Method Overriding and Abstract Classes in Java

Lecture 10

Agenda Points

- What is Method Overriding?
- What are Abstract Classes?
- Practical Code Examples
- Key Differences & Use Cases

Method Overriding

What is method overriding?

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

Key Points:

- Same method signature (name, parameters) in both superclass and subclass.
- Used to achieve runtime polymorphism.
- Allows subclass to provide its specific behavior.

Rules for Method Overriding

1. The method must have the same name and parameters as the parent class.
2. The return type must be the same or a subtype (covariant return type).
3. The method in the subclass must be at least as accessible (or more) as the method in the superclass (e.g., public → protected).
4. @**Override** annotation can be used to indicate that a method is overriding.

Practical Example of Method Overriding

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

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

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();  // Runtime polymorphism
        myDog.sound();  // Outputs: Dog barks
    }
}
```

Abstract Classes

What is an Abstract Class?

An abstract class in Java is a class that cannot be instantiated, but it can have abstract methods (methods without a body) and concrete methods (with implementation).

Purpose:

To provide a base class with common functionality, while allowing subclasses to provide specific implementations of abstract methods.

Rules for Abstract Classes

1. An abstract class cannot be instantiated.
2. It can have both abstract and concrete methods.
3. Subclasses must override all abstract methods or declare themselves as abstract.
4. Abstract methods do not have a body.

Practical Example of Abstract Class

```
abstract class Vehicle {
    // Abstract method (does not have a body)
    public abstract void start();

    // Regular method
    public void stop() {
        System.out.println("Vehicle stopped");
    }
}
class Car extends Vehicle {
    @Override
    public void start() {
        System.out.println("Car started");
    }
```

Practical Example of Abstract Class

```
public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();

// Polymorphism with abstract class

        myCar.start();  // Outputs: Car started
        myCar.stop();   // Outputs: Vehicle stopped
    }
}
```

Key Differences Between Method Overriding and Abstract Classes

Feature	Method Overriding	Abstract Class
Purpose	Customize behavior of an inherited method	Define a template for subclasses
Instantiation	Can instantiate class normally	Cannot instantiate abstract class
Method Body	Method has a body	Abstract method has no body
Requirement to Override	Optional, but @Override can be used	Mandatory to override abstract methods

Use Cases

Method Overriding:

- Useful for dynamic method dispatch and polymorphism.
- Example: Customizing behavior of a framework class in a subclass.

Abstract Classes:

- Provides a blueprint for concrete subclasses.
- Example: A template for different types of vehicles, employees, or products.

Key Takeaways

Method overriding allows runtime polymorphism and subclass-specific behavior.

Abstract classes provide a way to define abstract methods that must be implemented by subclasses.

Quiz Scenario 1: Animal Shelter

You are designing a system for an animal shelter. The system needs to manage different types of animals, and each animal type has a different sound. The shelter wants to ensure that no one can instantiate an animal directly, but each specific type of animal (e.g., Dog, Cat) should have a custom sound.

Question:

1. Create an abstract class Animal with an abstract method makeSound().
2. Implement two subclasses, Dog and Cat, that extend Animal and override makeSound() to print "Dog barks" and "Cat meows", respectively.
3. In the main method, instantiate both Dog and Cat, and call their makeSound() methods.

Quiz Scenario 1: Animal Shelter

```
// Abstract class Animal
abstract class Animal {
    // Abstract method
    public abstract void makeSound();
}

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

// Cat class extending Animal
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}
```

Quiz Scenario 1: Animal Shelter

```
// Main class to test the solution
public class Main {
    public static void main(String[] args) {
        // Creating Dog and Cat objects
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        // Calling makeSound() for each animal
        myDog.makeSound();  // Outputs: Dog barks
        myCat.makeSound();  // Outputs: Cat meows
    }
}
```

Quiz Scenario 2: Vehicle System

You are developing a vehicle system for a car rental service. The service rents out cars and motorcycles. Each vehicle type should have its own implementation of starting the engine, but all vehicles should have a stop method that prints "Vehicle stopped".

Question:

1. Create an abstract class Vehicle with an abstract method startEngine() and a concrete method stop().
2. Implement two subclasses, Car and Motorcycle, that override the startEngine() method to print "Car engine started" and "Motorcycle engine started", respectively.
3. In the main method, create objects for both Car and Motorcycle, and call both startEngine() and stop() methods for each.

Quiz Scenario 2: Vehicle System

```
// Abstract class Vehicle
abstract class Vehicle {
    // Abstract method
    public abstract void startEngine();

    // Concrete method
    public void stop() {
        System.out.println("Vehicle stopped");
    }
}

// Car class extending Vehicle
class Car extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Car engine started");
    }
}

// Motorcycle class extending Vehicle
class Motorcycle extends Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Motorcycle engine started");
    }
}
```

Quiz Scenario 2: Vehicle System

```
// Main class to test the solution
public class Main {
    public static void main(String[] args) {
        // Creating Car and Motorcycle objects
        Vehicle myCar = new Car();
        Vehicle myMotorcycle = new Motorcycle();

        // Calling startEngine() and stop() for each vehicle
        myCar.startEngine();  // Outputs: Car engine started
        myCar.stop();         // Outputs: Vehicle stopped

        myMotorcycle.startEngine();  // Outputs: Motorcycle engine started
        myMotorcycle.stop();         // Outputs: Vehicle stopped
    }
```

Quiz Scenario 3: Employee Management System

You are developing an employee management system. All employees have a common salary structure, but managers have a bonus component in their salary. You need to ensure that no one can instantiate an employee directly but can create objects of subclasses like Manager and Developer.

Question:

1. Create an abstract class Employee with an abstract method calculateSalary() and a concrete method displayRole().
2. Implement a subclass Manager that overrides calculateSalary() to include a bonus and a subclass Developer that calculates a regular salary.
3. In the main method, create objects for both Manager and Developer, and call their calculateSalary() and displayRole() methods.

Quiz Scenario 4: Payment Processing System

You are designing a payment processing system where payments can be made by different methods like Credit Card and PayPal. Each payment type has its own way of processing, but they all should have a method to verify the payment.

Question:

1. Create an abstract class PaymentMethod with an abstract method processPayment() and a concrete method verifyPayment().
2. Implement two subclasses, CreditCardPayment and PayPalPayment, that override the processPayment() method to print specific messages for each payment type.
3. In the main method, create objects for both payment types and call both processPayment() and verifyPayment() methods.