DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

Assignment:01

Subject Name: System Design
Subject Code: 23CSH-307

Submitted By: Jashanpreet Kaur
UID: 23BCS14043
Section/Group: 23BCSKRG-2A
Branch: BE-CSE

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

Answer:

In software design, interfaces and enumerations (enums) are important constructs that help in building well-structured, maintainable, and scalable systems. They promote principles such as abstraction, consistency, type safety, and clean design.

1. Role of Interfaces in Software Design

What is an Interface?

An interface defines a contract that a class must follow. It contains method declarations without implementation (in most languages like Java). Any class that implements an interface must provide implementations for all its methods.

Role of Interfaces

1. **Abstraction**
   - Interfaces hide implementation details and expose only essential behavior.
   - Helps designers focus on what a class does, not how it does it.
2. **Loose Coupling**
   - Code depends on interfaces, not concrete classes.
   - Makes systems flexible and easy to modify.
3. **Supports Multiple Inheritance**
   - A class can implement multiple interfaces.
   - Avoids problems of multiple class inheritance.
4. **Improves Testability**
   - Interfaces allow easy mocking and stubbing in unit tests.
5. **Scalability and Extensibility**
   - New implementations can be added without changing existing code.

Example of Interface

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

Design Explanation

- Payment interface defines a common payment contract.
- Different payment methods implement it.
- New payment modes (e.g., NetBanking) can be added without modifying existing logic.

2. Role of Enums in Software Design

What is an Enum?

An enum (enumeration) is a data type that defines a fixed set of constant values. It is used when a variable can take only predefined values.

Role of Enums

1. **Improves Type Safety**
   - Prevents invalid values from being assigned**.**
2. **Enhances Readability**
   - Makes code self-explanatory and meaningful**.**
3. **Eliminates Magic Numbers and Strings**
   - Avoids hard-coded values that are hard to maintain**.**
4. **Better Control Flow**
   - Useful in switch statements for decision-making.
5. **Consistent Domain Modeling**
   - Represents real-world states clearly.

Example of Enum

```
enum OrderStatus {
   PENDING,
   SHIPPED,
   DELIVERED,
   CANCELLED
}

class Order {
   OrderStatus status;
```

```
    Order(OrderStatus status) {
        this.status = status;
    }
}
```

Design Explanation

- OrderStatus enum restricts status values.
- Avoids invalid states like "shiped" or "done".
- Improves system reliability and clarity.

3. Using Interface and Enum Together (Real-World Design)

```
enum UserRole {
    ADMIN,
    CUSTOMER
}
interface User {
    void accessLevel();
}
class Admin implements User {
    public void accessLevel() {
        System.out.println("Full access granted");
    }
}
class Customer implements User {
    public void accessLevel() {
        System.out.println("Limited access granted");
    }
}
```

Design Benefit

- Enum controls roles
- Interface controls behavior
- Together they produce clean and scalable architecture

4. Conclusion

Interfaces and enums play a vital role in software design:

- Interfaces support abstraction, flexibility, and clean architecture.
- Enums ensure data consistency, type safety, and readable code.

Using both appropriately leads to robust, maintainable, and extensible software systems, which is a key goal of good software design.

Q2. Discuss how interfaces enable loose coupling with example.

Answer:

1. Meaning of Loose Coupling

Loose coupling refers to a design approach in which **components of a system are minimally dependent on each other**. Changes in one component **do not significantly affect** other components.

Interfaces play a major role in achieving loose coupling by allowing classes to **depend on abstractions rather than concrete implementations**.

2. How Interfaces Enable Loose Coupling

An **interface defines a contract** without specifying implementation details. Classes interact through the interface instead of directly depending on a specific class.

This leads to:

- Reduced dependency between modules
- Improved flexibility and maintainability
- Easier testing and extension of the system

3. Problem Without Interface (Tightly Coupled Design)

```
class CreditCardPayment {
    void pay(double amount) {
        System.out.println("Paid using Credit Card");
    }
}
class ShoppingCart {
    CreditCardPayment payment = new CreditCardPayment();

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

Issues

- ShoppingCart is directly dependent on CreditCardPayment
- Adding another payment method (UPI, Cash, etc.) requires modifying ShoppingCart
- Violates **Open–Closed Principle**

4. Solution Using Interface (Loosely Coupled Design)

Step 1: Create an Interface

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

Step 2: Implement the Interface

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

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

Step 3: Use Interface in Client Class

```
class ShoppingCart {
    Payment payment;

    ShoppingCart(Payment payment) {
        this.payment = payment;
```

```
    }

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

5. How Loose Coupling Is Achieved

- ShoppingCart depends on the **Payment interface**, not on a specific class
- Payment method can change **without modifying ShoppingCart**
- New payment types can be added easily

Example usage:

```
ShoppingCart cart1 = new ShoppingCart(new CreditCardPayment());
cart1.checkout(1000);
ShoppingCart cart2 = new ShoppingCart(new UpiPayment());
cart2.checkout(500);
```

6. Benefits of Loose Coupling Using Interfaces

1. **Flexibility**
    - Easily switch implementations at runtime
2. **Maintainability**
    - Changes in one class do not affect others
3. **Scalability**
    - New features can be added with minimal changes
4. **Better Testing**
    - Interfaces allow mocking during unit tests
5. **Clean Architecture**
    - Follows SOLID design principles

7. Conclusion

Interfaces enable loose coupling by separating **what a class does** from **how it does it**. By programming to an interface rather than a concrete class, software systems become **flexible, extensible, and easier to maintain**, which is essential for good software design.