("Processing PayPal Payment:");
86          System.out.println("Email: " + email);
87          System.out.println("Redirecting to PayPal...");
88          System.out.println("User authenticated");
89          System.out.println("Payment completed via PayPal");
90      }
91
92      // Line 85: Override with PayPal-specific fee
93      @Override
94      public double calculateFee() {
95          return amount * 0.029 + 0.30;  // 2.9% + $0.30 PayPal fee
                structure
96      }
97
98      public void sendConfirmationEmail() {
99          System.out.println("Confirmation email sent to: " + email);
100     }
101 }
102
103 // ==================== BANK TRANSFER PAYMENT ====================
104 class BankTransferPayment extends Payment {
105     private String accountNumber;
106     private String bankName;
```

```
107
108     public BankTransferPayment(double amount, String transactionId,
109                                String accountNumber, String bankName) {
110         super(amount, transactionId);
111         this.accountNumber = accountNumber;
112         this.bankName = bankName;
113     }
114
115     // Line 104: Different implementation for bank transfer
116     @Override
117     public void processPayment() {
118         System.out.println("Processing Bank Transfer:");
119         System.out.println("Bank: " + bankName);
120         System.out.println("Account: " + accountNumber);
121         System.out.println("Initiating transfer...");
122         System.out.println("Transfer may take 1-3 business days");
123     }
124
125     // Line 113: Lowest fee for bank transfers
126     @Override
127     public double calculateFee() {
128         return 5.00;  // Fixed fee for bank transfer
129     }
130
131     public void printTransferSlip() {
132         System.out.println("Bank Transfer Slip Generated");
133     }
134 }
135
136 // ==================== CRYPTOCURRENCY PAYMENT ====================
137 class CryptoPayment extends Payment {
138     private String walletAddress;
139     private String cryptocurrency;
140
141     public CryptoPayment(double amount, String transactionId,
142                          String walletAddress, String cryptocurrency) {
143         super(amount, transactionId);
144         this.walletAddress = walletAddress;
145         this.cryptocurrency = cryptocurrency;
146     }
147
148     // Line 131: Completely different implementation
149     @Override
150     public void processPayment() {
151         System.out.println("Processing " + cryptocurrency + " Payment:"
            );
152         System.out.println("Wallet: " + walletAddress);
153         System.out.println("Validating blockchain transaction...");
154         System.out.println("Transaction confirmed on blockchain!");
155         System.out.println("Payment immutable and secure");
156     }
157
158     // Line 140: Variable fee based on network congestion
159     @Override
160     public double calculateFee() {
161         double networkFee = amount * 0.01;  // 1% network fee
162         double minerFee = 2.50;             // Fixed miner fee
163         return networkFee + minerFee;
```

```
164     }
165
166     public void showTransactionHash() {
167         System.out.println("Blockchain TX Hash: 0x" +
168                            transactionId.hashCode());
169     }
170 }
171
172 // ==================== MAIN CLASS ====================
173 public class RuntimePolymorphismDemo {
174     public static void main(String[] args) {
175         System.out.println("=== RUNTIME POLYMORPHISM DEMONSTRATION ===\
            n");
176
177         // Line 159: Create array of Payment references
178         // Dynamic Method Dispatch: Reference type = Payment, Objects =
                Different types
179         Payment[] payments = new Payment[4];
180
181         payments[0] = new CreditCardPayment(100.0, "TXN001",
182                                             "1234567812345678", "John
                                                Doe");
183         payments[1] = new PayPalPayment(75.50, "TXN002", "user@example.
            com");
184         payments[2] = new BankTransferPayment(200.0, "TXN003",
185                                               "ACCT123456", "Bank of
                                                  America");
186         payments[3] = new CryptoPayment(150.0, "TXN004",
187                                         "0xABC123DEF456", "Ethereum");
188
189         System.out.println("=== PROCESSING PAYMENTS (DYNAMIC METHOD
            DISPATCH) ===\n");
190
191         // Line 173: Same method call, different behaviors at runtime
192         for (Payment payment : payments) {
193             System.out.println("--- Processing " +
194                                payment.getClass().getSimpleName() + " ---
                                   ");
195
196             // Line 177: Dynamic method dispatch - actual object's
                    method is called
197             payment.processPayment();  // Runtime decision based on
                    actual object type
198
199             // Line 180: Polymorphic fee calculation
200             double fee = payment.calculateFee();
201             System.out.printf("Transaction Fee: $%.2f\n", fee);
202
203             payment.displayDetails();  // Inherited, same for all
204
205             // Line 185: Call static method (not polymorphic)
206             Payment.paymentPolicy();
207
208             System.out.println();
209         }
210
211         System.out.println("=== DEMONSTRATING RUNTIME DECISION MAKING
            ===\n");
```

```
212
213         // Line 192: Payment reference, but actual object determined at
                runtime
214         Payment paymentRef;
215
216         // Simulate user choice
217         int userChoice = 2;  // 0-CC, 1-PayPal, 2-Bank, 3-Crypto
218
219         switch (userChoice) {
220             case 0:
221                 paymentRef = new CreditCardPayment(50.0, "TXN005",
222                                                    "8765432187654321", "
                                                       Alice");
223                 break;
224             case 1:
225                 paymentRef = new PayPalPayment(30.0, "TXN006", "
                       alice@example.com");
226                 break;
227             case 2:
228                 paymentRef = new BankTransferPayment(100.0, "TXN007",
229                                                      "ACCT987654", "Chase
                                                         ");
230                 break;
231             case 3:
232                 paymentRef = new CryptoPayment(75.0, "TXN008",
233                                                "0xXYZ789ABC456", "Bitcoin
                                                   ");
234                 break;
235             default:
236                 paymentRef = new Payment(10.0, "TXN009");
237         }
238
239         System.out.println("User selected: " +
240                            paymentRef.getClass().getSimpleName());
241         paymentRef.processPayment();  // Decision made at runtime!
242
243         System.out.println("\n=== TYPE-SPECIFIC OPERATIONS (DOWNCASTING
                ) ===\n");
244
245         // Line 221: Downcasting to access child-specific methods
246         for (Payment payment : payments) {
247             if (payment instanceof CreditCardPayment) {
248                 CreditCardPayment cc = (CreditCardPayment) payment;
249                 cc.generateReceipt();
250             }
251             else if (payment instanceof PayPalPayment) {
252                 PayPalPayment pp = (PayPalPayment) payment;
253                 pp.sendConfirmationEmail();
254             }
255             else if (payment instanceof BankTransferPayment) {
256                 BankTransferPayment bt = (BankTransferPayment) payment;
257                 bt.printTransferSlip();
258             }
259             else if (payment instanceof CryptoPayment) {
260                 CryptoPayment cp = (CryptoPayment) payment;
261                 cp.showTransactionHash();
262             }
263             System.out.println();
```

```
264         }
265
266         System.out.println("=== RUNTIME POLYMORPHISM BENEFITS ===\n");
267         System.out.println("1. Code Flexibility: Can add new payment
                types without modifying existing code");
268         System.out.println("2. Extensibility: Easy to add new payment
                methods");
269         System.out.println("3. Maintainability: Changes isolated to
                specific classes");
270         System.out.println("4. Reusability: Common interface for all
                payment types");
271         System.out.println("5. Dynamic Behavior: Actual method
                determined at runtime");
272
273         // Line 246: Demonstrating difference between static and
                dynamic binding
274         System.out.println("\n=== STATIC VS DYNAMIC BINDING ===");
275         System.out.println("Static Binding (Compile-time):");
276         System.out.println("  - Overloaded methods");
277         System.out.println("  - Static methods");
278         System.out.println("  - Final methods");
279         System.out.println("  - Private methods");
280         System.out.println("\nDynamic Binding (Runtime):");
281         System.out.println("  - Overridden methods (virtual methods)");
282         System.out.println("  - Actual object type determines method
                called");
283
284         // Line 257: Final demonstration
285         System.out.println("\n=== FINAL DEMONSTRATION ===");
286         Payment genericPayment = new Payment(10.0, "TXN010");
287         Payment creditCardPayment = new CreditCardPayment(10.0, "TXN011
                ",
288                                                           "
                                                              1111222233334444
                                                              ", "Bob");
289
290         System.out.println("\nGeneric Payment:");
291         genericPayment.processPayment();  // Calls Payment's version
292
293         System.out.println("\nCredit Card Payment (via Payment
                reference):");
294         creditCardPayment.processPayment();  // Calls CreditCardPayment
                's version
295     }
296 }
```

Listing 6: Runtime Polymorphism and Dynamic Method Dispatch

RuntimePolymorphismDemo Program Output

```
=== RUNTIME POLYMORPHISM DEMONSTRATION ===
=== PROCESSING PAYMENTS (DYNAMIC METHOD DISPATCH) =
--- Processing CreditCardPayment ---
Processing Credit Card Payment:Card Holder: John Doe,Card Number: ****-****-****-56
Verifying with bank...Payment authorized!
Transaction Fee: $3.50,Transaction: TXN001, Amount: $100.0,All payments must follow
--- Processing PayPalPayment ---
Processing PayPal Payment: Email: user@example.com
Redirecting to PayPal...User authenticated
Payment completed via PayPal
Transaction Fee: $2.49,Transaction: TXN002, Amount: $75.5
All payments must follow security guidelines
--- Processing BankTransferPayment ---
Processing Bank Transfer:Bank: Bank of America
Account: ACCT123456,Initiating transfer...
Transfer may take 1-3 business days, Transaction Fee: $5.00 ,Transaction: TXN003, A
All payments must follow security guidelines
--- Processing CryptoPayment ---
Processing Ethereum Payment:
Wallet: 0xABC123DEF456, Validating blockchain transaction..., Transaction confirmed
Transaction Fee: $4.00 ,Transaction: TXN004, Amount: $150.0 ,All payments must foll
=== DEMONSTRATING RUNTIME DECISION MAKING ===
User selected: BankTransferPayment
Processing Bank Transfer:, Bank: Chase
Account: ACCT987654 ,Initiating transfer...
Transfer may take 1-3 business days
=== TYPE-SPECIFIC OPERATIONS (DOWNCASTING) ===
Credit Card Receipt Generated, Total with fees: $103.50, Confirmation email sent to
Bank Transfer Slip Generated
Blockchain TX Hash: 0x1665753818
=== RUNTIME POLYMORPHISM BENEFITS ===
1. Code Flexibility: Can add new payment types without modifying existing code
2. Extensibility: Easy to add new payment methods
3. Maintainability: Changes isolated to specific classes
4. Reusability: Common interface for all payment types
5. Dynamic Behavior: Actual method determined at runtime
=== STATIC VS DYNAMIC BINDING ===
Static Binding (Compile-time):
  - Overloaded methods,   - Static methods
  - Final methods,   - Private methods
Dynamic Binding (Runtime):
  - Overridden methods (virtual methods)
  - Actual object type determines method called
=== FINAL DEMONSTRATION ===
Generic Payment:, Processing generic payment of $10.0, Credit Card Payment (via Pay
Processing Credit Card Payment:
Card Holder: Bob, Card Number: ****-****-****-4444
Amount: $10.0, Verifying with bank...
Payment authorized!
```

4 Abstraction

4.1 Abstract Classes and Abstract Methods

```java
// ==================== ABSTRACT CLASS ====================
// Line 4: Abstract class cannot be instantiated
// Serves as a template for concrete classes
abstract class Employee {
    protected int id;
    protected String name;
    protected double baseSalary;

    // Line 10: Constructor in abstract class
    // Called when concrete subclass object is created
    public Employee(int id, String name, double baseSalary) {
        this.id = id;
        this.name = name;
        this.baseSalary = baseSalary;
        System.out.println("Employee " + name + " created");
    }

    // Line 18: Abstract method - no implementation
    // MUST be implemented by concrete subclasses
    public abstract double calculateSalary();

    // Line 21: Abstract method for role-specific work
    public abstract void performDuties();

    // Line 24: Concrete method in abstract class
    // Provides common implementation for all subclasses
    public void displayBasicInfo() {
        System.out.println("ID: " + id + ", Name: " + name);
    }

    // Line 29: Another concrete method
    public double getBaseSalary() {
        return baseSalary;
    }

    // Line 33: Final concrete method - cannot be overridden
    public final void companyPolicy() {
        System.out.println("All employees must follow company policies"
            );
    }

    // Line 38: Static method in abstract class
    public static void showCompanyName() {
        System.out.println("Company: Tech Solutions Inc.");
    }
}

// ==================== FULL-TIME EMPLOYEE ====================
class FullTimeEmployee extends Employee {
    private double bonus;
    private int leavesTaken;

    public FullTimeEmployee(int id, String name, double baseSalary,
```

```
                            double bonus, int leavesTaken) {
        // Line 49: Must call parent constructor
        super(id, name, baseSalary);
        this.bonus = bonus;
        this.leavesTaken = leavesTaken;
    }

    // Line 55: MUST implement abstract method
    @Override
    public double calculateSalary() {
        double salary = baseSalary + bonus;
        // Deduct for leaves taken
        double dailyRate = baseSalary / 30;
        salary -= (leavesTaken * dailyRate);
        return salary;
    }

    // Line 63: MUST implement abstract method
    @Override
    public void performDuties() {
        System.out.println(name + " is working full-time (9 AM - 5 PM)"
            );
        System.out.println("Attending meetings, completing projects");
    }

    // Line 69: Additional method specific to full-time employees
    public void requestLeave(int days) {
        System.out.println(name + " requested " + days + " days leave")
            ;
        leavesTaken += days;
    }

    // Line 74: Override concrete method (optional)
    @Override
    public void displayBasicInfo() {
        super.displayBasicInfo();
        System.out.println("Type: Full-Time, Bonus: $" + bonus);
    }
}

// ==================== PART-TIME EMPLOYEE ====================
class PartTimeEmployee extends Employee {
    private int hoursWorked;
    private double hourlyRate;

    public PartTimeEmployee(int id, String name, double baseSalary,
                            int hoursWorked, double hourlyRate) {
        super(id, name, baseSalary);
        this.hoursWorked = hoursWorked;
        this.hourlyRate = hourlyRate;
    }

    // Line 90: Different implementation for part-time
    @Override
    public double calculateSalary() {
        return hoursWorked * hourlyRate;
    }

```

```
109     // Line 95: Different implementation
110     @Override
111     public void performDuties() {
112         System.out.println(name + " is working part-time (" +
113                            hoursWorked + " hours this month)");
114         System.out.println("Flexible schedule, task-based work");
115     }
116 
117     // Line 101: Additional method
118     public void logHours(int hours) {
119         hoursWorked += hours;
120         System.out.println(name + " logged " + hours + " additional
                hours");
121     }
122 
123     @Override
124     public void displayBasicInfo() {
125         super.displayBasicInfo();
126         System.out.println("Type: Part-Time, Hours: " + hoursWorked +
127                            ", Rate: $" + hourlyRate + "/hour");
128     }
129 }
130 
131 // ==================== CONTRACT EMPLOYEE ====================
132 class ContractEmployee extends Employee {
133     private int contractDuration;  // in months
134     private double projectBonus;
135 
136     public ContractEmployee(int id, String name, double baseSalary,
137                             int contractDuration, double projectBonus) {
138         super(id, name, baseSalary);
139         this.contractDuration = contractDuration;
140         this.projectBonus = projectBonus;
141     }
142 
143     // Line 122: Different salary calculation
144     @Override
145     public double calculateSalary() {
146         return baseSalary + projectBonus;
147     }
148 
149     // Line 127: Different duties
150     @Override
151     public void performDuties() {
152         System.out.println(name + " is working on contract (" +
153                            contractDuration + " months)");
154         System.out.println("Project-based work, specific deliverables")
                ;
155     }
156 
157     // Line 133: Additional method
158     public void extendContract(int additionalMonths) {
159         contractDuration += additionalMonths;
160         System.out.println("Contract extended by " + additionalMonths +
                " months");
161     }
162 
163     @Override
```

```
164     public void displayBasicInfo() {
165         super.displayBasicInfo();
166         System.out.println("Type: Contract, Duration: " +
                contractDuration +
167                            " months, Project Bonus: $" + projectBonus);
168     }
169 }
170 
171 // ==================== INTERN ====================
172 class Intern extends Employee {
173     private String university;
174     private int internshipDuration;  // in months
175 
176     public Intern(int id, String name, double baseSalary,
177                   String university, int internshipDuration) {
178         super(id, name, baseSalary);
179         this.university = university;
180         this.internshipDuration = internshipDuration;
181     }
182 
183     // Line 155: Interns have fixed stipend
184     @Override
185     public double calculateSalary() {
186         return baseSalary;  // Fixed stipend
187     }
188 
189     // Line 160: Intern-specific duties
190     @Override
191     public void performDuties() {
192         System.out.println(name + " is interning from " + university);
193         System.out.println("Learning, assisting, training (" +
194                            internshipDuration + " months)");
195     }
196 
197     // Line 166: Additional method
198     public void submitReport() {
199         System.out.println(name + " submitted internship report");
200     }
201 
202     @Override
203     public void displayBasicInfo() {
204         super.displayBasicInfo();
205         System.out.println("Type: Intern, University: " + university +
206                            ", Duration: " + internshipDuration + " months
                               ");
207     }
208 }
209 
210 // ==================== ABSTRACT CLASS WITH PARTIAL IMPLEMENTATION
    ====================
211 abstract class Vehicle {
212     protected String brand;
213     protected String model;
214     protected int year;
215 
216     public Vehicle(String brand, String model, int year) {
217         this.brand = brand;
218         this.model = model;
```

```
219         this.year = year;
220     }
221
222     // Abstract methods
223     public abstract void start();
224     public abstract void stop();
225     public abstract double calculateFuelEfficiency();
226
227     // Concrete methods
228     public void displayInfo() {
229         System.out.println(brand + " " + model + " (" + year + ")");
230     }
231
232     // Template method pattern using abstract methods
233     public final void operate() {
234         System.out.println("Operating vehicle:");
235         start();
236         System.out.println("Vehicle is running...");
237         System.out.println("Fuel Efficiency: " +
                calculateFuelEfficiency() + " km/l");
238         stop();
239     }
240 }
241
242 class Car extends Vehicle {
243     public Car(String brand, String model, int year) {
244         super(brand, model, year);
245     }
246
247     @Override
248     public void start() {
249         System.out.println("Turning key, car engine starts");
250     }
251
252     @Override
253     public void stop() {
254         System.out.println("Turning key, car engine stops");
255     }
256
257     @Override
258     public double calculateFuelEfficiency() {
259         return 15.5;  // km per liter
260     }
261 }
262
263 class Bike extends Vehicle {
264     public Bike(String brand, String model, int year) {
265         super(brand, model, year);
266     }
267
268     @Override
269     public void start() {
270         System.out.println("Kick starting the bike");
271     }
272
273     @Override
274     public void stop() {
275         System.out.println("Applying brakes, bike stops");
```

```
276     }
277
278     @Override
279     public double calculateFuelEfficiency() {
280         return 45.0;  // km per liter
281     }
282 }
283
284 // ==================== MAIN CLASS ====================
285 public class AbstractionDemo {
286     public static void main(String[] args) {
287         System.out.println("=== ABSTRACTION DEMONSTRATION ===\n");
288
289         // Line 231: Cannot instantiate abstract class
290         // Employee emp = new Employee(1, "Test", 1000);  //
                Compilation error
291
292         System.out.println("=== CREATING DIFFERENT TYPES OF EMPLOYEES
                ===\n");
293
294         // Line 236: Create array of Employee references
295         // Can hold any concrete subclass
296         Employee[] employees = new Employee[4];
297
298         employees[0] = new FullTimeEmployee(101, "Alice", 5000, 1000,
                2);
299         employees[1] = new PartTimeEmployee(102, "Bob", 0, 80, 25);
300         employees[2] = new ContractEmployee(103, "Charlie", 4000, 6,
                2000);
301         employees[3] = new Intern(104, "Diana", 1000, "MIT", 3);
302
303         System.out.println("\n=== PROCESSING EMPLOYEES USING
                ABSTRACTION ===\n");
304
305         // Line 247: Process all employees using abstract interface
306         double totalSalaryExpense = 0;
307
308         for (Employee emp : employees) {
309             System.out.println("--- " + emp.getClass().getSimpleName()
                    + " ---");
310
311             // Line 252: Call concrete method from abstract class
312             emp.displayBasicInfo();
313
314             // Line 255: Call abstract methods (different
                    implementations)
315             emp.performDuties();
316
317             // Line 258: Polymorphic salary calculation
318             double salary = emp.calculateSalary();
319             System.out.printf("Salary: $%.2f\n", salary);
320             totalSalaryExpense += salary;
321
322             // Line 263: Call final method
323             emp.companyPolicy();
324
325             // Line 266: Type-specific operations (downcasting)
326             if (emp instanceof FullTimeEmployee) {
```

```
                FullTimeEmployee ft = (FullTimeEmployee) emp;
                ft.requestLeave(1);
            } else if (emp instanceof PartTimeEmployee) {
                PartTimeEmployee pt = (PartTimeEmployee) emp;
                pt.logHours(10);
            } else if (emp instanceof ContractEmployee) {
                ContractEmployee ce = (ContractEmployee) emp;
                ce.extendContract(3);
            } else if (emp instanceof Intern) {
                Intern intern = (Intern) emp;
                intern.submitReport();
            }

            System.out.println();
        }

        System.out.println("=== COMPANY SUMMARY ===");
        Employee.showCompanyName();  // Static method
        System.out.printf("Total Monthly Salary Expense: $%.2f\n",
            totalSalaryExpense);
        System.out.println("Total Employees: " + employees.length);

        System.out.println("\n=== VEHICLE ABSTRACTION EXAMPLE ===");

        // Line 289: Vehicle abstraction demonstration
        Vehicle[] vehicles = new Vehicle[2];
        vehicles[0] = new Car("Toyota", "Camry", 2022);
        vehicles[1] = new Bike("Yamaha", "R15", 2023);

        for (Vehicle vehicle : vehicles) {
            System.out.println("\n--- " + vehicle.getClass().
                getSimpleName() + " ---");
            vehicle.displayInfo();
            vehicle.operate();  // Template method pattern
        }

        System.out.println("\n=== ABSTRACTION BENEFITS ===");
        System.out.println("1. Hides implementation details, shows only
            functionality");
        System.out.println("2. Forces subclasses to implement specific
            behavior");
        System.out.println("3. Allows common implementation in abstract
            class");
        System.out.println("4. Enables polymorphism and flexible code
            design");
        System.out.println("5. Template Method Pattern: Define
            algorithm skeleton");

        System.out.println("\n=== WHEN TO USE ABSTRACT CLASSES ===");
        System.out.println("1. When you want to share code among
            related classes");
        System.out.println("2. When you have classes with common state/
            behavior");
        System.out.println("3. When you want to declare non-static/non-
            final fields");
        System.out.println("4. When you need to define constructor");
        System.out.println("5. When you want to provide partial
            implementation");
```

```
374
375         // Line 314: Demonstrating abstract class as reference type
376         System.out.println("\n=== ABSTRACT CLASS AS REFERENCE TYPE ===");
377         Employee empRef = new FullTimeEmployee(105, "Eve", 6000, 1200, 0);
378         System.out.println("Reference type: Employee");
379         System.out.println("Actual object: FullTimeEmployee");
380         System.out.printf("Salary through abstract reference: $%.2f\n",
381                           empRef.calculateSalary());
382     }
383 }
```

Listing 7: Complete Abstraction Implementation

AbstractionDemo Program Output

```
=== ABSTRACTION DEMONSTRATION ===
=== CREATING DIFFERENT TYPES OF EMPLOYEES ===
Employee Alice created, Employee Bob created, Employee Charlie created, Employee Di
=== PROCESSING EMPLOYEES USING ABSTRACTION ===
--- FullTimeEmployee ---
ID: 101, Name: Alice, Type: Full-Time, Bonus: $1000.0, Alice is working full-time (
Attending meetings, completing projects
Salary: $5666.67, All employees must follow company policies
Alice requested 1 days leave
--- PartTimeEmployee ---
ID: 102, Name: Bob
Type: Part-Time, Hours: 80, Rate: $25.0/hour
Bob is working part-time (80 hours this month)
Flexible schedule, task-based work
Salary: $2000.00, All employees must follow company policies Bob logged 10 addition
--- ContractEmployee ---
ID: 103, Name: Charlie
Type: Contract, Duration: 6 months, Project Bonus: $2000.0
Charlie is working on contract (6 months)
Project-based work, specific deliverables
Salary: $6000.00 All employees must follow company policies Contract extended by 3
--- Intern ---
ID: 104, Name: Diana, Type: Intern, University: MIT, Duration: 3 months, Diana is i
Salary: $1000.00, All employees must follow company policies ,Diana submitted inter
=== COMPANY SUMMARY ===
Company: Tech Solutions Inc.
Total Monthly Salary Expense: $14666.67
Total Employees: 4
=== VEHICLE ABSTRACTION EXAMPLE === --- Car ---
Toyota Camry (2022), Operating vehicle:
Turning key, car engine starts Vehicle is running... Fuel Efficiency: 15.5 km/l
Turning key, car engine stops --- Bike ---
Yamaha R15 (2023) , Operating vehicle:
Kick starting the bike Vehicle is running...
Fuel Efficiency: 45.0 km/l Applying brakes, bike stops
=== ABSTRACTION BENEFITS ===
1. Hides implementation details, shows only functionality
2. Forces subclasses to implement specific behavior
3. Allows common implementation in abstract class
4. Enables polymorphism and flexible code design
5. Template Method Pattern: Define algorithm skeleton
=== WHEN TO USE ABSTRACT CLASSES ===
1. When you want to share code among related classes
2. When you have classes with common state/behavior
3. When you want to declare non-static/non-final fields
4. When you need to define constructor
5. When you want to provide partial implementation
=== ABSTRACT CLASS AS REFERENCE TYPE =
Reference type: Employee
Actual object: FullTimeEmployee
Salary through abstract reference: $6000.00
```

5 Interfaces

5.1 Defining and Implementing Interfaces

```java
// ==================== SIMPLE INTERFACE ====================
// Line 4: Interface definition
// All methods are implicitly public and abstract (before Java 8)
interface Playable {
    // Line 6: Constant (implicitly public, static, final)
    String TYPE = "Entertainment";

    // Line 9: Abstract method (no implementation)
    void play();

    // Line 12: Abstract method
    void stop();

    // Line 15: Default method (Java 8+)
    // Provides default implementation
    default void pause() {
        System.out.println("Pausing playback");
    }

    // Line 20: Static method (Java 8+)
    // Called using interface name
    static void showCategory() {
        System.out.println("Category: " + TYPE);
    }
}

// ==================== MULTIPLE INTERFACES ====================
interface VolumeControllable {
    int MAX_VOLUME = 100;
    int MIN_VOLUME = 0;

    void increaseVolume();
    void decreaseVolume();
    void setVolume(int level);

    // Default method
    default void mute() {
        System.out.println("Muted");
        setVolume(0);
    }
}

interface Connectable {
    void connect();
    void disconnect();

    default String getConnectionStatus() {
        return "Unknown";
    }
}

// ==================== EXTENDING INTERFACES ====================
interface AdvancedPlayable extends Playable {
```

```
54      void fastForward();
55      void rewind();
56      void record();
57  }
58
59  // ==================== CLASS IMPLEMENTING SINGLE INTERFACE
        ====================
60  class MusicPlayer implements Playable {
61      private String currentSong;
62
63      public MusicPlayer(String song) {
64          this.currentSong = song;
65      }
66
67      // Line 60: Must implement all abstract methods
68      @Override
69      public void play() {
70          System.out.println("Playing song: " + currentSong);
71          System.out.println("Duration: 3:45");
72      }
73
74      @Override
75      public void stop() {
76          System.out.println("Music stopped");
77      }
78
79      // Line 69: Can override default method (optional)
80      @Override
81      public void pause() {
82          System.out.println("Music paused at 2:15");
83      }
84
85      // Line 74: Additional method
86      public void changeSong(String newSong) {
87          this.currentSong = newSong;
88          System.out.println("Changed to: " + newSong);
89      }
90  }
91
92  // ==================== CLASS IMPLEMENTING MULTIPLE INTERFACES
        ====================
93  class SmartTV implements Playable, VolumeControllable, Connectable {
94      private int volume;
95      private boolean isConnected;
96      private String currentChannel;
97
98      public SmartTV(String channel) {
99          this.volume = 20;
100         this.isConnected = false;
101         this.currentChannel = channel;
102     }
103
104     // Implement Playable methods
105     @Override
106     public void play() {
107         System.out.println("Playing TV channel: " + currentChannel);
108     }
109
```

```
110     @Override
111     public void stop() {
112         System.out.println("TV turned off");
113     }
114 
115     // Implement VolumeControllable methods
116     @Override
117     public void increaseVolume() {
118         if (volume < MAX_VOLUME) {
119             volume += 5;
120             System.out.println("Volume increased to: " + volume);
121         } else {
122             System.out.println("Volume at maximum: " + MAX_VOLUME);
123         }
124     }
125 
126     @Override
127     public void decreaseVolume() {
128         if (volume > MIN_VOLUME) {
129             volume -= 5;
130             System.out.println("Volume decreased to: " + volume);
131         } else {
132             System.out.println("Volume at minimum: " + MIN_VOLUME);
133         }
134     }
135 
136     @Override
137     public void setVolume(int level) {
138         if (level >= MIN_VOLUME && level <= MAX_VOLUME) {
139             volume = level;
140             System.out.println("Volume set to: " + volume);
141         }
142     }
143 
144     // Implement Connectable methods
145     @Override
146     public void connect() {
147         isConnected = true;
148         System.out.println("SmartTV connected to WiFi");
149     }
150 
151     @Override
152     public void disconnect() {
153         isConnected = false;
154         System.out.println("SmartTV disconnected");
155     }
156 
157     @Override
158     public String getConnectionStatus() {
159         return isConnected ? "Connected" : "Disconnected";
160     }
161 
162     // Line 140: Override default method
163     @Override
164     public void pause() {
165         System.out.println("TV paused - showing pause screen");
166     }
167 
```

```
    // Additional methods
    public void changeChannel(String channel) {
        this.currentChannel = channel;
        System.out.println("Changed to channel: " + channel);
    }
}

// ==================== CLASS IMPLEMENTING EXTENDED INTERFACE ====================
class VideoRecorder implements AdvancedPlayable {
    private String currentVideo;
    private boolean isRecording;

    public VideoRecorder(String video) {
        this.currentVideo = video;
        this.isRecording = false;
    }

    // Implement Playable methods (inherited from parent interface)
    @Override
    public void play() {
        System.out.println("Playing video: " + currentVideo);
        System.out.println("Resolution: 1080p");
    }

    @Override
    public void stop() {
        System.out.println("Video stopped");
        if (isRecording) {
            isRecording = false;
            System.out.println("Recording stopped");
        }
    }

    // Implement AdvancedPlayable methods
    @Override
    public void fastForward() {
        System.out.println("Fast forwarding video 2x");
    }

    @Override
    public void rewind() {
        System.out.println("Rewinding video");
    }

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

    // Additional method
    public String getRecordingStatus() {
        return isRecording ? "Recording in progress" : "Not recording";
    }
}

```

```java
// ==================== INTERFACE WITH PRIVATE METHODS (Java 9+) ====================
interface Logger {
    void log(String message);

    // Private method for internal use (Java 9+)
    private void logWithTimestamp(String message) {
        System.out.println(java.time.LocalDateTime.now() + ": " + message);
    }

    // Default method using private method
    default void logInfo(String message) {
        logWithTimestamp("[INFO] " + message);
    }

    default void logError(String message) {
        logWithTimestamp("[ERROR] " + message);
    }
}

class FileLogger implements Logger {
    @Override
    public void log(String message) {
        System.out.println("Logging to file: " + message);
    }
}

// ==================== CLASS WITH INTERFACE AND ABSTRACT CLASS ====================
abstract class Device {
    protected String brand;
    protected String model;

    public Device(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }

    public abstract void powerOn();
    public abstract void powerOff();

    public void displayInfo() {
        System.out.println(brand + " " + model);
    }
}

// Line 220: Class extends abstract class AND implements interface
class SmartPhone extends Device implements Playable, Connectable {
    private String currentApp;

    public SmartPhone(String brand, String model) {
        super(brand, model);
        this.currentApp = "Home Screen";
    }

    // Implement Device abstract methods
    @Override
```

```
279     public void powerOn() {
280         System.out.println("SmartPhone booting up...");
281         System.out.println("Welcome to " + brand + " " + model);
282     }
283 
284     @Override
285     public void powerOff() {
286         System.out.println("Shutting down SmartPhone...");
287     }
288 
289     // Implement Playable methods
290     @Override
291     public void play() {
292         System.out.println("Playing media on " + currentApp);
293     }
294 
295     @Override
296     public void stop() {
297         System.out.println("Media stopped");
298     }
299 
300     // Implement Connectable methods
301     @Override
302     public void connect() {
303         System.out.println("Connected to cellular network");
304     }
305 
306     @Override
307     public void disconnect() {
308         System.out.println("Disconnected from network");
309     }
310 
311     // Additional methods
312     public void openApp(String appName) {
313         this.currentApp = appName;
314         System.out.println("Opened: " + appName);
315     }
316 }
317 
318 // ==================== MAIN CLASS ====================
319 public class InterfaceDemo {
320     public static void main(String[] args) {
321         System.out.println("=== INTERFACE DEMONSTRATION ===\n");
322 
323         // Line 265: Access interface constant
324         System.out.println("Interface Constant: " + Playable.TYPE);
325 
326         // Line 268: Call interface static method
327         Playable.showCategory();
328         System.out.println();
329 
330         System.out.println("=== SINGLE INTERFACE IMPLEMENTATION ===");
331         MusicPlayer player = new MusicPlayer("Imagine - John Lennon");
332         player.play();
333         player.pause();  // Overridden default method
334         player.stop();
335         player.changeSong("Bohemian Rhapsody - Queen");
336         player.play();
```

```
337         System.out.println();
338
339         System.out.println("=== MULTIPLE INTERFACE IMPLEMENTATION ===")
                ;
340         SmartTV tv = new SmartTV("Discovery");
341         tv.connect();
342         System.out.println("Connection Status: " + tv.
                getConnectionStatus());
343         tv.play();
344         tv.increaseVolume();
345         tv.increaseVolume();
346         tv.setVolume(50);
347         tv.mute();  // Default method from VolumeControllable
348         tv.pause(); // Overridden default method from Playable
349         tv.stop();
350         tv.disconnect();
351         System.out.println();
352
353         System.out.println("=== EXTENDED INTERFACE IMPLEMENTATION ===")
                ;
354         VideoRecorder recorder = new VideoRecorder("Vacation.mp4");
355         recorder.play();
356         recorder.fastForward();
357         recorder.rewind();
358         recorder.record();
359         System.out.println("Status: " + recorder.getRecordingStatus());
360         recorder.stop();
361         System.out.println();
362
363         System.out.println("=== INTERFACE AS REFERENCE TYPE ===");
364         // Line 298: Interface reference can hold any implementing
                object
365         Playable[] playableDevices = new Playable[3];
366         playableDevices[0] = new MusicPlayer("Shape of You - Ed Sheeran
                ");
367         playableDevices[1] = new SmartTV("National Geographic");
368         playableDevices[2] = new VideoRecorder("Tutorial.mp4");
369
370         System.out.println("Processing all playable devices:");
371         for (Playable device : playableDevices) {
372             System.out.println("\nDevice: " +
373                                device.getClass().getSimpleName());
374             device.play();
375             device.pause();
376             device.stop();
377         }
378         System.out.println();
379
380         System.out.println("=== CLASS WITH ABSTRACT CLASS AND INTERFACE
                ===");
381         SmartPhone phone = new SmartPhone("Apple", "iPhone 15");
382         phone.displayInfo();
383         phone.powerOn();
384         phone.connect();
385         phone.openApp("Spotify");
386         phone.play();
387         phone.pause();
388         phone.stop();
```

```
389         phone.powerOff();
390         System.out.println();
391
392         System.out.println("=== LOGGER INTERFACE WITH PRIVATE METHODS
                ===");
393         Logger logger = new FileLogger();
394         logger.logInfo("Application started");
395         logger.logError("File not found");
396         logger.log("Custom log message");
397         System.out.println();
398
399         System.out.println("=== INTERFACE BENEFITS ===");
400         System.out.println("1. Multiple Inheritance: Class can
                implement multiple interfaces");
401         System.out.println("2. Contract Enforcement: Forces
                implementing specific behavior");
402         System.out.println("3. Loose Coupling: Code depends on
                interface, not implementation");
403         System.out.println("4. Polymorphism: Interface reference can
                hold any implementing object");
404         System.out.println("5. Default Methods: Provide backward
                compatibility");
405         System.out.println("6. Static Methods: Utility methods in
                interfaces");
406
407         System.out.println("\n=== DIFFERENCES: ABSTRACT CLASS VS
                INTERFACE ===");
408         System.out.println("| Feature                | Abstract Class
                    | Interface                |");
409         System.out.println("
                |------------------------|--------------------------|--------------------------
                ");
410         System.out.println("| Multiple Inheritance   | No (extends one
                class)  | Yes (implements multiple)|");
411         System.out.println("| Variables              | Can have
                instance vars  | Only constants           |");
412         System.out.println("| Constructors           | Yes
                                | No                       |");
413         System.out.println("| Method Implementation  | Partial (
                abstract/concrete)| Java 8+: default/static|");
414         System.out.println("| Access Modifiers       | Any
                                | Public only (implicitly)|");
415         System.out.println("| When to Use            | IS-A
                relationship       | CAN-DO relationship     |");
416
417         System.out.println("\n=== REAL-WORLD ANALOGY ===");
418         System.out.println("Abstract Class: Vehicle (Car IS-A Vehicle)"
                );
419         System.out.println("Interface: Drivable (Car CAN-DO Drive)");
420         System.out.println("Interface: Flyable (Airplane CAN-DO Fly)");
421         System.out.println("Class Airplane extends Vehicle implements
                Flyable");
422
423         // Line 349: Testing instanceof with interfaces
424         System.out.println("\n=== INSTANCEOF WITH INTERFACES ===");
425         Object obj = new SmartTV("BBC");
426         System.out.println("obj instanceof SmartTV: " + (obj instanceof
                SmartTV));
```

```
427         System.out.println("obj instanceof Playable: " + (obj
                instanceof Playable));
428         System.out.println("obj instanceof VolumeControllable: " +
429                            (obj instanceof VolumeControllable));
430         System.out.println("obj instanceof Connectable: " + (obj
                instanceof Connectable));
431
432         // Line 357: Multiple interface references
433         System.out.println("\n=== MULTIPLE INTERFACE REFERENCES ===");
434         SmartTV myTV = new SmartTV("CNN");
435         Playable p = myTV;
436         VolumeControllable v = myTV;
437         Connectable c = myTV;
438
439         p.play();
440         v.increaseVolume();
441         c.connect();
442     }
443 }
```

Listing 8: Complete Interface Implementation

InterfaceDemo Program Output

```
=== INTERFACE DEMONSTRATION ===

Interface Constant: Entertainment
Category: Entertainment

=== SINGLE INTERFACE IMPLEMENTATION ===
Playing song: Imagine - John Lennon
Duration: 3:45
Music paused at 2:15
Music stopped
Changed to: Bohemian Rhapsody - Queen
Playing song: Bohemian Rhapsody - Queen
Duration: 3:45

=== MULTIPLE INTERFACE IMPLEMENTATION ===
SmartTV connected to WiFi
Connection Status: Connected
Playing TV channel: Discovery
Volume increased to: 25
Volume increased to: 30
Volume set to: 50
Muted
Volume set to: 0
TV paused - showing pause screen
TV turned off
SmartTV disconnected
```

```
=== EXTENDED INTERFACE IMPLEMENTATION ===
Playing video: Vacation.mp4
Resolution: 1080p
Fast forwarding video 2x
Rewinding video
Recording started
Status: Recording in progress
Video stopped
Recording stopped

=== INTERFACE AS REFERENCE TYPE ===
Processing all playable devices:

Device: MusicPlayer
Playing song: Shape of You - Ed Sheeran
Duration: 3:45
Music paused at 2:15
Music stopped
Device: SmartTV
Playing TV channel: National Geographic
TV paused - showing pause screen
TV turned off
Device: VideoRecorder
Playing video: Tutorial.mp4
Resolution: 1080p
Video paused - showing pause screen
Video stopped
=== CLASS WITH ABSTRACT CLASS AND INTERFACE ===
Apple iPhone 15
SmartPhone booting up...
Welcome to Apple iPhone 15
Connected to cellular network
Opened: Spotify
Playing media on Spotify
Media paused at 2:15
Media stopped
Shutting down SmartPhone...
=== LOGGER INTERFACE WITH PRIVATE METHODS ===
2024-01-15T14:30:45.123: [INFO] Application started
2024-01-15T14:30:45.124: [ERROR] File not found
Logging to file: Custom log message
=== INTERFACE BENEFITS ===
1. Multiple Inheritance: Class can implement multiple interfaces
2. Contract Enforcement: Forces implementing specific behavior
3. Loose Coupling: Code depends on interface, not implementation
4. Polymorphism: Interface reference can hold any implementing object
5. Default Methods: Provide backward compatibility
```

```
6. Static Methods: Utility methods in interfaces
=== DIFFERENCES: ABSTRACT CLASS VS INTERFACE ===
| Feature                | Abstract Class           | Interface                |
|------------------------|--------------------------|--------------------------|
| Multiple Inheritance   | No (extends one class)   | Yes (implements multiple)|
| Variables              | Can have instance vars   | Only constants           |
| Constructors           | Yes                      | No                       |
| Method Implementation  | Partial (abstract/concrete)| Java 8+: default/static|
| Access Modifiers       | Any                      | Public only (implicitly)|
| When to Use            | IS-A relationship        | CAN-DO relationship      |
=== REAL-WORLD ANALOGY ===
Abstract Class: Vehicle (Car IS-A Vehicle)
Interface: Drivable (Car CAN-DO Drive)
Interface: Flyable (Airplane CAN-DO Fly)
Class Airplane extends Vehicle implements Flyable

=== INSTANCEOF WITH INTERFACES ===
obj instanceof SmartTV: true
obj instanceof Playable: true
obj instanceof VolumeControllable: true
obj instanceof Connectable: true
=== MULTIPLE INTERFACE REFERENCES ===
Playing TV channel: CNN
Volume increased to: 5
SmartTV connected to WiFi
```

6 Packages

```java
// ==================== FILE 1: com/university/student/Student.java ====================
// Line 1: Package declaration - MUST be first statement
package com.university.student;

// Line 4: Import statement - use classes from other packages
import java.util.Date;

// Line 7: Public class - accessible from other packages
public class Student {
    private String studentId;
    private String name;
    private Date dateOfBirth;
    private String department;
    private double cgpa;

    // Package-private constructor
    Student(String studentId, String name) {
        this.studentId = studentId;
        this.name = name;
    }

```

```
    // Public constructor
    public Student(String studentId, String name, Date dob, String dept) {
        this(studentId, name);
        this.dateOfBirth = dob;
        this.department = dept;
    }

    // Public getters
    public String getStudentId() {
        return studentId;
    }

    public String getName() {
        return name;
    }

    public String getDepartment() {
        return department;
    }

    // Package-private getter
    Date getDateOfBirth() {
        return dateOfBirth;
    }

    // Protected method - accessible in subclasses and same package
    protected void setCGPA(double cgpa) {
        if (cgpa >= 0 && cgpa <= 10) {
            this.cgpa = cgpa;
        }
    }

    // Public method
    public double getCGPA() {
        return cgpa;
    }

    // Default (package-private) method
    void displayBasicInfo() {
        System.out.println("ID: " + studentId + ", Name: " + name);
    }

    // Public method
    public void displayFullInfo() {
        displayBasicInfo();
        System.out.println("Department: " + department);
        System.out.println("CGPA: " + cgpa);
    }
}

// ==================== FILE 2: com/university/student/GraduateStudent.java ====================
package com.university.student;

// Line 66: Same package, no import needed for Student
public class GraduateStudent extends Student {
    private String researchTopic;
```

```
78      private String supervisor;
79
80      public GraduateStudent(String studentId, String name,
81                             String researchTopic, String supervisor) {
82          // Line 71: Can call protected constructor from same package
83          super(studentId, name);
84          this.researchTopic = researchTopic;
85          this.supervisor = supervisor;
86      }
87
88      // Line 77: Can access protected method from parent
89      public void updateCGPA(double cgpa) {
90          setCGPA(cgpa);  // Protected method accessible
91      }
92
93      // Line 82: Cannot access package-private methods from different
            package
94      // void test() {
95      //     getDateOfBirth();  // Not accessible - package-private
96      // }
97
98      public void displayResearchInfo() {
99          System.out.println("Research Topic: " + researchTopic);
100         System.out.println("Supervisor: " + supervisor);
101     }
102 }
103
104 // ==================== FILE 3: com/university/faculty/Professor.java
        ====================
105 package com.university.faculty;
106
107 // Line 93: Different package, need import
108 import com.university.student.Student;
109 import java.util.ArrayList;
110 import java.util.List;
111
112 public class Professor {
113     private String professorId;
114     private String name;
115     private String department;
116     private List<Student> students;
117
118     public Professor(String professorId, String name, String department
            ) {
119         this.professorId = professorId;
120         this.name = name;
121         this.department = department;
122         this.students = new ArrayList<>();
123     }
124
125     // Line 107: Can access public methods from Student
126     public void addStudent(Student student) {
127         if (student.getDepartment().equals(this.department)) {
128             students.add(student);
129             System.out.println("Added student: " + student.getName());
130         }
131     }
132
```

```
    // Line 114: Cannot access package-private or protected members
    public void displayStudents() {
        System.out.println("\nProfessor: " + name);
        System.out.println("Department: " + department);
        System.out.println("Students under guidance:");

        for (Student student : students) {
            // Line 121: Can only call public methods
            System.out.println("  - " + student.getName() +
                               " (ID: " + student.getStudentId() + ")");

            // Cannot access these:
            // student.displayBasicInfo();  // Package-private
            // student.getDateOfBirth();    // Package-private
            // student.setCGPA(9.5);        // Protected
        }
    }
}

// ==================== FILE 4: com/university/courses/Course.java ====================
package com.university.courses;

import com.university.student.Student;

public class Course {
    private String courseCode;
    private String courseName;
    private int credits;
    private Student[] enrolledStudents;
    private int studentCount;

    public Course(String courseCode, String courseName, int credits) {
        this.courseCode = courseCode;
        this.courseName = courseName;
        this.credits = credits;
        this.enrolledStudents = new Student[50];
        this.studentCount = 0;
    }

    public void enrollStudent(Student student) {
        if (studentCount < enrolledStudents.length) {
            enrolledStudents[studentCount++] = student;
            System.out.println(student.getName() + " enrolled in " +
                courseName);
        }
    }

    public void displayCourseInfo() {
        System.out.println("\nCourse: " + courseCode + " - " +
            courseName);
        System.out.println("Credits: " + credits);
        System.out.println("Enrolled Students: " + studentCount);
    }
}

// ==================== FILE 5: com/university/UniversityApp.java ====================
```

```java
package com.university;

// Line 160: Import classes from different packages
import com.university.student.Student;
import com.university.student.GraduateStudent;
import com.university.faculty.Professor;
import com.university.courses.Course;
import java.util.Date;

// Line 167: Static import for utility classes
import static java.lang.Math.PI;
import static java.lang.System.out;

public class UniversityApp {
    public static void main(String[] args) {
        out.println("=== UNIVERSITY MANAGEMENT SYSTEM ===\n");

        // Line 174: Create students
        Student undergrad = new Student("UG2023001", "Alice Johnson",
                                        new Date(), "Computer Science");
        GraduateStudent grad = new GraduateStudent("PG2023001", "Bob
            Smith",
                                                   "AI in Healthcare", "
                                                      Dr. Wilson");

        // Line 179: Set CGPA - different access based on relationship
        // undergrad.setCGPA(8.5);  // Not accessible - protected
        grad.updateCGPA(9.2);       // Accessible through public method

        // Line 183: Create professor
        Professor prof = new Professor("PROF101", "Dr. James", "
            Computer Science");

        // Line 186: Add students to professor
        prof.addStudent(undergrad);
        prof.addStudent(grad);

        // Line 190: Create course
        Course cs101 = new Course("CS101", "Programming Fundamentals",
            4);
        cs101.enrollStudent(undergrad);
        cs101.enrollStudent(grad);

        // Line 195: Display information
        out.println("\n=== STUDENT INFORMATION ===");
        undergrad.displayFullInfo();
        out.println();
        grad.displayFullInfo();
        grad.displayResearchInfo();

        out.println("\n=== FACULTY INFORMATION ===");
        prof.displayStudents();

        out.println("\n=== COURSE INFORMATION ===");
        cs101.displayCourseInfo();

        // Line 207: Demonstrate static import
        out.println("\n=== STATIC IMPORT DEMONSTRATION ===");
```

```
241         out.println("Value of PI (static import): " + PI);
242         out.println("Area of circle with radius 5: " + (PI * 5 * 5));
243 
244         // Line 212: Package information
245         out.println("\n=== PACKAGE INFORMATION ===");
246         out.println("Student class package: " + Student.class.
                getPackageName());
247         out.println("Professor class package: " + Professor.class.
                getPackageName());
248         out.println("Course class package: " + Course.class.
                getPackageName());
249         out.println("Current class package: " + UniversityApp.class.
                getPackageName());
250 
251         // Line 219: Access modifiers demonstration
252         out.println("\n=== ACCESS MODIFIERS ACROSS PACKAGES ===");
253         out.println("Public: Accessible from anywhere");
254         out.println("Protected: Accessible in same package + subclasses
                ");
255         out.println("Default (Package-private): Accessible only in same
                 package");
256         out.println("Private: Accessible only in same class");
257 
258         // Line 226: CLASSPATH and compilation
259         out.println("\n=== COMPILATION AND EXECUTION ===");
260         out.println("To compile: javac -d . UniversityApp.java");
261         out.println("To run: java com.university.UniversityApp");
262         out.println("Directory structure matches package structure");
263 
264         // Line 232: Import entire package (not recommended for clarity
                )
265         // import com.university.student.*;
266 
267         // Line 235: Built-in package usage
268         out.println("\n=== BUILT-IN PACKAGE USAGE ===");
269         java.util.Scanner scanner = new java.util.Scanner(System.in);
270         out.print("Enter university name: ");
271         String uniName = scanner.nextLine();
272         out.println("Welcome to " + uniName + " University System!");
273         scanner.close();
274     }
275 }
276 
277 // ==================== FILE 6: DefaultPackageDemo.java
        ====================
278 // This file is in default package (no package declaration)
279 // Not recommended for production code
280 
281 class DefaultPackageDemo {
282     // Classes in default package can only be accessed by other classes
283     // in default package in the same directory
284 }
285 
286 // ==================== COMPILATION AND DIRECTORY STRUCTURE
        ====================
287 /*
288 Directory Structure:
289 university-system/
```

```
290             com/
291                   university/
292                       UniversityApp.java
293                       student/
294                             Student.java
295                             GraduateStudent.java
296                       faculty/
297                             Professor.java
298                       courses/
299                           Course.java
300             DefaultPackageDemo.java
301
302 Compilation Steps:
303 1. javac -d . com/university/UniversityApp.java
304    (This compiles all dependent files automatically)
305
306 2. java com.university.UniversityApp
307
308 OR compile separately:
309 1. javac -d . com/university/student/Student.java
310 2. javac -d . com/university/student/GraduateStudent.java
311 3. javac -d . com/university/faculty/Professor.java
312 4. javac -d . com/university/courses/Course.java
313 5. javac -d . com/university/UniversityApp.java
314 6. java com.university.UniversityApp
315 */
```

Listing 9: Complete Package Implementation with Multiple Files

UniversityApp Program Output

```
=== UNIVERSITY MANAGEMENT SYSTEM ===

Added student: Alice Johnson
Added student: Bob Smith
Alice Johnson enrolled in Programming Fundamentals
Bob Smith enrolled in Programming Fundamentals

=== STUDENT INFORMATION ===
ID: UG2023001, Name: Alice Johnson
Department: Computer Science
CGPA: 0.0

ID: PG2023001, Name: Bob Smith
Department: null
CGPA: 9.2
Research Topic: AI in Healthcare
Supervisor: Dr. Wilson

=== FACULTY INFORMATION ===

Professor: Dr. James
Department: Computer Science
Students under guidance:
  - Alice Johnson (ID: UG2023001)
  - Bob Smith (ID: PG2023001)

=== COURSE INFORMATION ===

Course: CS101 - Programming Fundamentals
Credits: 4
Enrolled Students: 2

=== STATIC IMPORT DEMONSTRATION ===
Value of PI (static import): 3.141592653589793
Area of circle with radius 5: 78.53981633974483

=== PACKAGE INFORMATION ===
Student class package: com.university.student
Professor class package: com.university.faculty
Course class package: com.university.courses
Current class package: com.university

=== ACCESS MODIFIERS ACROSS PACKAGES ===
Public: Accessible from anywhere
Protected: Accessible in same package + subclasses
Default (Package-private): Accessible only in same package
Private: Accessible only in same class

=== COMPILATION AND EXECUTION ===
To compile: javac -d . UniversityApp.java
To run: java com.university.UniversityApp
Directory structure matches package structure
```

7 Static Members

```java
// ==================== COMPANY EMPLOYEE SYSTEM ====================
class Employee {
    // Line 4: Instance variables - unique for each object
    private int id;
    private String name;
    private double salary;

    // Line 9: Static variable - shared by all objects
    // Tracks company-wide information
    private static int totalEmployees = 0;
    private static double totalSalaryExpense = 0.0;

    // Line 14: Static constant - shared, cannot be changed
    public static final String COMPANY_NAME = "Tech Solutions Inc.";
    public static final double MAX_SALARY = 1000000.0;

    // Line 18: Static block - executed when class is loaded
    static {
        System.out.println("Employee class loaded into memory");
        System.out.println("Company: " + COMPANY_NAME);
        System.out.println("Initializing static resources...");
        initializeStaticData();
    }

    // Line 25: Static method - can be called without object
    private static void initializeStaticData() {
        // Can only access static members
        totalEmployees = 0;
        totalSalaryExpense = 0.0;
        System.out.println("Static data initialized");
    }

    // Line 32: Instance constructor
    public Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;

        // Line 36: Validate salary against static constant
        if (salary <= MAX_SALARY) {
            this.salary = salary;
        } else {
            this.salary = MAX_SALARY;
            System.out.println("Salary capped at maximum: $" +
                MAX_SALARY);
        }

        // Line 44: Update static variables
        totalEmployees++;
        totalSalaryExpense += this.salary;

        System.out.println("Employee created: " + name +
                           " (Total employees: " + totalEmployees + ")");
    }

    // Line 51: Instance methods
```

```
55      public void displayInfo() {
56          System.out.println("ID: " + id + ", Name: " + name +
57                             ", Salary: $" + salary);
58      }
59
60      // Line 56: Instance method accessing static variable
61      public void showCompanyStats() {
62          System.out.println("Company: " + COMPANY_NAME);
63          System.out.println("You are employee #" + totalEmployees);
64          System.out.println("Total salary expense: $" +
                totalSalaryExpense);
65      }
66
67      // Line 63: Static method - utility method
68      public static double calculateAnnualExpense() {
69          // Can only access static members
70          return totalSalaryExpense * 12;
71      }
72
73      // Line 69: Static method - factory method pattern
74      public static Employee createManager(String name, double salary) {
75          System.out.println("Creating manager position...");
76          // Generate manager ID (M prefix)
77          int managerId = totalEmployees + 1000;
78          return new Employee(managerId, name, salary * 1.5); // Managers
                get 50% more
79      }
80
81      // Line 77: Static method to get company info
82      public static String getCompanyInfo() {
83          return COMPANY_NAME + " | Employees: " + totalEmployees +
84                 " | Monthly Expense: $" + totalSalaryExpense;
85      }
86
87      // Line 83: Getters for static variables
88      public static int getTotalEmployees() {
89          return totalEmployees;
90      }
91
92      public static double getTotalSalaryExpense() {
93          return totalSalaryExpense;
94      }
95
96      // Line 90: Static nested class
97      public static class EmployeeStats {
98          public static double getAverageSalary() {
99              if (totalEmployees == 0) return 0;
100             return totalSalaryExpense / totalEmployees;
101         }
102
103         public static void printStatistics() {
104             System.out.println("\n=== EMPLOYEE STATISTICS ===");
105             System.out.println("Total Employees: " + totalEmployees);
106             System.out.println("Total Monthly Salary: $" +
                    totalSalaryExpense);
107             System.out.printf("Average Salary: $%.2f\n",
                    getAverageSalary());
```

```
            System.out.println("Annual Expense: $" +
                calculateAnnualExpense());
        }
    }
}

// ==================== MATH UTILITY CLASS ====================
// Line 107: Utility class with only static members
// Often declared final with private constructor
final class MathUtils {
    // Line 110: Private constructor to prevent instantiation
    private MathUtils() {
        throw new AssertionError("Cannot instantiate utility class");
    }

    // Line 115: Static constants
    public static final double PI = 3.141592653589793;
    public static final double E = 2.718281828459045;

    // Line 119: Static methods for mathematical operations
    public static int add(int a, int b) {
        return a + b;
    }

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

    public static int factorial(int n) {
        if (n <= 1) return 1;
        return n * factorial(n - 1);
    }

    public static boolean isPrime(int number) {
        if (number <= 1) return false;
        for (int i = 2; i <= Math.sqrt(number); i++) {
            if (number % i == 0) return false;
        }
        return true;
    }

    public static double circleArea(double radius) {
        return PI * radius * radius;
    }

    // Line 140: Static block for initialization
    static {
        System.out.println("MathUtils class loaded");
    }
}

// ==================== BANK ACCOUNT WITH SERIAL NUMBER
    ====================
class BankAccount {
    // Line 147: Static variable for auto-incrementing account numbers
    private static int nextAccountNumber = 1000;

    // Line 150: Instance variables
```

```
164     private int accountNumber;
165     private String accountHolder;
166     private double balance;
167 
168     // Line 155: Static method to generate next account number
169     private static int generateAccountNumber() {
170         return nextAccountNumber++;
171     }
172 
173     public BankAccount(String accountHolder, double initialDeposit) {
174         this.accountNumber = generateAccountNumber();  // Use static
                method
175         this.accountHolder = accountHolder;
176         this.balance = initialDeposit;
177         System.out.println("Account #" + accountNumber + " created for
                " + accountHolder);
178     }
179 
180     // Line 165: Static method to get next available account number
181     public static int getNextAccountNumber() {
182         return nextAccountNumber;
183     }
184 
185     // Line 170: Instance methods
186     public void displayAccountInfo() {
187         System.out.println("Account #: " + accountNumber);
188         System.out.println("Holder: " + accountHolder);
189         System.out.println("Balance: $" + balance);
190     }
191 
192     // Line 177: Method using both instance and static data
193     public void showBankInfo() {
194         System.out.println("Next available account number: " +
                getNextAccountNumber());
195     }
196 }
197 
198 // ==================== SINGLETON PATTERN USING STATIC
        ====================
199 class DatabaseConnection {
200     // Line 185: Static variable to hold single instance
201     private static DatabaseConnection instance;
202 
203     // Line 188: Instance variables
204     private String connectionString;
205     private boolean isConnected;
206 
207     // Line 192: Private constructor - prevents external instantiation
208     private DatabaseConnection() {
209         this.connectionString = "jdbc:mysql://localhost:3306/mydb";
210         this.isConnected = false;
211         System.out.println("DatabaseConnection instance created");
212     }
213 
214     // Line 199: Static method to get singleton instance
215     public static DatabaseConnection getInstance() {
216         if (instance == null) {
217             instance = new DatabaseConnection();
```

```
218         }
219         return instance;
220     }
221 
222     // Line 206: Instance methods
223     public void connect() {
224         if (!isConnected) {
225             isConnected = true;
226             System.out.println("Connected to database: " +
                    connectionString);
227         }
228     }
229 
230     public void disconnect() {
231         if (isConnected) {
232             isConnected = false;
233             System.out.println("Disconnected from database");
234         }
235     }
236 
237     // Line 218: Static method to check if instance exists
238     public static boolean isInstanceCreated() {
239         return instance != null;
240     }
241 }
242 
243 // ==================== MAIN CLASS ====================
244 public class StaticMembersDemo {
245     // Line 225: Static variable in main class
246     private static int demoCounter = 0;
247 
248     // Line 228: Static block in main class
249     static {
250         System.out.println("StaticMembersDemo class loaded");
251         System.out.println("Initializing demonstration...");
252         demoCounter = 1;
253     }
254 
255     // Line 235: Static method in main class
256     public static void incrementCounter() {
257         demoCounter++;
258     }
259 
260     public static void main(String[] args) {
261         System.out.println("=== STATIC MEMBERS DEMONSTRATION ===\n");
262 
263         // Line 242: Access static members without object
264         System.out.println("Company Name (static constant): " +
                Employee.COMPANY_NAME);
265         System.out.println("Max Salary (static constant): $" + Employee
                .MAX_SALARY);
266 
267         System.out.println("\n=== CREATING EMPLOYEES ===");
268 
269         // Line 247: Create employee objects
270         Employee emp1 = new Employee(101, "Alice", 50000);
271         Employee emp2 = new Employee(102, "Bob", 60000);
272         Employee emp3 = new Employee(103, "Charlie", 75000);
```

```
273
274         // Line 252: Try to create employee with salary exceeding max
275         Employee emp4 = new Employee(104, "David", 1500000);
276
277         System.out.println("\n=== EMPLOYEE INFORMATION ===");
278         emp1.displayInfo();
279         emp2.displayInfo();
280         emp3.displayInfo();
281         emp4.displayInfo();
282
283         System.out.println("\n=== COMPANY STATISTICS ===");
284         // Line 261: Access static methods using class name
285         System.out.println("Total Employees: " + Employee.
               getTotalEmployees());
286         System.out.println("Total Monthly Salary: $" + Employee.
               getTotalSalaryExpense());
287         System.out.println("Annual Salary Expense: $" + Employee.
               calculateAnnualExpense());
288
289         // Line 266: Access static nested class
290         Employee.EmployeeStats.printStatistics();
291         System.out.printf("Average Salary: $%.2f\n", Employee.
               EmployeeStats.getAverageSalary());
292
293         System.out.println("\n=== USING FACTORY METHOD ===");
294         // Line 271: Static factory method
295         Employee manager = Employee.createManager("Eve", 80000);
296         manager.displayInfo();
297
298         System.out.println("\n=== MATH UTILITY CLASS ===");
299         // Line 276: Using utility class with only static methods
300         System.out.println("PI value: " + MathUtils.PI);
301         System.out.println("5! = " + MathUtils.factorial(5));
302         System.out.println("Is 17 prime? " + MathUtils.isPrime(17));
303         System.out.println("Area of circle radius 7: " + MathUtils.
               circleArea(7));
304
305         // Line 282: Cannot instantiate utility class
306         // MathUtils utils = new MathUtils();  // Compilation error
307
308         System.out.println("\n=== BANK ACCOUNT SERIAL NUMBERS ===");
309         // Line 286: Bank accounts get auto-incremented account numbers
310         BankAccount acc1 = new BankAccount("John", 1000);
311         BankAccount acc2 = new BankAccount("Sarah", 2500);
312         BankAccount acc3 = new BankAccount("Mike", 500);
313
314         System.out.println("\nAccount Information:");
315         acc1.displayAccountInfo();
316         acc2.displayAccountInfo();
317         acc3.displayAccountInfo();
318
319         System.out.println("\nNext available account number: " +
320                            BankAccount.getNextAccountNumber());
321
322         System.out.println("\n=== SINGLETON PATTERN ===");
323         // Line 300: Get singleton instance
324         DatabaseConnection db1 = DatabaseConnection.getInstance();
325         DatabaseConnection db2 = DatabaseConnection.getInstance();
```

```
326
327         System.out.println("db1 == db2: " + (db1 == db2));  // Same
                instance
328         System.out.println("Is instance created? " + DatabaseConnection
                .isInstanceCreated());
329
330         db1.connect();
331         db2.connect();  // Already connected
332
333         System.out.println("\n=== STATIC VARIABLE SHARING DEMONSTRATION
                ===");
334         // Line 310: Each employee sees same static data
335         emp1.showCompanyStats();
336         System.out.println();
337         emp2.showCompanyStats();
338
339         System.out.println("\n=== STATIC IMPORT DEMONSTRATION ===");
340         // Using static import from Math class (not shown in imports)
341         double root = Math.sqrt(25);
342         double power = Math.pow(2, 8);
343         System.out.println("Square root of 25: " + root);
344         System.out.println("2^8: " + power);
345
346         System.out.println("\n=== STATIC VS INSTANCE ===");
347         System.out.println("Static Members:");
348         System.out.println("  - Belong to class, not objects");
349         System.out.println("  - One copy shared by all objects");
350         System.out.println("  - Can be accessed without object");
351         System.out.println("  - Can only access static members");
352
353         System.out.println("\nInstance Members:");
354         System.out.println("  - Belong to objects");
355         System.out.println("  - Each object has its own copy");
356         System.out.println("  - Require object to access");
357         System.out.println("  - Can access both static and instance
                members");
358
359         System.out.println("\n=== MEMORY ALLOCATION ===");
360         System.out.println("Static variables: Stored in Method Area");
361         System.out.println("Instance variables: Stored in Heap with
                object");
362         System.out.println("Static blocks: Executed once when class
                loads");
363         System.out.println("Instance blocks: Executed for each object
                creation");
364
365         // Line 337: Demonstrate main class static members
366         System.out.println("\n=== MAIN CLASS STATIC MEMBERS ===");
367         System.out.println("Demo Counter: " + demoCounter);
368         incrementCounter();
369         System.out.println("Demo Counter after increment: " +
                demoCounter);
370     }
371 }
```

Listing 10: Complete Static Members Implementation

StaticMembersDemo Program Output

```
Employee class loaded into memory
Company: Tech Solutions Inc.
Initializing static resources...
Static data initialized
MathUtils class loaded
StaticMembersDemo class loaded
Initializing demonstration...

=== STATIC MEMBERS DEMONSTRATION ===

Company Name (static constant): Tech Solutions Inc.
Max Salary (static constant): $1000000.0

=== CREATING EMPLOYEES ===
Employee created: Alice (Total employees: 1)
Employee created: Bob (Total employees: 2)
Employee created: Charlie (Total employees: 3)
Salary capped at maximum: $1000000.0
Employee created: David (Total employees: 4)

=== EMPLOYEE INFORMATION ===
ID: 101, Name: Alice, Salary: $50000.0
ID: 102, Name: Bob, Salary: $60000.0
ID: 103, Name: Charlie, Salary: $75000.0
ID: 104, Name: David, Salary: $1000000.0

=== COMPANY STATISTICS ===
Total Employees: 4
Total Monthly Salary: $1185000.0
Annual Salary Expense: $1.422E7

=== EMPLOYEE STATISTICS ===
Total Employees: 4
Total Monthly Salary: $1185000.0
Average Salary: $296250.00
Annual Expense: $1.422E7
Average Salary: $296250.00

=== USING FACTORY METHOD ===
Creating manager position...
Employee created: Eve (Total employees: 5)
ID: 1005, Name: Eve, Salary: $120000.0

=== MATH UTILITY CLASS ===
PI value: 3.141592653589793
5! = 120
```

```
Is 17 prime? true
Area of circle radius 7: 153.93804002589985

=== BANK ACCOUNT SERIAL NUMBERS ===
Account #1000 created for John
Account #1001 created for Sarah
Account #1002 created for Mike

Account Information:
Account #: 1000
Holder: John
Balance: $1000.0
Account #: 1001
Holder: Sarah
Balance: $2500.0
Account #: 1002
Holder: Mike
Balance: $500.0

Next available account number: 1003

=== SINGLETON PATTERN ===
DatabaseConnection instance created
db1 == db2: true
Is instance created? true
Connected to database: jdbc:mysql://localhost:3306/mydb

=== STATIC VARIABLE SHARING DEMONSTRATION ===
Company: Tech Solutions Inc.
You are employee #5
Total salary expense: $1305000.0

Company: Tech Solutions Inc.
You are employee #5
Total salary expense: $1305000.0

=== STATIC IMPORT DEMONSTRATION ===
Square root of 25: 5.0
2^8: 256.0

=== STATIC VS INSTANCE ===
Static Members:
  - Belong to class, not objects
  - One copy shared by all objects
  - Can be accessed without object
  - Can only access static members
```

```
Instance Members:
  - Belong to objects
  - Each object has its own copy
  - Require object to access
  - Can access both static and instance members

=== MEMORY ALLOCATION ===
Static variables: Stored in Method Area
Instance variables: Stored in Heap with object
Static blocks: Executed once when class loads
Instance blocks: Executed for each object creation

=== MAIN CLASS STATIC MEMBERS ===
Demo Counter: 1
Demo Counter after increment: 2
```

8 Final Keyword

```java
// ==================== FINAL VARIABLES ====================
class Constants {
    // Line 4: Final instance variable - must be initialized
    private final int MAX_VALUE;

    // Line 7: Final static variable - constant
    public static final double PI = 3.141592653589793;

    // Line 10: Final instance variable initialized in constructor
    private final String DATABASE_URL;

    // Line 13: Final array reference (contents can change, reference cannot)
    private final int[] SCORES = new int[3];

    public Constants() {
        // Line 17: Must initialize final instance variables in constructor
        MAX_VALUE = 100;
        DATABASE_URL = "jdbc:mysql://localhost:3306/testdb";

        // Line 21: Initialize array elements (allowed)
        SCORES[0] = 95;
        SCORES[1] = 87;
        SCORES[2] = 92;
    }

    // Line 27: Method to demonstrate final parameter
    public double calculateArea(final double radius) {
        // radius = radius * 2;  // Cannot modify final parameter
        return PI * radius * radius;
    }

    public void displayConstants() {
```

```
        System.out.println("MAX_VALUE: " + MAX_VALUE);
        System.out.println("PI: " + PI);
        System.out.println("DATABASE_URL: " + DATABASE_URL);
        System.out.println("Scores: " + java.util.Arrays.toString(
            SCORES));

        // Line 38: Can modify array contents (reference is final, not
            contents)
        SCORES[0] = 99;  // Allowed
        // SCORES = new int[5];  // Not allowed - cannot change
            reference
    }
}

// ==================== FINAL METHOD ====================
class Vehicle {
    protected String brand;
    protected String model;

    public Vehicle(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }

    // Line 52: Final method - cannot be overridden
    public final void startEngine() {
        System.out.println(brand + " " + model + " engine started");
        System.out.println("Standard engine starting procedure");
    }

    // Line 58: Non-final method - can be overridden
    public void displayInfo() {
        System.out.println("Vehicle: " + brand + " " + model);
    }

    // Line 63: Final method with parameters
    public final double calculateDepreciation(final int years) {
        final double DEPRECIATION_RATE = 0.1;  // Final local variable
        double depreciation = 0;
        for (int i = 0; i < years; i++) {
            depreciation += 10000 * DEPRECIATION_RATE;
        }
        return depreciation;
    }
}

// Line 74: Car extends Vehicle but cannot override final methods
class Car extends Vehicle {
    private int numberOfDoors;

    public Car(String brand, String model, int doors) {
        super(brand, model);
        this.numberOfDoors = doors;
    }

    // Line 82: Can override non-final method
    @Override
    public void displayInfo() {
```

```
88          super.displayInfo();
89          System.out.println("Type: Car, Doors: " + numberOfDoors);
90      }
91
92      // Line 88: Cannot override final method
93      // public void startEngine() { }  // Compilation error
94
95      // Line 91: New method is allowed
96      public void openSunroof() {
97          System.out.println("Sunroof opened");
98      }
99  }
100
101 // ==================== FINAL CLASS ====================
102 // Line 97: Final class - cannot be extended
103 final class SecurityManager {
104     private final String SECURITY_KEY;
105
106     public SecurityManager(String key) {
107         this.SECURITY_KEY = key;
108     }
109
110     public void authenticate() {
111         System.out.println("Authenticating with key: " +
112                            SECURITY_KEY.substring(0, 3) + "...");
113     }
114
115     public final void encryptData() {
116         System.out.println("Data encrypted using AES-256");
117     }
118 }
119
120 // Line 113: Cannot extend final class
121 // class AdvancedSecurityManager extends SecurityManager { }  // Error
122
123 // ==================== IMMUTABLE CLASS USING FINAL
    ====================
124 // Line 117: Immutable class - state cannot be changed after creation
125 final class ImmutablePerson {
126     // Line 119: All fields are private and final
127     private final String name;
128     private final int age;
129     private final java.util.List<String> hobbies;  // Reference is
        final
130
131     // Line 123: Constructor initializes all fields
132     public ImmutablePerson(String name, int age, java.util.List<String>
          hobbies) {
133         this.name = name;
134         this.age = age;
135         // Line 127: Create defensive copy for mutable fields
136         this.hobbies = new java.util.ArrayList<>(hobbies);
137     }
138
139     // Line 131: Only getters, no setters
140     public String getName() {
141         return name;
142     }
```

```
143
144     public int getAge() {
145         return age;
146     }
147
148     // Line 138: Return defensive copy for mutable fields
149     public java.util.List<String> getHobbies() {
150         return new java.util.ArrayList<>(hobbies);
151     }
152
153     @Override
154     public String toString() {
155         return "ImmutablePerson{name='" + name + "', age=" + age +
156                 ", hobbies=" + hobbies + "}";
157     }
158 }
159
160 // ==================== FINAL IN INHERITANCE HIERARCHY
    ====================
161 class Shape {
162     // Line 151: Final method with algorithm that shouldn't change
163     public final double calculateArea() {
164         return getLength() * getWidth();
165     }
166
167     // Line 156: Abstract methods to be implemented by subclasses
168     protected double getLength() {
169         return 0;
170     }
171
172     protected double getWidth() {
173         return 0;
174     }
175 }
176
177 // Line 164: Rectangle provides implementation for abstract methods
178 class Rectangle extends Shape {
179     private final double length;
180     private final double width;
181
182     public Rectangle(double length, double width) {
183         this.length = length;
184         this.width = width;
185     }
186
187     @Override
188     protected double getLength() {
189         return length;
190     }
191
192     @Override
193     protected double getWidth() {
194         return width;
195     }
196
197     // Line 179: Cannot override final method
198     // public double calculateArea() { return length * width * 2; }  //
        Error
```

```java
}

// ==================== FINAL WITH STATIC ====================
class Configuration {
    // Line 185: Final static - compile-time constant
    public static final int MAX_USERS = 1000;

    // Line 188: Final static initialized in static block
    public static final String DATABASE_URL;

    static {
        DATABASE_URL = System.getenv("DB_URL");
        if (DATABASE_URL == null) {
            // Cannot assign to final variable here if not initialized
            // DATABASE_URL = "default";  // Would need to be declared without initialization
        }
    }

    // Line 199: Blank final static
    public static final String API_KEY;

    static {
        API_KEY = "secret_" + System.currentTimeMillis();
    }
}

// ==================== MAIN CLASS ====================
public class FinalKeywordDemo {
    // Line 208: Final variable in main class
    private final String DEMO_NAME = "Final Keyword Demonstration";

    public static void main(String[] args) {
        System.out.println("=== FINAL KEYWORD DEMONSTRATION ===\n");

        // Line 214: Final local variable
        final int MAX_ATTEMPTS = 3;
        System.out.println("Max Attempts (final local): " + MAX_ATTEMPTS);
        // MAX_ATTEMPTS = 5;  // Cannot change final variable

        // Line 219: Constants class demonstration
        System.out.println("\n=== FINAL VARIABLES ===");
        Constants constants = new Constants();
        System.out.println("PI constant: " + Constants.PI);
        constants.displayConstants();

        double area = constants.calculateArea(7.0);
        System.out.printf("Area of circle radius 7: %.2f\n", area);

        System.out.println("\n=== FINAL METHOD ===");
        Vehicle vehicle = new Vehicle("Generic", "V100");
        vehicle.startEngine();  // Final method
        vehicle.displayInfo();

        Car car = new Car("Toyota", "Camry", 4);
        car.startEngine();  // Inherits final method
        car.displayInfo();  // Overridden method
```

```
255         car.openSunroof();
256
257         // Line 236: Polymorphism with final method
258         Vehicle v = new Car("Honda", "Civic", 2);
259         v.startEngine();  // Still calls Vehicle's final method
260         v.displayInfo();  // Calls Car's overridden method
261
262         System.out.println("\n=== FINAL CLASS ===");
263         SecurityManager security = new SecurityManager("secret123");
264         security.authenticate();
265         security.encryptData();
266
267         System.out.println("\n=== IMMUTABLE CLASS ===");
268         java.util.List<String> hobbies = new java.util.ArrayList<>();
269         hobbies.add("Reading");
270         hobbies.add("Swimming");
271
272         ImmutablePerson person = new ImmutablePerson("Alice", 30,
                hobbies);
273         System.out.println("Original: " + person);
274
275         // Line 252: Try to modify the list
276         hobbies.add("Gaming");  // Original list modified
277         System.out.println("After modifying original list: " + person);
278
279         // Line 256: Try to modify through getter
280         java.util.List<String> personHobbies = person.getHobbies();
281         personHobbies.add("Cooking");  // Modifies copy, not original
282         System.out.println("After modifying copy: " + person);
283
284         System.out.println("\n=== FINAL IN INHERITANCE ===");
285         Rectangle rect = new Rectangle(5, 3);
286         System.out.println("Rectangle area: " + rect.calculateArea());
287
288         System.out.println("\n=== FINAL STATIC CONSTANTS ===");
289         System.out.println("MAX_USERS: " + Configuration.MAX_USERS);
290         System.out.println("API_KEY: " + Configuration.API_KEY.
                substring(0, 10) + "...");
291
292         System.out.println("\n=== FINAL ARRAY/COLLECTION ===");
293         // Line 269: Final array reference
294         final int[] numbers = {1, 2, 3, 4, 5};
295         System.out.println("Original array: " + java.util.Arrays.
                toString(numbers));
296
297         // Can modify contents
298         numbers[0] = 10;
299         System.out.println("After modifying element: " + java.util.
                Arrays.toString(numbers));
300
301         // Cannot reassign reference
302         // numbers = new int[]{6, 7, 8};  // Compilation error
303
304         System.out.println("\n=== FINAL WITH STRINGS ===");
305         final String greeting = "Hello";
306         System.out.println("Original: " + greeting);
307
308         // Strings are immutable anyway, but reference cannot change
```

```
309         String newGreeting = greeting + " World";
310         System.out.println("New string: " + newGreeting);
311         System.out.println("Original unchanged: " + greeting);
312 
313         System.out.println("\n=== FINAL BENEFITS ===");
314         System.out.println("1. Constants: Create immutable values (PI,
                MAX_SIZE)");
315         System.out.println("2. Thread Safety: Final variables are
                thread-safe");
316         System.out.println("3. Performance: Compiler optimizations for
                final");
317         System.out.println("4. Design: Prevent method overriding or
                class extension");
318         System.out.println("5. Immutable Classes: Create thread-safe
                immutable objects");
319 
320         System.out.println("\n=== FINAL PARAMETERS ===");
321         System.out.println("Final parameters prevent accidental
                modification:");
322         System.out.println("  - Makes method implementation clearer");
323         System.out.println("  - Helps in lambda expressions and
                anonymous classes");
324 
325         System.out.println("\n=== WHEN TO USE FINAL ===");
326         System.out.println("1. Variables: When value shouldn't change")
                ;
327         System.out.println("2. Methods: When implementation shouldn't
                be overridden");
328         System.out.println("3. Classes: When class shouldn't be
                extended");
329         System.out.println("4. Parameters: When parameter shouldn't be
                modified");
330         System.out.println("5. Static: For constants shared across all
                instances");
331 
332         // Line 303: Demonstrate blank final
333         System.out.println("\n=== BLANK FINAL VARIABLES ===");
334         class BlankFinalDemo {
335             final int value;  // Blank final
336 
337             BlankFinalDemo(int val) {
338                 value = val;  // Must initialize in constructor
339             }
340 
341             void display() {
342                 System.out.println("Blank final value: " + value);
343             }
344         }
345 
346         BlankFinalDemo demo = new BlankFinalDemo(42);
347         demo.display();
348 
349         // Line 317: Final in try-with-resources
350         System.out.println("\n=== FINAL IN TRY-WITH-RESOURCES ===");
351         try (final java.io.StringReader reader = new java.io.
                StringReader("Test")) {
352             System.out.println("Using final resource in try-with-
                    resources");
```

```
353             } catch (Exception e) {
354                 e.printStackTrace();
355             }
356         }
357
358         // Line 326: Final method in main class
359         public final void finalMethodDemo() {
360             System.out.println("This is a final method");
361         }
362 }
```

Listing 11: Complete Final Keyword Implementation

FinalKeywordDemo Program Output

```
=== FINAL KEYWORD DEMONSTRATION ===

Max Attempts (final local): 3

=== FINAL VARIABLES ===
PI constant: 3.141592653589793
MAX_VALUE: 100
PI: 3.141592653589793
DATABASE_URL: jdbc:mysql://localhost:3306/testdb
Scores: [99, 87, 92]
Area of circle radius 7: 153.94

=== FINAL METHOD ===
Generic V100 engine started
Standard engine starting procedure
Vehicle: Generic V100
Toyota Camry engine started
Standard engine starting procedure
Vehicle: Toyota Camry
Type: Car, Doors: 4
Sunroof opened
Honda Civic engine started
Standard engine starting procedure
Vehicle: Honda Civic
Type: Car, Doors: 2

=== FINAL CLASS ===
Authenticating with key: sec...
Data encrypted using AES-256

=== IMMUTABLE CLASS ===
Original: ImmutablePerson{name='Alice', age=30, hobbies=[Reading, Swimming]}
After modifying original list: ImmutablePerson{name='Alice', age=30, hobbies=[Readi
After modifying copy: ImmutablePerson{name='Alice', age=30, hobbies=[Reading, Swimm
```

```
=== FINAL IN INHERITANCE ===
Rectangle area: 15.0

=== FINAL STATIC CONSTANTS ===
MAX_USERS: 1000
API_KEY: secret_170...

=== FINAL ARRAY/COLLECTION ===
Original array: [1, 2, 3, 4, 5]
After modifying element: [10, 2, 3, 4, 5]

=== FINAL WITH STRINGS ===
Original: Hello
New string: Hello World
Original unchanged: Hello

=== FINAL BENEFITS ===
1. Constants: Create immutable values (PI, MAX_SIZE)
2. Thread Safety: Final variables are thread-safe
3. Performance: Compiler optimizations for final
4. Design: Prevent method overriding or class extension
5. Immutable Classes: Create thread-safe immutable objects

=== FINAL PARAMETERS ===
Final parameters prevent accidental modification:
  - Makes method implementation clearer
  - Helps in lambda expressions and anonymous classes

=== WHEN TO USE FINAL ===
1. Variables: When value shouldn't change
2. Methods: When implementation shouldn't be overridden
3. Classes: When class shouldn't be extended
4. Parameters: When parameter shouldn't be modified
5. Static: For constants shared across all instances

=== BLANK FINAL VARIABLES ===
Blank final value: 42

=== FINAL IN TRY-WITH-RESOURCES ===
Using final resource in try-with-resources
```

Comprehensive Project: University Management System

```
// ==================== PACKAGE STRUCTURE ====================
// File: com/university/core/Person.java
```

```java
package com.university.core;

import java.util.Date;

public abstract class Person {
    protected final int id;
    protected final String name;
    protected final Date dateOfBirth;
    protected String email;

    public Person(int id, String name, Date dateOfBirth, String email) {
        this.id = id;
        this.name = name;
        this.dateOfBirth = dateOfBirth;
        this.email = email;
    }

    public abstract String getRole();

    public final void displayBasicInfo() {
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Email: " + email);
        System.out.println("Role: " + getRole());
    }

    // Getters
    public int getId() { return id; }
    public String getName() { return name; }
    public Date getDateOfBirth() { return dateOfBirth; }
    public String getEmail() { return email; }

    // Setter
    public void setEmail(String email) { this.email = email; }
}

// File: com/university/interfaces/Teachable.java
package com.university.interfaces;

public interface Teachable {
    void teachCourse(String courseCode);
    void gradeStudent(int studentId, double grade);
    void holdOfficeHours();

    default void displayTeachingSchedule() {
        System.out.println("Teaching schedule not set");
    }

    static String getTeachingGuidelines() {
        return "All teachers must follow university teaching guidelines";
    }
}

// File: com/university/interfaces/Learnable.java
package com.university.interfaces;

```

```java
public interface Learnable {
    void enrollCourse(String courseCode);
    void submitAssignment(String courseCode, String assignment);
    double calculateGPA();

    default void displayEnrolledCourses() {
        System.out.println("No courses enrolled");
    }
}

// File: com/university/faculty/Professor.java
package com.university.faculty;

import com.university.core.Person;
import com.university.interfaces.Teachable;
import java.util.ArrayList;
import java.util.List;

public class Professor extends Person implements Teachable {
    private final String department;
    private final String employeeId;
    private double salary;
    private List<String> coursesTeaching;
    private static int professorCount = 0;

    public Professor(int id, String name, java.util.Date dob,
                     String email, String department, double salary) {
        super(id, name, dob, email);
        this.department = department;
        this.salary = salary;
        this.coursesTeaching = new ArrayList<>();
        this.employeeId = "PROF" + (++professorCount);
    }

    @Override
    public String getRole() {
        return "Professor";
    }

    // Implement Teachable interface
    @Override
    public void teachCourse(String courseCode) {
        if (!coursesTeaching.contains(courseCode)) {
            coursesTeaching.add(courseCode);
            System.out.println(name + " is now teaching " + courseCode)
                ;
        }
    }

    @Override
    public void gradeStudent(int studentId, double grade) {
        System.out.println(name + " graded student " + studentId +
                           " with grade: " + grade);
    }

    @Override
    public void holdOfficeHours() {
        System.out.println(name + " is holding office hours");
```

```
    }

    @Override
    public void displayTeachingSchedule() {
        System.out.println("\nTeaching Schedule for " + name);
        System.out.println("Department: " + department);
        System.out.println("Courses: " + coursesTeaching);
    }

    // Additional methods
    public void conductResearch(String topic) {
        System.out.println(name + " conducting research on: " + topic);
    }

    public void publishPaper(String paperTitle) {
        System.out.println(name + " published paper: " + paperTitle);
    }

    // Static method
    public static int getProfessorCount() {
        return professorCount;
    }

    // Getters
    public String getDepartment() { return department; }
    public String getEmployeeId() { return employeeId; }
    public double getSalary() { return salary; }
    public List<String> getCoursesTeaching() { return new ArrayList<>(
        coursesTeaching); }
}

// File: com/university/students/Student.java
package com.university.students;

import com.university.core.Person;
import com.university.interfaces.Learnable;
import java.util.HashMap;
import java.util.Map;

public class Student extends Person implements Learnable {
    private final String studentId;
    private final String major;
    private int semester;
    private Map<String, Double> grades;
    private static int studentCount = 0;

    public Student(int id, String name, java.util.Date dob,
                   String email, String major, int semester) {
        super(id, name, dob, email);
        this.major = major;
        this.semester = semester;
        this.grades = new HashMap<>();
        this.studentId = "STU" + String.format("%06d", ++studentCount);
    }

    @Override
    public String getRole() {
        return "Student";
```

```
173     }
174
175     // Implement Learnable interface
176     @Override
177     public void enrollCourse(String courseCode) {
178         if (!grades.containsKey(courseCode)) {
179             grades.put(courseCode, 0.0);
180             System.out.println(name + " enrolled in " + courseCode);
181         }
182     }
183
184     @Override
185     public void submitAssignment(String courseCode, String assignment)
        {
186         if (grades.containsKey(courseCode)) {
187             System.out.println(name + " submitted " + assignment +
188                                " for " + courseCode);
189         }
190     }
191
192     @Override
193     public double calculateGPA() {
194         if (grades.isEmpty()) return 0.0;
195         double total = 0.0;
196         for (double grade : grades.values()) {
197             total += grade;
198         }
199         return total / grades.size();
200     }
201
202     @Override
203     public void displayEnrolledCourses() {
204         System.out.println("\nEnrolled Courses for " + name);
205         System.out.println("Student ID: " + studentId);
206         System.out.println("Major: " + major + ", Semester: " +
            semester);
207         System.out.println("Courses: " + grades.keySet());
208         System.out.println("Current GPA: " + calculateGPA());
209     }
210
211     // Additional methods
212     public void updateGrade(String courseCode, double grade) {
213         if (grades.containsKey(courseCode)) {
214             grades.put(courseCode, grade);
215             System.out.println("Updated grade for " + courseCode + ": "
                    + grade);
216         }
217     }
218
219     public void advanceSemester() {
220         semester++;
221         System.out.println(name + " advanced to semester " + semester);
222     }
223
224     // Static method
225     public static int getStudentCount() {
226         return studentCount;
227     }
```

```
228
229     // Getters
230     public String getStudentId() { return studentId; }
231     public String getMajor() { return major; }
232     public int getSemester() { return semester; }
233     public Map<String, Double> getGrades() { return new HashMap<>(
            grades); }
234 }
235
236 // File: com/university/faculty/TeachingAssistant.java
237 package com.university.faculty;
238
239 import com.university.core.Person;
240 import com.university.interfaces.Teachable;
241 import com.university.interfaces.Learnable;
242
243 public class TeachingAssistant extends Person implements Teachable,
        Learnable {
244     private final String department;
245     private final String studentId;
246     private String assignedCourse;
247     private double stipend;
248
249     public TeachingAssistant(int id, String name, java.util.Date dob,
250                              String email, String department, String
                                    studentId) {
251         super(id, name, dob, email);
252         this.department = department;
253         this.studentId = studentId;
254     }
255
256     @Override
257     public String getRole() {
258         return "Teaching Assistant";
259     }
260
261     // Implement Teachable interface
262     @Override
263     public void teachCourse(String courseCode) {
264         this.assignedCourse = courseCode;
265         System.out.println(name + " (TA) assisting in teaching " +
                courseCode);
266     }
267
268     @Override
269     public void gradeStudent(int studentId, double grade) {
270         System.out.println(name + " graded student " + studentId +
271                            " for " + assignedCourse);
272     }
273
274     @Override
275     public void holdOfficeHours() {
276         System.out.println(name + " holding TA office hours");
277     }
278
279     // Implement Learnable interface
280     @Override
281     public void enrollCourse(String courseCode) {
```

```
282         System.out.println(name + " enrolled as student in " +
                courseCode);
283     }
284
285     @Override
286     public void submitAssignment(String courseCode, String assignment)
            {
287         System.out.println(name + " submitted " + assignment +
288                            " for course " + courseCode);
289     }
290
291     @Override
292     public double calculateGPA() {
293         return 3.8;  // Example GPA
294     }
295
296     // Additional methods
297     public void setStipend(double stipend) {
298         this.stipend = stipend;
299         System.out.println(name + "'s stipend set to: $" + stipend);
300     }
301
302     public void displayTAInfo() {
303         displayBasicInfo();
304         System.out.println("Department: " + department);
305         System.out.println("Student ID: " + studentId);
306         System.out.println("Assigned Course: " + assignedCourse);
307         System.out.println("Stipend: $" + stipend);
308         System.out.println("GPA: " + calculateGPA());
309     }
310 }
311
312 // File: com/university/courses/Course.java
313 package com.university.courses;
314
315 import com.university.interfaces.Teachable;
316 import com.university.interfaces.Learnable;
317 import java.util.ArrayList;
318 import java.util.List;
319
320 public final class Course {
321     private final String courseCode;
322     private final String courseName;
323     private final int credits;
324     private final String department;
325     private Teachable instructor;
326     private final List<Learnable> enrolledStudents;
327
328     public Course(String courseCode, String courseName,
329                   int credits, String department) {
330         this.courseCode = courseCode;
331         this.courseName = courseName;
332         this.credits = credits;
333         this.department = department;
334         this.enrolledStudents = new ArrayList<>();
335     }
336
337     public void assignInstructor(Teachable instructor) {
```

```
        this.instructor = instructor;
        System.out.println(instructor + " assigned to teach " +
            courseCode);
    }

    public void enrollStudent(Learnable student) {
        if (!enrolledStudents.contains(student)) {
            enrolledStudents.add(student);
            System.out.println(student + " enrolled in " + courseCode);
        }
    }

    public void conductClass() {
        System.out.println("\n=== Conducting Class: " + courseCode + "
            ===");
        System.out.println("Course: " + courseName);
        System.out.println("Instructor: " + instructor);

        if (instructor != null) {
            instructor.teachCourse(courseCode);
        }

        System.out.println("Enrolled Students: " + enrolledStudents.
            size());
        for (Learnable student : enrolledStudents) {
            student.submitAssignment(courseCode, "Assignment 1");
        }

        System.out.println("Class completed");
    }

    public void displayCourseInfo() {
        System.out.println("\n=== Course Information ===");
        System.out.println("Code: " + courseCode);
        System.out.println("Name: " + courseName);
        System.out.println("Credits: " + credits);
        System.out.println("Department: " + department);
        System.out.println("Instructor: " + instructor);
        System.out.println("Enrolled Students: " + enrolledStudents.
            size());
    }

    // Getters
    public String getCourseCode() { return courseCode; }
    public String getCourseName() { return courseName; }
    public int getCredits() { return credits; }
    public String getDepartment() { return department; }
    public Teachable getInstructor() { return instructor; }
    public List<Learnable> getEnrolledStudents() {
        return new ArrayList<>(enrolledStudents);
    }
}

// File: com/university/UniversityMain.java
package com.university;

import com.university.core.Person;
import com.university.faculty.Professor;
```

```java
import com.university.faculty.TeachingAssistant;
import com.university.students.Student;
import com.university.courses.Course;
import com.university.interfaces.Teachable;
import com.university.interfaces.Learnable;
import java.util.Date;

public class UniversityMain {
    public static void main(String[] args) {
        System.out.println("=== UNIVERSITY MANAGEMENT SYSTEM ===\n");

        // Create date objects
        Date dob1 = new Date(90, 0, 15);  // Jan 15, 1990
        Date dob2 = new Date(95, 5, 20);  // Jun 20, 1995
        Date dob3 = new Date(98, 8, 10);  // Sep 10, 1998

        // Create professors
        System.out.println("=== CREATING FACULTY ===");
        Professor prof1 = new Professor(101, "Dr. Smith", dob1,
                                        "smith@univ.edu", "Computer
                                           Science", 85000);
        Professor prof2 = new Professor(102, "Dr. Johnson", dob1,
                                        "johnson@univ.edu", "Mathematics
                                           ", 90000);

        // Create students
        System.out.println("\n=== CREATING STUDENTS ===");
        Student student1 = new Student(201, "Alice Brown", dob3,
                                       "alice@student.edu", "Computer
                                          Science", 3);
        Student student2 = new Student(202, "Bob Wilson", dob3,
                                       "bob@student.edu", "Mathematics",
                                          2);
        Student student3 = new Student(203, "Charlie Davis", dob3,
                                       "charlie@student.edu", "Computer
                                          Science", 4);

        // Create teaching assistant
        System.out.println("\n=== CREATING TEACHING ASSISTANT ===");
        TeachingAssistant ta = new TeachingAssistant(301, "David Lee",
           dob2,
                                                     "david@univ.edu",
                                                     "Computer Science",
                                                        "STU000004");
        ta.setStipend(5000);

        // Create courses
        System.out.println("\n=== CREATING COURSES ===");
        Course cs101 = new Course("CS101", "Introduction to Programming
           ",
                                  4, "Computer Science");
        Course math201 = new Course("MATH201", "Calculus III",
                                    3, "Mathematics");

        // Assign instructors
        cs101.assignInstructor(prof1);
        math201.assignInstructor(prof2);

```

```
442         // TA assists in course
443         ta.teachCourse("CS101");
444 
445         // Enroll students
446         System.out.println("\n=== ENROLLING STUDENTS ===");
447         cs101.enrollStudent(student1);
448         cs101.enrollStudent(student3);
449         cs101.enrollStudent(ta);  // TA is also a student
450 
451         math201.enrollStudent(student2);
452         math201.enrollStudent(student1);  // Student taking multiple
                courses
453 
454         // Conduct classes
455         System.out.println("\n=== CONDUCTING CLASSES ===");
456         cs101.conductClass();
457         math201.conductClass();
458 
459         // Display information
460         System.out.println("\n=== DISPLAYING INFORMATION ===");
461 
462         System.out.println("\n--- Professor Information ---");
463         prof1.displayBasicInfo();
464         prof1.displayTeachingSchedule();
465         prof1.conductResearch("Artificial Intelligence");
466         prof1.publishPaper("Deep Learning Advances");
467 
468         System.out.println("\n--- Student Information ---");
469         student1.displayBasicInfo();
470         student1.displayEnrolledCourses();
471         student1.updateGrade("CS101", 85.5);
472         student1.updateGrade("MATH201", 92.0);
473         student1.displayEnrolledCourses();
474         student1.advanceSemester();
475 
476         System.out.println("\n--- Teaching Assistant Information ---");
477         ta.displayTAInfo();
478 
479         System.out.println("\n--- Course Information ---");
480         cs101.displayCourseInfo();
481         math201.displayCourseInfo();
482 
483         // Demonstrate polymorphism
484         System.out.println("\n=== POLYMORPHISM DEMONSTRATION ===");
485 
486         Person[] people = new Person[4];
487         people[0] = prof1;
488         people[1] = student1;
489         people[2] = ta;
490         people[3] = prof2;
491 
492         System.out.println("\nProcessing all people:");
493         for (Person person : people) {
494             System.out.println("\n--- " + person.getClass().
                    getSimpleName() + " ---");
495             person.displayBasicInfo();
496 
497             // Type checking and casting
```

```
498                 if (person instanceof Teachable) {
499                     Teachable teacher = (Teachable) person;
500                     teacher.holdOfficeHours();
501                 }
502 
503                 if (person instanceof Learnable) {
504                     Learnable learner = (Learnable) person;
505                     System.out.println("GPA: " + learner.calculateGPA());
506                 }
507             }
508 
509             // Interface static method
510             System.out.println("\n=== INTERFACE STATIC METHOD ===");
511             System.out.println(Teachable.getTeachingGuidelines());
512 
513             // Static counters
514             System.out.println("\n=== UNIVERSITY STATISTICS ===");
515             System.out.println("Total Professors: " + Professor.
                    getProfessorCount());
516             System.out.println("Total Students: " + Student.getStudentCount
                    ());
517 
518             // Final demonstration
519             System.out.println("\n=== FINAL AND ABSTRACT DEMONSTRATION ==="
                    );
520             System.out.println("Abstract Person class provides common
                    structure");
521             System.out.println("Final methods ensure consistent behavior");
522             System.out.println("Final Course class cannot be extended");
523 
524             System.out.println("\n=== SYSTEM ARCHITECTURE ===");
525             System.out.println("1. Abstract Person class with common
                    attributes");
526             System.out.println("2. Interfaces for capabilities (Teachable,
                    Learnable)");
527             System.out.println("3. Concrete classes implementing interfaces
                    ");
528             System.out.println("4. Final Course class for immutability");
529             System.out.println("5. Packages for modular organization");
530             System.out.println("6. Static members for university-wide data"
                    );
531     }
532 }
```

Listing 12: Complete University Management System Using All OOP Concepts

UniversityMain Program Output

```
=== UNIVERSITY MANAGEMENT SYSTEM ===

=== CREATING FACULTY ===

=== CREATING STUDENTS ===

=== CREATING TEACHING ASSISTANT ===
```

```
David Lee's stipend set to: $5000.0

=== CREATING COURSES ===
com.university.faculty.Professor@... assigned to teach CS101
com.university.faculty.Professor@... assigned to teach MATH201

=== ENROLLING STUDENTS ===
com.university.students.Student@... enrolled in CS101
com.university.students.Student@... enrolled in CS101
com.university.students.Student@... enrolled in CS101
com.university.students.Student@... enrolled in MATH201
com.university.students.Student@... enrolled in MATH201

=== CONDUCTING CLASSES ===

=== Conducting Class: CS101 ===
Course: Introduction to Programming
Instructor: com.university.faculty.Professor@...
Dr. Smith is now teaching CS101
Enrolled Students: 3
Alice Brown submitted Assignment 1 for CS101
Charlie Davis submitted Assignment 1 for CS101
David Lee submitted Assignment 1 for course CS101
Class completed

=== Conducting Class: MATH201 ===
Course: Calculus III
Instructor: com.university.faculty.Professor@...
Dr. Johnson is now teaching MATH201
Enrolled Students: 2
Bob Wilson submitted Assignment 1 for MATH201
Alice Brown submitted Assignment 1 for MATH201
Class completed

=== DISPLAYING INFORMATION ===

--- Professor Information ---
ID: 101
Name: Dr. Smith
Email: smith@univ.edu
Role: Professor

Teaching Schedule for Dr. Smith
Department: Computer Science
Courses: [CS101]
Dr. Smith conducting research on: Artificial Intelligence
Dr. Smith published paper: Deep Learning Advances
```

```
--- Student Information ---
ID: 201
Name: Alice Brown
Email: alice@student.edu
Role: Student

Enrolled Courses for Alice Brown
Student ID: STU000001
Major: Computer Science, Semester: 3
Courses: [MATH201, CS101]
Current GPA: 0.0
Updated grade for CS101: 85.5
Updated grade for MATH201: 92.0

Enrolled Courses for Alice Brown
Student ID: STU000001
Major: Computer Science, Semester: 3
Courses: [MATH201, CS101]
Current GPA: 88.75
Alice Brown advanced to semester 4

--- Teaching Assistant Information ---
ID: 301
Name: David Lee
Email: david@univ.edu
Role: Teaching Assistant
Department: Computer Science
Student ID: STU000004
Assigned Course: CS101
Stipend: $5000.0
GPA: 3.8

--- Course Information ---

=== Course Information ===
Code: CS101
Name: Introduction to Programming
Credits: 4
Department: Computer Science
Instructor: com.university.faculty.Professor@...
Enrolled Students: 3

=== Course Information ===
Code: MATH201
Name: Calculus III
Credits: 3
```

```
Department: Mathematics
Instructor: com.university.faculty.Professor@...
Enrolled Students: 2

=== POLYMORPHISM DEMONSTRATION ===

Processing all people:

--- Professor ---
ID: 101
Name: Dr. Smith
Email: smith@univ.edu
Role: Professor
Dr. Smith is holding office hours

--- Student ---
ID: 201
Name: Alice Brown
Email: alice@student.edu
Role: Student
GPA: 88.75

--- TeachingAssistant ---
ID: 301
Name: David Lee
Email: david@univ.edu
Role: Teaching Assistant
David Lee holding TA office hours
GPA: 3.8

--- Professor ---
ID: 102
Name: Dr. Johnson
Email: johnson@univ.edu
Role: Professor
Dr. Johnson is holding office hours

=== INTERFACE STATIC METHOD ===
All teachers must follow university teaching guidelines

=== UNIVERSITY STATISTICS ===
Total Professors: 2
Total Students: 3

=== FINAL AND ABSTRACT DEMONSTRATION ===
Abstract Person class provides common structure
Final methods ensure consistent behavior
```

```
Final Course class cannot be extended

=== SYSTEM ARCHITECTURE ===
1. Abstract Person class with common attributes
2. Interfaces for capabilities (Teachable, Learnable)
3. Concrete classes implementing interfaces
4. Final Course class for immutability
5. Packages for modular organization
6. Static members for university-wide data
```

Unit Summary and Assessment

Key Concepts Mastered

1. **Inheritance:** IS-A relationships, extends keyword, super keyword, method overriding
2. **Polymorphism:** Compile-time (overloading) vs Runtime (overriding), dynamic method dispatch
3. **Abstraction:** Abstract classes, abstract methods, partial implementation
4. **Interfaces:** Multiple inheritance, default/static methods, implementation contracts
5. **Packages:** Modular organization, access control, import statements
6. **Static Members:** Class-level variables/methods, static blocks, utility classes
7. **Final Keyword:** Constants, immutable classes, preventing inheritance/overriding

Practical Applications

- Design class hierarchies for real-world systems
- Implement polymorphic behavior using interfaces and abstract classes
- Create modular, maintainable code using packages
- Develop utility classes with static members
- Build immutable classes for thread-safe applications
- Apply design patterns using advanced OOP principles

Assessment Methods

Component	Weightage
Programming Assignments	25%
Laboratory Implementation	20%
Mid-Term Examination	25%
End-Term Examination	30%

Practice Problems

1. Design a Shape hierarchy with abstract methods for area/volume calculation
2. Implement a Payment system using interfaces for different payment methods
3. Create a Library Management System with packages for books, members, transactions
4. Build a Configuration Manager using Singleton pattern and final variables
5. Design an Employee hierarchy with abstract class and multiple interfaces
6. Implement a Calculator with method overloading for different operations

References

1. "Effective Java" by Joshua Bloch
2. "Java Concurrency in Practice" by Brian Goetz
3. Oracle Java Tutorials: Interfaces and Inheritance
4. GeeksforGeeks: OOP Concepts in Java
5. Stack Overflow: Common Java Patterns

End of Unit II: Advanced OOP Principles in Java