# DEPARTMENT OF
# COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

## Assignment 1

**Student Name: Vinayak Arora** **UID: 23BCS12507**

**Branch: BE-CSE** **Section/Group: KRG-3B**

**Semester: 6th** **Date of Performance: 4/02/26**

**Subject Name: System Design** **Subject Code:23CSH-314**

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

**Ans:**

### Definitions:

• **Interface:** An interface acts as a **contract** that defines a set of methods that a class must implement, without providing the logic for those methods. It allows different classes to be treated uniformly based on what they can do rather than how they do it.

• **Enum (Enumeration):** An enum is a special data type used to define a **collection of constants**. It ensures type safety by restricting a variable to have one of only a few predefined values.

**Role in a Payment Processing System:** In a payment system, **interfaces** allow the application to support multiple payment providers (like Stripe, PayPal, or Square) using a standardized structure. **Enums** are used to represent fixed categories, such as the status of a transaction or the type of payment method used.

### Example Code:

```
// Enum defining fixed payment statuses
public enum PaymentStatus {
    PENDING,
    COMPLETED,
    FAILED,
    REFUNDED
}

// Interface defining the contract for any payment provider
public interface IPaymentProcessor {
    void processPayment(double amount);
```

```
    PaymentStatus getStatus();
}

// Concrete implementation for PayPal
public class PayPalProcessor implements IPaymentProcessor {
    public void processPayment(double amount) {
        System.out.println("Processing $" + amount + " via PayPal.");
    }

    public PaymentStatus getStatus() {
        return PaymentStatus.COMPLETED;
    }
}
```

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

## Ans:

**Definition of Loose Coupling: Loose coupling** is a design goal where components are independent of each other. By relying on interfaces instead of concrete classes, a change in one part of the system (like changing a payment vendor) does not require changes in the parts that use it.

## Application in Payment Processing:

If a `CheckoutService` class is directly tied to a `StripeProcessor` class, it is "tightly coupled." If the business decides to switch to PayPal, the `CheckoutService` code must be rewritten. However, by using an **interface** (`IPaymentProcessor`), the `CheckoutService` only knows that it is talking to *something* that can process payments.

## Example Code:

```
// High-Level Module: CheckoutService
public class CheckoutService {
    private final IPaymentProcessor paymentProcessor;

    // Loose Coupling: The service depends on the Interface, not a specific
class
    public CheckoutService(IPaymentProcessor processor) {
        this.paymentProcessor = processor;
    }

    public void completeOrder(double total) {
        // The service doesn't care if it's Stripe, PayPal, or Bank Transfer
        paymentProcessor.processPayment(total);
        System.out.println("Order finalized with status: " +
paymentProcessor.getStatus());
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        // We can easily swap PayPalProcessor for any other implementation
        IPaymentProcessor myProcessor = new PayPalProcessor();
```

```
        CheckoutService checkout = new CheckoutService(myProcessor);

        checkout.completeOrder(150.00);
    }
}
```