Lab Sheet: Final Keyword, Polymorphism, and Abstraction in Java

Course: Introduction to Programming Language II (Java)
Topic: Final Keyword, Reference Type vs. Object Type, Dynamic Dispatch, Abstraction, Abstract Classes, Interfaces
Level: Beginner to Intermediate
Duration: 3 hours

Prepared by: Ashraful Islam Paran, Dept. of CSE, CUET

1. Objectives

By the end of this lab, students will be able to:

- Understand and apply the `final` keyword to variables, methods, and classes.
- Differentiate between reference type and object type in Java.
- Implement dynamic dispatch and runtime polymorphism.
- Use abstract classes and interfaces for abstraction.
- Resolve ambiguities like the diamond problem using interfaces.
- Practice coding with examples and exercises on these topics.

2. The final Keyword

The `final` keyword in Java is used to restrict modifications. It can be applied to variables (making them constants), methods (preventing overriding), and classes (preventing inheritance). **Key Points:**

- **Final Variables:** Constants that cannot be reassigned. Must be initialized at declaration or in a constructor (for non-static) or static block (for static).
- **Blank Final Variables:** Declared without initialization but must be assigned once in a constructor.
- **Static Blank Final Variables:** Assigned once in a static block.
- **Final Methods:** Cannot be overridden in subclasses.
- **Final Classes:** Cannot be extended.

3. Example 1: Final Variable

```
public class FinalVariableExample {
    public static void main(String[] args) {
        final int MAX = 100;
        System.out.println("MAX: " + MAX);
        // MAX = 200; //     Error
    }
}
```

Output:

```
MAX: 100
```

4. Example 2: Blank Final Variable

```java
class BlankFinalExample {
    final int speed; // blank final
    BlankFinalExample(int s) {
        speed = s; // must assign value here
    }
    void showSpeed() {
        System.out.println("Speed: " + speed);
    }
    public static void main(String[] args) {
        BlankFinalExample obj = new BlankFinalExample(90);
        obj.showSpeed(); // Output: Speed: 90
    }
}
```

Output:

```
Speed: 90
```

5. Example 3: Static Blank Final Variable

```java
class StaticBlankFinalExample {
    static final int MAX;
    static {
        MAX = 500; // must initialize here
    }
    public static void main(String[] args) {
        System.out.println("MAX: " + MAX); // Output: MAX: 500
    }
}
```

Output:

```
MAX: 500
```

6. Example 4: Final Method

```
class Parent {
    final void show() {
        System.out.println("Final method in Parent");
    }
}

class Child extends Parent {
    // void show() { } //     Error: cannot override final method
}

public class FinalMethodExample {
    public static void main(String[] args) {
        new Child().show(); // Output: Final method in Parent
    }
}
```

Output:

```
Final method in Parent
```

7. Example 5: Final Class

```
final class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

// class Car extends Vehicle {} //     Error

public class FinalClassExample {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.run(); // Output: Vehicle is running
    }
}
```

Output:

```
Vehicle is running
```

Task 2.1 – Final Variable Practice (10 min)

a. Create a class `Constants` with final variables for PI (3.14159) and GRAVITY (9.8). Write a main method to print them. Attempt to reassign one and note the error.

Sample Code:

```
public class Constants {
    public static void main(String[] args) {
        final double PI = 3.14159;
        final double GRAVITY = 9.8;
        System.out.println("PI: " + PI);
        System.out.println("GRAVITY: " + GRAVITY);
        // PI = 3.14; //  Error
    }
}
```

Task 2.2 – Blank Final in Constructor (10 min)

a. Create a class `Person` with a blank final String `name`. Initialize it in the constructor. Add a method to display the name.

Sample Code:

```
class Person {
    final String name;
    Person(String n) {
        name = n;
    }
    void display() {
        System.out.println("Name: " + name);
    }
    public static void main(String[] args) {
        Person p = new Person("Alice");
        p.display(); // Output: Name: Alice
    }
}
```

Task 2.3 – Static Blank Final (10 min)

a. Create a class `Config` with a static blank final int `PORT`. Initialize it in a static block to 8080. Print it in main.

Sample Code:

```
class Config {
    static final int PORT;
    static {
        PORT = 8080;
    }
    public static void main(String[] args) {
        System.out.println("PORT: " + PORT); // Output: PORT: 8080
    }
}
```

Task 2.4 – Final Method and Class (10 min)

a. Create a final class `MathUtils` with a final method `add(int a, int b)` that returns the sum. Try to extend the class and override the method (note errors).

Sample Code:

```
final class MathUtils {
    final int add(int a, int b) {
        return a + b;
    }
}

// class ExtendedMath extends MathUtils {} //     Error

public class MathTest {
    public static void main(String[] args) {
        MathUtils mu = new MathUtils();
        System.out.println(mu.add(5, 3)); // Output: 8
    }
}
```

8. Reference Type vs. Object Type and Dynamic Dispatch

Reference type is the declared type of a variable (compile-time), while object type is the actual type in memory (runtime). Dynamic dispatch resolves overridden methods at runtime based on object type. **Key Points:**

- **Reference Type:** Determines accessible methods/fields at compile-time. From left side of assignment.
- **Object Type:** Determines which overridden methods run at runtime. From right side of assignment.
- **Dynamic Dispatch:** Overridden methods are called based on object type (runtime polymorphism).
- Variables and non-overridden methods are resolved by reference type.
- Fields are not polymorphic; resolved by reference type.
- Casting allows access to subclass members.

9. Example 6: Basic Reference vs Object Type

```
// Reference type: Parent
Parent p = new Child(); // Object type: Child
```

10. Example 7: Parent-Child Example

```
class Parent {
    int x = 10;
    void speak() {
        System.out.println("Parent speaks");
    }
}
class Child extends Parent {
    int x = 20;
    @Override
    void speak() {
        System.out.println("Child speaks");
    }
    void onlyChildDoes() {
        System.out.println("Child-only method");
    }
}
public class Test {
    public static void main(String[] args) {
        Parent p = new Child(); // ref: Parent, obj: Child
        System.out.println(p.x); //     10 (reference type wins)
        p.speak(); //     Child speaks (object type wins)
        // p.onlyChildDoes(); //     Compile error (Parent doesn't have
            ↪ this method)
        ((Child) p).onlyChildDoes(); //     OK after casting
    }
}
```

Output:

```
10
Child speaks
Child-only method
```

11. Example 8: Animal-Dog-Cat Example

```java
class Animal {
    String name = "Animal";
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    String name = "Dog";
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
class Cat extends Animal {
    String name = "Cat";
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal a1 = new Dog(); // parent reference, child object
        Animal a2 = new Cat(); // parent reference, child object
        Animal a3 = new Animal(); // parent reference, parent object
        // --- Dynamic Dispatch in action ---
        a1.sound(); // Dog barks
        a2.sound(); // Cat meows
        a3.sound(); // Animal makes a sound
        // --- Variable behavior ---
        System.out.println(a1.name); // Animal
        System.out.println(a2.name); // Animal
        System.out.println(a3.name); // Animal
    }
}
```

Output:

```
Dog barks
Cat meows
Animal makes a sound
Animal
Animal
Animal
```

Task 3.1 – Reference vs Object Type (12 min)

a. Create classes `Vehicle` (with int speed = 50, void move()) and `Car` extends Vehicle (speed = 100, override move()). Use Vehicle ref = new Car() and print speed/move.

Sample Code:

```
class Vehicle {
    int speed = 50;
    void move() {
        System.out.println("Vehicle moves");
    }
}
class Car extends Vehicle {
    int speed = 100;
    @Override
    void move() {
        System.out.println("Car drives");
    }
}
public class VehicleTest {
    public static void main(String[] args) {
        Vehicle v = new Car();
        System.out.println(v.speed); // 50 (reference type)
        v.move(); // Car drives (object type)
    }
}
```

Task 3.2 – Dynamic Dispatch with Casting (12 min)

a. Add a method `honk()` to Car only. Use casting to call it from Vehicle reference.

Sample Code:

```
// Extend from above
class Car extends Vehicle {
    // ... (previous)
    void honk() {
        System.out.println("Car honks");
    }
}
public class VehicleTest {
    public static void main(String[] args) {
        Vehicle v = new Car();
        // v.honk(); //     Compile error
        ((Car) v).honk(); // Car honks
    }
}
```

Task 3.3 – Multiple Subclasses (12 min)

a. Create `Bird` extends Animal (name = "Bird", override sound() to "Bird chirps"). Use Animal refs for Dog, Cat, Bird and call sound()/print name.

Sample Code:

```
// Extend from Animal example
class Bird extends Animal {
    String name = "Bird";
    @Override
    void sound() {
        System.out.println("Bird chirps");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal a4 = new Bird();
        a4.sound(); // Bird chirps
        System.out.println(a4.name); // Animal
    }
}
```

12. Abstraction, Abstract Classes, and Interfaces

Abstraction hides implementation details. Achieved via abstract classes (partial abstraction) or interfaces (full abstraction). **Key Points:**

- **Abstract Class:** Can have abstract (no body) and concrete methods, fields, constructors. Cannot instantiate. Subclasses must implement abstract methods.
- **Interface:** All methods abstract/public by default, variables public static final. Supports multiple inheritance. No constructors/instance fields.
- **Differences:** Interfaces are 100% abstract, abstract classes can have defaults.
- **Diamond Problem:** Java avoids with classes; resolves in interfaces by overriding.
- Multiple inheritance via interfaces is safe due to no implementations.

13. Example 9: Abstract Class Basic Example

```
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();
    // Concrete method
    void sleep() {
        System.out.println("Animal is sleeping...");
    }
}
// Concrete subclass
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
public static void main(String[] args) {
    Dog d = new Dog();
    d.sound(); // Output: Dog barks
    d.sleep(); // Output: Animal is sleeping...
}
```

Output:

```
Dog barks
Animal is sleeping...
```

14. Example 10: Abstract Subclass Example

```
abstract class Animal {
    abstract void sound();
    void sleep() {
        System.out.println("Animal is sleeping...");
    }
}
abstract class Dog extends Animal {
    void bark() {
        System.out.println("Dog barking behavior");
    }
}
class Beagle extends Dog {
    @Override
    void sound() {
        System.out.println("Beagle barks");
    }
}
public class MainClass {
    public static void main(String[] args) {
        Dog d = new Beagle(); // Dog reference to Beagle object
        d.sound(); // Output: Beagle barks
        d.bark(); // Output: Dog barking behavior
        d.sleep(); // Output: Animal is sleeping...
    }
}
```

Output:

```
Beagle barks
Dog barking behavior
Animal is sleeping...
```

15. Example 11: Vehicle Abstract Example

```
abstract class Vehicle {
    abstract void start();
    abstract void stop();
    void fuelType() {
        System.out.println("Uses fuel");
    }
}
class Car extends Vehicle {
    void start() {
        System.out.println("Car starts with key");
    }
    void stop() {
        System.out.println("Car stops with brakes");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle v = new Car();
        v.start(); // Car starts with key
        v.stop(); // Car stops with brakes
        v.fuelType(); // Uses fuel
    }
}
```

Output:

```
Car starts with key
Car stops with brakes
Uses fuel
```

16. Example 12: Shape Abstract with Constructor

```
abstract class Shape {
    String color;
    Shape(String color) {
        this.color = color;
    }
    abstract double area();
    void displayColor() {
        System.out.println("Color: " + color);
    }
}
class Circle extends Shape {
    double radius;
    Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }
    double area() {
        return Math.PI * radius * radius;
    }
}
```

17. Example 13: Employee Abstract Example

```
abstract class Employee {
    abstract void calculateSalary();
}
class FullTimeEmployee extends Employee {
    void calculateSalary() {
        System.out.println("Salary: Fixed monthly salary");
    }
}
class PartTimeEmployee extends Employee {
    void calculateSalary() {
        System.out.println("Salary: Hourly based salary");
    }
}
public class Main {
    public static void main(String[] args) {
        Employee e1 = new FullTimeEmployee();
        Employee e2 = new PartTimeEmployee();
        e1.calculateSalary(); // Salary: Fixed monthly salary
        e2.calculateSalary(); // Salary: Hourly based salary
    }
}
```

Output:

```
Salary: Fixed monthly salary
Salary: Hourly based salary
```

18. Example 14: Interface Basic Example

```
interface Flyable {
    // Variable must be static and final
    int MAX_ALTITUDE = 10000; // implicitly public static final
    void fly(); // implicitly public and abstract
}
```

19. Example 15: Bank Interface Example

```
interface Bank {
    double INTEREST_RATE = 5.5;
    void calculateInterest();
}

class CityBank implements Bank {
    public void calculateInterest() {
        System.out.println("Interest Rate: " + INTEREST_RATE + "%");
    }
}
```

20. Example 16: Shape Interface Example

```
interface Shape {
    double pi = 3.1416;
    double area();
}
class Circle implements Shape {
    double r = 2;
    public double area() {
        return pi * r * r;
    }
}
public class Main {
    public static void main(String[] args) {
        Circle c = new Circle();
        System.out.println(c.area());
        System.out.println(Circle.pi);
        System.out.println(Shape.pi);
    }
}
```

21. Example 17: Multiple Interfaces Example

```
interface Printable {
    void print();
}
interface Scannable {
    void scan();
}
class Printer implements Printable, Scannable {
    public void print() {
        System.out.println("Printing document");
    }
    public void scan() {
        System.out.println("Scanning document");
    }
}
```

22. Example 18: Payment Interface Example

```
interface Payment {
    void pay();
}
class CreditCardPayment implements Payment {
    public void pay() {
        System.out.println("Paid using credit card");
    }
}
class MobileBankingPayment implements Payment {
    public void pay() {
        System.out.println("Paid using mobile banking");
    }
}
public class Main {
    public static void main(String[] args) {
        Payment p1 = new CreditCardPayment();
        p1.pay(); // Paid using credit card
        Payment p2 = new MobileBankingPayment();
        p2.pay(); // Paid using mobile banking
    }
}
```

Output:

```
Paid using credit card
Paid using mobile banking
```

23. Example 19: Abstract Class + Interface Example

```
abstract class Animal {
    abstract void sound();
    void sleep() {
        System.out.println("Sleeping...");
    }
}
// Interface
interface Pet {
    void play();
}
class Dog extends Animal implements Pet {
    public void sound() {
        System.out.println("Bark");
    }
    public void play() {
        System.out.println("Plays fetch");
    }
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound(); // Output: Bark
        d.play(); // Output: Plays fetch
        d.sleep(); // Output: Sleeping...
    }
}
```

Output:

```
Bark
Plays fetch
Sleeping...
```

24. Example 20: Diamond Problem Example (Classes - Error)

```
class A {
    void show() {
        System.out.println("A's show");
    }
}

class C extends A {
    void show() {
        System.out.println("C's show");
    }
}

class B extends A {
    void show() {
        System.out.println("B's show");
    }
}

// class D extends B, C { // Ambiguous: which show() method to inherit? }

// Compile error: Java does not allow multiple inheritance with classes
```

25. Example 21: Diamond Problem Solution (Interfaces)

```
interface A {
    default void show() {
        System.out.println("Interface A");
    }
}
interface B {
    default void show() {
        System.out.println("Interface B");
    }
}
class Demo implements A, B {
    public void show() {
        A.super.show();
    }
}
public class Main {
    public static void main(String[] args) {
        Demo d = new Demo();
        d.show(); // Interface A
    }
}
```

Output:

```
Interface A
```

26. Example 22: Multiple Inheritance with Interfaces Example

```
interface Flyable {
    void fly();
}
interface Swimmable {
    void swim();
}
class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Duck is flying");
    }
    public void swim() {
        System.out.println("Duck is swimming");
    }
    public static void main(String[] args) {
        Duck d = new Duck();
        d.fly(); // Output: Duck is flying
        d.swim(); // Output: Duck is swimming
    }
}
```

Output:

```
Duck is flying
Duck is swimming
```

Task 4.1 – Abstract Class Practice (15 min)

a. Create abstract class `Device` with abstract void powerOn() and concrete void checkBattery(). Subclass `Phone` implements powerOn().

Sample Code:

```
abstract class Device {
    abstract void powerOn();
    void checkBattery() {
        System.out.println("Battery level: 100%");
    }
}
class Phone extends Device {
    @Override
    void powerOn() {
        System.out.println("Phone powering on");
    }
}
public class DeviceTest {
    public static void main(String[] args) {
        Device d = new Phone();
        d.powerOn(); // Phone powering on
        d.checkBattery(); // Battery level: 100%
    }
}
```

Task 4.2 – Abstract with Constructor (15 min)

a. Create abstract `Fruit` with String color (via constructor) and abstract void taste(). Subclass `Apple`.

Sample Code:

```
abstract class Fruit {
    String color;
    Fruit(String color) {
        this.color = color;
    }
    abstract void taste();
    void displayColor() {
        System.out.println("Color: " + color);
    }
}
class Apple extends Fruit {
    Apple(String color) {
        super(color);
    }
    @Override
    void taste() {
        System.out.println("Sweet");
    }
}
public class FruitTest {
    public static void main(String[] args) {
        Apple a = new Apple("Red");
        a.taste(); // Sweet
        a.displayColor(); // Color: Red
    }
}
```

Task 4.3 – Interface Practice (15 min)

a. Create interface `Drawable` with void draw() and constant int SIZE = 10. Implement in class `Square`.

Sample Code:

```
interface Drawable {
    int SIZE = 10;
    void draw();
}
class Square implements Drawable {
    public void draw() {
        System.out.println("Drawing square of size " + SIZE);
    }
}
public class DrawTest {
    public static void main(String[] args) {
        Square s = new Square();
        s.draw(); // Drawing square of size 10
    }
}
```

Task 4.4 – Multiple Interfaces (15 min)

a. Create interfaces `Readable` (void read()) and `Writable` (void write()). Class `Book` implements both.

Sample Code:

```
interface Readable {
    void read();
}
interface Writable {
    void write();
}
class Book implements Readable, Writable {
    public void read() {
        System.out.println("Reading book");
    }
    public void write() {
        System.out.println("Writing book");
    }
}
public class BookTest {
    public static void main(String[] args) {
        Book b = new Book();
        b.read(); // Reading book
        b.write(); // Writing book
    }
}
```

Task 4.5 – Abstract Class + Interface (15 min)

a. Abstract class `Gadget` (abstract void operate()) + interface `Chargeable` (void charge()). Class `Laptop` extends/implements both.

Sample Code:

```
abstract class Gadget {
    abstract void operate();
}
interface Chargeable {
    void charge();
}
class Laptop extends Gadget implements Chargeable {
    @Override
    void operate() {
        System.out.println("Laptop operating");
    }
    @Override
    void charge() {
        System.out.println("Laptop charging");
    }
}
public class GadgetTest {
    public static void main(String[] args) {
        Laptop l = new Laptop();
        l.operate(); // Laptop operating
        l.charge(); // Laptop charging
    }
}
```

Task 4.6 – Diamond Problem Resolution (15 min)

a. Interfaces `SoundA` and `SoundB` both with default void makeSound(). Class `Speaker` implements both and overrides.

Sample Code:

```
interface SoundA {
    default void makeSound() {
        System.out.println("Sound A");
    }
}
interface SoundB {
    default void makeSound() {
        System.out.println("Sound B");
    }
}
class Speaker implements SoundA, SoundB {
    @Override
    public void makeSound() {
        SoundA.super.makeSound(); // Choose A
    }
}
public class SoundTest {
    public static void main(String[] args) {
        Speaker s = new Speaker();
        s.makeSound(); // Sound A
    }
}
```

27. Key Differences: Abstract Class vs Interface

Interface	Abstract Class
Java interface are implicitly abstract and cannot have implementations	A Java abstract class can h
Variables declared in a Java interface is by default final	An abstract class may conta
Members of a Java interface are public by default	A Java abstract class can h
Java interface should be implemented using keyword "implements"	A Java abstract class shoul
An interface can extend another Java interface only	an abstract class can extend
Interface is absolutely abstract and cannot be instantiated	A Java abstract class also c
java interfaces are slow as it requires extra indirection	Comparatively fast

28. Summary Table

Concept	Description	Example
Final Variable	Constant, cannot reassign	final int MAX = 100;
Final Method	Cannot override	final void show()
Final Class	Cannot extend	final class Vehicle
Reference Type	Compile-time type	Parent p = new Child(); (Parent)
Object Type	Runtime type	Parent p = new Child(); (Child)
Dynamic Dispatch	Runtime method resolution	p.speak() calls Child's version
Abstract Class	Partial abstraction	abstract class Animal
Interface	Full abstraction	interface Flyable
Diamond Problem	Ambiguity in multiple inheritance	Resolved by overriding in interfaces

29. Final Task

Design a system using abstraction and polymorphism:

- Abstract class `Shape` with abstract `area()` and `perimeter()`.
- Concrete classes `Circle` and `Rectangle`.
- Interface `Resizable` with `resize(double factor)`.
- Make `Rectangle` implement `Resizable`.
- Use polymorphism to compute areas in a list of shapes.

Use `final` where appropriate.

30. Practice Questions

1. Explain the difference between reference type and object type.
2. What is dynamic dispatch, and when does it occur?
3. Why can't abstract classes be instantiated?

4. How does Java handle multiple inheritance?
5. Write a program using an interface with default methods and resolve a diamond problem.
6. What happens if a subclass doesn't implement an abstract method?

Keep Coding!