Inheritance, Polymorphism, Abstract Classes & Interfaces

Object-Oriented Programming in Java

PT821: Object-Oriented Programming

State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

1. Inheritance
2. Polymorphism
3. Abstract Classes
4. Interfaces
5. Combining All Concepts
6. Summary

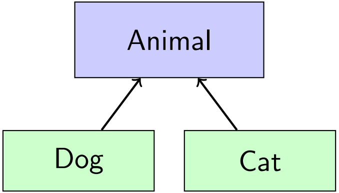
What is Inheritance?

Definition

Inheritance is a mechanism where a new class **inherits** properties and behaviors from an existing class.

Key Terms:

- **Parent/Super Class** - The class being inherited from
- **Child/Sub Class** - The class that inherits



Why Use Inheritance?

Benefits:

1. **Code Reusability** - Write once, use many times
2. **Method Overriding** - Customize inherited behavior
3. **Extensibility** - Easy to add new features
4. **Maintainability** - Changes in one place

Real-World Analogy

A child inherits traits from parents but can have their own unique characteristics!

The extends Keyword

Syntax

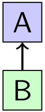
```java
class ChildClass extends ParentClass {
    // Child class body
}
```

Example:

```java
class Animal {
    String name;
    void eat() {
        System.out.println(name + " is eating");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println(name + " says Woof!");
    }
}
```

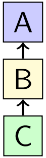
Types of Inheritance in Java

1. **Single**



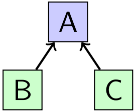
One parent, one child

2. **Multilevel**



Chain of inheritance

3. **Hierarchical**



One parent, many children

Note

Java does NOT support multiple inheritance with classes (use interfaces instead).

The super Keyword

The super keyword refers to the parent class.

Three Uses:

1. Call Parent Constructor

```java
class Dog extends Animal {
    Dog(String name) {
        super(name); // calls Animal()
    }
}
```

2. Access Parent Method

```java
void display() {
    super.display(); // parent's
    // then child's code
}
```

3. Access Parent Variable: `super.variableName`

Complete Inheritance Example

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

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

    void introduce() {
        System.out.println("I am " + name + ", " + age + " years old");
    }
}

class Student extends Person {
    String studentId;

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

    void study() {
```

What is Polymorphism?

Definition

Polymorphism = ”many forms” - the ability of an object to take different forms.

Two Types:

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

Real-World Example

A person can be a student, employee, and parent - same person, different roles!

Method Overloading (Compile-time)

Same method name, different parameters

```java
class Calculator {
    // Two integers
    int add(int a, int b) {
        return a + b;
    }

    // Three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Two doubles
    double add(double a, double b) {
        return a + b;
    }
}
```

Rules

Must differ in: number of parameters, type of parameters, or order of parameters.

Method Overriding (Runtime)

Child class provides specific implementation of parent's method

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

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Woof! Woof!");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow!");
    }
}
```

@Override Annotation

Polymorphism in Action

```java
public class Main {
    public static void main(String[] args) {
        // Parent reference, child objects
        Animal myAnimal;

        myAnimal = new Dog();
        myAnimal.makeSound();  // Output: Woof! Woof!

        myAnimal = new Cat();
        myAnimal.makeSound();  // Output: Meow!

        // Array of Animals
        Animal[] animals = {new Dog(), new Cat(), new Dog()};
        for (Animal a : animals) {
            a.makeSound();  // Each calls its own version!
        }
    }
}
```

Key Point

The same method call produces different results based on the actual object type!

Overloading vs Overriding

Aspect	**Overloading**	**Overriding**
When decided	Compile-time	Runtime
Where	Same class	Parent-Child
Parameters	Must differ	Must be same
Return type	Can differ	Must be same/covariant
Keyword	None	@Override

What is an Abstract Class?

Definition

An **abstract class** is a class that cannot be instantiated and may contain abstract methods (methods without implementation).

Key Characteristics:

- Declared with `abstract` keyword
- **Cannot** create objects directly
- **Can** have both abstract and concrete methods
- **Can** have constructors and instance variables
- Child classes **must** implement all abstract methods

Purpose

Provides a common base with some implementation, forcing subclasses to complete the rest.

Abstract Class Syntax

```java
abstract class Shape {
    String color;

    // Constructor
    Shape(String color) {
        this.color = color;
    }

    // Abstract method - no body!
    abstract double calculateArea();

    // Concrete method - has body
    void displayColor() {
        System.out.println("Color: " + color);
    }
}
```

Note

Abstract methods end with semicolon - no curly braces!

Implementing Abstract Class

```java
class Circle extends Shape {
    double radius;

    Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

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

class Rectangle extends Shape {
    double width, height;

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

Using Abstract Classes

```java
public class Main {
    public static void main(String[] args) {
        // Shape s = new Shape("Red");  // ERROR! Cannot instantiate

        Shape circle = new Circle("Red", 5.0);
        Shape rect = new Rectangle("Blue", 4.0, 6.0);

        System.out.println("Circle area: " + circle.calculateArea());
        System.out.println("Rectangle area: " + rect.calculateArea());

        circle.displayColor();  // Inherited concrete method
    }
}
```

Output:

```
Circle area: 78.54
Rectangle area: 24.0
Color: Red
```

What is an Interface?

Definition

An **interface** is a contract that defines what a class must do, but not how it does it.

Key Characteristics:

- All methods are **public abstract** by default
- All variables are **public static final** (constants)
- A class **implements** an interface
- A class can implement **multiple** interfaces
- Cannot have constructors

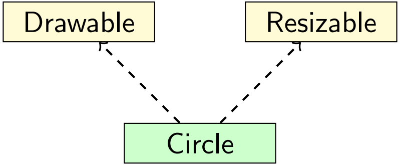
Real-World Analogy

Like a contract or job description - it says what must be done, but not how to do it.

Interface Syntax

```java
interface Drawable {
    void draw();  // public abstract by default
}

interface Resizable {
    void resize(double factor);
    double getSize();
}

// Implementing interfaces
class Circle implements Drawable, Resizable {
    double radius = 5.0;

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

    @Override
    public void resize(double factor) {
        radius *= factor;
    }
```

Multiple Interface Implementation



Drawable

Resizable

Circle

```java
class Circle implements Drawable, Resizable {
    // Must implement ALL methods from BOTH interfaces
}
```

This is how Java achieves ”multiple inheritance”

A class can extend only ONE class but implement MANY interfaces!

Abstract Class vs Interface

Feature	**Abstract Class**	**Interface**
Methods	Abstract + Concrete	Abstract only*
Variables	Any type	Constants only
Constructor	Yes	No
Inheritance	extends (single)	implements (multiple)
Access modifiers	Any	Public only
Use when	IS-A relationship	CAN-DO capability

*Java 8+ allows default and static methods in interfaces

When to Use What?

Use Abstract Class when:

- Classes share common code
- Need non-public members
- Need constructors
- Want to provide default behavior

Example: Animal → Dog, Cat

Use Interface when:

- Unrelated classes need same behavior
- Need multiple inheritance
- Define a contract/capability
- Want loose coupling

Example: Comparable, Serializable

Real-World Example: Payment System

```java
// Interface - defines capability
interface Payable {
    void processPayment(double amount);
}

// Abstract class - common base
abstract class Payment implements Payable {
    protected String transactionId;

    Payment() {
        this.transactionId = generateId();
    }

    private String generateId() {
        return "TXN" + System.currentTimeMillis();
    }

    abstract void validate();  // Each payment validates differently
}
```

Payment System (Continued)

```java
class CreditCard extends Payment {
    private String cardNumber;

    CreditCard(String cardNumber) {
        super();
        this.cardNumber = cardNumber;
    }

    @Override
    void validate() {
        System.out.println("Validating card: " + cardNumber);
    }

    @Override
    public void processPayment(double amount) {
        validate();
        System.out.println("Processing $" + amount + " via Credit Card");
    }
}

class MobileMoney extends Payment {
    private String phoneNumber;
    // Similar implementation...
```

Using the Payment System

```java
public class PaymentDemo {
    public static void main(String[] args) {
        // Polymorphism - same interface, different implementations
        Payable[] payments = {
            new CreditCard("1234-5678-9012-3456"),
            new MobileMoney("+255-123-456-789"),
            new BankTransfer("SUZA-ACCOUNT-001")
        };

        double amount = 50000.0;

        for (Payable payment : payments) {
            payment.processPayment(amount);
            System.out.println("---");
        }
    }
}
```

Key Benefit

Easy to add new payment methods without changing existing code!

Key Takeaways

Inheritance

- Code reuse via `extends`
- `super` for parent access
- Single inheritance only

Polymorphism

- Overloading = same name, different params
- Overriding = new implementation
- Runtime flexibility

Abstract Classes

- Cannot instantiate
- Mix of abstract + concrete
- For IS-A relationships

Interfaces

- Pure contract
- Multiple implementation
- For CAN-DO capabilities

Best Practices

1. **Favor composition over inheritance** when possible
2. **Program to interfaces**, not implementations
3. **Use @Override** annotation always
4. **Keep inheritance hierarchies shallow** (max 3-4 levels)
5. **Don't use inheritance** just for code reuse
6. **Abstract classes** for related classes with shared code
7. **Interfaces** for unrelated classes with common behavior

Practice Exercises

1. Create a **Vehicle** hierarchy with Car, Motorcycle, Bicycle
2. Implement a **Shape** system with Circle, Rectangle, Triangle
3. Design a **Bank Account** system with Savings, Checking accounts
4. Create an **E-commerce** product system with Books, Electronics
5. Implement a **Zoo** management system with different animals

Challenge

For each exercise, use a combination of inheritance, abstract classes, and interfaces!

Thank You!

Questions?

PT821: Object-Oriented Programming
State University of Zanzibar (SUZA)