Assignment No: 3

Ques 1 Answer:

```
import java.util.Scanner;
import java.util.*;
class BankAccount {
    private String accountHolderName;
    private double balance;
    public BankAccount(String accountHolderName) {
        this.accountHolderName = accountHolderName;
        this.balance = 0.0;
    }
    public String getAccountHolderName() {
        return accountHolderName;
    }
    public double getBalance() {
        return balance;
    }
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposit successful. Current balance: " + balance);
        } else {
            System.out.println("Invalid amount. Deposit failed.");
        }
    }
    public void withdraw(double amount) {
        if (amount > 0) {
            if (amount <= balance) {
                balance -= amount;
                System.out.println("Withdrawal successful. Current balance: " + balance);
            } else {
                System.out.println("Insufficient balance. Withdrawal failed.");
            }
        } else {
            System.out.println("Invalid amount. Withdrawal failed.");
        }
    }
}
```

```
public class BankingSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter account holder name: ");
        String accountHolderName = scanner.nextLine();
        BankAccount account = new BankAccount(accountHolderName);
        System.out.println("Welcome, " + account.getAccountHolderName() + " ! ");
        System.out.println("Your current balance is: " + account.getBalance());

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

        System.out.print("Enter the amount to withdraw: ");
        double withdrawalAmount = scanner.nextDouble();
        account.withdraw(withdrawalAmount);
    }
}
```

Que2.Answer:

```
class ParentClass {
    public void display() {
        System.out.println("This is the display method of the parent class.");
    }
}
class ChildClass extends ParentClass {
    @Override
    public void display() {
        System.out.println("This is the display method of the child class.");
    }
}
public class InheritanceExample {
    public static void main(String[] args) {
        // Create an instance of the parent class
        ParentClass parent = new ParentClass();
        parent.display();
        ChildClass child = new ChildClass();

                child.display();
    }
}
```

Ques.3Answer

```
class Fruit {
   public void taste () {
      System.out.println("Fruit makes a taste ");
   }
}

class Mango extends Fruit {
   @Override
   public void taste () {
      System.out.println("Mango Sweet");
   }
}

class Orange extends Fruit{
   @Override
   public void taste () {
      System.out.println("Orange Sour");
   }
}

public class PolymorphismExample {
   public static void main(String[] args) {
      Fruit fruit1 = new Fruit();
        Fruit fruit2 = new Mango();
            Fruit fruit3 = new Orange ();
      fruit l1. taste ();  // call
      fruit 2. taste ();  //
      fruit 3. taste ();  //
}
```

Ques 4.Answer:

```
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 PolymorphismExample {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        int result1 = calculator.add(5, 10);
        System.out.println("Result 1: " + result1);

        int result2 = calculator.add(2, 4, 6);
        System.out.println("Result 2: " + result2);

        double result3 = calculator.add(2.5, 3.5);
        System.out.println("Result 3: " + result3);
    }
}
```

Que5.Answer :

Encapsulation: Expose a relatively well-defined interface with the state and behaviour of a class. Therefore, it can be changed internally without affecting the rules that implement the class.

Interface-based programming: Functionality that involves interfaces rather than specific applications. Define the interfaces that define the contract for the group of classes. By coding against interfaces, you can easily switch between different implementations without affecting the code that implements the interface.

Dependency Injection (DI): Use dependency injection to give an external class the required dependencies. Instead of creating dependencies in a class, pass them through constructor parameters, method parameters, or setter methods. This approach offers great flexibility, as functions can be easily loaded without modifying the underlying classes.

Inverting a container of control (IoC): Use IoC containers, such as the Spring framework's Application Context, to control creation and dependency injection. IoC containers build and connect threads, reducing the interdependence between classes.

Designed design: Implement scheduling strategies such as workspaces, dependency injection, and task managers that encourage loose coupling and modular design.

Limit Dependencies: Reduce the number of dependencies between classes. Avoid tight coupling by reducing direct dependencies on concrete classes and instead rely on abstractions or interfaces.

Follow SOLID principles: SOLID principles (single responsibility, open-closure, Liskov substitution, interface separation, dependency inversion) contribute to loose coupling and maintainable code Following this principle promotes modularity, separation of concerns, and flexibility . . . .

It's about it

Ques6.Answer:
Encapsulation in Java provides several benefits, including:

Data Hiding: Encapsulation allows you to hide the internal details and implementation of a class. By using access modifiers (such as private, protected, and public), you can control the visibility of class members. This prevents direct access to internal data and provides a level of abstraction, enhancing data security and integrity.

Access Control: Encapsulation allows you to specify the level of access to class members. By using access modifiers, you can restrict access to sensitive or implementation-specific details. This enables you to enforce data encapsulation and maintain proper encapsulation boundaries.

Code Organization and Maintainability: Encapsulation promotes code organization and modular design. By encapsulating related data and behavior within a class, you create self-contained and reusable components. This improves code maintainability, as changes to the implementation can be confined within the class, reducing the impact on other parts of the codebase.

Flexibility and Evolution: Encapsulation provides flexibility in evolving the internal implementation of a class. As long as the public interface remains the same, you can modify or refactor the internal implementation without affecting the code that uses the class. This allows for easier code evolution, bug fixing, and enhancement without breaking existing client code.

Code Reusability: Encapsulated classes with well-defined interfaces can be easily reused in different parts of the codebase or in other projects. By exposing only the necessary public methods, you create reusable components that can be integrated into various systems without exposing implementation details.

Testing and Debugging: Encapsulation facilitates easier testing and debugging. Since the internal state of an object is encapsulated, you can create focused tests that verify the behavior of specific methods without needing to know the intricacies of the internal implementation. Additionally, encapsulation can help in isolating and identifying issues within a class, making debugging more manageable.

Overall, encapsulation in Java improves code maintainability, reusability, flexibility, and security. It promotes modular design, enhances code organization, and allows for easier evolution and testing of software systems.

Que.7Answer:
Primitive data types: Java includes primitive data types such as int, boolean, and char, which are not objects. There are no associated methods or properties like these types of things. However, Java provides wrapper classes (Integer, Boolean, Character, etc.) to treat these primitives as objects when needed.

Static members: Allows the declaration of static members (variables and methods) in Java classes. Static members are associated with the class itself rather than individual instances of the class. They can be accessed without creating an object, they are not bound by the principles of instance level condensation and polymorphism.

Procedural Programming: While Java is mainly object-oriented, it also supports procedural programming. Creating functions and methods outside of Java classes (through static methods in utility classes) allows you to perform programming tasks without having to implement an object.

Inheritance Restriction: Java supports only one inheritance, which means that a class can inherit from only one parent class. This is different from languages like C++ that support many properties. However, Java supports a wide range of interfaces, enabling multiple properties to be accessed through the use of interfaces.

It should be noted that although Java incorporates these object-oriented features, it adheres to basic OOP principles such as encapsulation, inheritance, polymorphism, and abstraction The design of Java was influenced l by the desire for the desire for backward compatibility with procedural programming.

Que8.Answer:
Java offers many advantages over abstraction, e.g.

1. **Simplifies complexity**: Abstraction helps simplify complex systems by focusing on important features and hiding unnecessary information. It allows you to represent real-world objects, systems, or concepts in a flexible and logical way. By abstracting away low-level implementation details, you can work with higher-level concepts, making the code easier to read, understand, and maintain.

2. **Encapsulates Implementation**: Abstraction allows you to hold the implementation information of a class or module. By defining abstract classes, interfaces, and methods, you can provide a high-level contract or blueprint for functionality without exposing the underlying functionality. This separation between abstraction and implementation encourages modularity, flexibility, and maintainability.

3. **Enables flexibility and extensibility**: Abstraction enables flexibility and extensibility in software architecture. By coding against abstract classes or interfaces instead of concrete implementations, you can easily change or add different implementations without impacting the code that uses the abstractions This allows for code reusability, adaptability, and simple changes or enhancements to the system.

4. **Supports polymorphism**: Abstraction is closely related to polymorphism, which is a fundamental principle of resource management. By planning against abstract classes or interfaces, you can write code that is agnostic to a particular implementation and work with different objects based on the same abstraction. This allows the code to be modified, reused, and adapted.

Que 9.Answer:
Abstraction is a fundamental concept in object-oriented programming that involves simplifying complex systems by focusing on essential aspects while hiding unnecessary details. It allows you to represent real-world entities, systems, or concepts in a simplified and understandable manner.
Here's an example to explain abstraction:
Suppose we want to model a Zoo in a Java program. We can start by defining an abstract class called **Aeroplane** that represents the common characteristics and behaviours of all animals in the zoo:

```
abstract class Aeroplane {
   protected String name;

   public Aeroplane (String name) {
      this.name = name;
   }

   public abstract void makeSound();

   public void travel() {
      System.out.println(name + " is traveling.");
   }
}
```

Que10 .Answer:

***A* final class in Java has the following characteristics:**

1. **Inheritance Restriction**: A final class cannot be subclasses or extended by any other class. This means that it cannot serve as a superclass for any other class. Attempting to extend a final class will result in a compilation error.
2. **Method Overriding Restriction**: All methods within a final class are implicitly final as well. This means that methods in a final class cannot be overridden by subclasses.
3. **Immutable Behaviour**: A final class is often used to create immutable objects, where the state of the object cannot be changed after it is created. By making the class final, it ensures that no subclass can modify its behavior or state.
4. **Performance Optimization**: Since a final class cannot be extended, the Java compiler can make certain optimizations. For example, it can inline method calls and perform other performance-related optimizations that are not possible with non-final classes.