Interface Comparison to Abstract classes &
Case studies

Core Design Decision: Abstract Class vs. Interface

An abstract class creates a tightly-coupled lineage, while an interface defines a loosely-coupled contract.

The wrong choice here leads to brittle, inflexible systems that are difficult to evolve.

Comparison

Attribute	Abstract Class	Interface
Purpose	Defines a common identity for a family of related classes (an **is-a** relationship).	Defines a role or capability for a class (a **can-do** relationship).
Methods	Can contain both `abstract` and concrete (implemented) methods.	All methods are implicitly `public` and `abstract` (unless using `default` or `static` methods as an exception).
Variables	Can contain instance variables (state) that are inherited by subclasses.	Can only contain `public static final` constants.
Inheritance	A class can `extend` only one abstract class.	A class can `implement` multiple interfaces.
Constructors	Has constructors, which are called by subclasses via `super()` during instantiation.	Has no constructors.

Decision-Making Framework

- **Choose an abstract class when your design requires:**
  - Creating a set of closely related classes that share a common identity.
  - Providing common, implemented functionality (code) that subclasses can inherit and use directly.
  - Maintaining state (instance variables) that is common to all subclasses.
- **• Choose an interface when your design requires:**
  - Unrelated classes to implement a common set of behaviors (e.g., Runnable, Serializable).
  - Defining a role or capability that a class can adopt in addition to its primary identity from its superclass.
  - A class to inherit behavior from multiple sources (simulating multiple inheritance).

Case Study: Solving the "Pet Shop" Problem

- Interfaces allow us to design for *roles* or *capabilities*, completely decoupling a class's abilities from its position in an inheritance hierarchy. Imagine a design dilemma where you need to add Pet behaviors (like beFriendly() and play()) to only some Animal subclasses, such as Dog and Cat, but not to Hippo or Lion.
- Flawed Approach #1: Adding Concrete Methods to Animal
  - Adding these methods to the Animal superclass would incorrectly grant pet behaviors to wild animals. A Lion or Hippo would inherit methods like play(), which violates the design's integrity.
- Flawed Approach #2: Adding Abstract Methods to Animal
  - Making the pet methods abstract in the Animal class is even worse. This would force every concrete subclass—including Hippo and Lion—to provide "do-nothing" implementations. This pollutes their public contract with irrelevant methods and creates unnecessary boilerplate code.

Pet Interface

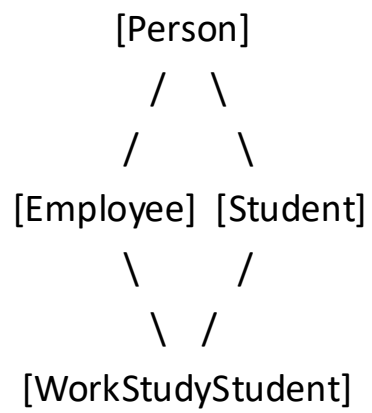
The ideal solution is the Pet interface.

This approach defines a Pet "role" contractually. A Dog class can extend Animal and also implements Pet.

This design correctly applies the behavior only where it belongs and avoids the "Deadly Diamond of Death" a problem associated with multiple inheritance of state and implementation.

"Deadly Diamond of Death" Problem Explained

- Imagine this class structure:
  - You have a base class named Person. It has a method called work().
  - Two classes, Employee and Student, inherit from Person.
  - Now, you create a new class, WorkStudyStudent, that inherits from *both* Employee and Student.
- Here is the "diamond":



- The ambiguity arises when you call the work() method on a WorkStudyStudent object. Which work() method should it run?
  - Does it inherit the one from the Employee path?
  - Does it inherit the one from the Student path?
  - Does it get *two copies* of the work() method, causing a conflict?

Cross-Hierarchy Polymorphism

The most powerful feature of interfaces is their ability to allow objects from **completely different inheritance trees to be treated polymorphically**. An object's eligibility is based on the role it plays (implements Pet), not its ancestry (extends Animal).

Case Study1: Sensor Monitoring System

- **A company needs to develop a system to monitor various types of sensors (e.g., temperature, humidity, motion sensors) within an industrial environment. The system should be able to handle different sensor types and easily integrate new types as they are developed.**
- **Problem**:
  Each sensor type has its own way of collecting and transmitting data. The challenge is to design a system that can work with all these different sensors and remain open for future sensor types without requiring significant changes to the existing codebase.
- **Solution Using Interfaces:**
  1. Define a Sensor Interface: This interface declares methods for common sensor functionalities, such as `readData`.
  2. Implement the Interface for Each Sensor Type: Different classes are created for each sensor type (TemperatureSensor, HumiditySensor, MotionSensor), each implementing the Sensor interface.
  3. Use Polymorphism and Dependency Injection: The monitoring system uses the Sensor interface type to interact with different sensors, allowing new sensor types to be integrated seamlessly.

```java
//Sensor Interface
public interface Sensor {
    String readData();
}

//TemperatureSensor Implementation
public class TemperatureSensor implements Sensor {
    @Override
    public String readData() {
        // Logic to read temperature data
        return "Temperature: 22°C";
    }
}

//HumiditySensor Implementation
public class HumiditySensor implements Sensor {
    @Override
    public String readData() {
        // Logic to read humidity data
        return "Humidity: 45%";
    }
}

//MotionSensor Implementation
public class MotionSensor implements Sensor {
    @Override
    public String readData() {
        // Logic to detect motion
        return "Motion detected";
    }
}
```

```java
//Sensor Monitoring
public class SensorMonitor {
    private List<Sensor> sensors;

    public SensorMonitor() {
        sensors = new ArrayList<>();
    }

    public void addSensor(Sensor sensor) {
        sensors.add(sensor);
    }

    public void monitorSensors() {
        for (Sensor sensor : sensors) {
            System.out.println(sensor.readData());
        }
    }

    public static void main(String[] args) {
        SensorMonitor monitor = new SensorMonitor();
        monitor.addSensor(new TemperatureSensor());
        monitor.addSensor(new HumiditySensor());
        monitor.addSensor(new MotionSensor());

        monitor.monitorSensors();
    }
}
```

Benefits and Conclusion CS1

Benefits:

- **Flexibility**: Easy to add new sensor types without altering existing code.
- **Scalability**: The system can accommodate an increasing number of sensor types.
- **Maintainability**: Isolates changes to individual sensor implementations.

Conclusion:

- This case study illustrates the power of interfaces in building a sensor monitoring system that is both adaptable to various sensor types and extendable for future developments. Interfaces in Java enable a design that is clean, modular, and adheres to the principles of good software engineering, facilitating easy maintenance and expansion.

Case Study2: Payment Processing System

A software company is developing a payment processing system that needs to support multiple payment methods like credit cards, PayPal, and cryptocurrencies. To create a flexible and extensible system, the development team decides to use interfaces.

- Problem:
  The system must be capable of processing different types of payments, each with its own processing logic. However, it should be designed in a way that adding new payment methods in the future is straightforward and doesn't require major changes to the existing codebase.
- Solution Using Interfaces:
  1. Define a Payment Interface: This interface declares a method for processing payments, such as `processPayment`.
  2. Implement the Interface for Each Payment Type: Separate classes are created for each payment method (CreditCard, PayPal, Crypto), and each class implements the Payment interface.
  3. Use Polymorphism: The system uses the Payment interface type to handle all payments, allowing for flexibility and the ability to add new payment methods easily.

```java
//Payment Interface
public interface Payment {
    boolean processPayment(double amount);
}

//CreditCard Implementation
public class CreditCard implements Payment {
    @Override
    public boolean processPayment(double amount) {
        // Specific logic for processing credit card payment
        System.out.println("Processing credit card payment: $" + amount);
        return true;
    }
}

//PayPal Implementation
public class PayPal implements Payment {
    @Override
    public boolean processPayment(double amount) {
        // Specific logic for processing PayPal payment
        System.out.println("Processing PayPal payment: $" + amount);
        return true;
    }
}

//crypto Implementation
public class Crypto implements Payment {
    @Override
    public boolean processPayment(double amount) {
        // Specific logic for processing cryptocurrency payment
        System.out.println("Processing crypto payment: $" + amount);
        return true;
    }
}
```

```java
//Payment Processing
public class PaymentProcessor {
    public void process(Payment paymentMethod, double amount) {
        if (paymentMethod.processPayment(amount)) {
            System.out.println("Payment processed successfully.");
        } else {
            System.out.println("Payment failed.");
        }
    }

    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();
        processor.process(new CreditCard(), 100.0);
        processor.process(new PayPal(), 200.0);
        processor.process(new Crypto(), 300.0);
    }
}
```

Case Study3: University Extracurricular Activities Management System

A university wishes to develop a software system to manage and track student participation in various extracurricular activities, such as clubs, sports teams, and academic societies. Each type of activity has unique characteristics and requirements.

- **Problem**:
  The challenge is to design a system that can handle different types of extracurricular activities, each with its own set of functionalities, while also being adaptable for new types of activities that may be introduced in the future.
- **Solution Using Interfaces:**
  - Define an Activity Interface: This interface declares methods relevant to extracurricular activities, such as `registerStudent`, `scheduleEvent`, and `getActivityDetails`.
  - Implement the Interface for Each Activity Type: Different classes are created for each type of activity (Club, SportsTeam, AcademicSociety), each implementing the Activity interface.
  - Use Polymorphism: The system uses the Activity interface type to handle all activities, allowing for a unified approach to manage diverse activities.

```java
//Activity Interface
public interface Activity {
    void registerStudent(String studentId);
    void scheduleEvent(String eventDetails);
    String getActivityDetails();
}

//Club Implementation
public class Club implements Activity {
    @Override
    public void registerStudent(String studentId) {
        // Logic to register a student in a club
        System.out.println("Student " + studentId + " registered in Club.");
    }

    @Override
    public void scheduleEvent(String eventDetails) {
        // Logic to schedule a club event
        System.out.println("Club event scheduled: " + eventDetails);
    }

    @Override
    public String getActivityDetails() {
        return "Club Activity Details";
    }
}

//SportsTeam Implementation
public class SportsTeam implements Activity {
    @Override
    public void registerStudent(String studentId) {
        // Logic to register a student in a sports team
        System.out.println("Student " + studentId + " registered in Sports Team.");
    }

    @Override
    public void scheduleEvent(String eventDetails) {
        // Logic to schedule a sports team event
        System.out.println("Sports team event scheduled: " + eventDetails);
    }

    @Override
    public String getActivityDetails() {
        return "Sports Team Activity Details";
    }
}

//AcademicSociety Implementation
public class AcademicSociety implements Activity {
    @Override
    public void registerStudent(String studentId) {
        // Logic to register a student in an academic society
        System.out.println("Student " + studentId + " registered in Academic Society.");
    }

    @Override
    public void scheduleEvent(String eventDetails) {
        // Logic to schedule an academic society event
        System.out.println("Academic Society event scheduled: " + eventDetails);
    }

    @Override
    public String getActivityDetails() {
        return "Academic Society Activity Details";
    }
}

//Activity Management
public class ActivityManager {
    private List<Activity> activities;

    public ActivityManager() {
        activities = new ArrayList<>();
    }

    public void addActivity(Activity activity) {
        activities.add(activity);
    }

    public void manageActivities() {
        for (Activity activity : activities) {
            System.out.println(activity.getActivityDetails());
            activity.registerStudent("12345");
            activity.scheduleEvent("Event on Friday");
        }
    }

    public static void main(String[] args) {
        ActivityManager manager = new ActivityManager();
        manager.addActivity(new Club());
        manager.addActivity(new SportsTeam());
        manager.addActivity(new AcademicSociety());

        manager.manageActivities();
    }
}
```

Questions?