DEPARTMENT OF

Discover. Learn. Empower.

COMPUTER SCIENCE & ENGINEERING

ASSIGNMENT - 01

Name :- Archita Srivastava **UID :- 23BCS12459**

Section :- KRG-2B **Subject :- System Design**

Question 1.

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

Answer 1.

Interfaces

Interfaces define contracts that specify what methods a class must implement, without dictating how they should be implemented. They enable polymorphism, abstraction, and loose coupling in software design.

Key Roles of Interfaces:

1. **Defining Contracts:** Interfaces establish a set of methods that implementing classes must provide, ensuring consistency across different implementations.

2. **Achieving Abstraction:** They hide implementation details and expose only the essential behaviors, allowing code to work with abstractions rather than concrete types.

3. **Enabling Polymorphism:** Different classes can implement the same interface in different ways, allowing objects to be treated uniformly through the interface type.

4. **Supporting Multiple Inheritance:** In languages like Java and C#, a class can implement multiple interfaces, providing flexibility that single-inheritance class hierarchies cannot offer.

Example in Java:

```
// Interface defining a contract for payment

processing interface PaymentProcessor {     boolean

processPayment(double amount);     void

refund(double amount);

    String getPaymentMethod();
}
// Implementation for Credit Card payments class

CreditCardProcessor implements PaymentProcessor {
 private String cardNumber;     public

CreditCardProcessor(String cardNumber) {

this.cardNumber = cardNumber;

    }
    @Override     public boolean

processPayment(double amount) {

        System.out.println("Processing $" + amount + " via Credit Card");
        // Credit card processing logic

return true;

    }
    @Override     public void

refund(double amount) {

        System.out.println("Refunding $" + amount + " to Credit Card");
    }
    @Override            public   String

getPaymentMethod() {          return

"Credit Card";

    }
}
// Implementation for PayPal payments class

PayPalProcessor implements PaymentProcessor {

private String email;     public

PayPalProcessor(String email) {         this.email =

email;

    }
    @Override     public boolean

processPayment(double amount) {

        System.out.println("Processing $" + amount + " via PayPal");
        // PayPal processing logic

return true;

    }
```

```
   @Override     public void

refund(double amount) {

        System.out.println("Refunding $" + amount + " to PayPal account");     }
   @Override            public   String

getPaymentMethod() {          return

"PayPal";

   }


// Client code works with the interface, not specific implementations

class PaymentService {     public void checkout(PaymentProcessor

processor, double amount) {         if

(processor.processPayment(amount)) {

            System.out.println("Payment successful using " + processor.getPaymentMethod());
        }
   }
}
// Main class  public class Main {

public static void main(String[]

args) {

        // Create payment service
        PaymentService paymentService = new PaymentService();
        // Process payment with Credit Card
        PaymentProcessor creditCard = new CreditCardProcessor("1234-5678-9012-3456");

paymentService.checkout(creditCard, 150.00);

        System.out.println(); // Blank line for readability
        // Process payment with PayPal
        PaymentProcessor paypal = new PayPalProcessor("user@example.com");

paymentService.checkout(paypal, 75.50);

        System.out.println(); // Blank line for readability
        // Demonstrate refund functionality

creditCard.refund(50.00);

paypal.refund(25.00);

   }
```

```
  }
```

Output:

```
Processing $150.0 via Credit Card
Payment successful using Credit Card


Processing $75.5 via PayPal
Payment successful using PayPal


Refunding $50.0 to Credit Card
Refunding $25.0 to PayPal account
```

Enums

Enums (enumerations) are special data types that represent a fixed set of named constants. They improve code readability, type safety, and maintainability by replacing magic numbers or strings with meaningful named values.

Key Roles of Enums:

1. **Type Safety:** Enums prevent invalid values from being assigned, as only predefined constants are allowed.
2. **Readability:** They make code self-documenting by using descriptive names instead of arbitrary numbers or strings.
3. **Maintainability:** Centralizing related constants makes it easier to modify or extend the set of valid values.
4. **Switch Statement Support:** Enums work seamlessly with switch statements, enabling clear control flow logic.

Example in Java:

```
// Enum representing order status

enum OrderStatus {

   PENDING,
   CONFIRMED,
   SHIPPED,
   DELIVERED,
   CANCELLED
}
```

```
                                        //
Enum with additional data and methods enum

PaymentMethod {

   CREDIT_CARD("Credit Card", 2.5),
   DEBIT_CARD("Debit Card", 1.5),
   PAYPAL("PayPal", 3.0),
   BANK_TRANSFER("Bank

Transfer", 0.5);     private final String

displayName;     private final double

processingFee;

   // Constructor
   PaymentMethod(String displayName, double

processingFee) {         this.displayName = displayName;

this.processingFee = processingFee;

   }
   public String getDisplayName() {

return displayName;

   }
   public double getProcessingFee() {

return processingFee;

   }
   public double calculateFee(double amount) {

return amount * (processingFee / 100);

   }
}
// Using enums in a class class Order {     private int orderId;     private

OrderStatus status;     private PaymentMethod paymentMethod;     private

double amount;     public Order(int orderId, double amount, PaymentMethod

paymentMethod) {

      this.orderId = orderId;
      this.amount = amount;

this.paymentMethod = paymentMethod;

this.status = OrderStatus.PENDING;
```

```
     }
   public void updateStatus(OrderStatus

newStatus) {        this.status = newStatus;

switch (status) {            case CONFIRMED:

             System.out.println("Order #" + orderId + " confirmed. Processing payment...");
             break;

case SHIPPED:

             System.out.println("Order #" + orderId + " has been shipped!");
             break;

case DELIVERED:

             System.out.println("Order #" + orderId + " delivered successfully.");
             break;
          case CANCELLED:
             System.out.println("Order #" + orderId + " has been cancelled.");
             break;

default:

             System.out.println("Order #" + orderId + " status: " + status);
      }
   }
   public void displayPaymentInfo() {        double fee

= paymentMethod.calculateFee(amount);

      System.out.println("Payment Method: " + paymentMethod.getDisplayName());
      System.out.println("Processing Fee: $" + fee);
      System.out.println("Total: $" + (amount + fee));
   }
}
// Main class to test the code
public class Main {     public static

void main(String[] args) {

      // Create an order with Credit Card payment
      Order order1 = new Order(12345, 100.00,

PaymentMethod.CREDIT_CARD);        System.out.println("=== Order 1

===");        order1.updateStatus(OrderStatus.CONFIRMED);

order1.displayPaymentInfo();

      System.out.println();
```

```
order1.updateStatus(OrderStatus.SHIPPED);

order1.updateStatus(OrderStatus.DELIVERED);

System.out.println("\n=== Order 2 ===");

      // Create another order with PayPal payment
      Order order2 = new Order(67890, 200.00,

PaymentMethod.PAYPAL);

order2.updateStatus(OrderStatus.CONFIRMED);

order2.displayPaymentInfo();        System.out.println();

order2.updateStatus(OrderStatus.CANCELLED);

      System.out.println("\n=== Order 3 ===");
      // Create order with Bank Transfer
      Order order3 = new Order(11111, 500.00, PaymentMethod.BANK_TRANSFER);

order3.displayPaymentInfo();

   }
}
```

Output:

```
=== Order 1 ===
Order #12345 confirmed. Processing payment...
Payment Method: Credit Card
Processing Fee: $2.5
Total: $102.5
Order #12345 has been shipped!
Order #12345 delivered successfully.
=== Order 2 ===
Order #67890 confirmed. Processing payment...
Payment Method: PayPal
Processing Fee: $6.0
Total: $206.0
Order #67890 has been cancelled.
=== Order 3 ===
Payment Method: Bank Transfer
Processing Fee: $2.5
Total: $502.5
```

Question 2.

Discuss how interfaces enable loose coupling with example.

Answer 2.

Loose coupling is a design principle where components in a system have minimal dependencies on each other's internal implementations. Components interact through well-defined contracts (interfaces) rather than concrete implementations, making the system more flexible, maintainable, and testable.

How Interfaces Enable Loose Coupling

Interfaces enable loose coupling by:

1. **Separating "what" from "how"** - Defining what operations are available without specifying how they're implemented
2. **Reducing dependencies** - Code depends on abstractions (interfaces) rather than concrete classes
3. **Enabling substitutability** - Different implementations can be swapped without changing client code
4. **Facilitating testing** - Mock implementations can easily replace real ones for testing

Example : Payment Processing

```
// Interface - contract for

payment interface Payment {

void pay(double amount);

}
// Cash payment implementation class

CashPayment implements Payment {

@Override     public void pay(double

amount) {

        System.out.println("Paid $" + amount + " in cash");
    }
}
// Card payment implementation class

CardPayment implements Payment {

    @Override     public void

pay(double amount) {

        System.out.println("Paid $" + amount + " by card");
    }
}
// Shop class - loosely coupled to Payment interface class Shop {

public void checkout(Payment payment, double amount) {
```

```
System.out.println("Processing checkout...");

payment.pay(amount);  // Works with any Payment implementation

        System.out.println("Thank you!\n");
    }
}
// Main class public class

SimplePaymentExample {     public

static void main(String[] args) {

        Shop shop = new Shop();
        // Use cash payment
        Payment cash = new CashPayment();

shop.checkout(cash, 50.0);



        // Switch to card payment - Shop class doesn't need to

change!         Payment card = new CardPayment();

shop.checkout(card, 75.0);

    }
}
```

Output:

```
Processing checkout...
Paid $50.0 in cash
Thank you!
Processing checkout...
Paid $75.0 by card
Thank you!
```

Benefits of Loose Coupling Through Interfaces

1. Flexibility and Extensibility

- New notification methods can be added without modifying OrderService
- Easy to support multiple notification channels

2. Easier Testing

3. Runtime Flexibility

- Notification method can be changed dynamically based on user preferences, business rules, or configuration

4. **Single Responsibility Principle**

- OrderService focuses on order processing
- Each notification class focuses on its specific delivery mechanism

5. Open/Closed Principle

- System is open for extension (new notification types) but closed for modification (no changes to existing code)