Assignment-1[Set-1]

Name-Astik Joshi **Subject**-System Design

UID-23BCS10627 **Subject code**-23CSH-314

Q1. Explain the role of interfaces and enums in software design with proper examples.

Ans:

An interface represents a blueprint that defines a set of behaviors which a class must implement. It describes what actions are required without specifying how those actions are performed. Interfaces mainly consist of abstract method declarations and constant values.

Importance of Interfaces

Abstraction

Interfaces expose only the required functionality while hiding implementation details.

Design advantage: Simplifies complex systems by separating "what" from "how".

Loose Coupling

When classes interact through interfaces instead of concrete implementations, dependencies are reduced.

Design advantage: Code becomes easier to modify and maintain.

Multiple Inheritance Support

Since Java does not allow multiple inheritance through classes, interfaces provide a workaround by allowing a class to implement multiple interfaces.

Design advantage: Encourages flexible and reusable designs.

Polymorphism

Different classes implementing the same interface can be treated uniformly.

Design advantage: Enables runtime behavior changes.

Better Testability

Interfaces allow developers to substitute real implementations with mock objects during testing.

Design advantage: Facilitates isolated and effective unit testing.

Interface Example: Payment Processing

Step 1: Define the Interface

```
interface PaymentMethod {
    void makePayment(double amount);
}
```

Step 2: Implement the Interface

```
class CardPayment implements PaymentMethod {
    public void makePayment(double amount) {
        System.out.println("Payment of " + amount + " made using Card");
    }
}

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

Step 3: Use Interface in Service Layer

```
class PaymentProcessor {
    private PaymentMethod method;

    PaymentProcessor(PaymentMethod method) {
        this.method = method;
    }

    void executePayment(double amount) {
        method.makePayment(amount);
    }
}
```

This design allows new payment modes to be added without changing existing logic.

An enum (enumeration) is a data type that defines a limited and fixed set of constant values. It is useful when a variable should only hold predefined options.

Importance of Enums:

1. **Type Safety**

Enums ensure that only valid values are assigned.

This avoids errors caused by invalid strings or numbers.

2. **Code Readability**

Named constants make the code easier to understand and self-explanatory.

3. **Centralized Value Management**

All valid options are maintained at a single location.

This simplifies updates and maintenance.

4. **Behavior Support**

Enums can include fields and methods, enabling logic to be encapsulated within them.

5. **Eliminates Magic Values**

Prevents the use of hard-coded literals scattered across the codebase.

Enum Example: Order Status

Step 1: Define Enum

```
enum OrderState {
   PLACED,
   SHIPPED,
   DELIVERED,
   CANCELLED
}
```

Step 2: Use Enum in a Class

```
class Order {
   private OrderState state;

   Order(OrderState state) {
       this.state = state;
   }

   void displayStatus() {
       System.out.println("Current order state: " + state);
   }
}
```

Enums make state management safer and more readable.

Q2. Discuss how interfaces enable loose coupling with example.

Ans:
Loose coupling refers to a design approach where system components depend as little as possible on one another. In such systems, modifying one component does not force changes in other components, resulting in better maintainability and scalability.

Interfaces are a key mechanism for achieving loose coupling. By defining a common contract, interfaces allow classes to interact without being aware of each other's concrete implementations. This ensures that components rely on abstractions instead of specific classes.

Example: Tight Coupling Scenario

```
class CardPayment {
    void pay(double amount) {
        System.out.println("Paid using Card");
    }
}


class PaymentService {
    CardPayment payment = new CardPayment();

    void process(double amount) {
        payment.pay(amount);
    }
}
```

Issues with this design:

- PaymentService is directly dependent on CardPayment
- Introducing new payment methods requires code changes
- Violates the Open–Closed Principle

➞ This results in tight coupling

Loose Coupling Using Interface

Step 1: Define Interface

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

Step 2: Implement Interface

```
class CardPayment implements Payment {
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Card");
    }
}


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

Step 3: Use Interface in Service

```
class PaymentService {
    private Payment payment;


    PaymentService(Payment payment) {
        this.payment = payment;
    }
    void process(double amount) {
        payment.pay(amount);
    }
}
```

Step 4: Swap Implementation Without Code Change

```
Payment payment = new UpiPayment();
PaymentService service = new PaymentService(payment);
service.process(500);
```

The service logic remains unchanged while behavior changes dynamically.

Conclusion:

Interfaces promote loose coupling by separating behavior definition from implementation. This results in a modular, flexible, and scalable system that is easy to extend, test, and maintain—making interfaces a cornerstone of effective software design.