Assignment - 1

Que 1. Explain the role of interfaces and enums in software design with proper examples.

Ans: Interfaces in Software Design

An **interface** defines a contract that specifies what operations a class must provide, without dictating how those operations are implemented. It represents behavior at an abstract level.

Role of Interfaces

The primary role of interfaces is to enable **abstraction and loose coupling**. Instead of depending on concrete classes, higher-level components depend on interfaces. This makes the system easier to extend, modify, and test.

Key benefits include:

- **Decoupling**: Clients are isolated from implementation details.
- **Polymorphism**: Multiple implementations can be used interchangeably.
- **Extensibility**: New implementations can be added without changing existing code.
- **Testability**: Interfaces allow mocking or stubbing in unit tests.

Example :->

```
public interface PaymentProcessor {
  void processPayment(double amount);
}
public class CreditCardProcessor implements PaymentProcessor {
  public void processPayment(double amount) {
    System.out.println("Paid by credit card: " + amount);
  }
}
public class CheckoutService {
  private PaymentProcessor processor;
```

```
    public CheckoutService(PaymentProcessor processor) {
        this.processor = processor;
    }

    public void checkout(double amount) {
        processor.processPayment(amount);
    }
}
```

Enums in Software Design

An **enum** represents a fixed, finite set of predefined values. Enums are used to model **states, modes, or categories** within a domain.

Role of Enums

Enums improve **type safety, readability, and correctness**. Instead of relying on strings or numeric constants, enums ensure that only valid values can exist.

Key benefits include:

- **Compile-time safety**: Invalid values are prevented.
- **Clarity**: Code becomes self-documenting.
- **Domain modeling**: Business concepts are represented explicitly.
- **Centralization**: Valid options are defined in one place.

Example :->

```
public enum OrderStatus {
    NEW,
    PROCESSING,
    SHIPPED,
    DELIVERED,
    CANCELLED
}
```

```
public class Order {
    private OrderStatus status = OrderStatus.NEW;
    public void ship() {
        if (status == OrderStatus.PROCESSING) {
            status = OrderStatus.SHIPPED;
        }
    }
}
```

Que 2. Discuss how interfaces enable loose coupling with example.

Ans: Interfaces enable **loose coupling** by allowing components to depend on **abstractions instead of concrete implementations**. Loose coupling means that changes in one part of a system have minimal impact on others, making the software easier to maintain, extend, and test.

An **interface** defines a contract that specifies what operations are available without revealing how they are performed. When a class uses an interface, it does not need to know the details of the underlying implementation. This separation reduces direct dependencies between components.

Example:->

```
public interface NotificationService {
    void send(String message);
}
public class EmailNotificationService implements NotificationService {
    public void send(String message) {
        System.out.println("Sending email: " + message);
    }
}
```

```
public class NotificationManager {
    private final NotificationService service;


    public NotificationManager(NotificationService service) {
        this.service = service;
    }


    public void notifyUser(String message) {
        service.send(message);
    }
}
```

In this design, NotificationManager depends only on the NotificationService interface, not on a specific notification method. If the system later needs SMS or pushes notifications, new implementations can be added without modifying NotificationManager.

This approach follows the **Dependency Inversion Principle**, improves testability through mock implementations, and ensures that high-level logic remains stable even as low-level details change.