OOM PREVIOUS YEAR SOLUTIONS

VISHAL SINGH (IFI2022003)

Question 1: Write with examples:

i) Role of Use case Diagrams , CRC and Class Diagrams in Object Oriented Modelling.

Solution:
a) Use Case Diagrams:

Role: Use Case Diagrams are used to capture and depict the interactions between different actors (users or external systems) and a system. They illustrate the various ways the system can be used and the different types of users involved.

Example: Consider an online shopping system. The actors could be "Customer," "Administrator," and "Supplier." The use cases might include "Place Order," "Track Order," and "Manage Product Catalog." The diagram would show how these actors interact with the system through these use cases.

b) CRC (Class-Responsibility-Collaboration) Cards:

Role: CRC cards are a brainstorming tool used in the early stages of object-oriented design. Each card represents a class and outlines its responsibilities and collaborations with other classes.

Example: In a banking system, you might have a class called "Account." The CRC card for this class could list responsibilities such as "manage balance," "process transactions," and collaborations like "collaborates with Transaction class." These cards help in identifying and organizing classes and their relationships.

c) Class Diagrams:

Role: Class Diagrams provide a visual representation of the static structure and relationships among classes in a system. They show the attributes, methods, and associations between classes.

Example: Continuing with the banking system, a Class Diagram might include classes such as "Account," "Transaction," and "Customer." Associations between these classes, such as "one-to-many" between "Customer" and "Account," would be represented. Attributes and methods of each class would also be specified.

Question 2: What is abstraction in OOPS, write the only effect that the Abstract keyword has on a class? What are the two main effects of declaring a method abstract?

Solution:

In Java, Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essential units are not displayed to the user.

Effect of the Abstract Keyword on a Class:

In OOP, the abstract keyword is used to declare an abstract class. An abstract class is a class that cannot be instantiated on its own and may contain abstract methods, as well as concrete methods. The main effect of declaring a class as abstract is that it cannot be instantiated directly; instead, it serves as a base class for other classes.

Example with code:

```
abstract class Animal {

   // abstract methods and/or concrete methods

   abstract void makeSound();

}

class Dog extends Animal {

   // implementation of abstract method

   void makeSound() {

      System.out.println("Woof!");

   }

}
```

```
// Animal animal = new Animal(); // This would be an error, as Animal is abstract.

Animal dog = new Dog();  // Correct, using polymorphism

dog.makeSound();  // Outputs: Woof!
```

Two Main Effects of Declaring a Method Abstract:

1) Forcing Subclasses to Provide Implementation:

⇨ When a method is declared as abstract in an abstract class, it means that the method has no implementation in the abstract class and must be implemented by any concrete subclass that extends the abstract class.

⇨ This enforces a contract, ensuring that all subclasses provide their own implementation for the abstract method.

Example with Code:

```
abstract class Shape {
    abstract double calculateArea();
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

2) Preventing Instantiation of the Abstract Class:

⇨ Abstract classes with abstract methods cannot be instantiated directly. This is because an abstract class may have incomplete or undefined methods, and it doesn't make sense to create objects of such classes.

⇨ Subclasses must be created to provide concrete implementations for the abstract methods before objects can be instantiated.

Example with code:

```
abstract class Vehicle {
   abstract void start();
}

class Car extends Vehicle {
   void start() {
      System.out.println("Car started");
   }
}
```

Question 3): Why is functional Independence significant in object oriented software design ,explain Cohesion and coupling in Java with example.

Solution:

FUNCTIONAL INDEPENDENCE IN OBJECT-ORIENTED SOFTWARE DESIGN:

Functional independence is a crucial aspect of object-oriented software design because it promotes modularization and encapsulation. In an object-oriented system, functional independence refers to the idea that each class or module should have a well-defined and limited set of responsibilities. This separation of concerns helps in creating modular, maintainable, and reusable code.

When classes or modules are functionally independent, changes to one part of the system are less likely to impact other parts, reducing the risk of unintended consequences. It also enhances code readability, understandability, and makes it easier to test and debug individual components.

COHESION IN JAVA:

Cohesion is a measure of how strongly the elements (methods or fields) within a module or class are related to each other. High cohesion implies that the elements within the module are closely related and work together to achieve a common purpose. On the other hand, low cohesion indicates that the elements within the module may not be strongly related, and the module may have multiple responsibilities.

COUPLING IN JAVA:

Coupling measures the degree of dependence between modules or classes. Low coupling is desirable because it indicates that changes to one module are less likely to affect another. High coupling implies a strong dependency between modules, making the system more rigid and less maintainable.

Question: Write Output of the code and what type of inheritance present in the following code.

```
class A{
  static{
    System.out.println("First");
  }
}
class B extends A{
  static{
    System.out.println("Second");
  }
}
class C extends B{
  static{
    System.out.println("Third");
  }
}
public class MainClass {
  public static void main(String[] args) {
    C c = new C();
  }}
```

Solution: Multilevel Inheritence and Output:

```
First
Second
Third
```

Question: Discuss public, private , protected and default access modifiers with examples.

Solution:

Access modifiers in Java control the visibility or accessibility of classes, methods, and other members within a program. There are four types of access modifiers in Java:

1.Public:

•The public modifier makes a class, method, or field accessible from any other class.

•Example:

```
public class PublicExample {
    public int publicField;

    public void publicMethod() {
        System.out.println("This is a public method.");
    }
}
```

The PublicExample class and its members can be accessed from any other class.

2.Private:

•The private modifier restricts the access of a class, method, or field to the class that declares it. It is not accessible from outside the class.

•Example:

```
public class PrivateExample {
    private int privateField;

    private void privateMethod() {
        System.out.println("This is a private method.");
    }
}
```

The PrivateExample class's privateField and privateMethod can only be accessed within the same class.

3.Protected:

•The protected modifier allows access within the same package and by subclasses, regardless of the package. It is more restrictive than public but less restrictive than private.

•Example:

```
public class ProtectedExample {
    protected int protectedField;

    protected void protectedMethod() {
        System.out.println("This is a protected method.");
    }
}
```

The ProtectedExample class and its members can be accessed by any class in the same package and by subclasses, even if they are in a different package.

4.Default (Package-Private):

•If no access modifier is specified, the default access level is package-private. This means that the class, method, or field is only accessible within the same package.

•Example:

```
class DefaultExample {
    int packagePrivateField;

    void packagePrivateMethod() {
        System.out.println("This is a package-private method.");
    }
}
```

The DefaultExample class and its members can be accessed only within the same package.

Question: Some programmers prefer not to use protected access, because they Believe It Breaks the encapsulation of the Superclass. Discuss the relative merits of using protected access versus using private access in superclass.

Solution:

The choice between using protected access versus private access in a superclass involves trade-offs and depends on the design goals, maintainability, and extensibility of the software. Let's discuss the relative merits of each:

USING PROTECTED ACCESS IN SUPERCLASS:

Pros:

●Facilitates Subclassing: protected access allows subclasses to access the members of the superclass. This is beneficial when you want to provide a certain level of extensibility, allowing subclasses to override or extend the behaviour of the superclass.

●Flexibility for Subclass Modifications: Subclasses can access protected members, enabling them to modify or extend the behavior of the superclass without violating encapsulation.

Cons:

●Breaks Encapsulation to Some Extent: Some argue that using protected access might break encapsulation because it exposes internal details of the superclass to its subclasses. Subclasses have access to the implementation details, potentially leading to tight coupling and making it harder to change the superclass without affecting its subclasses.

●Security Concerns: If there are security concerns or a need for strict encapsulation, using protected members might not be advisable.

USING PRIVATE ACCESS IN SUPERCLASS:

Pros:

●Enhances Encapsulation: private access ensures that the internal details of the superclass are hidden from subclasses. This promotes stronger encapsulation and reduces the risk of unintended interference by subclasses.

●Easier Maintenance: Since the internal implementation details are hidden, it becomes easier to modify the internals of the superclass without affecting its subclasses. This supports more straightforward maintenance.

Cons:

●Less Extensibility: private members are not accessible to subclasses, which can limit the extensibility of the superclass. Subclasses cannot directly modify or extend the behavior of the superclass.

- May Lead to Code Duplication: Without the ability for subclasses to access and modify certain aspects of the superclass, developers might resort to code duplication or other less clean solutions to achieve the desired functionality in subclasses.

Question: What are the similarities and differences between Interfaces and Abstract Class in Java.

Solution:

SIMILARITIES BETWEEN INTERFACES AND ABSTRACT CLASSES:

- **Abstraction:** Both interfaces and abstract classes provide a way to achieve abstraction by declaring methods without specifying their implementation details.

- **Cannot be Instantiated:** Instances of both interfaces and abstract classes cannot be created directly. They need to be extended or implemented by concrete classes.

- **Support for Polymorphism:** Both interfaces and abstract classes support polymorphism. Subclasses or implementing classes can be used interchangeably with their parent types.

- **Declaration of Methods:** Both can declare abstract methods that must be implemented by concrete subclasses or implementing classes.

DIFFERENCES BETWEEN INTERFACES AND ABSTRACT CLASSES:

- **Method Implementation:** In an abstract class, you can have both abstract and non-abstract (concrete) methods. In an interface, all methods are implicitly abstract and do not have an implementation until provided by the implementing class.

- **Multiple Inheritance:** A class in Java can extend only one abstract class, but it can implement multiple interfaces. This allows Java to achieve a form of multiple inheritance through interfaces.

- **Access Modifiers:** Abstract classes can have different access modifiers for their methods, including public, protected, private, or package-private (default). In contrast, all methods in an interface are implicitly public and abstract.
- **Fields (Variables):** Abstract classes can have instance variables (fields) that can be inherited by their subclasses. Interfaces, on the other hand, can only have constants (public static final variables).
- **Constructors:** Abstract classes can have constructors, and they are called when a concrete subclass is instantiated. Interfaces cannot have constructors.
- **Default Methods and Static Methods:** Java 8 introduced default methods and static methods in interfaces, allowing them to have method implementations. Abstract classes can also have method implementations, but they are not called default methods.
- **Intent and Use Cases:** Abstract classes are often used when there is a common base implementation shared among subclasses. Interfaces are used to define contracts that classes agree to fulfill, regardless of their inheritance hierarchy.
- **Extending and Implementing:** A class uses the extends keyword to inherit from an abstract class, and the implements keyword to implement an interface.

Question: How does the polymorphism enable you to program "in the general" rather than "in the specific". Discuss the key advantages of programming "in the general".

Solution: Polymorphism in object-oriented programming allows you to program "in the general" rather than "in the specific" through the use of a common interface or base class that can be shared by multiple classes. This concept is often referred to as "polymorphic behaviour."

HERE ARE THE KEY ADVANTAGES OF PROGRAMMING "IN THE GENERAL":

Code Reusability: When you design your code to work with polymorphism, you can create generalized classes and methods that operate on a common interface or base class. This enables you to reuse the same code with different implementations, promoting a more efficient and maintainable codebase.

Flexibility and Extensibility: Programming in the general allows you to create flexible systems that can be easily extended. New classes can be added without modifying existing code, as long as they adhere to the common interface or inherit from the base class. This promotes a modular and scalable architecture.

Interchangeability of Objects: Polymorphism allows objects of different classes to be treated as objects of a common type. This interchangeability simplifies the code and allows you to write methods that can accept a variety of objects as parameters, as long as they conform to the expected interface or inheritance hierarchy.

Simplified Code Maintenance: When changes or updates are needed, you can modify the common interface or base class, and the changes will automatically apply to all the classes that implement or inherit from it. This reduces the risk of introducing errors and makes code maintenance more straightforward.

Enhanced Readability and Abstraction: Code that operates in the general is often more readable and abstract. By focusing on the common characteristics shared by different classes, you can create higher-level abstractions that make it easier to understand the overall structure and behavior of the system.

Improved Collaboration: In a team environment, programming in the general promotes collaboration and parallel development. Different team members can work on specific classes or modules without interfering with each other, as long as they adhere to the agreed-upon interfaces or base classes.

Adherence to Design Principles: Polymorphism aligns well with key design principles such as the Open/Closed Principle and the Dependency Inversion Principle. It encourages code that is open for extension but closed for modification and promotes dependency on abstractions rather than concrete implementations.

Enhanced Testability: Code designed with polymorphism often leads to more modular and testable systems. Unit testing becomes more straightforward since you can isolate and test individual components without being overly concerned about the specifics of the implementations.

Question: State whether 'True' or 'false' in the following. Give reason in one sentence.

(i)Class diagram show the behavioral view of a software application.

(ii) Objects of the same class have same data elements and methods.

(iii) Unified modelling language (UML) is an object oriented programming language.

(iv) Polymorphism is an effect that stems from inheritance.

Solution: (i) **False:** Class diagrams primarily show the static structure (class structure) of a software application, not the behavioral view.

(ii) **True:** Objects of the same class have the same structure (data elements) and behaviour (methods) as defined by the class.

(iii) **False:** Unified Modelling Language (UML) is a modelling language used for visualizing, specifying, constructing, and documenting the artifacts of software systems, and it is not a programming language.

(iv) **False:** Polymorphism is a concept that allows objects of different types to be treated as objects of a common type; it is not solely dependent on inheritance but can also be achieved through interfaces in object-oriented programming.