



**CHANDIGARH  
UNIVERSITY**  
Discover. Learn. Empower.

# ASSIGNMENT

Subject: System Design

Subject Code: 23CSH-314

Date: 04/02/2026

**Submitted To:**

Prof. Alok Kumar

**Submitted By:**

Aditya Singh

23BCS11827

KRG 2B

## ASSIGNMENT – 1 (SET-1)

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

⇒ Roles of Interfaces:

### **1. Achieving Abstraction**

Interfaces hide implementation details and expose only essential behavior.

Design Benefit: Reduces complexity and increases maintainability.

**Eg:**

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

### **2. Supporting Multiple Inheritance**

A class can implement multiple interfaces, avoiding ambiguity problems of multiple inheritance.

**Eg:**

```
interface Printable {  
    void print();  
}
```

```
interface Scannable {  
    void scan();  
}
```

```
class Printer implements Printable, Scannable {  
    public void print() {  
        System.out.println("Printing document");  
    }  
  
    public void scan() {  
        System.out.println("Scanning document");  
    }  
}
```

### 3. Loose Coupling

Interfaces allow the system to depend on abstractions, not concrete implementations.

Design Benefit: Easier testing, flexibility, and replacement of components.

**Eg:**

```
class ShoppingCart {  
    Payment paymentMethod;  
  
    ShoppingCart(Payment paymentMethod) {  
        this.paymentMethod = paymentMethod;  
    }  
  
    void checkout(double amount) {  
        paymentMethod.pay(amount);  
    }  
}
```

### 4. Enabling Polymorphism

Different classes can implement the same interface differently.

**Eg:**

```
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");  
    }  
}
```

⇒ Roles of Enums:

## 1. Representing Fixed Sets of Values

Enums prevent invalid values and make code safer.

**Eg:**

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

## 2. Improving Readability and Maintainability

Enums replace magic numbers or strings with meaningful names.

**Eg:**

Bad Design:

```
if(status == 3) { ... }
```

Good Design:

```
if(status == OrderStatus.DELIVERED) { ... }
```

## 3. Type Safety

The compiler prevents invalid assignments.

**Eg:**

```
OrderStatus status = OrderStatus.PLACED;  
// status = "SHIPPED"; ✗ Compile-time error
```

## 4. Adding Behavior to Constants

Enums can have methods and fields.

**Eg:**

```
enum TrafficLight {  
    RED(60),  
    YELLOW(5),  
    GREEN(45);  
}
```

```
int time;
```

```
TrafficLight(int time) {  
    this.time = time;  
}
```

```
int getTime() {  
    return time;  
}  
}
```

## **Q2 Discuss how interfaces enable loose coupling with example.**

Loose coupling means that different parts of a software system depend on abstractions rather than concrete implementations. When components are loosely coupled, changes in one part of the system have minimal or no impact on other parts.

### **Role of Interfaces in Loose Coupling**

An interface defines a contract (what to do) without revealing how it is done. Classes interact through the interface instead of directly depending on specific classes.

### **Tightly Coupled Design (Problem)**

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

```
class ShoppingCart {  
    CreditCardPayment payment = new CreditCardPayment();  
  
    void checkout(double amount) {  
        payment.pay(amount);  
    }  
}
```

**Issues:**

- ShoppingCart is directly dependent on CreditCardPayment
- Changing payment method requires modifying ShoppingCart
- Hard to test and extend

This is tight coupling.

## Loosely Coupled Design Using Interface (Solution)

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

### Step 3: Use the Interface in Client Class

```
class ShoppingCart {  
    Payment paymentMethod;  
  
    ShoppingCart(Payment paymentMethod) {  
        this.paymentMethod = paymentMethod;  
    }  
}
```

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

## How This Achieves Loose Coupling

- ShoppingCart depends only on the Payment interface, not on concrete classes
- Any new payment method can be added without changing ShoppingCart
- Objects can be easily swapped at runtime

```
Payment p1 = new CreditCardPayment();  
Payment p2 = new UpiPayment();
```

```
ShoppingCart cart1 = new ShoppingCart(p1);  
ShoppingCart cart2 = new ShoppingCart(p2);
```