Advanced Programming (OOP)

Module 3: Advanced Features of OOP

SDB

Spring 2025

Module 3: Topics

- Polymorphism: Method overloading and overriding.
- Inheritance: Using superclass and subclass.
- Abstract Classes and Interfaces.
- Copying and Cloning Objects.
- Using Wrapper Classes and Streams.

Outline

What is Inheritance?

Definition: Inheritance is a mechanism in which one class (subclass) derives properties and behaviors from another class (superclass).

Key Benefits:

- Code Reusability: Reuse fields and methods of the parent class.
- Extensibility: Add new features to existing classes without modifying them.
- Polymorphism: Achieve dynamic method invocation.

Real-World Usage of Inheritance

Example Scenario 1: Employee Management System

- Superclass: ‘Employee‘
- Subclasses: ‘Manager‘, ‘Developer‘, ‘Tester‘
- Shared Attributes: ‘name‘, ‘id‘, ‘department‘
- Specific Behaviors: ‘assignTask()‘ for Manager, ‘writeCode()‘ for Developer.

Example Scenario 2: Vehicle Hierarchy

- Superclass: ‘Vehicle‘
- Subclasses: ‘Car‘, ‘Bike‘, ‘Truck‘
- Shared Attributes: ‘speed‘, ‘fuelCapacity‘
- Specific Behaviors: ‘loadCargo()‘ for Truck, ‘ride()‘ for Bike.

Types of Inheritance in Java

Supported Types:

- Single Inheritance: One class inherits from another.
- Multilevel Inheritance: A class inherits from a class that itself inherits from another class.
- Hierarchical Inheritance: Multiple classes inherit from a single superclass.

Not Supported:

- Multiple Inheritance: Java does not allow a class to inherit from more than one class directly to avoid ambiguity caused by the diamond problem.

Why Multiple Inheritance is Not Supported

The Diamond Problem:

- Occurs when a class inherits from two classes that have a common ancestor.
- Leads to ambiguity: Which implementation of the shared method should be used?

Solution in Java:

- Use Interfaces: Multiple inheritance of type is achieved by implementing multiple interfaces.

Single Inheritance Example

Definition: A class inherits from a single superclass.

```
class Animal {
    void eat() {System.out.println("Animal eats food.");}
}

class Dog extends Animal {
    void bark() {System.out.println("Dog barks.");}
}

public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // Inherited method
        dog.bark();
    }
}
```

Listing 1: Single Inheritance

Multilevel Inheritance Example

Definition: A class inherits from a class that itself inherits from another class.

```
class Animal {
    void eat() {System.out.println("Animal eats food.");}
}

class Mammal extends Animal {
    void walk() {System.out.println("Mammal walks.");}
}

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

public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();
        dog.walk();
        dog.bark();
    }
}
```

Listing 2: Multilevel Inheritance

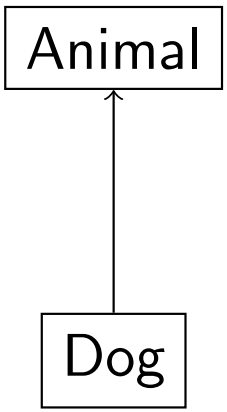
Hierarchical Inheritance Example

Definition: Multiple classes inherit from a single superclass.

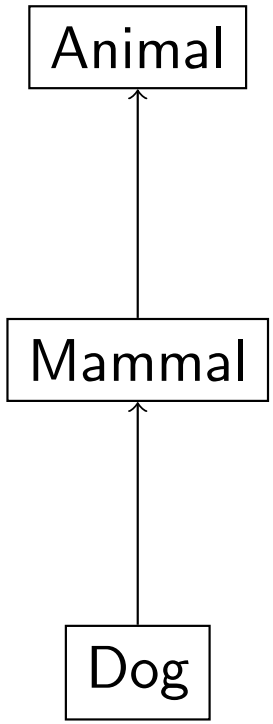
```
class Animal {
    void eat() {System.out.println("Animal eats food.");}
}

class Dog extends Animal {
    void bark() {System.out.println("Dog barks.");}
}

class Cat extends Animal {
    void meow() {System.out.println("Cat meows.");}
}

public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();

        dog.eat();
        dog.bark();
        cat.eat();
        cat.meow();
    }
}
```

Listing 3: Hierarchical Inheritance

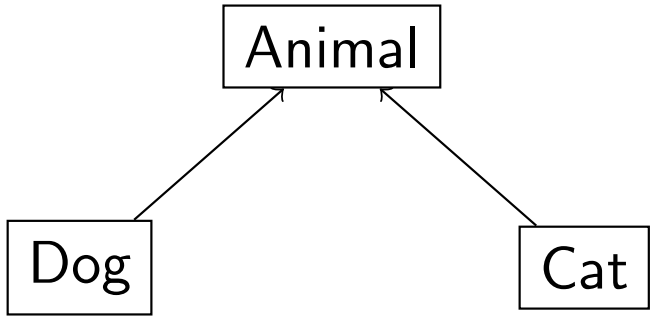
Visualizing Inheritance



Single Inheritance:



Multilevel Inheritance:



Hierarchical Inheritance:

Access Specifiers and Inheritance: Single Inheritance I

Single Inheritance:

- **Public:** Inherited and accessible everywhere.
- **Protected:** Inherited and accessible within the same package and subclasses.
- **Default:** Inherited but accessible only within the same package.
- **Private:** Not inherited or accessible.

Access Specifiers and Inheritance: Single Inheritance II

Example:

```
class Parent {
    public int publicVar = 10;
    private int privateVar = 20;
    protected int protectedVar = 30;
    int defaultVar = 40;
}

class Child extends Parent {
    void display() {
        System.out.println(publicVar); // Accessible
        // System.out.println(privateVar); // Not Accessible
        System.out.println(protectedVar); // Accessible
        System.out.println(defaultVar); // Accessible (same package)
    }
}
```

Listing 4: Single Inheritance Access

Access Specifiers and Inheritance: Multilevel Inheritance I

Multilevel Inheritance:

- **Public:** Inherited and accessible everywhere.
- **Protected:** Inherited and accessible within the package and in subclasses at all levels.
- **Default:** Accessible only within the package.
- **Private:** Not inherited or accessible.

Access Specifiers and Inheritance: Multilevel Inheritance II

Example:

```
class GrandParent {
    public int publicVar = 10;
    protected int protectedVar = 20;
    int defaultVar = 30;
    private int privateVar = 40;
}

class Parent extends GrandParent {
}

class Child extends Parent {
    void display() {
        System.out.println(publicVar); // Accessible
        System.out.println(protectedVar); // Accessible
        // System.out.println(defaultVar); // Not Accessible (if in another package)
        // System.out.println(privateVar); // Not Accessible
    }
}
```

Listing 5: Multilevel Inheritance Access

Access Specifiers and Inheritance: Hierarchical Inheritance I

Hierarchical Inheritance:

- **Public:** Inherited and accessible everywhere.
- **Protected:** Inherited and accessible within the package and in subclasses.
- **Default:** Inherited but accessible only within the same package.
- **Private:** Not inherited or accessible.

Access Specifiers and Inheritance: Hierarchical Inheritance II

Example:

```
class Parent {
    public int publicVar = 10;
    protected int protectedVar = 20;
    int defaultVar = 30;
    private int privateVar = 40;
}
class ChildA extends Parent {
    void displayA() {
        System.out.println(publicVar); // Accessible
        System.out.println(protectedVar); // Accessible
        System.out.println(defaultVar); // Accessible (same package)
    }
}
class ChildB extends Parent {
    void displayB() {
        System.out.println(publicVar); // Accessible
        System.out.println(protectedVar); // Accessible
        // System.out.println(defaultVar); // Not Accessible (if in another package)
    }
}
```

Listing 6: Hierarchical Inheritance Access

Constructors in Inheritance

Key Points:

- A subclass constructor always calls the constructor of its superclass.
- If no ‘super()‘ is explicitly called, the default (no-argument) constructor of the superclass is invoked.
- Parameterized constructors in the superclass must be explicitly called using ‘super(parameters)‘.

Importance:

- Ensures proper initialization of inherited attributes.
- Prevents uninitialized fields in the parent class.

Using ‘super()‘ in Constructors

Example:

```
class Parent {
    String name;

    Parent(String name) {
        this.name = name;
        System.out.println("Parent constructor called.");
    }
}
class Child extends Parent {
    int age;

    Child(String name, int age) {
        super(name); // Call to superclass constructor
        this.age = age;
        System.out.println("Child constructor called.");
    }
}
public class Test {
    public static void main(String[] args) {
        Child c = new Child("Alice", 10);
        // Output:
        // Parent constructor called.
        // Child constructor called.
    }
}
```

Listing 7: Using super()

Default Constructor and Inheritance

Behavior:

- If a superclass does not have a no-argument constructor, the subclass must explicitly call a parameterized constructor.
- Failure to do so results in a compile-time error.

Example:

```
class Parent {
    int value;

    Parent(int value) {
        this.value = value;
    }
}
class Child extends Parent {
    Child(int value) {
        super(value); // Must call superclass constructor
    }
}
public class Test {
    public static void main(String[] args) {
        Child c = new Child(42);
        System.out.println("Value: " + c.value);
    }
}
```

Listing 8: No-Argument Constructor Missing

Constructor Chaining in Inheritance I

Definition: A sequence of constructor calls across the inheritance hierarchy.

Example:

```
class GrandParent {
    GrandParent() {
        System.out.println("GrandParent constructor called.");
    }
}

class Parent extends GrandParent {
    Parent() {
        super(); // Calls GrandParent constructor
        System.out.println("Parent constructor called.");
    }
}

class Child extends Parent {
    Child() {
        super(); // Calls Parent constructor
        System.out.println("Child constructor called.");
    }
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
```

Constructor Chaining in Inheritance II

```
        // Output:
        // GrandParent constructor called.
        // Parent constructor called.
        // Child constructor called.
    }
}
```

Listing 9: Constructor Chaining

Best Practices: Constructors and Inheritance

- Always use ‘super()‘ explicitly when initializing parent class attributes.
- Keep constructors simple to ensure maintainability.
- Avoid calling overridable methods within constructors to prevent unexpected behavior.
- Ensure consistent initialization of attributes in both parent and child classes.

Common Mistakes and Issues in Inheritance I

1. Incorrect Use of Access Specifiers:

- Declaring fields as ‘private‘ in the parent class makes them inaccessible to child classes.
- Solution: Use ‘protected‘ for fields meant to be inherited.

Example:

```
class Parent {
    private int value = 10; // Not accessible in Child
}

class Child extends Parent {
    void showValue() {
        // System.out.println(value); // Compile-time error
    }
}
```

Listing 10: Access Specifier Issue

Common Mistakes and Issues in Inheritance II

2. Overusing Inheritance:

- Inheriting from a class unnecessarily, leading to a fragile hierarchy.
- Solution: Use composition instead of inheritance when the relationship is not "is-a".

Example:

```
class Engine {
    void start() {
        System.out.println("Engine starts");
    }
}

// Instead of:
class Car extends Engine { }
// Prefer composition:
class Car {
    private Engine engine = new Engine();
    void startCar() {
        engine.start();
    }
}
```

Listing 11: Overusing Inheritance

Common Mistakes and Issues in Inheritance III

3. Constructor Issues:

- Forgetting to call the superclass constructor explicitly when needed.
- Solution: Use ‘super()‘ to initialize parent class fields.

Example:

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

class Child extends Parent {
    Child(String name) {
        super(name); // Explicit call to superclass constructor
    }
}
```

Listing 12: Constructor Issue

Common Mistakes and Issues in Inheritance IV

4. Overriding Mistakes:

- Incorrectly overriding methods, leading to bugs or loss of functionality.
- Solution: Use the '@Override' annotation to ensure proper overriding.

Example:

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

class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child class method");
    }
}
```

Listing 13: Overriding Mistake

Common Mistakes and Issues in Inheritance V

5. Diamond Problem:

- Attempting multiple inheritance with classes, causing ambiguity.
- Solution: Use interfaces to achieve multiple inheritance of type.

Example:

```
class A {
    void show() {System.out.println("Class A method");}
}
class B extends A {
    void show() {System.out.println("Class B method");}
}
// Java prevents this scenario:
// class C extends A, B { }

interface I1 {void show();}
interface I2 {void show();}

class C implements I1, I2 {
    public void show() {
        System.out.println("Resolved in Class C");
    }
}
```

Listing 14: Diamond Problem Example

Common Mistakes and Issues in Inheritance VI

6. Tight Coupling:

- Child classes tightly coupled with parent classes, making maintenance difficult.
- Solution: Favor loose coupling by using interfaces or abstract classes for flexibility.

Example:

```
class A {
    void service() {System.out.println("Service from A");}
}
class B extends A {
    void useService() {service(); // Tight coupling}
}
// Use an interface instead for loose coupling
interface Service {void perform();}
class C implements Service {
    public void perform() {
        System.out.println("Service from C");
    }
}
```

Listing 15: Tight Coupling

Common Mistakes and Issues in Inheritance VII

7. Unintended Behavior:

- Forgetting to account for inherited methods, leading to unexpected behaviors.
- Solution: Carefully design parent classes with inheritance in mind.

Example:

```
class Parent {
    void calculate() {System.out.println("Parent calculation");}
}
class Child extends Parent {
    void calculate() {System.out.println("Child-specific calculation");}
}
class Test {
    public static void main(String[] args) {
        Parent p = new Child(); // Unintended method execution
        p.calculate();
    }
}
```

Listing 16: Unintended Behavior

Common Mistakes and Issues in Inheritance VIII

8. Fragile Base Class Problem:

- Modifying a base class breaks the functionality of child classes.
- Solution: Design base classes to be stable and avoid frequent changes.

Example:

```
class Base {
    void display() {System.out.println("Base display");}
}
class Derived extends Base {
    void display() {System.out.println("Derived display");}
}
class Test {
    public static void main(String[] args) {
        Base obj = new Derived();
        obj.display();
    }
}

// Changing Base class implementation may impact Derived class
```

Listing 17: Fragile Base Class

Java Annotations I

Definition: Annotations provide metadata about the code, used to give instructions to the compiler or runtime.

Common Annotations:

- **@Override:** Indicates a method overrides a superclass method.
- **@Deprecated:** Marks a method or class as deprecated.
- **@SuppressWarnings:** Suppresses specific compiler warnings.
- **@FunctionalInterface:** Marks an interface as a functional interface.

Java Annotations II

Example:

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

class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child method");
    }
}
```

Listing 18: Using Annotations

Outline

Introduction to Polymorphism in Java

Definition: Polymorphism allows objects to take many forms, enabling a single interface to be used with different underlying types.

Key Concepts:

- **Compile-Time Polymorphism:** Achieved through method overloading.
- **Runtime Polymorphism:** Achieved through method overriding.
- Promotes code flexibility and reusability.

Real-World Analogy:

- A single word like "run" can have different meanings based on context (e.g., a person running, a program running).

Supported Types of Polymorphism

Definition: Polymorphism allows methods or objects to take many forms.
Types Supported in Java:

- **Compile-Time Polymorphism:** Achieved via method overloading.
- **Runtime Polymorphism:** Achieved via method overriding.

Not Supported:

- **Operator Overloading:** Not supported to maintain code simplicity (e.g., unlike C++).

Compile-Time Polymorphism Example

Definition: Resolved at compile time using method overloading.

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

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

public class Test {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10)); // Calls int version
        System.out.println(calc.add(5.5, 2.2)); // Calls double version
    }
}
```

Listing 19: Method Overloading

Runtime Polymorphism Example

Definition: Resolved at runtime using method overriding.

```java
class Animal {
    void speak() {
        System.out.println("Animal speaks.");
    }
}

class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("Dog barks.");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        animal.speak(); // Calls Dog's speak method
    }
}
```

Listing 20: Method Overriding

Polymorphism: Method Overloading

Definition: Polymorphism allows methods to perform different tasks based on the parameters passed.

Advantages:

- Simplifies code by allowing multiple methods with the same name.
- Enhances readability and reusability.

Explanation: Method overloading occurs when two or more methods in the same class share the same name but have different parameters (type, number, or both). This enables the same method name to perform different tasks depending on the arguments passed.

Overloading in Java

Definition: Overloading allows multiple methods in the same class to have the same name but different parameter lists.

Key Characteristics:

- Methods must differ in the number, type, or order of parameters.
- Return type alone cannot differentiate overloaded methods.
- Provides compile-time polymorphism.

Method Overloading Example

Example:

```
class Calculator {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }
    // Method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
    // Method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
}
public class Test {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3)); // Calls add(int, int)
        System.out.println(calc.add(2, 3, 4)); // Calls add(int, int, int)
        System.out.println(calc.add(2.5, 3.5)); // Calls add(double, double)
    }
}
```

Listing 21: Method Overloading

Constructor Overloading

Definition: Overloading constructors allows the creation of objects with different initializations.

Example:

```
class Box {
    double width, height, depth;
    // Default constructor
    Box() { width = height = depth = 0;}
    // Parameterized constructor
    Box(double w, double h, double d) {
        width = w;  height = h;  depth = d;
    }
    double volume() { return width * height * depth;}
}

public class Test {
    public static void main(String[] args) {
        Box box1 = new Box();
        Box box2 = new Box(2, 3, 4);
        System.out.println("Volume of box1: " + box1.volume());
        System.out.println("Volume of box2: " + box2.volume());
    }
}
```

Listing 22: Constructor Overloading

Rules for Method Overloading

- Methods must have the same name but different parameter lists.
- Parameter lists can differ in:
  - Number of parameters.
  - Type of parameters.
  - Order of parameters (if types are different).
- Cannot overload methods by return type alone.

Common Mistakes in Overloading

1. **Confusion with Overriding:**
   - Overloading occurs within the same class, whereas overriding occurs between parent and child classes.
2. **Ambiguity Errors:**
   - If two overloaded methods match equally well for a call, the compiler throws an error.

Example:

```
class Example {
    void show(int a, float b) { System.out.println("int-float version");}
    void show(float a, int b) { System.out.println("float-int version");}
}
public class Test {
    public static void main(String[] args) {
        Example ex = new Example();
        // ex.show(10, 10); // Ambiguity: int matches both
        ex.show(float(10), 10) // Outputs float-int version
    }
}
```

Listing 23: Ambiguity in Overloading

Polymorphism: Method Overriding

Definition: Subclasses provide specific implementations of a method defined in the superclass.

Advantages:

- Enables dynamic method invocation at runtime.
- Promotes code flexibility and extensibility.

Explanation: Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass. The overriding method must have the same name, return type, and parameters as the method in the superclass.

Method Overriding in Java

Definition: Method overriding allows a subclass to provide a specific implementation of a method already defined in its superclass.

Key Characteristics:

- The method in the subclass must have the same name, return type, and parameters as in the superclass.
- Used to achieve runtime polymorphism.
- The overridden method in the superclass must not be declared as ‘private‘ or ‘final‘.

Example: Method Overriding

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Polymorphism
        myAnimal.sound(); // Output: Dog barks
    }
}
```

Listing 24: Method Overriding

Rules for Method Overriding

- The method must have the same name, parameter list, and return type as in the superclass.
- The method in the subclass cannot have a more restrictive access modifier than the method in the superclass (e.g., a ‘protected‘ method in the superclass cannot be overridden as ‘private‘).
- The ‘@Override‘ annotation is optional but highly recommended to avoid mistakes.
- Static methods cannot be overridden (they are hidden instead).
- Constructors cannot be overridden.

Common Mistakes in Method Overriding I

1. Incorrect Method Signature:

- If the method signature does not match exactly, it results in method overloading instead of overriding.

Example:

```
class Parent {
    void display() {
        System.out.println("Parent display");
    }
}

class Child extends Parent {
    void display(int value) { // Overloading, not overriding
        System.out.println("Child display with value: " + value);
    }
}
```

Listing 25: Incorrect Method Signature

Common Mistakes in Method Overriding II

2. **Missing '@Override':**
   - Not using '@Override' may lead to silent errors if the method signature does not match.
3. **More Restrictive Access Modifier:**
   - Overridden methods cannot have a more restrictive access level.

Example:

```
class Parent {
    protected void display() {
        System.out.println("Parent display");
    }
}

class Child extends Parent {
    // void display() { // Error: Cannot reduce access level
    //     System.out.println("Child display");
    // }
}
```

Listing 26: Access Modifier Issue

Common Mistakes in Method Overriding III

4. Overriding Static Methods:

- Static methods are not overridden; they are hidden.

Example:

```
class Parent {
    static void display() {
        System.out.println("Parent static display");
    }
}

class Child extends Parent {
    static void display() {
        System.out.println("Child static display");
    }
}

public class Test {
    public static void main(String[] args) {
        Parent.display(); // Output: Parent static display
        Child.display(); // Output: Child static display
    }
}
```

Listing 27: Static Method Hiding

Outline

Introduction to Abstract Classes

Definition: An abstract class in Java is a class that cannot be instantiated and is designed to be extended by subclasses.

Key Characteristics:

- Can include both abstract methods (without implementation) and concrete methods (with implementation).
- Provides a blueprint for subclasses.
- Declared using the ‘abstract‘ keyword.
- Reduces code duplication by allowing shared functionality.

Explanation: An abstract class provides a common base for related classes. Subclasses must implement abstract methods, while they can inherit concrete methods directly.

Use Case: *Define a common interface or shared behavior across related classes.*

Abstract Class Syntax

Syntax:

```
abstract class Shape {
    abstract void draw(); // Abstract method

    public void description() {
        System.out.println("This is a shape.");
    }
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle.");
    }
}

public class Test {
    public static void main(String[] args) {
        Shape s = new Circle();
        s.description();
        s.draw();
    }
}
```

Listing 28: Abstract Class Example

Abstract Class Common Queries

Can an Abstract Class have Static Methods?

- Yes, an abstract class can have static methods.
- Static methods belong to the class, not instances of the class.

Can an Abstract Class have a Constructor?

- Yes, an abstract class can have a constructor.
- The constructor is called when an instance of a subclass is created.

Can an Abstract Class have Nested Classes?

- Yes, an abstract class can have nested classes.
- Nested classes are inner classes that are defined inside another class.

Abstract Class vs Interface

Comparison:

Feature	**Abstract Class**	**Interface**
Methods	Both abstract and concrete	Abstract (default) and default methods (Java 8+)
Multiple Inheritance	Single inheritance	Can implement multiple interfaces
Access Modifiers	Can have any access modifier	Methods are public by default
Fields	Can include non-static and static fields	Only static final constants
Constructors	Allowed	Not Allowed

Abstract Classes: Practical Example

Scenario: Payment System

Abstract Class: Provides a template for different payment methods.

```
abstract class Payment {
    abstract void processPayment();

    void paymentDetails() {
        System.out.println("Processing payment details...");
    }
}

class CreditCardPayment extends Payment {
    void processPayment() {
        System.out.println("Processing credit card payment");
    }
}
class PayPalPayment extends Payment {
    void processPayment() {
        System.out.println("Processing PayPal payment");
    }
}
```

Listing 29: Payment Example

Advanced Example: Abstract Class with Fields and Constructor

```
abstract class Animal {
    String name;

    Animal(String name) { this.name = name;}
    abstract void sound();
    public void eat() { System.out.println(name + " is eating.");}
}

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

    @Override
    void sound() { System.out.println(name + " barks.");}
}

public class Test {
    public static void main(String[] args) {
        Animal a = new Dog("Buddy");
        a.eat();
        a.sound();
    }
}
```

Listing 30: Abstract Class with Constructor

When to Use Abstract Classes

Scenarios:

- When common behavior or structure needs to be shared across related classes.
- When you want to partially implement functionality and enforce subclasses to provide specific implementations.
- When a class should not be instantiated on its own but serves as a base class.

```
abstract class Vehicle {
    abstract void start();
    public void stop() { System.out.println("Vehicle stopped.");}
}
class Car extends Vehicle {
    @Override
    void start() { System.out.println("Car is starting.");}
}
class Bike extends Vehicle {
    @Override
    void start() { System.out.println("Bike is starting.");}
}
```

Listing 31: Abstract Class for Vehicles

Advantages and Disadvantages of Abstract Classes

Advantages:

- Provides a clear contract for subclasses.
- Allows code reuse through concrete methods.
- Encapsulates shared behavior and state.
- Supports single inheritance while maintaining flexibility.

Disadvantages:

- Does not support multiple inheritance.
- Can become rigid if not designed properly.
- More restrictive compared to interfaces for flexibility.

Common Mistakes with Abstract Classes

1. Instantiating Abstract Classes:

```
abstract class Example {}

public class Test {
    public static void main(String[] args) {
        // Example obj = new Example(); // Error: Cannot instantiate
    }
}
```

2. Forgetting to Implement Abstract Methods:

```
abstract class Parent {
    abstract void display();
}

class Child extends Parent {
    // Error: Child must implement abstract method
}
```

Best Practices for Abstract Classes

- Use abstract classes for shared behavior and inheritance.
- Avoid making an abstract class overly specific.
- Use interfaces if multiple inheritance is required.
- Document the purpose of each abstract method.

Outline

Introduction to Interfaces in Java

Definition: An interface in Java is a blueprint for a class that defines a set of methods that the implementing class must provide.

Key Characteristics:

- Declared using the ‘interface‘ keyword.
- All variables (**properties**) are static, final, public, and initialized.
- All methods are implicitly ‘abstract‘ and ‘public‘ (before Java 8).
- From Java 8 onwards, can include default and static methods.
- Supports multiple inheritance through implementation.
- Ensures standardization across implementing classes.

Explanation: An interface contains only method declarations (abstract methods) and static or final fields. Implementing classes must provide functionality for all declared methods.

Use Case: *Define a contract for classes without enforcing a specific implementation hierarchy.*

Basic Syntax of an Interface

Syntax:

```
interface Animal {
    void eat();
    void sleep();
}

class Dog implements Animal {
    public void eat() {
        System.out.println("Dog is eating.");
    }

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

public class Test {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.eat();
        dog.sleep();
    }
}
```

Interfaces vs Abstract Classes

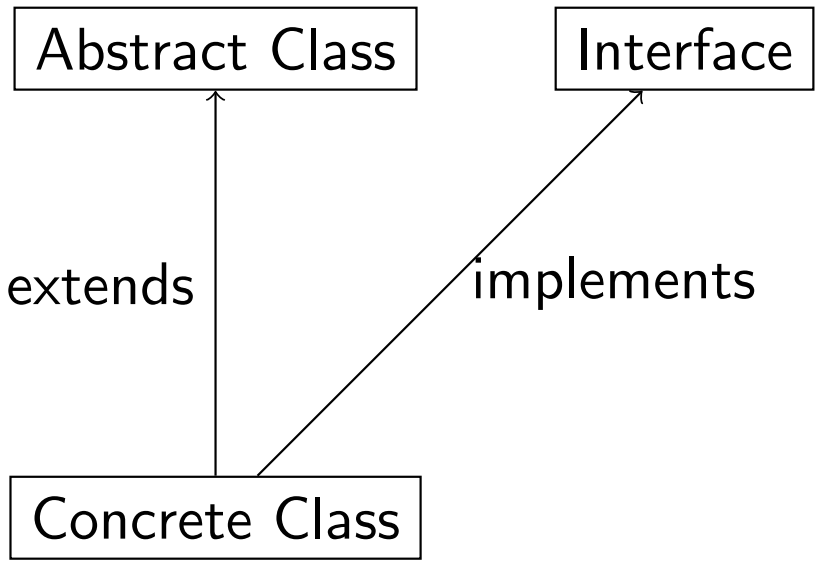
Comparison:

Feature	**Interface**	**Abstract Class**
Methods	Abstract, default, static	Abstract and concrete
Fields	‘public static final‘	Any type (static, instance)
Inheritance	Multiple inheritance	Single inheritance
Constructors	Not allowed	Allowed
Default Methods	Allowed (Java 8+)	Allowed
Multiple Implementations	Can implement multiple interfaces	Cannot extend multiple abstract classes

Abstract Class vs Interface

Key Differences:

- Abstract classes can have state, interfaces cannot.
- Abstract classes can have protected methods, interfaces cannot.
- A class can implement multiple interfaces, but extend only one abstract class.



Interfaces: Practical Example

Scenario: Multiple Behaviors for Robots

Interface: Defines specific behaviors a robot must implement.

```
interface Walkable {
    void walk();
}
interface Talkable {
    void talk();
}

class Robot implements Walkable, Talkable {
    public void walk() {
        System.out.println("Robot walking...");
    }
    public void talk() {
        System.out.println("Robot talking...");
    }
}
```

Listing 32: Robot Example

Advanced Example: Interface Extending Another Interface

Example:

```
interface Animal {
    void eat();
}

interface Mammal extends Animal {
    void giveBirth();
}

class Dolphin implements Mammal {
    public void eat() {
        System.out.println("Dolphin is eating fish.");
    }
    public void giveBirth() {
        System.out.println("Dolphin gives birth to live young.");
    }
}
```

Listing 33: Extending Interfaces

Common Mistakes with Interfaces

1. Forgetting ‘public‘ on Methods:

```
interface Example {
    // void method(); // Error: Method is not public
    public void method();
}
```

Listing 34: Access Modifier Error

2. Attempting to Instantiate an Interface:

```
interface Animal {}

public class Test {
    public static void main(String[] args) {
        // Animal a = new Animal(); // Error: Cannot instantiate
    }
}
```

Listing 35: Instantiation Error

When to Use Interfaces

Scenarios:

- When you need to define a contract for unrelated classes.
- For achieving multiple inheritance.
- When sharing constants across classes.
- To ensure loose coupling and flexibility in design.

Advantages and Disadvantages of Interfaces

Advantages:

- Promotes flexibility and modular design.
- Enables multiple inheritance.
- Provides a mechanism for achieving loose coupling.
- Standardizes behavior across classes.

Disadvantages:

- Lack of implementation can lead to redundant code.
- Can make the code harder to follow if overused.
- Limited to ‘public static final‘ fields.

Best Practices for Interfaces

- Use interfaces for defining contracts and APIs.
- Prefer interfaces over abstract classes when multiple inheritance is needed.
- Avoid adding too many default methods to an interface.
- Use meaningful names for interfaces (e.g., ‘Runnable‘, ‘Comparable‘).
- Document the purpose of each interface and its methods.

Outline

The ‘Object‘ Class in Java

Definition: The ‘Object‘ class is the root class of the Java class hierarchy. Every class in Java implicitly extends this class.

Key Methods Provided by ‘Object‘:

- **‘toString()‘:** Returns a string representation of the object.
- **‘equals(Object obj)‘:** Compares the object with another for equality.
- **‘hashCode()‘:** Returns a hash code value for the object.
- **‘getClass()‘:** Returns the runtime class of the object.
- **‘clone()‘:** Creates a copy of the object (if the class implements ‘Cloneable‘).
- **‘finalize()‘:** Invoked by the garbage collector before an object is destroyed.
- **‘wait()‘, ‘notify()‘, ‘notifyAll()‘:** Used for thread communication.

Example: Using ‘toString()‘ and ‘equals()‘

Example:

```
class Example {
    int id;  String name;
    Example(int id, String name) {
        this.id = id;
        this.name = name;
    }
    @Override
    public String toString() {
        return "Example{id=" + id + ", name='" + name + "'}";
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Example example = (Example) obj;
        return id == example.id && name.equals(example.name);
    }
}
public class Test {
    public static void main(String[] args) {
        Example e1 = new Example(1, "Alice");
        Example e2 = new Example(1, "Alice");
        System.out.println(e1); // Outputs: Example{id=1, name='Alice'}
        System.out.println(e1.equals(e2)); // Outputs: true
    }
}
```

Listing 36: Overriding Object Methods

Cloning Objects I

Definition: Cloning creates a copy of an object.
Advantages:

- Useful for creating object backups.
- Reduces the overhead of creating new instances from scratch.

Explanation: Cloning is achieved by implementing the ‘Cloneable‘ interface and overriding the ‘clone()‘ method from the ‘Object‘ class. This creates a shallow copy of the object.

Cloning Objects II

Example:

```
class Student implements Cloneable {
    String name;
    int rollNo;
    Student(String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Student s1 = new Student("Alice", 101);
        Student s2 = (Student) s1.clone();
        System.out.println(s2.name + " " + s2.rollNo);
    }
}
```

Listing 37: Cloning Example

Best Practices: Working with ‘Object‘ Methods

- Always override ‘toString()‘ to provide meaningful string representations of your objects.
- Implement ‘equals()‘ and ‘hashCode()‘ together to ensure consistency when using objects in collections.
- Use ‘getClass()‘ to implement type-specific logic dynamically.
- Be cautious when overriding ‘clone()‘ as it requires implementing the ‘Cloneable‘ interface.

Outline

Shallow Copy vs. Deep Copy in Java

Shallow Copy:

- Copies the object's fields but not the objects referenced by those fields.
- Changes to the referenced objects affect both the original and the copied object.
- Typically achieved using 'Object.clone()'.

Deep Copy:

- Creates a new object and recursively copies all objects referenced by the fields.
- Changes to the referenced objects do not affect the original or vice versa.
- Requires explicit implementation.

Example: Shallow Copy

Example:

```
class Address {
    String city;
    Address(String city) {
        this.city = city;
    }
}
class Person implements Cloneable {
    String name; Address address;
    Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }
    @Override protected Object clone() throws CloneNotSupportedException {
        return super.clone(); // Shallow copy
    }
}

public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Address addr = new Address("New York");
        Person p1 = new Person("Alice", addr);
        Person p2 = (Person) p1.clone();
        p2.address.city = "San Francisco";
        System.out.println(p1.address.city); // Outputs: San Francisco
    }
}
```

Listing 38: Shallow Copy

Example: Deep Copy

Example:

```
class Address {
    String city;
    Address(String city) { this.city = city; }
    @Override protected Object clone() throws CloneNotSupportedException {
        return new Address(this.city);
    }
}
class Person implements Cloneable {
    String name;    Address address;
    Person(String name, Address address) { this.name = name; this.address = address; }
    @Override protected Object clone() throws CloneNotSupportedException {
        Person cloned = (Person) super.clone();
        cloned.address = (Address) this.address.clone(); // Deep copy
        return cloned;
    }
}
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Address addr = new Address("New York");
        Person p1 = new Person("Alice", addr);
        Person p2 = (Person) p1.clone();
        p2.address.city = "San Francisco";
        System.out.println(p1.address.city); // Outputs: New York
    }
}
```

Listing 39: Deep Copy

Cloning: ‘Cloneable‘ vs. Copy Constructor

Using ‘Cloneable‘:

- Provides a shallow copy by default.
- Requires implementing the ‘Cloneable‘ interface and overriding ‘clone()‘.
- May require deep copying for complex objects.

Using Copy Constructor:

- Explicitly defines how to copy fields and objects.
- More control over the copying process.
- Easier to implement deep copies.

Example: Cloning with Copy Constructor

Example:

```
class Address {
    String city;
    Address(String city) {this.city = city;}
    Address(Address other) {this.city = other.city;}
}

class Person {
    String name; Address address;
    Person(String name, Address address) {
        this.name = name;
        this.address = new Address(address); // Deep copy
    }
    Person(Person other) {
        this.name = other.name;
        this.address = new Address(other.address); // Deep copy
    }
}
public class Test {
    public static void main(String[] args) {
        Address addr = new Address("New York");
        Person p1 = new Person("Alice", addr);
        Person p2 = new Person(p1); // Copy constructor
        p2.address.city = "San Francisco";
        System.out.println(p1.address.city); // Outputs: New York
    }
}
```

Listing 40: Copy Constructor

Advantages and Disadvantages: ‘Cloneable‘ vs. Copy Constructor I

Advantages of ‘Cloneable‘:

- Built-in mechanism for object cloning.
- Provides a default shallow copy implementation.
- Supports cloning of arrays directly.

Disadvantages of ‘Cloneable‘:

- Requires handling ‘CloneNotSupportedException‘.
- Deep copying needs manual implementation for non-primitive fields.
- Potentially violates encapsulation by accessing private fields.

Advantages and Disadvantages: ‘Cloneable‘ vs. Copy Constructor II

Advantages of Copy Constructor:

- Full control over the copying process.
- Can easily implement deep copies.
- Avoids the pitfalls of ‘Cloneable‘ (e.g., exception handling).

Disadvantages of Copy Constructor:

- Requires explicit implementation for each class.
- Does not support polymorphic copying automatically.
- More verbose compared to ‘Cloneable‘ for simple cases.

Outline

Introduction to Wrapper Classes in Java

Definition: Wrapper classes provide an object representation for primitive data types in Java.

Primitive Types and Their Wrappers:

- ‘byte‘ - ‘Byte‘
- ‘short‘ - ‘Short‘
- ‘int‘ - ‘Integer‘
- ‘long‘ - ‘Long‘
- ‘float‘ - ‘Float‘
- ‘double‘ - ‘Double‘
- ‘char‘ - ‘Character‘
- ‘boolean‘ - ‘Boolean‘

Examples: Using Wrapper Classes

Example: Autoboxing and Unboxing

```
public class WrapperExample {
    public static void main(String[] args) {
        // Autoboxing: Primitive to Wrapper
        Integer intWrapper = 10;

        // Unboxing: Wrapper to Primitive
        int intValue = intWrapper;

        System.out.println("Wrapper: " + intWrapper);
        System.out.println("Primitive: " + intValue);
    }
}
```

Listing 41: Autoboxing and Unboxing

Advanced Usage: Wrapper Classes

Example: Conversion and Utilities

```
public class WrapperUtilities {
    public static void main(String[] args) {
        // Parsing Strings to Primitives
        int parsedInt = Integer.parseInt("123");
        System.out.println("Parsed Integer: " + parsedInt
            );

        // Converting to Binary and Hexadecimal Strings
        String binary = Integer.toBinaryString(10);
        String hex = Integer.toHexString(255);
        System.out.println("Binary: " + binary);
        System.out.println("Hexadecimal: " + hex);
    }
}
```

Listing 42: Wrapper Utilities

When to Use Wrapper Classes

- When working with collections (e.g., ‘ArrayList‘, ‘HashMap‘) that require objects.
- To leverage utility methods (e.g., ‘Integer.parseInt()‘ for string-to-integer conversion).
- For null values to represent "no value" (not possible with primitives).
- When working with frameworks that require objects (e.g., Java Streams, Reflection APIs).
- For type conversions and string representations of numbers.

Advantages and Disadvantages of Wrapper Classes

Advantages:

- Enable the use of primitives in collections and frameworks.
- Provide utility methods for type conversion and operations.
- Allow representation of null for missing or optional values.
- Support conversions to various formats (binary, hex, etc.).

Disadvantages:

- Additional memory overhead compared to primitives.
- Autoboxing and unboxing may lead to performance issues in loops.
- Risk of ‘NullPointerException‘ if a null wrapper is unboxed.
- Increased complexity for developers unfamiliar with the nuances of wrappers.

Best Practices for Wrapper Classes

- Prefer primitives over wrappers for performance-critical code.
- Use ‘Optional‘ for nullable values instead of wrappers.
- Minimize autoboxing/unboxing in performance-sensitive scenarios.
- Use utility methods provided by wrapper classes for type conversion.
- Avoid unnecessary conversions between primitives and wrappers.

Outline

Introduction to Java Streams

Definition: Java Streams are a part of the Java 8 Stream API that enable functional-style operations on collections and sequences of data.

Key Features:

- Declarative programming for data processing.
- Supports operations like ‘filter‘, ‘map‘, ‘reduce‘, ‘sorted‘, etc.
- Can be sequential or parallel for better performance.

Types of Streams

- **Stream:** Handles objects.
- **IntStream, LongStream, DoubleStream:** Handle primitives to avoid boxing overhead.

Example:

```
import java.util.stream.*;

public class StreamExample {
    public static void main(String[] args) {
        Stream<Integer> objectStream = Stream.of(1, 2, 3, 4);
        IntStream intStream = IntStream.of(1, 2, 3, 4);
    }
}
```

Listing 43: Creating Streams

Examples of Stream Operations

Example: Filtering and Mapping

```java
import java.util.*;
import java.util.stream.*;

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

        names.stream()
             .filter(name -> name.startsWith("A"))
             .map(String::toUpperCase)
             .forEach(System.out::println);
    }
}
// Output: ALICE
```

Listing 44: Stream Example

Example: Reducing and Collecting

```
import java.util.*;
import java.util.stream.*;

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

        // Reduce Example
        int sum = numbers.stream().reduce(0, Integer::sum);
        System.out.println("Sum: " + sum);

        // Collect Example
        List<Integer> squaredNumbers = numbers.stream()
                                              .map(n -> n * n)
                                              .collect(Collectors.toList());
        System.out.println("Squared: " + squaredNumbers);
    }
}
```

Listing 45: Stream Reduce and Collect

When to Use Streams

- For complex data transformations or aggregations.
- When working with large datasets where parallelism can improve performance.
- To simplify and make code more readable compared to traditional loops.
- When immutable or stateless data transformations are needed.

Advantages and Disadvantages of Streams

Advantages

- Cleaner and more declarative code.
- Built-in support for parallelism.
- Reduces boilerplate code for data operations.
- Encourages immutability and stateless programming.

Disadvantages

- Not ideal for operations with side-effects.
- Can be less intuitive for developers new to functional programming.
- Overhead of creating streams for small datasets.
- Debugging stream operations can be challenging.

Best Practices for Streams

- Use parallel streams cautiously; ensure thread-safety of operations.
- Avoid modifying external state in stream operations.
- Use primitive streams (‘IntStream‘, etc.) to avoid boxing overhead.
- Chain multiple operations to leverage the power of streams.
- Use terminal operations (‘forEach‘, ‘collect‘, etc.) judiciously to conclude stream pipelines.

Outline

Introduction

Object References and Type Compatibility:
In object-oriented programming, it's essential to understand which object references can hold which type of objects.

Base Class Reference

Base Class Reference A base class reference can hold objects of its own type or any of its derived classes.

```
Animal animal = new Dog(); // OK
Animal animal = new Cat(); // OK
Animal animal = new Animal(); // OK
```

Note: "Animal" is the base class, and "Dog" and "Cat" are derived classes.

Derived Class Reference

Derived Class Reference A derived class reference can only hold objects of its own type or any of its own derived classes.

```
Dog dog = new Dog(); // OK
Dog dog = new GoldenRetriever(); // OK (if GoldenRetriever is a subclass of Dog)
Dog dog = new Animal(); // Error (Animal is a superclass of Dog)
```

Note: "Dog" is a derived class, and "GoldenRetriever" is a subclass of "Dog".

Abstract Class Reference

Abstract Class Reference An abstract class reference can hold objects of its own type or any of its concrete subclasses.

```
Shape shape = new Circle(); // OK
Shape shape = new Rectangle(); // OK
Shape shape = new Shape(); // Error (Shape is abstract)
```

Note: "Shape" is an abstract class, and "Circle" and "Rectangle" are concrete subclasses.

Interface Reference

Interface Reference An interface reference can hold objects of any class that implements the interface.

```
Printable printable = new Document(); // OK (if Document implements Printable)
Printable printable = new Image(); // OK (if Image implements Printable)
```

Note: "Printable" is an interface, and "Document" and "Image" are classes that implement the "Printable" interface.

Interface Extending Another Interface

Interface Extending Another Interface: If an interface extends another interface, a class that implements the child interface also implements the parent interface.

```
interface Printable {
    void print();
}

interface ColorPrintable extends Printable {
    void printColor();
}

class Document implements ColorPrintable {
    @Override
    public void print() {
        // implement print
    }

    @Override
    public void printColor() {
        // implement printColor
    }
}

Printable printable = new Document(); // OK
ColorPrintable colorPrintable = new Document(); // OK
```

Grandparent Class Reference

Grandparent Class Reference: A grandparent class reference can hold objects of its own type, its child classes, or its grandchild classes.

```
Grandparent grandparent = new Grandparent(); // OK
Grandparent grandparent = new Parent(); // OK
Grandparent grandparent = new Child(); // OK
```

Summary

Summary: In summary, the following object references can hold the following types of objects:

- Base class reference: base class, derived classes
- Derived class reference: derived class, its own derived classes
- Abstract class reference: abstract class, non-abstract derived classes
- Interface reference: any class that implements the interface
- Grandparent class reference: grandparent class, child classes, grandchild classes
- Interface extending another interface: a class that implements the child interface also implements the parent interface

Outline

Static Methods in Overloading and Overriding I

Overloading:

- Static methods can be overloaded like regular methods.
- The method signature (name + parameters) must differ.

Example 1: Different Parameter Types

```
class Example {
    static void display(int a) {
        System.out.println("Integer: " + a);
    }

    static void display(String s) {
        System.out.println("String: " + s);
    }

    public static void main(String[] args) {
        Example.display(10);
        Example.display("Hello");
    }
}
```

Listing 46: Static Method Overloading with Different Types

Static Methods in Overloading and Overriding II

Example 2: Different Number of Parameters

```
class Example {
    static void display(int a) {
        System.out.println("One parameter: " + a);
    }

    static void display(int a, int b) {
        System.out.println("Two parameters: " + (a + b));
    }

    public static void main(String[] args) {
        Example.display(5);
        Example.display(5, 10);
    }
}
```

Listing 47: Static Method Overloading with Different Parameters

Static Methods in Overriding I

Key Points:

- Static methods cannot be overridden; they are hidden.
- The method belongs to the class, not the object.
- If a subclass defines a static method with the same signature, it hides the superclass method.

Example 1: Hiding Static Methods

```
class Parent {
    static void display() { System.out.println("Parent static display");}
}
class Child extends Parent {
    static void display() { System.out.println("Child static display");}
}
public class Test {
    public static void main(String[] args) {
        Parent.display(); // Output: Parent static display
        Child.display();  // Output: Child static display
    }
}
```

Listing 48: Static Method Hiding

Static Methods in Overriding II

Example 2: Static Method Behavior with References

```
class Parent {
    static void show() { System.out.println("Parent show");}
}
class Child extends Parent {
    static void show() { System.out.println("Child show");}
}
public class Test {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.show(); // Output: Parent show (static binding)
    }
}
```

Listing 49: Static Methods Using References

Final Methods in Overloading and Overriding I

Overloading:

- Final methods can be overloaded like any other method.

Example 1: Overloading with Different Parameters

```
class Example {
    final void display(int a) { System.out.println("Integer: " + a);}
    final void display(String s) { System.out.println("String: " + s);}

    public static void main(String[] args) {
        Example obj = new Example();
        obj.display(10);
        obj.display("Hello");
    }
}
```

Listing 50: Final Method Overloading

Example 2: Overloading with Different Return Types

Final Methods in Overloading and Overriding II

```
class Example {
    final int display(int a) { return a * 2;}
    final String display(String s) { return "Hello, " + s;}

    public static void main(String[] args) {
        Example obj = new Example();
        System.out.println(obj.display(5));
        System.out.println(obj.display("World"));
    }
}
```

Listing 51: Overloading with Return Types

Final Methods in Overriding

Key Points:

- Final methods cannot be overridden.
- Prevents alteration of critical functionality in subclasses.

Example: Attempting to Override a Final Method

```
class Parent {
    final void display() {
        System.out.println("Parent display");
    }
}

class Child extends Parent {
    // void display() { // Error: Cannot override final method
    //     System.out.println("Child display");
    // }
}
```

Listing 52: Final Method Example

Access Specifiers in Overloading and Overriding

Overloading:

- Access specifiers do not affect method overloading.
- Methods with the same name but different parameter lists can have different access modifiers.

Example:

```
class Example {
    public void display(int a) {
        System.out.println("Public Integer: " + a);
    }

    private void display(String s) {
        System.out.println("Private String: " + s);
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.display(10);
        // obj.display("Hello"); // Error: Cannot access private method
    }
}
```

Listing 53: Access Specifier with Overloading

Access Specifiers in Overriding I

Key Points:

- The access specifier of an overriding method cannot be more restrictive than the method in the superclass.
- This ensures that the overridden method is at least as accessible as the original.

Example 1: Valid Access Modifiers

```
class Parent {
    protected void display() { System.out.println("Parent display");}
}
class Child extends Parent {
    @Override
    public void display() { System.out.println("Child display");}
}
public class Test {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.display(); // Output: Child display
    }
}
```

Listing 54: Access Specifier with Overriding

Access Specifiers in Overriding II

Example 2: Invalid Access Modifiers

```
class Parent {
    public void display() { System.out.println("Parent display");}
}

class Child extends Parent {
    // protected void display() { // Error: Cannot reduce visibility
    //     System.out.println("Child display");
    // }
}
```

Listing 55: Invalid Overriding with Access Specifiers

Outline

Abstract Class with Generics

Syntax:

```
abstract class Container<T> {
    private T value;

    public Container(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    abstract void process();
}

class StringContainer extends Container<String> {
    public StringContainer(String value) {
        super(value);
    }

    @Override
    void process() {
        System.out.println("Processing string: " + getValue());
    }
}
```

Listing 56: Abstract Class with Generics

Abstract Class with Nested Classes

Syntax:

```
abstract class University {
    abstract void displayInfo();

    public static class Department {
        private String name;

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

        public void displayDepartment() {
            System.out.println("Department: " + name);
        }
    }
}

class MIT extends University {
    @Override
    void displayInfo() {
        System.out.println("MIT University");
    }

    public static void main(String[] args) {
        University.Department csDepartment = new University.Department("Computer
            Science");
        csDepartment.displayDepartment();
    }
}
```

Abstract Class with Lambda Expressions

Syntax:

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
}

abstract class Calculator {
    abstract MathOperation getOperation();

    public int calculate(int a, int b) {
        return getOperation().operation(a, b);
    }
}

class AdditionCalculator extends Calculator {
    @Override
    MathOperation getOperation() {
        return (a, b) -> a + b; // Lambda expression
    }
}
```

Listing 58: Abstract Class with Lambda Expressions

Abstract Class with Lambda Expressions

Syntax:

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
}

abstract class Calculator {
    abstract MathOperation getOperation();

    public int calculate(int a, int b) {
        return getOperation().operation(a, b);
    }
}

class AdditionCalculator extends Calculator {
    @Override
    MathOperation getOperation() {
        return (a, b) -> a + b; // Lambda expression
    }
}
```

Listing 59: Abstract Class with Lambda Expressions

Outline

Interfaces vs Abstract Classes After Java 8 and 9

Key Differences Introduced in Java 8 and 9:

- Interfaces can now have **default methods** (Java 8) that provide concrete implementations.
- Interfaces can contain **static methods** (Java 8) that belong to the interface itself.
- Interfaces in Java 9 introduced **private methods**, allowing code reuse within interfaces.
- Abstract classes remain the choice for partially implemented functionality with instance variables.

Example: Default and Static Methods in Interfaces (Java 8)

```
interface Vehicle {
    void start();

    default void stop() {
        System.out.println("Vehicle stopped.");
    }

    static void maintenance() {
        System.out.println("Performing maintenance.");
    }
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting.");
    }
}

public class Test {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start();
        car.stop();
        Vehicle.maintenance();
    }
}
```

Listing 60: Default and Static Methods

Example: Private Methods in Interfaces (Java 9)

```java
interface Logger {
    default void log(String message) {
        printMessage("LOG: " + message);
    }

    private void printMessage(String msg) {
        System.out.println(msg);
    }
}

class ApplicationLogger implements Logger {}

public class Test {
    public static void main(String[] args) {
        ApplicationLogger logger = new ApplicationLogger();
        logger.log("Application started.");
    }
}
```

Listing 61: Private Methods in Interface

How Java Handles the Diamond Problem with Default Methods

The Diamond Problem: In traditional multiple inheritance, if a class inherits methods with the same signature from multiple parent classes, it creates ambiguity about which method to execute.

Java's Solution:

- If a class implements multiple interfaces that provide default methods with the same signature, the class must explicitly override the method to resolve ambiguity.
- The class can call a specific interface's default method using 'InterfaceName.super.methodName()'.

Example: Resolving Diamond Problem in Java

```
interface A {
    default void show() {
        System.out.println("Interface A");
    }
}

interface B {
    default void show() {
        System.out.println("Interface B");
    }
}

class C implements A, B {
    public void show() {
        System.out.println("Resolving conflict in C");
        A.super.show(); // Call specific interface's method
    }
}

public class Test {
    public static void main(String[] args) {
        C obj = new C();
        obj.show();
    }
}
```

Listing 62: Diamond Problem Resolution

Key Takeaways for Handling Diamond Problem

- Java does not allow multiple inheritance of classes to prevent ambiguity.
- Default methods in interfaces provide multiple inheritance-like behavior.
- If an implementing class inherits conflicting default methods, it must override them explicitly.
- The ‘InterfaceName.super.methodName()‘ syntax allows calling a specific interface’s method.

Introduction to Functional Interfaces

What is a Functional Interface?

- An interface that has only one abstract method.
- Can have multiple default and static methods.
- **Example:** Runnable, ActionListener, Comparator

Lambda Expressions (Introduced in Java 8): A shorthand way to represent an instance of a functional interface.

```
Runnable r = () -> System.out.println("Hello World");
r.run();
```

Method References (Introduced in Java 8): A shorthand way to represent an instance of a functional interface.

```
Comparator<String> c = String::compareTo;
System.out.println(c.compare("hello", "world"));
```

Benefits of Functional Interfaces

Benefits

- Simplifies code and reduces boilerplate.
- Improves readability and maintainability.
- Enables more concise and expressive code.

Example:

```
public interface MathOperation {
  int operation(int a, int b);
}

MathOperation addition = (a, b) -> a + b;
System.out.println(addition.operation(2, 3));
```

Listing 63: Functional Interface Example

Best Practices for Functional Interfaces

Best Practices

- Use functional interfaces to represent single-method interfaces.
- Use lambda expressions and method references to represent instances of functional interfaces.
- Keep functional interfaces simple and focused on a single task.

Example:

```
public interface Logger {
  void log(String message);
}

Logger logger = message -> System.out.println(message);
logger.log("Hello World");
```

Listing 64: Functional Interface Example

Interface with Anonymous Class

Example:

```
// Define an interface
interface Printable {
  void print(String message);
}

// Create an instance of the interface using an anonymous class
Printable printer = new Printable() {
  @Override
  public void print(String message) {
    System.out.println(message);
  }
};

// Use the instance
printer.print("Hello World");
```

Listing 65: Interface with Anonymous Class

Takeaways

How it works:

- We define an interface ‘Printable‘ with a single method ‘print‘.
- We create an instance of the interface using an anonymous class.
- The anonymous class implements the ‘print‘ method.
- We use the instance to call the ‘print‘ method.

Benefits:

- We don’t need to create a separate class file for the implementation.
- The implementation is defined inline, making the code more concise.
- We can use the instance immediately after defining it.

Real-World Example

```java
// Define an interface for a button click listener
interface ButtonClickListener {
  void onClick();
}

// Create a button and add a click listener using an
    anonymous class
Button button = new Button("Click me");
button.addActionListener(new ButtonClickListener() {
  @Override
  public void onClick() {
    System.out.println("Button clicked");
  }
});
```

Outline

Quick Review

Interface Inheritance

Interface Inheritance:

- An interface can extend another interface using the ‘extends‘ keyword.
- The child interface inherits all the methods of the parent interface.
- A class that implements the child interface must provide an implementation for all the methods in the parent interface.

```
public interface ParentInterface {
    void method1();
}

public interface ChildInterface extends ParentInterface {
    void method2();
}

public class MyClass implements ChildInterface {
    @Override
    public void method1() {
        // implementation
    }

    @Override
    public void method2() {
        // implementation
    }
}
```

Class Implementing Multiple Interfaces

Class Implementing Multiple Interfaces:

- A class can implement multiple interfaces using the ‘implements‘ keyword.
- The class must provide an implementation for all the methods in all the interfaces.

```
public interface Interface1 {
    void method1();
}

public interface Interface2 {
    void method2();
}

public class MyClass implements Interface1, Interface2 {
    @Override
    public void method1() {
        // implementation
    }

    @Override
    public void method2() {
        // implementation
    }
}
```

Abstract Class Implementing an Interface

Abstract Class Implementing an Interface:

- An abstract class can implement an interface using the ‘implements‘ keyword.
- The abstract class must provide an implementation for all the methods in the interface, or declare them as abstract.

```
public interface MyInterface {
    void method1();
}

public abstract class MyAbstractClass implements MyInterface {
    @Override
    public void method1() {
        // implementation
    }
}

public class MyClass extends MyAbstractClass {
    // no need to implement method1()
}
```

Type Casting

Type Casting:

- Type casting is used to convert an object reference to a different type.
- There are two types of type casting: upcasting and downcasting.

```
public class Animal {
    //...
}

public class Dog extends Animal {
    //...
}

Animal animal = new Dog();
Dog dog = (Dog) animal; // downcasting
```

Instanceof Operator

Instanceof Operator:

- The instanceof operator is used to check if an object is an instance of a particular class or interface.
- It returns true if the object is an instance of the class or interface, and false otherwise.

```
public class Animal {
    //...
}

public class Dog extends Animal {
    //...
}

Animal animal = new Dog();
if (animal instanceof Dog) {
    System.out.println("animal is a Dog");
}
```

Polymorphism

Polymorphism:

- Polymorphism is the ability of an object to take on multiple forms, depending on the context in which it is used.
- There are two types of polymorphism: method overloading and method overriding.

```
public class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

public class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

Animal animal = new Dog();
animal.sound(); // outputs "Dog barks"
```

Method Overloading

Method Overloading:

- Method overloading is a form of polymorphism where multiple methods with the same name can be defined, but with different parameters.
- The method to be called is determined by the number and types of parameters passed to it.

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

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

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

Method Overriding

Method Overriding:

- Method overriding is a form of polymorphism where a subclass provides a specific implementation for a method that is already defined in its superclass.
- The method in the subclass has the same name, return type, and parameters as the method in the superclass.

```
public class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

public class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

Java 8 Default Methods

Java 8 Default Methods:

- Java 8 introduced default methods, which allow interfaces to provide a default implementation for methods.
- Default methods are used to add new functionality to an interface without breaking existing code.

```
public interface MyInterface {
    default void method1() {
        System.out.println("Default implementation");
    }
}

public class MyClass implements MyInterface {
    // no need to implement method1()
}
```

Java 9 Private Methods in Interfaces

Java 9 Private Methods in Interfaces:

- Java 9 introduced private methods in interfaces, which allow interfaces to provide private helper methods.
- Private methods are used to encapsulate implementation details and improve code organization.

```
public interface MyInterface {
    default void method1() {
        helperMethod();
    }

    private void helperMethod() {
        System.out.println("Helper method");
    }
}
```

Discussion Scenarios

1. **Abstract Class vs Interface:**
   - When would you choose an abstract class over an interface for a plugin system?
2. **Polymorphism:**
   - Can you design a dynamic discount system where behavior changes based on user type (e.g., Student, Senior Citizen)?
3. **Streams:**
   - How can you use streams to process large datasets efficiently?

Exercises for Students

1. **Polymorphism:**
   - Create a class hierarchy for ‘Vehicle‘ with subclasses ‘Car‘ and ‘Bike‘. Implement method overriding for ‘start()‘ and ‘stop()‘.
   - Extend the hierarchy to include a ‘Bus‘ class with additional methods.
2. **Abstract Classes:**
   - Create an abstract class ‘Employee‘ with attributes ‘name‘ and ‘id‘. Add abstract methods for calculating salary.
   - Extend the class for ‘Manager‘ and ‘Developer‘.
   - Add a ‘Tester‘ subclass to calculate test case completion rates.
3. **Cloning:**
   - Implement a ‘Product‘ class with attributes ‘name‘ and ‘price‘. Demonstrate cloning to create duplicates.
   - Extend the program to include deep cloning for a ‘ProductBundle‘ class containing multiple ‘Product‘ objects.

Discussion Questions

- How does method overriding enhance polymorphism?
- What scenarios require abstract classes versus interfaces?
- What are the challenges of using cloning in real-world applications?
- Can you think of scenarios where deep cloning is essential?

Polymorphism: Discussion Points

Inheritance:

- Why does Java not support multiple inheritance for classes?
- How do interfaces provide an alternative?

Polymorphism:

- Why is operator overloading not supported in Java?
- Can you think of real-world examples where polymorphism is useful?