Abstract Class & Interfaces

Premanand S

Assistant Professor,
School of Electronics Engineering
Vellore Institute of Technology, Chennai Campus

premanand.s@vit.ac.in

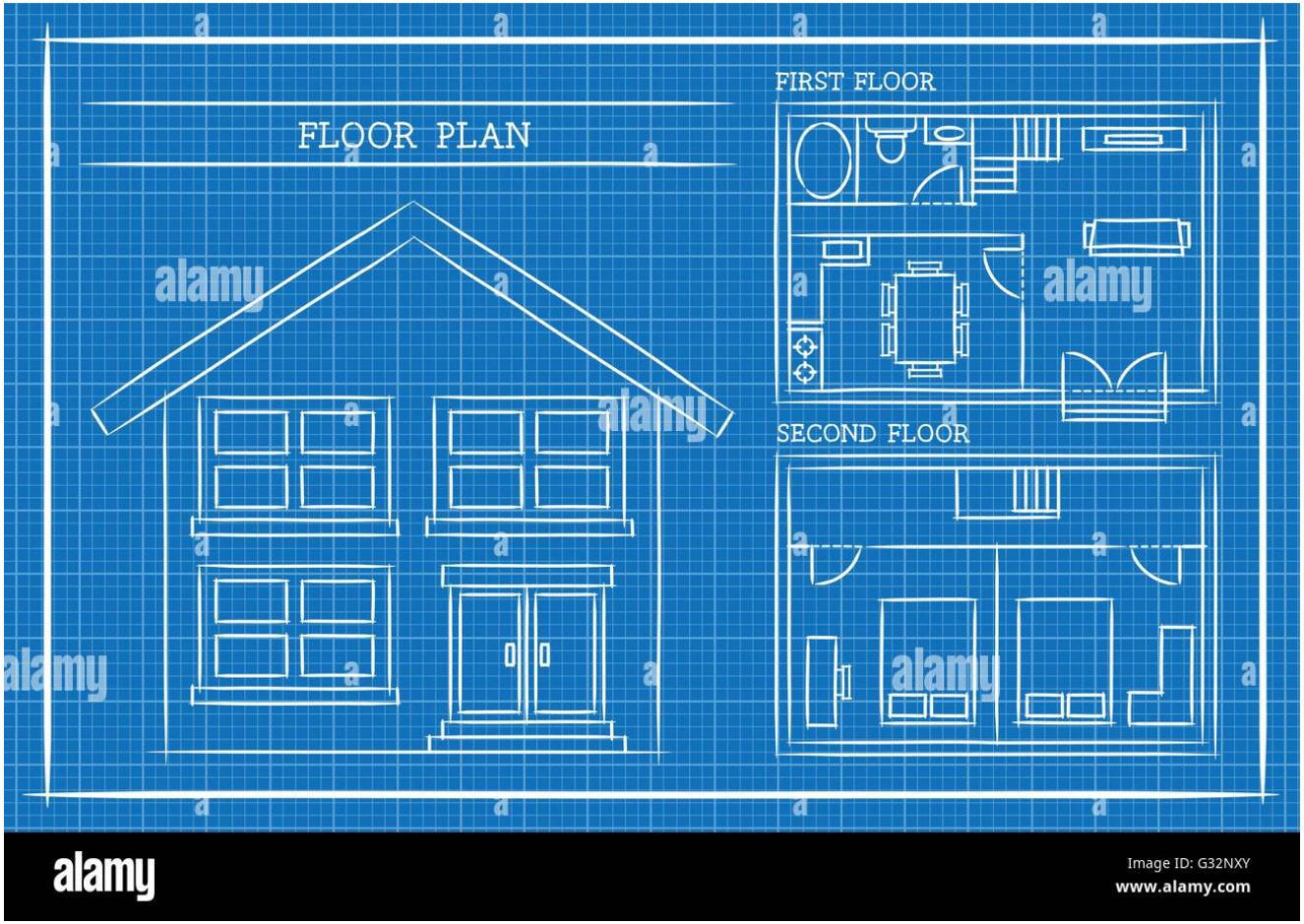
March 24, 2025

Contents - Module 4

- Inheritance
- Polymorphism
- **Abstract Class**
- **Interfaces**

Abstract Class

Understanding



What is Abstract Class?

- An abstract class in Java acts as a blueprint for other classes,
  - You can't create objects of it
  - It can have both abstract methods (without implementation) and concrete methods (with implementation).
  - Subclasses inherit the blueprint and implement the abstract methods.

Basic Syntax of Abstract Class

Example (Understanding)

```
// Abstract class
abstract class Shape {
    // Abstract method (no implementation)
    abstract void draw();

    // Concrete method (with implementation)
    void printShape() {
        System.out.println("This is a shape.");
    }
}
```

Basic Syntax of Abstract Class

Example (Understanding)

```
// Subclass implementing the abstract method
class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle.");
    }
}
```

Basic Syntax of Abstract Class

Example (Understanding)

```
// Main class
public class Main {
    public static void main(String[] args) {
        Shape myShape = new Circle();
        myShape.draw();
        myShape.printShape();
    }
}
```

Abstract Method?

- An abstract method is a method declared without a body.
- It is declared in an abstract class or an interface.
- Subclasses or implementing classes must override it to provide a definition.
- Characteristics,
  - Declared using the abstract keyword.
  - No implementation in the parent class.
  - Must be implemented by subclasses.

Basic Syntax of Abstract Method

Example (Understanding)

```
abstract class Shape {
    // Abstract method - no body
    abstract void draw();
}

abstract class RobotPart {
    // Abstract method - no body
    abstract void movePart();  // Subclasses define movement
}
```

Concrete Method?

- A concrete method is a method that has a body/implementation
- It defines behavior that can be inherited by subclasses.
- It can be either in an abstract class or a normal class.
- Characteristics,
  - Has a method body with logic.
  - Can be inherited by subclasses.
  - Provides reusable functionality.

Basic Syntax of Concrete Method

Example (Understanding)

```
abstract class Shape {
    // Concrete method - has a body
    void printShape() {
        System.out.println("This is a shape.");
    }
}

abstract class RobotPart {
    // Concrete method - has body
    void attachPart() {
        System.out.println("Attaching part to the robot.");
    }
}
```

Why Do We Need Abstract Classes?

- Enforcing a Common Contract (Partial Abstraction)
- Avoiding Code Duplication (Code Reusability)
- Flexibility and Scalability (Future-Proofing Code)
- Facilitating Polymorphism (Dynamic Method Invocation)
- Improving Code Maintainability (DRY Principle)

1. Enforcing a Common Contract (KGF Franchise)

- **Problem:** When multiple classes share similar behavior but require specific implementations for some methods, you need a way to enforce that behavior.
  - An abstract class defines abstract methods that must be implemented by subclasses.
  - It ensures that all subclasses follow a common structure while allowing flexibility in implementation.

1. Enforcing a Common Contract

Example (Understanding)

```
// Abstract class as the KFC Franchise Agreement
abstract class KFCFranchise {
    // Abstract method -
    //All outlets must serve the signature chicken
    abstract void serveChicken();

    // Concrete method - Common standards
    // (like ambiance, service quality)
    void followBrandGuidelines() {
        System.out.println("Following KFC brand guidelines:
        Hygiene, Quality, and Customer Satisfaction.");
    }
}
```

Example (Understanding)

```
class KFCChennai extends KFCFranchise {
    @Override
    void serveChicken() {
        System.out.println("KFC Chennai: Serving spicy
        chicken with extra masala! ");
    }
}
```

Example (Understanding)

```
class KFCDelhi extends KFCFranchise {
    @Override
    void serveChicken() {
        System.out.println("KFC Delhi: Serving chicken
        with butter naan combo! ");
    }
}
```

Example (Understanding)

```
class KFCMumbai extends KFCFranchise {
    @Override
    void serveChicken() {
        System.out.println("KFC Mumbai: Serving chicken
        with a tangy twist! ");
    }
}
```

Example (Understanding)

```
public class Main {
    public static void main(String[] args) {
        KFCFranchise chennaiOutlet = new KFCChennai();
        chennaiOutlet.serveChicken();
        chennaiOutlet.followBrandGuidelines();

        KFCFranchise delhiOutlet = new KFCDelhi();
        delhiOutlet.serveChicken();
        delhiOutlet.followBrandGuidelines();

        KFCFranchise mumbaiOutlet = new KFCMumbai();
        mumbaiOutlet.serveChicken();
        mumbaiOutlet.followBrandGuidelines();
    }}
```

2. Avoiding Code Duplication (Zomato Swiggy Uber)

- **Problem:** Without an abstract class, you would need to duplicate common logic across multiple subclasses.
  - Define concrete methods in the abstract class to provide shared functionality.
  - Subclasses inherit this behavior, reducing code duplication.

2. Avoiding Code Duplication

Example (Understanding)

```
abstract class FoodDelivery {
    // Abstract method (to be implemented by partners)
    abstract void deliverOrder(String order);

    // Concrete method (shared by all partners)
    void sendNotification(String order) {
        System.out.println("Notification: Your order '" +
        order + "' is on the way! ");
    }
}
```

2. Avoiding Code Duplication

Example (Understanding)

```
class Swiggy extends FoodDelivery {
    @Override
    void deliverOrder(String order) {
        System.out.println("Swiggy: Delivering " + order
        + " with fast delivery! ");
    }
}
```

2. Avoiding Code Duplication

Example (Understanding)

```
class Zomato extends FoodDelivery {
    @Override
    void deliverOrder(String order) {
        System.out.println("Zomato: Delivering " + order
        + " with special packaging! ");
    }}
```

2. Avoiding Code Duplication

Example (Understanding)

```
class UberEats extends FoodDelivery {
    @Override
    void deliverOrder(String order) {
        System.out.println("Uber Eats: Delivering " + order
        + " with route optimization!");
    }
}
```

2. Avoiding Code Duplication

Example (Understanding)

```
public class Main {
    public static void main(String[] args) {
        FoodDelivery swiggy = new Swiggy();
        swiggy.deliverOrder("Pizza");
        swiggy.sendNotification("Pizza");

        FoodDelivery zomato = new Zomato();
        zomato.deliverOrder("Burger");
        zomato.sendNotification("Burger");

        FoodDelivery uberEats = new UberEats();
        uberEats.deliverOrder("Fries");
        uberEats.sendNotification("Fries");
    }}
```

3. Flexibility and Scalability (Zomato Swiggy Uber)

- **Problem:** As applications grow, new functionality or types need to be added without modifying existing code.
  - Abstract classes make it easier to add new subclasses with minimal changes.
  - Polymorphism allows dynamically selecting the appropriate subclass at runtime.

Example (Understanding)

```
class Rapido extends FoodDelivery {
    @Override
    void deliverOrder(String order) {
        System.out.println("Rapido: Delivering " + order
        + " using electric bikes!");
    }
}
```

Example (Understanding)

```
class Dunzo extends FoodDelivery {
    @Override
    void deliverOrder(String order) {
        System.out.println("Dunzo: Delivering " + order
        + " with lightning speed! ");
    }
}
```

3. Flexibility and Scalability

Example (Understanding)

```
public class Main {
    public static void main(String[] args) {
        String[][] orders = {
            {"Pizza", "Swiggy"},
            {"Burger", "Zomato"},
            {"Fries", "UberEats"},
            {"Sandwich", "Dunzo"},
            {"Pasta", "Rapido"}
        };

        for (String[] order : orders) {
            FoodDelivery partner = getDeliveryPartner(order[1]
            partner.deliverOrder(order[0]);
            partner.sendNotification(order[0]);
        }    }
```

Example (Understanding)

```
static FoodDelivery getDeliveryPartner(String partnerName)
    switch (partnerName) {
        case "Swiggy":
            return new Swiggy();
        case "Zomato":
            return new Zomato();
        case "UberEats":
            return new UberEats();
        case "Dunzo":
            return new Dunzo();
        case "Rapido":
            return new Rapido();
        default:
            throw new IllegalArgumentException("Unknown
            delivery partner: " + partnerName);
```

4. Facilitating Polymorphism (Payment Option)

- **Problem:** When you need to treat different types of objects uniformly but execute their specific behavior dynamically at runtime.
  - Abstract classes allow creating parent class references that point to subclass objects.
  - This facilitates dynamic method invocation (method overriding)

Example (Understanding)

```
abstract class Payment {
    abstract void processPayment(double amount);
    void printReceipt(double amount) {
        System.out.println("Receipt: Payment of " +
        amount + " processed successfully.");
    }
}
```

Example (Understanding)

```
class CreditCardPayment extends Payment {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing credit
        card payment of " + amount);
    }
}
```

Example (Understanding)

```
class UpiPayment extends Payment {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing UPI
        payment of " + amount);
    }
}
```

Example (Understanding)

```
class PayPalPayment extends Payment {
    @Override
    void processPayment(double amount) {
        System.out.println("Processing PayPal
        payment of " + amount);
    }
}
```

Example (Understanding)

```
public class PaymentSystem {
    public static void main(String[] args) {
        Payment payment1 = new CreditCardPayment();
        Payment payment2 = new UpiPayment();
        Payment payment3 = new PayPalPayment();

        payment1.processPayment(1000.0);
        payment1.printReceipt(1000.0);

        payment2.processPayment(500.0);
        payment2.printReceipt(500.0);

        payment3.processPayment(750.0);
        payment3.printReceipt(750.0);
    }}
```

5. Improving Code Maintainability (DRY Principle) (Coffee shop)

- **Problem:** If common functionality is duplicated across multiple classes, any change requires modifying multiple places.
  - Define shared logic in an abstract class once and allow subclasses to inherit it.
  - Apply the DRY (Don't Repeat Yourself) principle to minimize maintenance effort.

5. Improving Code Maintainability (DRY Principle)

Example (Understanding)

```
// Parent abstract class
abstract class CoffeeHouse {
    // Abstract method to be implemented by outlets
    abstract void serveDrink();

    // Concrete method to print receipt (same for all outlets)
    void printReceipt(String drink) {
        System.out.println("Receipt: Your " + drink
        + " is ready! ");
    }
}
```

5. Improving Code Maintainability (DRY Principle)

Example (Understanding)

```
class NewYorkOutlet extends CoffeeHouse {
    @Override
    void serveDrink() {
        System.out.println("Serving Espresso in New York!");
    }
}
```

5. Improving Code Maintainability (DRY Principle)

Example (Understanding)

```
class ParisOutlet extends CoffeeHouse {
    @Override
    void serveDrink() {
        System.out.println("Serving Cappuccino in Paris!");
    }
}
```

5. Improving Code Maintainability (DRY Principle)

Example (Understanding)

```
class MumbaiOutlet extends CoffeeHouse {
    @Override
    void serveDrink() {
        System.out.println("Serving Masala Chai in Mumbai! ");
    }
}
```

5. Improving Code Maintainability (DRY Principle)

Example (Understanding)

```
public class Main {
    public static void main(String[] args)
        CoffeeHouse nyOutlet = new NewYorkOutlet();
        nyOutlet.serveDrink();
        nyOutlet.printReceipt("Espresso");

        CoffeeHouse parisOutlet = new ParisOutlet();
        parisOutlet.serveDrink();
        parisOutlet.printReceipt("Cappuccino");

        CoffeeHouse mumbaiOutlet = new MumbaiOutlet();
        mumbaiOutlet.serveDrink();
        mumbaiOutlet.printReceipt("Masala Chai");
    }
}
```

When Should You Use Abstract Classes?

- When you want to enforce a contract that subclasses must follow.
- When you need to share common functionality across multiple classes.
- When you expect future extensions of your class hierarchy.
- When you want to use polymorphism to treat multiple related types uniformly.

Can an Abstract Class Have a Constructor?

Example (Understanding)

```
// Abstract parent class
abstract class CarBlueprint {
    String modelName;
    String registrationNumber;

    // Constructor for common initialization logic
    CarBlueprint(String modelName) {
        this.modelName = modelName;
        this.registrationNumber = generateRegistrationNumber();
        System.out.println("Car " + modelName + "
        registered with number: " + registrationNumber);
    }
```

Can an Abstract Class Have a Constructor?

Example (Understanding)

```
// Abstract method to define engine specs (to be implement
abstract void defineEngineSpecs();

// Common method for all cars
void showCarDetails() {
    System.out.println("Car Model: " + modelName);
    System.out.println("Registration Number: "
    + registrationNumber);
}

// Private method to simulate registration number generati
private String generateRegistrationNumber() {
    return "REG-" + (int)(Math.random() * 10000);
}}
```

Can an Abstract Class Have a Constructor?

Example (Understanding)

```
class CarModelA extends CarBlueprint {
    // Constructor for CarModelA
    CarModelA() {
        super("Model A");
    }

    @Override
    void defineEngineSpecs() {
        System.out.println("Engine: 2.0L
        Turbocharged Engine for Model A.");
    }
}
```

Can an Abstract Class Have a Constructor?

Example (Understanding)

```
class CarModelB extends CarBlueprint {
    // Constructor for CarModelB
    CarModelB() {
        super("Model B");
    }

    @Override
    void defineEngineSpecs() {
        System.out.println("Engine: 3.0L
        V6 Engine for Model B.");
    }
}
```

Can an Abstract Class Have a Constructor?

Example (Understanding)

```
public class Main {
    public static void main(String[] args) {

        CarBlueprint modelA = new CarModelA();
        modelA.defineEngineSpecs();
        modelA.showCarDetails();

        CarBlueprint modelB = new CarModelB();
        modelB.defineEngineSpecs();
        modelB.showCarDetails();
    }
}
```

Final Method in Abstract Class?

Example (Understanding)

```
// Abstract parent class
abstract class Restaurant {
    // Abstract method to be implemented by franchises
    abstract void prepareSpecialDish();

    // Final method - cannot be overridden
    final void serveSignatureDish() {
        System.out.println("Serving Gourmet Heaven's
        Signature Dish: Spaghetti Carbonara");
    }

    // Common method that can be modified if necessary
    void printBill(double amount) {
        System.out.println("Total Bill: " + amount);
    }}
```

Final Method in Abstract Class?

Example (Understanding)

```
class OutletA extends Restaurant {
    @Override
    void prepareSpecialDish() {
        System.out.println("OutletA: Preparing
        Margherita Pizza ");
    }
}
```

Final Method in Abstract Class?

Example (Understanding)

```
class OutletB extends Restaurant {
    @Override
    void prepareSpecialDish() {
        System.out.println("OutletB: Preparing
        Butter Chicken");
    }
}
```

Final Method in Abstract Class?

Example (Understanding)

```
public class Main {
    public static void main(String[] args) {
        Restaurant outletA = new OutletA();
        outletA.serveSignatureDish();
        outletA.prepareSpecialDish();

        Restaurant outletB = new OutletB();
        outletB.serveSignatureDish();
        outletB.prepareSpecialDish();
    }
}
```

Problem Practice 1

- **Problem:** A drawing application needs to handle multiple types of shapes:
  - Circle, Rectangle, and Triangle.
  - Create an abstract class Shape with:
    - An abstract method calculateArea() and
    - A concrete method displayShape().
- **Task:** Implement subclasses Circle, Rectangle, and Triangle with specific formulas to calculate area.
- **Task:** Create a method that accepts a list of shapes and calculates the area dynamically.
- **Bonus Challenge:** How would you add a Square without modifying existing code?

Solution Practice 1

Example (Understanding)

```
import java.util.ArrayList;
import java.util.List;
abstract class Shape {
    abstract double calculateArea();

    void displayShape(String shapeName) {
        System.out.println("Drawing a " + shapeName);
    }
}
class Circle extends Shape {
    private double radius;

    Circle(double radius) {
        this.radius = radius;
    }
    @Override
```

Solution Practice 1

Example (Understanding)

```
    @Override
    void displayShape(String shapeName) {
        super.displayShape(shapeName);
        System.out.println("Radius: " + radius);
    } }

class Rectangle extends Shape {
    private double length, width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    double calculateArea() {
```

Solution Practice 1

Example (Understanding)

```
    @Override
    void displayShape(String shapeName) {
        super.displayShape(shapeName);
        System.out.println("Length: " + length + ", Width: " +
    } }

class Triangle extends Shape {
    private double base, height;

    Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    @Override
    double calculateArea() {
```

Solution Practice 1

Example (Understanding)

```
    @Override
    void displayShape(String shapeName) {
        super.displayShape(shapeName);
        System.out.println("Base: " + base + ", Height: " + he
    } }
public class Main {
    public static void main(String[] args) {
        List<Shape> shapes = new ArrayList<>();
        shapes.add(new Circle(5.0));
        shapes.add(new Rectangle(4.0, 6.0));
        shapes.add(new Triangle(3.0, 8.0));
        calculateAndDisplayAreas(shapes);
    }
```

Solution Practice 1

Example (Understanding)

```
    public static void calculateAndDisplayAreas(List<Shape> sh
        for (Shape shape : shapes) {
            shape.displayShape(shape.getClass().getSimpleName(
            double area = shape.calculateArea();
            System.out.println("Area: " + area);
        }
    }
}
```

Interfaces

What is an Interface?

- An interface in Java is like a contract that defines a set of rules that a class must follow.
  - It contains only abstract methods (Java 7).
  - It can also have default methods and static methods (Java 8+).
  - A class that implements an interface agrees to provide the behavior defined by that interface.

Basic Syntax of Interface

Example (Understanding)

```
interface Animal {
    void eat();  // Abstract method
    void sleep(); // Abstract method
}

class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog eats bones.");
    }

    @Override
    public void sleep() {
        System.out.println("Dog sleeps in a kennel.");
    }}
```

Basic Syntax of Interface

Example (Understanding)

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

Why Do We Use Interfaces?

- To Achieve Full Abstraction
- To Support Multiple Inheritance
- To Ensure Loose Coupling

1. To Achieve Full Abstraction

Example (Understanding)

```
interface Animal {
    void sound();  // Abstract method for sound
    void move();   // Abstract method for movement
}

class Dog implements Animal {
    // Implementation of sound method
    public void sound() {
        System.out.println("Dog barks.");
    }

    public void move() {
        System.out.println("Dog runs.");
    }
}
```

1. To Achieve Full Abstraction

Example (Understanding)

```
class Bird implements Animal {
    public void sound() {
        System.out.println("Bird chirps.");
    }

    public void move() {
        System.out.println("Bird flies.");
    }
}
```

1. To Achieve Full Abstraction

Example (Understanding)

```
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.sound();  // Output: Dog barks.
        dog.move();   // Output: Dog runs.

        System.out.println("----------------------");

        Animal bird = new Bird();
        bird.sound();  // Output: Bird chirps.
        bird.move();   // Output: Bird flies.
    }
}
```

2. To Support Multiple Inheritance

Example (Understanding)

```
interface Printer {
    void printDocument();}

interface Scanner {
    void scanDocument();}

class MultiFunctionDevice implements Printer, Scanner {
    public void printDocument() {
        System.out.println("Printing document...");
    }

    public void scanDocument() {
        System.out.println("Scanning document...");
    }}
```

2. To Support Multiple Inheritance

Example (Understanding)

```
public class Main {
    public static void main(String[] args) {
        MultiFunctionDevice mfd = new MultiFunctionDevice();
        mfd.printDocument();
        mfd.scanDocument();
    }
}
```

3. To Ensure Loose Coupling

Example (Understanding)

```
interface Payment {
    void processPayment(double amount);
}

class CreditCardPayment implements Payment {
    public void processPayment(double amount) {
        System.out.println("Payment of " + amount
        + " made using Credit Card.");
    }
}

class UpiPayment implements Payment {
    public void processPayment(double amount) {
        System.out.println("Payment of " + amount
        + " made using UPI.");
    }
```

3. To Ensure Loose Coupling

Example (Understanding)

```
public class Main {
    public static void process(Payment payment, double amount)
    {
        payment.processPayment(amount);
    }

    public static void main(String[] args) {
        Payment ccPayment = new CreditCardPayment();
        Payment upiPayment = new UpiPayment();

        process(ccPayment, 1000.0);
        process(upiPayment, 500.0);
    }
}
```

Practice Question

- Design a Media Player that can:
  - Play MP3 files.
  - Play Video files.
- Define an interface MediaPlayer with play() method.
- Create classes MP3Player and VideoPlayer that implement MediaPlayer.
- Add a default method stop() to the interface.

Solution Practice 1

Example (Understanding)

```
interface MediaPlayer {
    void play(String filename);

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

class MP3Player implements MediaPlayer {
    @Override
    public void play(String filename) {
        System.out.println("Playing MP3 file: " + filename);
    }}
```

Solution Practice 1

Example (Understanding)

```
class VideoPlayer implements MediaPlayer {
    @Override
    public void play(String filename) {
        System.out.println("Playing video file: " + filename);
    }}

public class Main {
    public static void main(String[] args) {
        MediaPlayer mp3Player = new MP3Player();
        mp3Player.play("song.mp3");
        mp3Player.stop();
        MediaPlayer videoPlayer = new VideoPlayer();
        videoPlayer.play("movie.mp4");
        videoPlayer.stop();          }}
```