Assignment 01

Name-Raj Kumar Singh **Subject**-System Design

UID-23BCS11393 **Subject code**-23CSH-314

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

An **interface** defines a **contract** that a class must follow.
It specifies **what to do**, not **how to do it**.

An interface can contain:

- Method declarations (abstract by default)
- Constants (public static final)

Role of Interfaces in Software Design

1. Abstraction

Interfaces hide implementation details and expose only essential behavior.

Design benefit: Reduces system complexity.

2. Loose Coupling

Classes depend on interfaces, not concrete implementations.

Design benefit: Easier maintenance and modification.

3. Supports Multiple Inheritance

Java does not allow multiple class inheritance, but a class can implement multiple interfaces.

Design benefit: Flexible design.

4. Enables Polymorphism

Different classes can be treated uniformly using a common interface.

Design benefit: Runtime flexibility.

5. Improves Testability

Interfaces allow mocking or stubbing during testing.

Design benefit: Easier unit testing.

Interface Example (Payment System)

Step 1: Define Interface

```
interface Payment {
```

```
    void pay(double amount);
}
```

Step 2: Implement Interface

```
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");
    }
}
```

Step 3: Use Interface (Loose Coupling)

```
class PaymentService {
    private Payment payment;

    PaymentService(Payment payment) {
        this.payment = payment;
    }

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

An **enum (enumeration)** is a special data type that represents a **fixed set of constants**.

Role of Enums in Software Design

1. Type Safety

Enums prevent invalid values.

Without enum:

```
String status = "DELIVEREDD";
```

With enum:

OrderStatus status = OrderStatus.DELIVERED;

2. Improves Readability

Enums give meaningful names to values.

Design benefit: Self-documenting code.

3. Centralized Control

All valid values are defined in one place.

Design benefit: Easy modification and maintenance.

4. Supports Behavior

Enums can have methods, fields, and constructors.

Design benefit: Cleaner logic than switch-case or if-else.

5. Prevents Magic Values

Avoids hard-coded numbers or strings.

Enum Example (Order Management System)

Step 1: Define Enum

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

Step 2: Use Enum in Class

```
class Order {
   OrderStatus status;

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

```
    void showStatus() {
        System.out.println("Order status: " + status);
    }
}
```

Q2. Discuss how interfaces enable loose coupling with example.

Loose coupling is a software design principle in which components of a system have minimal dependency on one another. In a loosely coupled system, changes in one component do not significantly affect other components, thereby improving maintainability and flexibility.

Interfaces play a crucial role in enabling loose coupling by acting as an abstraction layer between different parts of a software system. An interface defines a set of methods that a class must implement, without specifying how those methods should be implemented. This allows the interacting classes to depend on the interface rather than on concrete implementations.

When a class uses an interface instead of a specific class, it becomes independent of the actual implementation. Multiple classes can implement the same interface in different ways, and the dependent class can work with any of these implementations without requiring modification. As a result, the system becomes more flexible and extensible.

Interfaces also support the principle of "programming to an abstraction rather than an implementation." This principle reduces tight coupling by ensuring that high-level modules are not directly dependent on low-level modules, but instead interact through well-defined contracts.

Furthermore, interfaces improve testability by allowing the use of mock or stub implementations during testing. This makes it easier to test individual components in isolation. They also promote adherence to object-oriented design principles such as the Open–Closed Principle, where software entities are open for extension but closed for modification.

In conclusion, interfaces enable loose coupling by separating the definition of behavior from its implementation. This leads to a modular, maintainable, scalable, and flexible software design.

Tight Coupling (without interface)

Example: Payment System

```
class CreditCardPayment {

    void pay(double amount) {

        System.out.println("Paid using Credit Card");

    }

}
```

```
class PaymentService {
    CreditCardPayment payment = new CreditCardPayment();

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

Problems:

- PaymentService is **directly dependent** on CreditCardPayment
- Cannot switch to UPI, NetBanking, etc. without changing code
- Violates **Open–Closed Principle**

➞ This is **tight coupling**

Loose Coupling Using Interface

Step 1: Create Interface

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

Step 2: Implement Interface

```
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");
    }
```

```
}
```

Step 3: Use Interface in Service Class

```
class PaymentService {
    private Payment payment;

    PaymentService(Payment payment) {
        this.payment = payment;
    }

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

Step 4: Change Behavior Without Changing Code

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

PaymentService code remains **unchanged**

Only the implementation is swapped