Assignment 3 (Core Java)

Q1. Write a simple Banking System program by using OOPs concept where you can get account Holder name balance etc?

Sol:-

```
import java.util.Scanner;

// define class
class BankAccount {
    private String accountHolderName;
    private double balance;

    // constructor
    public BankAccount(String accountHolderName, double initialBalance) {
        this.accountHolderName = accountHolderName;
        this.balance = initialBalance;
    }

    public String getAccountHolderName() {
        return accountHolderName;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposit of " + amount + " successful.");
    }

    public void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawal of " + amount + " successful.");
        } else {
            System.out.println("Insufficient balance. Withdrawal failed.");
        }
    }
}

public class bankingSystemProgram {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter account holder name: ");
        String accountHolderName = scanner.nextLine();

        System.out.print("Enter initial balance: ");
        double initialBalance = scanner.nextDouble();

        BankAccount bankAccount = new BankAccount(accountHolderName, initialBalance);

        System.out.println("Account holder name: " + bankAccount.getAccountHolderName());
        System.out.println("Initial balance: " + bankAccount.getBalance());

        System.out.print("Enter amount to deposit: ");
        double depositAmount = scanner.nextDouble();
        bankAccount.deposit(depositAmount);

        System.out.print("Enter amount to withdraw: ");
        double withdrawalAmount = scanner.nextDouble();
        bankAccount.withdraw(withdrawalAmount);

        System.out.println("Current balance: " + bankAccount.getBalance());
    }
}
```

Q2. Write a Program where you inherit method from parent class and show method Overridden Concept?

Sol:-

```
class Parent {
    public void display() {
        System.out.println("This is the parent class");
    }
}

class Child extends Parent {
    @Override
    public void display() {
        System.out.println("This is the child class");
    }
}

public class overrideen {
    public static void main(String[] args) {

        Parent parent = new Parent();
        parent.display();

        Child child = new Child();
        child.display();

        // Upcasting - Child object assigned to a Parent reference
        Parent upcastedChild = new Child();
        upcastedChild.display();
    }
}
```

Q3. Write a program to show run time polymorphism in java?

Sol:-

```

class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class polymorphismAssignment {
    public static void main(String[] args) {

        Animal animal1 = new Animal();
        Animal animal2 = new Dog();
        Animal animal3 = new Cat();

        animal1.sound();
        animal2.sound();
        animal3.sound();
    }
}
```

Q4. Write a program to show Compile time polymorphism in java?

Sol:-

```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }

    public double add(double a, double b) {
        return a + b;
    }
}

public class compileTimePolymorphism {
    public static void main(String[] args) {

       Calculator calculator = new Calculator();

        int sum1 = calculator.add(5, 10);
        int sum2 = calculator.add(2, 4, 6);
        double sum3 = calculator.add(2.5, 3.7);

        System.out.println("Sum1: " + sum1);
        System.out.println("Sum2: " + sum2);
        System.out.println("Sum3: " + sum3);
    }
}
```

Q5. Achieve loose coupling in java by using OOPs concept?

Sol:-

Encapsulation:- Encapsulating related data and behavior within a class helps in achieving loose coupling. By hiding the internal implementation details and providing a well-defined interface, objects can interact with each other through method calls, promoting loose coupling.

Abstraction:- Using abstraction, you can define abstract classes or interfaces to represent common behavior or contracts. By depending on abstractions rather than concrete implementations, classes can communicate with each other through abstract types, reducing direct dependencies and achieving loose coupling.

Inheritance:- Inheritance can be used to create specialized classes that inherit behavior and properties from a base class. This allows objects to be treated polymorphically, enabling loose coupling between classes by depending on the base class or interfaces.

Interface-based programming:- Programming against interfaces rather than concrete classes promotes loose coupling. By depending on interfaces, you can interchange different implementations without affecting the code that uses them. This reduces the coupling between classes and allows for more flexibility and extensibility.

==========

Q6. What is the benefit of encapsulation in java?

Sol:-

Data Hiding:- Encapsulation allows you to hide the internal implementation details of a class from external code. By declaring the class's fields (data) as private, you prevent direct access to them from outside the class. This protects the integrity and consistency of the object's data and ensures that it can only be modified through controlled methods (getters and setters). This way, encapsulation provides data security and prevents unintended modifications or misuse of the object's state.

Modularity and Maintainability:- Encapsulation promotes modular design by encapsulating related data and behavior within a class. This helps in organizing and structuring the codebase, making it easier to understand, maintain, and extend. Changes to the internal implementation of a class can be isolated within the class, without affecting other parts of the program. This enhances code maintainability, as modifications are localized and do not have a ripple effect on other classes or modules.

Code Flexibility and Reusability:- Encapsulation allows you to define a public interface (methods) for interacting with an object while keeping the internal details hidden. This provides flexibility as you can change the internal implementation without impacting the code that uses the object. The external code interacts with the object through the defined interface, enabling loose coupling and enabling easy swapping of implementations. Encapsulation also supports code reusability by encapsulating common behavior within classes, which can be reused across different parts of the codebase or in other projects.

Encourages Good Programming Practices:- Encapsulation encourages the use of getter and setter methods to access and modify the object's state. This promotes the principle of information hiding and adheres to the concept of encapsulation. It helps enforce consistent access and modification rules for the object's data, facilitating data validation, error checking, and maintaining the integrity of the object's state. Encapsulation also promotes code readability and understandability by providing a clear and consistent way to interact with objects.

==========

Q7. Is java a 100% Object oriented Programming language? If no why ?

Sol:-
No, Java is not considered a 100% Object-Oriented Programming (OOP) language. While Java is predominantly an object-oriented language, there are a few features and aspects of the language that deviate from strict OOP principles. Here are some reasons why Java is not considered 100% pure OOP:

1.Primitive Data Types:- Java includes primitive data types like **int**, **double**, **boolean**, etc., which are not objects. These primitive types do not inherit from a common class or have associated methods. However, Java provides wrapper classes (such as **Integer**, **Double**, **Boolean**) that allow these primitive types to be used as objects.

2.Static Methods:- Java allows the declaration of static methods that are associated with the class itself rather than with an instance of the class. Static methods can be called without creating an object of the class, and they can access static variables. Since static methods are not tied to a specific object, they do not adhere to the instance-based nature of OOP.

3.Lack of Multiple Inheritance:- Java does not support multiple inheritance, where a class can inherit from multiple parent classes. Instead, Java uses interfaces to achieve a form of multiple inheritance through interface implementation. While interfaces provide a way to define contracts and achieve polymorphism, they are distinct from classes and do not support code implementation.

4.Access Modifiers:- Java employs access modifiers (**public**, **protected**, **private**, and default access) that control the visibility and accessibility of class members (fields, methods, constructors). These modifiers can limit or restrict the full openness and encapsulation expected in pure OOP.

==========

Q8. What are the advantages of abstraction in java?

Sol:- Abstraction in Java is a crucial concept in object-oriented programming that allows you to create abstract classes and interfaces to represent common behavior and define contracts. It offers several advantages:

1.Simplifies Complexity:- Abstraction helps in managing complexity by hiding unnecessary implementation details and focusing on the essential features and behaviors. It allows you to represent real-world concepts and entities in a simplified and understandable manner. By abstracting away low-level implementation complexities, you can focus on higher-level design and problem-solving.

2.Provides a Clear Interface:- Abstraction provides a clear and well-defined interface for interacting with objects. By defining abstract methods in interfaces or abstract classes, you establish a contract that specifies what behavior is expected without getting into the specifics of how it is implemented. This separation between the interface and implementation promotes modularity, flexibility, and code maintainability.

3.Promotes Code Reusability:- Abstraction facilitates code reuse. By creating abstract classes or interfaces, you can define common behavior that can be implemented by multiple classes. This promotes code reuse, as multiple classes can implement the same interface or extend the same abstract class, inheriting the common behavior. Abstraction helps in building modular and extensible code, reducing redundancy and promoting efficient development.

4.Enables Polymorphism:- Abstraction plays a significant role in achieving polymorphism, one of the fundamental principles of object-oriented programming. Polymorphism allows objects to be treated as instances of their own class or as instances of their parent class or interface. By relying on abstraction, you can write code that works with generic types or abstract types, enabling flexibility and extensibility in handling different implementations of the same behavior.

5.Supports Loose Coupling:- Abstraction helps in achieving loose coupling between components in a system. By depending on abstractions (interfaces or abstract classes) rather than concrete implementations, you reduce direct dependencies between classes. This promotes flexibility, as you can easily switch implementations without affecting the code that uses those abstractions. Loose coupling improves code maintainability, testability, and the ability to adapt to changing requirements.

==========

Q9. What is an abstraction explained with an Example?

Sol:-

Abstraction in Java is a concept that allows you to represent complex real-world entities in a simplified and understandable manner by hiding unnecessary implementation details. It involves creating abstract classes and interfaces to define common behavior and contracts.

Example:-

```java
// Abstract class representing a Shape
abstract class Shape {
    public abstract void draw(); // Abstract method to draw the shape

    public void displayArea() {
        System.out.println("Area: " + calculateArea()); // Concrete method to display the area
    }

    public abstract double calculateArea(); // Abstract method to calculate the area
}

// Concrete class representing a Circle
class Circle extends Shape {
    private double radius;

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

    public void draw() {
        System.out.println("Drawing a circle");
    }

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

// Concrete class representing a Rectangle
class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public void draw() {
        System.out.println("Drawing a rectangle");
    }

    public double calculateArea() {
        return length * width;
    }
}

public class AbstractionExample {
    public static void main(String[] args) {
        Shape circle = new Circle(5.0);
        circle.draw();
        circle.displayArea();

        Shape rectangle = new Rectangle(4.0, 6.0);
        rectangle.draw();
        rectangle.displayArea();
    }
}
```

Q10. What is the final class in Java?

Sol:-

In Java, a final class is a class that cannot be subclassed or extended by other classes. Once a class is declared as final, it cannot be inherited by any other class. The final keyword is used to mark a class as final.

1.Inheritance Restriction:- When a class is declared as final, it signifies that it is complete and cannot be extended further. This restriction prevents other classes from inheriting from the final class.

2.Method Overriding Restriction:- When a class is marked as final, it also implies that its methods cannot be overridden by subclasses. The final methods within a final class cannot be modified or overridden in any subclass.

3.Design and Security:- Final classes are commonly used in Java to achieve design goals or ensure security. By marking a class as final, you indicate that it is not designed to be subclassed and is intended to be used as-is.

4.Efficiency:- Final classes can offer improved performance in certain scenarios. Since they cannot be extended or overridden, the compiler can optimize code execution, and runtime checks for method invocations and polymorphism can be avoided.