**What is an Interface?**

In Java, an interface is a type that defines a contract for classes that implement it.

It specifies a set of abstract methods that implementing classes must provide, as well as constants (static final fields).

Interfaces allow you to define a common set of methods that multiple classes can share, promoting code reusability and supporting a form of multiple inheritance in Java.

Here are some key points about Java interfaces:

1. **Declaration**: An interface is declared using the **interface** keyword, followed by the interface's name and a block of abstract method declarations and constant fields.

For example:

public interface MyInterface {

void someMethod();

int anotherMethod();

String INTERFACE\_CONSTANT = "This is a constant";

}

1. **Abstract** **Methods**:
   1. Methods declared in an interface do not have method bodies (i.e., they are abstract methods) and do not contain any implementation details.
   2. Classes that implement the interface must provide concrete implementations for all the methods declared in the interface.
2. **Multiple** **Inheritance**:
   1. Unlike classes, which can only extend a single class, a Java class can implement multiple interfaces.
   2. This allows a class to inherit behavior and method signatures from multiple sources.
3. **Access** **Modifiers**:
   1. By default, methods in an interface are implicitly **public** and **abstract**, and fields are implicitly **public**, **static**, and **final**.
   2. You can explicitly specify access modifiers, but you can't use **private** or **protected** for methods.
4. **Implementing** **an** **Interface**:
   1. To implement an interface, a class uses the **implements** keyword, and it must provide concrete implementations for all the methods declared in the interface.

For example:

public class MyClass implements MyInterface {

@Override

public void someMethod() {

// Implementation of someMethod

}

@Override

public int anotherMethod() {

// Implementation of anotherMethod

return 42;

}

}

1. **Interface** **Inheritance**:
   1. Interfaces can extend other interfaces using the **extends** keyword.
   2. This allows you to build more complex hierarchies of interfaces.
2. **Polymorphism**:
   1. Interfaces are commonly used to achieve polymorphism, where you can interact with objects of different classes through a common interface.
   2. This promotes flexibility and code extensibility.
3. **Constants**:
   1. Interfaces can contain constants, which are **public**, **static**, and **final** by default.
   2. They can be accessed using the interface name (e.g., **MyInterface.INTERFACE\_CONSTANT**).

Java interfaces are a fundamental part of the language and play a crucial role in achieving abstraction, code organization, and reusability in object-oriented programming.

They are widely used in the Java standard library and in Java applications to define contracts for various classes and ensure consistency and interoperability.

**Where we will be using an interface in java?**

Interfaces in Java are used in various scenarios to define contracts that classes must adhere to.

They are a powerful tool for achieving abstraction, code organization, and reusability.

Here are some common use cases for interfaces in Java:

1. **Defining APIs**: When you want to define a common set of methods that classes should implement, use an interface to specify the API contract. This is particularly useful in libraries and frameworks where you want to provide a consistent interface for users to implement.
2. **Polymorphism**: Interfaces enable polymorphism, allowing you to write code that works with objects of different classes as long as they implement the same interface. This promotes flexibility and code extensibility.
3. **Multiple Inheritance**: Since Java doesn't support multiple inheritance of classes (i.e., a class can't extend multiple classes), interfaces provide a way to inherit behavior from multiple sources. A class can implement multiple interfaces to gain functionality from each of them.
4. **Java Collections Framework**: Many classes in the Java Collections Framework (e.g., ArrayList, HashSet, TreeMap) implement various interfaces, such as List, Set, and Map. This allows for consistent usage of collections, regardless of the specific implementation.
5. **Event Handling**: Interfaces are commonly used in event handling frameworks, where multiple event listeners can implement a common interface to handle events in a standardized way.
6. **Plugin Architecture**: In applications with a plugin architecture, interfaces define the contract that plugins must adhere to. This allows developers to create plugins that are easily integrated into the main application.
7. **Database Access**: In Java Database Connectivity (JDBC), interfaces like Connection and Statement define the contract for interacting with databases. This enables different database vendors to provide their own implementations while adhering to the standard JDBC interface.
8. **Dependency Injection**: Many dependency injection frameworks use interfaces to define the services or components that can be injected into other classes. This promotes decoupling and makes it easy to switch implementations.
9. **Testing and Mocking**: Interfaces are valuable in unit testing. You can create mock implementations of interfaces to simulate behavior for testing without relying on the actual implementations.
10. **Adapters and Wrappers**: Interfaces can be used to create adapter and wrapper classes that allow existing classes to work with new functionality. For example, the Java.io package uses interfaces to define the contract for various I/O operations, making it possible to use a consistent API for file, network, and other types of I/O.
11. **Callback Mechanisms**: Interfaces are used in callback mechanisms, where a class provides a callback interface that other classes can implement to receive notifications or handle events.

In summary, interfaces in Java are used in a wide range of situations where you want to define a common contract for classes, achieve polymorphism, promote code reusability, and ensure interoperability between different components in a system. They are a fundamental part of Java's object-oriented programming model and are instrumental in creating well-structured, extensible, and maintainable code.

**How to define an interface?**

To define an interface in Java, you use the **interface** keyword followed by the name of the interface, a block of method declarations, and potentially constant fields.

Here's the basic syntax for defining an interface:

public interface MyInterface {  
 // Abstract method declarations

void method1();

int method2(String parameter);

// Constant field (implicitly public, static, final)

String INTERFACE\_CONSTANT = "This is a constant";

}

Let's break down the components of this syntax:

1. **public**:
   1. You can specify an access modifier for the interface.
   2. In this example, it's **public**, which means the interface is accessible from other classes in other packages.
   3. You can also use **package-private** access (no modifier) for an interface, but **public** is the most common choice.
2. **interface**:
   1. This keyword is used to declare that you're defining an interface.
3. **MyInterface**:
   1. This is the name of the interface.
   2. You should follow Java's naming conventions for classes, which usually involve using CamelCase for class and interface names.
4. **Method** **Declarations**:
   1. Inside the interface, you declare abstract methods.
   2. These method declarations specify the method name, return type, and parameters (if any), but they don't contain method bodies (i.e., no code implementation).
   3. Classes that implement the interface must provide concrete implementations for these methods.
5. **Constant** **Fields**:
   1. You can include constant fields in an interface.
   2. By default, they are **public**, **static**, and **final**, so you don't need to specify these modifiers explicitly.
   3. Constants are often written in ALL\_CAPS with underscores separating words (e.g., **INTERFACE\_CONSTANT**).

Here's a simple example of an interface with a single method:

public interface Printable {

void print();

}

In this example, any class that implements the **Printable** interface must provide a concrete implementation of the **print** method.

To implement an interface in a class, you use the **implements** keyword, as shown below:

public class MyPrinter implements Printable {

@Override

public void print() {

// Implementation of the print method

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

}

}

In this example, the **MyPrinter** class implements the **Printable** interface and provides an implementation of the **print** method.

**Advantages of interfaces**

Interfaces in Java offer several advantages that contribute to better code organization, maintainability, and flexibility in object-oriented programming.

Here are some of the key advantages of using interfaces:

1. **Abstraction**:
   1. Interfaces allow you to define a contract for classes to adhere to without specifying the implementation details.
   2. This promotes a high level of abstraction, allowing you to focus on what a class should do rather than how it does it.
2. **Multiple Inheritance**:
   1. Java supports multiple interface inheritance, which means a class can implement multiple interfaces.
   2. This is in contrast to inheriting from multiple classes, which Java does not allow.
   3. Multiple inheritance through interfaces allows for code reuse and flexibility in class design.
3. **Polymorphism**:
   1. Interfaces enable polymorphism, which means you can write code that works with objects of different classes as long as they implement the same interface.
   2. This promotes code flexibility and extensibility.
4. **Code Reusability**:
   1. By defining a common set of methods in an interface, you can create multiple classes that implement the same interface.
   2. This reusability reduces code duplication and promotes a more modular and maintainable codebase.
5. **Loose Coupling**:
   1. Interfaces promote loose coupling between components in a system.
   2. Classes that depend on interfaces only need to know the contract defined by the interface, rather than the concrete implementation details of the implementing classes.
   3. This enhances code maintainability and allows for easier component replacement.
6. **Testing and Mocking**:
   1. Interfaces are valuable in unit testing, as they allow you to create mock implementations of interfaces to simulate behavior for testing without relying on the actual implementations.
   2. This is crucial for writing effective unit tests.
7. **API Design**:
   1. When designing libraries or frameworks, interfaces define the public API contract, ensuring that users of your code adhere to a specified structure.
   2. This enhances code consistency and reliability in large projects.
8. **Plugin Architecture**:
   1. In applications that support plugins or extensions, interfaces define the contract that plugins must adhere to.
   2. This allows for the dynamic loading of plugins without requiring modifications to the core application.
9. **Simplifies Code Maintenance**:
   1. Interfaces make it easier to maintain and evolve your code.
   2. When you need to change the behavior of a class, you can do so without affecting classes that rely on its interface, provided you don't change the interface itself.
10. **Promotes Design by Contract**:
    1. Interfaces help establish a clear "contract" between classes, specifying what a class should provide and what other classes can expect.
    2. This makes it easier to reason about code behavior and interactions.
11. **Decoupling from Implementation**:
    1. Interfaces decouple the code that uses a service or functionality from the code that provides that service or functionality.
    2. This separation allows for more flexible design and future changes.
12. **Enforces Consistency**:
    1. By defining a common set of methods in an interface, you ensure that implementing classes adhere to a specific structure and behavior.
    2. This helps in maintaining consistency across the codebase.

In summary, interfaces in Java provide a powerful mechanism for defining contracts, achieving code reusability, supporting polymorphism, and promoting loose coupling.

They are fundamental to building maintainable, extensible, and well-organized software systems, making them an essential part of Java's object-oriented programming paradigm.

**Examples**

1. **Shapes**

Here's an example of a Java program that uses interfaces to demonstrate a simple geometric shape hierarchy with interfaces for common shape properties.

This program defines an interface for geometric shapes and two classes that implement the interface for specific shapes (Circle and Rectangle).

// Define an interface for geometric shapes

public interface **Shape** {

double getArea();

double getPerimeter();

}

// Implement the Shape interface for a circle

public class **Circle implements Shape** {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

@Override

public double getArea() {

return Math.PI \* radius \* radius;

}

@Override

public double getPerimeter() {

return 2 \* Math.PI \* radius;

}

}

// Implement the Shape interface for a rectangle

public class Rectangle **implements** Shape {

private double width;

private double height;

public Rectangle(double width, double height) {

this.width = width;

this.height = height;

}

@Override

public double getArea() {

return width \* height;

}

@Override

public double getPerimeter() {

return 2 \* (width + height);

}

}

public class ShapeDemo {

public static void main(String[] args) {

// Create instances of Circle and Rectangle

Circle circle = new Circle(5.0);

Rectangle rectangle = new Rectangle(4.0, 6.0);

// Calculate and display properties of the shapes

System.out.println("Circle:");

System.out.println("Area: " + circle.getArea());

System.out.println("Perimeter: " + circle.getPerimeter());

System.out.println("\nRectangle:");

System.out.println("Area: " + rectangle.getArea());

System.out.println("Perimeter: " + rectangle.getPerimeter());

}

}

In this program:

1. We define an interface **Shape** with two methods: **getArea()** and **getPerimeter()**.
2. These methods are used to calculate the area and perimeter of a shape.
3. We create two classes, **Circle** and **Rectangle**, that implement the **Shape** interface.
4. Each class provides its own implementation of the **getArea()** and **getPerimeter()** methods.
5. In the **ShapeDemo** class, we create instances of **Circle** and **Rectangle** and demonstrate how to calculate and display the area and perimeter of these shapes.

This program showcases how interfaces can be used to define a common contract (the **Shape** interface) that multiple classes (in this case, **Circle** and **Rectangle**) must adhere to.

It allows for polymorphic behavior and a consistent way to work with different geometric shapes.

1. **Accounts**

Here's a more extensive Java program that uses interfaces and classes to model a simple banking system.

The program defines interfaces for bank accounts and demonstrates the concept of inheritance and polymorphism in a banking context:

// Define an interface for a bank account

public interface **BankAccount** {

void deposit(double amount);

void withdraw(double amount);

double getBalance();

void displayAccountInfo();

}

// Implement the BankAccount interface for a savings account

public class **SavingsAccount** implements **BankAccount** {

private double balance;

private double interestRate;

public SavingsAccount(double initialBalance, double interestRate) {

this.balance = initialBalance;

this.interestRate = interestRate;

}

@Override

public void deposit(double amount) {

balance += amount;

}

@Override

public void withdraw(double amount) {

if (balance >= amount) {

balance -= amount;

} else {

System.out.println("Insufficient funds.");

}

}

@Override

public double getBalance() {

return balance;

}

@Override

public void displayAccountInfo() {

System.out.println("Savings Account Balance: $" + balance);

}

public void applyInterest() {

balance += balance \* interestRate;

}

}

// Implement the BankAccount interface for a checking account

public class **CheckingAccount** implements **BankAccount** {

private double balance;

private double overdraftLimit;

public CheckingAccount(double initialBalance, double overdraftLimit) {

this.balance = initialBalance;

this.overdraftLimit = overdraftLimit;

}

@Override

public void deposit(double amount) {

balance += amount;

}

@Override

public void withdraw(double amount) {

if (balance - amount >= -overdraftLimit) {

balance -= amount;

} else {

System.out.println("Exceeded overdraft limit.");

}

}

@Override

public double getBalance() {

return balance;

}

@Override

public void displayAccountInfo() {

System.out.println("Checking Account Balance: $" + balance);

}

}

public class BankDemo {

public static void main(String[] args) {

SavingsAccount savingsAccount = new SavingsAccount(1000.0, 0.05);

CheckingAccount checkingAccount = new CheckingAccount(500.0, 100.0);

// Deposit and withdraw from savings account

savingsAccount.deposit(200.0);

savingsAccount.withdraw(50.0);

// Deposit and withdraw from checking account

checkingAccount.deposit(300.0);

checkingAccount.withdraw(700.0);

// Apply interest to the savings account

savingsAccount.applyInterest();

// Display account information

savingsAccount.displayAccountInfo();

checkingAccount.displayAccountInfo();

}

}

In this program:

1. We define the **BankAccount** interface, which includes methods for depositing, withdrawing, getting the balance, and displaying account information.
2. We create two classes, **SavingsAccount** and **CheckingAccount**, which implement the **BankAccount** interface.
3. Each class provides its own implementation of the methods.
4. In the **BankDemo** class, we create instances of both **SavingsAccount** and **CheckingAccount** and perform various banking operations, such as deposits and withdrawals.
5. We also demonstrate the use of the **applyInterest** method specific to the **SavingsAccount** class.

This program illustrates how interfaces can be used to define a common contract for different types of bank accounts while allowing for specialized behavior within each account type.

It demonstrates inheritance, polymorphism, and encapsulation in the context of a simple banking system.

1. **Products**

Here's a more complex Java program that uses interfaces and classes to model a basic e-commerce system.

This program defines interfaces for products and users, as well as classes for various types of products and users, demonstrating how interfaces can be used to organize a complex system:

// Define an interface for products

public interface **Product** {

String getName();

double getPrice();

}

// Implement the Product interface for different product types

public class **Book** implements **Product** {

private String title;

private double price;

public Book(String title, double price) {

this.title = title;

this.price = price;

}

@Override

public String getName() {

return title;

}

@Override

public double getPrice() {

return price;

}

}

public class **Electronic** implements **Product** {

private String name;

private double price;

public Electronic(String name, double price) {

this.name = name;

this.price = price;

}

@Override

public String getName() {

return name;

}

@Override

public double getPrice() {

return price;

}

}

// Define an interface for users

public interface **User** {

String getUsername();

void purchase(Product product);

}

// Implement the User interface for different user roles

public class **Customer** implements **User** {

private String username;

private double balance;

public Customer(String username, double initialBalance) {

this.username = username;

this.balance = initialBalance;

}

@Override

public String getUsername() {

return username;

}

@Override

public void purchase(Product product) {

if (balance >= product.getPrice()) {

System.out.println(username + " purchased " + product.getName() + " for $" + product.getPrice());

balance -= product.getPrice();

} else {

System.out.println("Insufficient balance for the purchase.");

}

}

}

public class **Admin** implements **User** {

private String username;

public Admin(String username) {

this.username = username;

}

@Override

public String getUsername() {

return username;

}

@Override

public void purchase(Product product) {

System.out.println("Admin " + username + " made a purchase for " + product.getName() + " (no cost)");

}

}

public class ECommerceDemo {

public static void main(String[] args) {

Product book = new Book("Java Programming", 39.99);

Product laptop = new Electronic("Laptop", 799.99);

User customer = new Customer("john\_doe", 1000.0);

User admin = new Admin("admin\_user");

customer.purchase(book);

customer.purchase(laptop);

admin.purchase(laptop);

}

}

In this program:

1. We define the **Product** interface, which includes methods to get the name and price of a product. We also implement this interface for two product types: **Book** and **Electronic**.
2. We define the **User** interface, which includes methods to get the username and make a purchase. We implement this interface for two user roles: **Customer** and **Admin**.
3. In the **ECommerceDemo** class, we create instances of various products and users. We simulate purchases by customers and administrators, demonstrating how different user roles interact with products.
4. Customers have a balance and can make purchases, while administrators can make purchases without a cost.

This program illustrates how interfaces can be used to define common behaviors for different types of products and users in an e-commerce system. It demonstrates how polymorphism allows us to treat different user roles uniformly, despite differences in their purchase behavior.

1. **Audio Players**

Here's an example of a Java program where a class inherits from more than one interface. In this case, we'll create a simple music player that implements multiple interfaces for different functionalities:

// Define an interface for playing audio

public interface **AudioPlayer** {

void play();

void pause();

void stop();

}

// Define an interface for managing playlists

public interface **PlaylistManager** {

void createPlaylist(String playlistName);

void addSongToPlaylist(String playlistName, String songName);

void playPlaylist(String playlistName);

}

// Implement the AudioPlayer and PlaylistManager interfaces in a MusicPlayer class

public class **MusicPlayer** implements **AudioPlayer**, **PlaylistManager** {

private boolean isPlaying;

private String currentSong;

private String currentPlaylist;

@Override

public void play() {

if (!isPlaying) {

System.out.println("Playing: " + currentSong);

isPlaying = true;

}

}

@Override

public void pause() {

if (isPlaying) {

System.out.println("Paused: " + currentSong);

isPlaying = false;

}

}

@Override

public void stop() {

if (isPlaying) {

System.out.println("Stopped: " + currentSong);

isPlaying = false;

}

}

@Override

public void createPlaylist(String playlistName) {

System.out.println("Created a new playlist: " + playlistName);

}

@Override

public void addSongToPlaylist(String playlistName, String songName) {

System.out.println("Added " + songName + " to playlist: " + playlistName);

}

@Override

public void playPlaylist(String playlistName) {

currentPlaylist = playlistName;

System.out.println("Now playing playlist: " + playlistName);

}

public void loadSong(String songName) {

currentSong = songName;

}

}

public class **MusicPlayerDemo** {

public static void main(String[] args) {

MusicPlayer player = new MusicPlayer();

player.createPlaylist("My Favorites");

player.addSongToPlaylist("My Favorites", "Song 1");

player.addSongToPlaylist("My Favorites", "Song 2");

player.loadSong("Song 1");

player.play();

player.loadSong("Song 2");

player.play();

player.pause();

player.playPlaylist("My Favorites");

player.stop();

}

}

**Example 05: Vehicle Rental System**

**Step 1: Define Interfaces**

// Interface for all vehicles

public interface **Vehicle** {

void start();

void stop();

void accelerate();

void brake();

}

// Interface for maintenance services

public interface **Maintenance** {

void service();

}

**Step 2: Create Base Classes**

// Base class for all automobiles

public abstract class **Automobile** implements **Vehicle** {

@Override

public void accelerate() {

System.out.println("Automobile is accelerating.");

}

@Override

public void brake() {

System.out.println("Automobile is braking.");

}

}

// Base class for all motorcycles

public abstract class **Motorcycle** implements **Vehicle** {

@Override

public void accelerate() {

System.out.println("Motorcycle is accelerating.");

}

@Override

public void brake() {

System.out.println("Motorcycle is braking.");

}

}

**Step 3: Create Concrete Classes**

// Concrete class for Car

public class **Car** extends **Automobile** implements **Maintenance** {

@Override

public void start() {

System.out.println("Car is starting.");

}

@Override

public void stop() {

System.out.println("Car is stopping.");

}

@Override

public void service() {

System.out.println("Car is being serviced.");

}

void park() {

System.out.println("Car is parked.");

}

}

// Concrete class for Truck

public class **Truck** extends **Automobile** implements **Maintenance** {

@Override

public void start() {

System.out.println("Truck is starting.");

}

@Override

public void stop() {

System.out.println("Truck is stopping.");

}

@Override

public void service() {

System.out.println("Truck is being serviced.");

}

void loadCargo() {

System.out.println("Truck is loading cargo.");

}

}

// Concrete class for Motorcycle

public class **SportsBike** extends **Motorcycle** {

@Override

public void start() {

System.out.println("Sports Bike is starting.");

}

@Override

public void stop() {

System.out.println("Sports Bike is stopping.");

}

void wheelie() {

System.out.println("Sports Bike is doing a wheelie.");

}

}

**Step 4: Main Application**

// Main application to demonstrate vehicle rental and service

public class **VehicleRentalSystem** {

public static void main(String[] args) {

// Rent a car

Car myCar = new Car();

rentAndOperateVehicle(myCar);

// Rent a truck

Truck myTruck = new Truck();

rentAndOperateVehicle(myTruck);

// Rent a sports bike

SportsBike myBike = new SportsBike();

rentAndOperateVehicle(myBike);

}

// Method to rent and operate any vehicle

public static void rentAndOperateVehicle(Vehicle vehicle) {

// Rent the vehicle

System.out.println("Renting the vehicle...");

// Operate the vehicle

vehicle.start();

vehicle.accelerate();

vehicle.brake();

vehicle.stop();

// Service the vehicle if it implements Maintenance

if (vehicle instanceof Maintenance) {

Maintenance maintenanceVehicle = (Maintenance) vehicle;

maintenanceVehicle.service();

}

// Specific actions based on vehicle type

if (vehicle instanceof Car) {

Car car = (Car) vehicle;

car.park();

}

else if (vehicle instanceof Truck) {

Truck truck = (Truck) vehicle;

truck.loadCargo();

}

else if (vehicle instanceof SportsBike) {

SportsBike bike = (SportsBike) vehicle;

bike.wheelie();

}

System.out.println("Returning the vehicle...");

}

}

**Explanation:**

* **Interfaces (Vehicle and Maintenance)**: Define behaviors (start(), stop(), accelerate(), brake(), service()) that classes must implement.
* **Abstract Classes (Automobile and Motorcycle)**: Provide common behavior (accelerate() and brake()) that concrete classes can inherit.
* **Concrete Classes (Car, Truck, SportsBike)**:
  + Implement interfaces (Vehicle and Maintenance) and inherit from appropriate base classes (Automobile for cars and trucks, Motorcycle for sports bikes).
  + Implement specific behaviors (start(), stop(), service(), park(), loadCargo(), wheelie()).
* **Main Application (VehicleRentalSystem)**:
  + Demonstrates renting and operating different types of vehicles (Car, Truck, SportsBike) using polymorphism.
  + Calls rentAndOperateVehicle() method to handle renting, operating, servicing, and returning vehicles, demonstrating dynamic method invocation based on the actual object type.

This example showcases a more comprehensive program using multiple interfaces, inheritance, and polymorphism in Java, simulating a vehicle rental system with various types of vehicles and their specific functionalities.

**Example 07: University Management System**

Let's create a more complex program that simulates a university management system.

In this system, we'll have different types of personnel (students, faculty), various departments, courses, and enrollment management.

This example will utilize inheritance and interfaces extensively to demonstrate a structured object-oriented design in Java.

**Step 1: Define Interfaces and Base Classes**

// Interface for any person in the university

public interface **Person** {

String getName();

String getEmail();

}

// Interface for a student

public interface **Student** extends **Person** {

void enrollCourse(String courseCode);

void dropCourse(String courseCode);

}

// Interface for a faculty member

public interface **Faculty** extends **Person** {

void teachCourse(String courseCode);

}

// Base class for any university member

public abstract class **UniversityMember** implements **Person** {

private String name;

private String email;

public UniversityMember(String name, String email) {

this.name = name;

this.email = email;

}

@Override

public String getName() {

return name;

}

@Override

public String getEmail() {

return email;

}

}

**Step 2: Create Concrete Classes**

// Concrete class for a student

public class **StudentImpl** extends **UniversityMember** implements **Student** {

private List<String> enrolledCourses;

public StudentImpl(String name, String email) {

super(name, email);

this.enrolledCourses = new ArrayList<>();

}

@Override

public void enrollCourse(String courseCode) {

enrolledCourses.add(courseCode);

System.out.println(getName() + " enrolled in course " + courseCode);

}

@Override

public void dropCourse(String courseCode) {

enrolledCourses.remove(courseCode);

System.out.println(getName() + " dropped course " + courseCode);

}

}

// Concrete class for a faculty member

public class **FacultyImpl** extends **UniversityMember** implements **Faculty** {

public FacultyImpl(String name, String email) {

super(name, email);

}

@Override

public void teachCourse(String courseCode) {

System.out.println(getName() + " is teaching course " + courseCode);

}

}

// Enum for different departments

enum **Department** {

COMPUTER\_SCIENCE,

ELECTRICAL\_ENGINEERING,

MECHANICAL\_ENGINEERING,

BIOLOGY,

PHYSICS

}

// Class representing a course

public class **Course** {

private String courseCode;

private String courseName;

private Department department;

public Course(String courseCode, String courseName, Department department) {

this.courseCode = courseCode;

this.courseName = courseName;

this.department = department;

}

public String getCourseCode() {

return courseCode;

}

public String getCourseName() {

return courseName;

}

public Department getDepartment() {

return department;

}

}

**Step 3: University Management System**

// Main application for University Management System

public class **UniversityManagementSystem** {

public static void main(String[] args) {

// Create students

Student student1 = new StudentImpl("Alice", "alice@example.com");

Student student2 = new StudentImpl("Bob", "bob@example.com");

// Create faculty

Faculty faculty1 = new FacultyImpl("Dr. Smith", "smith@example.com");

Faculty faculty2 = new FacultyImpl("Prof. Johnson", "johnson@example.com");

// Create courses

Course cs101 = new Course("CS101", "Introduction to Computer Science", Department.COMPUTER\_SCIENCE);

Course ee201 = new Course("EE201", "Electrical Circuits", Department.ELECTRICAL\_ENGINEERING);

Course bio301 = new Course("BIO301", "Cell Biology", Department.BIOLOGY);

// Enroll students in courses

student1.enrollCourse(cs101.getCourseCode());

student1.enrollCourse(ee201.getCourseCode());

student2.enrollCourse(bio301.getCourseCode());

// Faculty teach courses

faculty1.teachCourse(cs101.getCourseCode());

faculty2.teachCourse(ee201.getCourseCode());

// Drop a course

student1.dropCourse(ee201.getCourseCode());

}

}

**Explanation:**

* **Interfaces (Person, Student, Faculty)**: Define common behaviors for university members and specific behaviors for students and faculty.
* **Base Class (UniversityMember)**: Implements common attributes (name, email) and methods (getName(), getEmail()) for all university members.
* **Concrete Classes (StudentImpl, FacultyImpl)**:
  + Implement specific behaviors (enrollCourse(), dropCourse() for students; teachCourse() for faculty).
  + Inherit from UniversityMember to reuse common attributes and methods.
* **Additional Classes (Course, Department)**:
  + Course represents a course with attributes (courseCode, courseName, department).
  + Department is an enum representing different departments in the university.
* **Main Application (UniversityManagementSystem)**:
  + Creates instances of students, faculty, and courses.
  + Demonstrates interactions such as enrolling students in courses, faculty teaching courses, and dropping a course by a student.

This program simulates a simplified university management system where students can enroll and drop courses, faculty members can teach courses, and courses are organized by departments. It showcases how Java's object-oriented features such as interfaces, inheritance, and polymorphism can be used to model complex real-world scenarios effectively.