Assignment 3 Solution | JAVA

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

Solution Code:

```
package in.ineuron.pptAssignmentJAVA_03;

import java.util.Scanner;

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) {
		balance += amount;
		System.out.println(amount + " deposited successfully.");
	}

	public void withdraw(double amount) {
		if (balance >= amount) {
			balance -= amount;
			System.out.println(amount + " withdrawn successfully.");
		} else {
			System.out.println("Insufficient balance.");
		}
	}
}
```

```
public class BankingSystem_1 {
    @SuppressWarnings("resource")
    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("Bank account created successfully!");
        System.out.println("Account holder name: " +
        account.getAccountHolderName());

        while (true) {
            System.out.println("\n1. Check Balance");
            System.out.println("2. Deposit");
            System.out.println("3. Withdraw");
            System.out.println("4. Exit");
            System.out.print("Enter your choice (1-4): ");
            int choice = scanner.nextInt();

            switch (choice) {
            case 1:
                System.out.println("Account balance: " + account.getBalance());
                break;
            case 2:
                System.out.print("Enter the amount to deposit: ");
                double depositAmount = scanner.nextDouble();
                account.deposit(depositAmount);
                break;
            case 3:
                System.out.print("Enter the amount to withdraw: ");
                double withdrawAmount = scanner.nextDouble();
                account.withdraw(withdrawAmount);
                break;
            case 4:
                System.out.println("Thank you for using the banking system.");
                System.exit(0);
            default:
                System.out.println("Invalid choice. Please enter a number from 1 to
                4.");
            }
        }
    }
}
```

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

Solution Code:

```
package in.ineuron.pptAssignmentJAVA_03;

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

class Cat extends Animal {
	@Override
	public void makeSound() {
		System.out.println("Meow! The cat purrs.");
	}

	public void scratch() {
		System.out.println("The cat scratches.");
	}
}

public class AnimalTest_2 {
	public static void main(String[] args) {
		Animal animal = new Animal();
		animal.makeSound(); // Output: The animal makes a sound.

		Cat cat = new Cat();
		cat.makeSound(); // Output: Meow! The cat purrs.
		cat.scratch(); // Output: The cat scratches.

		// Upcasting
		Animal anotherAnimal = new Cat();
		anotherAnimal.makeSound(); // Output: Meow! The cat purrs.

		// Downcasting
		Cat anotherCat = (Cat) anotherAnimal;
		anotherCat.scratch(); // Output: The cat scratches.
	}
}
```

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

Solution Code:

```
package in.ineuron.pptAssignmentJAVA_03;

class Shape {
	public void draw() {
		System.out.println("Drawing a shape.");
	}
}

class Circle extends Shape {
	@Override
	public void draw() {
		System.out.println("Drawing a circle.");
	}
}

class Rectangle extends Shape {
	@Override
	public void draw() {
		System.out.println("Drawing a rectangle.");
	}
}

class Triangle extends Shape {
	@Override
	public void draw() {
		System.out.println("Drawing a triangle.");
	}
}

public class PolymorphismExample_2 {
	public static void main(String[] args) {
		Shape shape1 = new Circle();
		Shape shape2 = new Rectangle();
		Shape shape3 = new Triangle();

		shape1.draw(); // Output: Drawing a circle.
		shape2.draw(); // Output: Drawing a rectangle.
		shape3.draw(); // Output: Drawing a triangle.
	}
}
```

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

Solution Code:

```
package in.ineuron.pptAssignmentJAVA_03;

public class PolymorphismExample_4 {
	public void display(String message) {
		System.out.println("Displaying message: " + message);
	}

	public void display(String message, int times) {
		for (int i = 0; i < times; i++) {
			System.out.println("Displaying message: " + message);
		}
	}

	public void display(double number) {
		System.out.println("Displaying number: " + number);
	}

	public static void main(String[] args) {
		PolymorphismExample_4 polymorphism = new PolymorphismExample_4();

		polymorphism.display("Hello!"); // Output: Displaying message: Hello!
		polymorphism.display("Hi!", 3);
			// Output: Displaying message: Hi! (printed 3 times)
		polymorphism.display(3.14); // Output: Displaying number: 3.14
	}
}
```

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

Solution Code:

To achieve loose coupling in Java using object-oriented programming (OOP) concepts, you can employ various techniques and design patterns. Here are a few key approaches:

1. **Encapsulation**: Encapsulation is a fundamental concept in OOP that promotes loose coupling. It involves bundling data and methods together within a class and exposing only necessary interfaces to the outside world. By encapsulating the internal details of a class, you reduce dependencies on its implementation, allowing for easier modifications and updates without affecting other parts of the system.

2. **Abstraction**: Abstraction focuses on hiding implementation details and providing a simplified and generalized interface. By designing classes and interfaces based on abstractions, you can reduce dependencies on specific implementations and create loosely coupled systems. Clients interact with objects through abstract interfaces, unaware of the underlying implementation.

3. **Interface-based programming**: Programming to interfaces instead of concrete implementations is another way to achieve loose coupling. By defining interfaces that provide a contract for behavior, you can write code that depends on the interface rather than a specific implementation. This allows for easy substitution of different implementations without affecting the clients that use the interface.

4. **Dependency Injection (DI)**: Dependency Injection is a design pattern that promotes loose coupling by externalizing object creation and dependency management. Instead of creating dependent objects within a class, dependencies are "injected" from external sources. This approach reduces the direct coupling between classes and enables easier testing, maintainability, and flexibility in the system.

5. **Observer pattern**: The Observer pattern allows objects to communicate with each other without strong dependencies. In this pattern, there is a subject (or observable) object that maintains a list of dependent objects (observers) and notifies them of any changes. This way, the subject and observers are loosely coupled, as the subject doesn't need to know the specific details of its observers, and observers can be added or removed without impacting the subject.

By applying these principles and techniques, you can achieve loose coupling in your Java codebase. This leads to more modular, maintainable, and flexible systems that are easier to understand, test, and extend.

💡 Q6. What is the benefit of encapsulation in java?

Answer:

Encapsulation in Java provides several benefits that contribute to the development of robust and maintainable code. Here are 10 key advantages of encapsulation:

1. **Data hiding**: Encapsulation allows you to hide the internal details of a class from other parts of the program. This prevents direct access to sensitive data, ensuring that it is only accessed and modified through controlled methods, thereby maintaining data integrity and security.

2. **Modularity**: Encapsulation promotes modularity by organizing code into smaller, self-contained units (classes). Each class encapsulates its own data and behavior, making it easier to understand, test, and modify without affecting other parts of the system.

3. **Code reusability**: Encapsulation enables code reusability by providing well-defined interfaces through which other objects interact with a class. This allows for the creation of reusable components that can be easily integrated into different parts of an application.

4. **Code maintenance and flexibility**: Encapsulation facilitates code maintenance and flexibility. By encapsulating implementation details, changes made to one part of a class do not impact other parts. This minimizes the risk of introducing bugs and allows for easier updates, modifications, and enhancements to the codebase.

5. **Encapsulation of complex operations**: Encapsulation allows you to encapsulate complex operations within a class, exposing a simple and easy-to-use interface to the outside world. This simplifies the usage of complex functionality and reduces the cognitive load on developers working with the class.

6. **Enhances code readability**: Encapsulation improves code readability by making the purpose and behavior of a class more explicit. The public interface of a class acts as a documentation of its functionality, making it easier for other developers to understand and use the class.

7. **Facilitates information hiding:** Encapsulation supports information hiding, allowing you to control the visibility and accessibility of class members. By hiding unnecessary implementation details, you can reduce complexity and make code easier to understand and maintain.

8. **Improves code organization:** Encapsulation helps in organizing code by grouping related data and behavior into cohesive units (classes). This logical organization enhances code structure and makes it easier to navigate and manage the codebase.

9. **Enables concurrent development:** Encapsulation enables multiple developers to work concurrently on different parts of a program. As long as the public interface of a class remains consistent, developers can independently work on their respective components without interfering with each other.

10. **Facilitates debugging and troubleshooting**: Encapsulation simplifies debugging and troubleshooting processes. By encapsulating data and behavior within classes, issues can be isolated to specific units, making it easier to identify and resolve problems without affecting the rest of the system.

Overall, encapsulation in Java offers numerous benefits that contribute to the development of more maintainable, reusable, and modular code, leading to increased productivity and code quality.

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

Answer:

No, Java is not considered a 100% object-oriented programming (OOP) language. While Java is primarily based on object-oriented principles, there are a few factors that prevent it from being considered 100% pure OOP. Here are a few reasons why:

1. **Primitive Data Types**: Java includes primitive data types such as `int`, `double`, `boolean`, etc., which are not objects. These primitive types are not instances of classes and do not have associated methods or properties. However, Java provides wrapper classes (e.g., `Integer`, `Double`, `Boolean`) to convert primitive types into objects, allowing them to be treated as objects when needed.

2. **Static Members and Methods**: Java supports static members (variables and methods) that belong to the class itself rather than specific instances of the class. Static members are not associated with objects and can be accessed without creating an instance. Since they are not tied to object instances, static members do not adhere to the principles of pure OOP.

3. **Procedural Programming:** Java allows procedural programming constructs in addition to OOP. It includes features like control statements (`if`, `for`, `while`), which are not exclusive to OOP. These constructs enable procedural-style programming alongside OOP concepts, offering flexibility in coding approaches.

4. **Lack of Multiple Inheritance:** Java does not support multiple inheritance, where a class can inherit from multiple parent classes. This is a design decision made to prevent complexities and conflicts arising from diamond inheritance issues. However, Java supports multiple interface implementation, which provides a form of multiple inheritance through interface inheritance.

While Java is not strictly 100% pure OOP, it is considered a predominantly object-oriented programming language. The language incorporates OOP principles extensively and provides a robust framework for building object-oriented applications.

💡 Q8.What are the advantages of abstraction in java?

Answer:

Abstraction in Java provides several advantages that contribute to the development of efficient and maintainable code. Here are the key advantages of abstraction in Java:

1. **Simplified Complexity**: Abstraction allows you to represent complex systems or concepts at a higher level of abstraction. By focusing on essential features and hiding unnecessary details, abstraction simplifies the understanding and usage of complex systems, making the code more manageable and less prone to errors.

2. **Modularity and Code Organization**: Abstraction promotes modularity by dividing a system into smaller, self-contained units. Each module encapsulates its own functionality and abstracts away implementation details, allowing developers to work on individual modules independently. This improves code organization, readability, and maintainability.

3. **Code Reusability:** Abstraction facilitates code reusability. By defining abstract classes and interfaces, you can create reusable components that provide well-defined interfaces to interact with. These components can be easily integrated into different parts of an application, saving development time and effort.

4. **Flexibility and Extensibility:** Abstraction allows for flexibility and extensibility in code. By programming to interfaces and abstract classes rather than concrete implementations, you can introduce new implementations or modify existing ones without affecting the code that depends on them. This reduces the impact of changes and improves the overall flexibility of the system.

5. **Loose Coupling:** Abstraction promotes loose coupling by decoupling the implementation details of a class or module from its usage. By interacting with objects through abstract interfaces, dependencies on specific implementations are reduced, resulting in code that is easier to maintain, test, and extend.

6. **Encapsulation of Implementation**: Abstraction encapsulates the implementation details of a class, exposing only necessary interfaces to the outside world. This provides a clear separation between how an object behaves (interface) and how it is implemented (implementation), allowing for changes in implementation without affecting the clients that use the interface.

7. **Enhanced Maintainability:** Abstraction improves code maintainability by allowing changes to be localized to specific modules or components. When implementation details are abstracted away, modifications or updates can be made within the scope of a module without affecting other parts of the system. This makes the codebase more maintainable and minimizes the risk of introducing bugs or unintended side effects.

8. **Polymorphism and Dynamic Binding:** Abstraction enables the use of polymorphism and dynamic binding in Java. By programming to abstract types (interfaces or abstract classes), you can write code that can work with different implementations interchangeably. This

promotes code flexibility, modularity, and allows for runtime determination of the specific implementation to be used.

Overall, abstraction in Java provides numerous advantages that contribute to code organization, reusability, flexibility, and maintainability. It allows developers to focus on essential concepts and hide unnecessary complexities, leading to more efficient and robust software development.

💡 Q9.What is an abstraction explained with an Example?

Solution Code:

Abstraction is a fundamental concept in object-oriented programming (OOP) that allows you to represent complex systems or concepts at a higher level of simplicity. It involves hiding unnecessary details and focusing on the essential features, making it easier to understand and work with the system.

To explain abstraction with an example, let's consider the concept of a "Vehicle" in a transportation system. A vehicle can be a car, a bike, or a truck. Each of these vehicles has certain common characteristics and behaviors, such as moving, accelerating, and stopping.

Using abstraction, we can create an abstract class or interface called "Vehicle" that defines these common features and behaviors without going into the specific implementation details. It serves as a blueprint for all types of vehicles in the system.

Here's an example of how abstraction can be used to create an abstract class "Vehicle" in Java:

```
abstract class Vehicle {
   // Abstract methods defining common behaviors
   public abstract void move();
   public abstract void accelerate();
   public abstract void stop();
}

class Car extends Vehicle {
   @Override
   public void move() {
    System.out.println("Car is moving.");
   }

   @Override
   public void accelerate() {
      System.out.println("Car is accelerating.");
   }

   @Override
   public void stop() {
```

```
        System.out.println("Car has stopped.");
    }
}

class Bike extends Vehicle {
    @Override
    public void move() {
        System.out.println("Bike is moving.");
    }

    @Override
    public void accelerate() {
        System.out.println("Bike is accelerating.");
    }

    @Override
    public void stop() {
        System.out.println("Bike has stopped.");
    }
}
```

In this example, the abstract class "Vehicle" defines three abstract methods: `move()`, `accelerate()`, and `stop()`. These methods represent the common behaviors of all vehicles, but the specific implementation is left to the subclasses.

The subclasses `Car` and `Bike` extend the abstract class "Vehicle" and provide their own implementation for the abstract methods. They define how a car and a bike move, accelerate, and stop.

By using abstraction, we can now work with vehicles at a higher level of abstraction without worrying about the specific details of each vehicle type. For example, we can create instances of the `Car` and `Bike` classes and call their methods without needing to know the internal implementation details of those methods.

```
public class AbstractionExample {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.move();        // Output: Car is moving.
        car.accelerate();  // Output: Car is accelerating.
        car.stop();        // Output: Car has stopped.

        Vehicle bike = new Bike();
        bike.move();       // Output: Bike is moving.
        bike.accelerate(); // Output: Bike is accelerating.
        bike.stop();       // Output: Bike has stopped.
    }
}
```

In this example, we can treat both the `Car` and `Bike` objects as instances of the abstract class "Vehicle" and call their common methods without needing to know the specific implementation details of each vehicle type. This simplifies the code and allows for easy extensibility and flexibility in the transportation system.

💡 Q10.What is the final class in Java?

Answer:

In Java, the `final` keyword can be applied to classes, methods, and variables. When applied to a class, it denotes that the class cannot be subclassed or extended further. Thus, a `final` class is a class that cannot have any subclasses.

Here are a few key points regarding final classes in Java:

1. **Inheritance restriction**: When a class is declared as `final`, it cannot be extended by any other class. This means that no other class can be derived from a final class. It ensures that the final class cannot be overridden or modified in any way.

2. **Immutable behavior**: Final classes are often used to create immutable objects, where the state of the object cannot be changed once it is created. By combining a final class with final member variables and methods, you can achieve immutability in Java.

3. **Performance optimization**: Since a final class cannot be extended, the Java compiler can perform certain optimizations. For example, it may inline method calls and eliminate dynamic dispatch, resulting in improved performance.

4. **Security considerations:** Final classes can be useful in security-sensitive contexts. By preventing subclassing, it ensures that the behavior of the class cannot be modified, which can be important in scenarios where tampering or malicious modifications need to be avoided.

5. **Design intent and clarity**: Marking a class as final can also communicate design intent and prevent unintended subclassing. It serves as a clear indication to other developers that the class is not meant to be extended.

Here's an example that demonstrates the usage of a final class in Java:

```
final class MyFinalClass {
   // Class implementation
}

// Error: Cannot inherit from final class 'MyFinalClass'
class Subclass extends MyFinalClass {
   // Subclass implementation
}
```

In this example, the class `MyFinalClass` is declared as `final`, indicating that it cannot be extended. When attempting to create a subclass `Subclass` that extends `MyFinalClass`, a compilation error occurs, as subclassing a final class is not allowed.

By using the `final` keyword on a class, you can ensure that the class cannot be subclassed, providing better control over the class's behavior, security, and design.