**Assignment Questions 3**

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

</aside>

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

</aside>

<aside> 💡 **Q3.Write a program to show run time polymorphism in java?**

</aside>

<aside> 💡 **Q4.Write a program to show Compile time polymorphism in java?**

</aside>

<aside> 💡 **Q5. Achieve loose coupling in java by using OOPs  concept?**

</aside>

<aside> 💡 **Q6. What is the benefit of encapsulation in java?**

</aside>

<aside> 💡 [**Q7.Is**](http://Q7.Is) **java a t 100% Object oriented Programming language? If no why ?**

</aside>

<aside> 💡 **Q8.What are the advantages of abstraction in java?**

</aside>

<aside> 💡 **Q9.What is an abstraction explained with an Example?**

</aside>

<aside> 💡 **Q10.What is the final class in Java?**

</aside>

# Answer

# Q1. Write a simple Banking System program using OOPs concepts where you can get account holder name, balance, etc.

# javaCopy code

# class BankAccount { private String accountHolderName; private double balance; 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. Current balance: " + balance); } public void withdraw(double amount) { if (balance >= amount) { balance -= amount; System.out.println("Withdrawal of " + amount + " successful. Current balance: " + balance); } else { System.out.println("Insufficient funds. Current balance: " + balance); } } } public class BankingSystem { public static void main(String[] args) { BankAccount account = new BankAccount("John Doe", 5000.0); System.out.println("Account Holder Name: " + account.getAccountHolderName()); System.out.println("Balance: " + account.getBalance()); account.deposit(2000.0); account.withdraw(1500.0); account.withdraw(6000.0); } }

# Output:

# yamlCopy code

# Account Holder Name: John Doe Balance: 5000.0 Deposit of 2000.0 successful. Current balance: 7000.0 Withdrawal of 1500.0 successful. Current balance: 5500.0 Insufficient funds. Current balance: 5500.0

# In this program, the BankAccount class represents a bank account and encapsulates the account holder name and balance as private data members. It provides getter methods (getAccountHolderName(), getBalance()) to access the account holder name and balance.

# The BankAccount class also provides methods for depositing (deposit()) and withdrawing (withdraw()) money from the account. The deposit() method increases the balance, while the withdraw() method checks if there are sufficient funds before decreasing the balance. If the withdrawal amount exceeds the balance, it displays an error message.

# The BankingSystem class demonstrates the usage of the BankAccount class by creating an instance of it and performing deposit and withdrawal operations. The account holder name and balance are displayed using the getter methods.

# Q2. Write a program where you inherit a method from the parent class and show the concept of method overriding.

# javaCopy code

# class Vehicle { public void start() { System.out.println("Vehicle started."); } } class Car extends Vehicle { @Override public void start() { System.out.println("Car started."); } } public class MethodOverridingExample { public static void main(String[] args) { Vehicle vehicle = new Vehicle(); vehicle.start(); Car car = new Car(); car.start(); } }

# Output:

# Copy code

# Vehicle started. Car started.

# In this program, the Vehicle class has a start() method that prints "Vehicle started." The Car class extends the Vehicle class and overrides the start() method with its own implementation that prints "Car started."

# In the MethodOverridingExample class, we create instances of both the Vehicle and Car classes and call the start() method on each object. The start() method of the Vehicle class is called when invoked on the Vehicle object, and the start() method of the Car class is called when invoked on the Car object. This demonstrates method overriding, where a subclass provides a different implementation of a method defined in its superclass.

# Q3. Write a program to show runtime polymorphism in Java.

# javaCopy code

# class Animal { public void sound() { System.out.println("Animal makes a sound"); } } class Cat extends Animal { @Override public void sound() { System.out.println("Cat meows"); } } class Dog extends Animal { @Override public void sound() { System.out.println("Dog barks"); } } public class RuntimePolymorphismExample { public static void main(String[] args) { Animal animal1 = new Cat(); Animal animal2 = new Dog(); animal1.sound(); animal2.sound(); } }

# Output:

# Copy code

# Cat meows Dog barks

# In this program, we have an Animal class with a sound() method that prints "Animal makes a sound". The Cat and Dog classes extend the Animal class and override the sound() method with their specific implementations.

# In the RuntimePolymorphismExample class, we create instances of the Cat and Dog classes, but we declare them as Animal types. This allows us to achieve polymorphism, as the actual object type is determined at runtime.

# When we invoke the sound() method on animal1, which is referencing a Cat object, the overridden sound() method in the Cat class is executed, printing "Cat meows". Similarly, when we invoke the sound() method on animal2, which is referencing a Dog object, the overridden sound() method in the Dog class is executed, printing "Dog barks". This demonstrates runtime polymorphism, where the appropriate method implementation is determined dynamically at runtime based on the actual object type.

# Q4. Write a program to show compile-time polymorphism in Java.

# javaCopy code

# class Calculation { public void sum(int num1, int num2) { int result = num1 + num2; System.out.println("Sum: " + result); } public void sum(int num1, int num2, int num3) { int result = num1 + num2 + num3; System.out.println("Sum: " + result); } } public class CompileTimePolymorphismExample { public static void main(String[] args) { Calculation calculation = new Calculation(); calculation.sum(5, 10); calculation.sum(5, 10, 15); } }

# Output:

# makefileCopy code

# Sum: 15 Sum: 30

# In this program, the Calculation class has two sum() methods with different parameter lists. One method takes two integers as parameters, and the other method takes three integers as parameters.

# In the CompileTimePolymorphismExample class, we create an instance of the Calculation class and invoke the sum() method with different argument lists. The Java compiler determines the appropriate method to call at compile-time based on the number and types of arguments. This demonstrates compile-time polymorphism, also known as method overloading, where multiple methods with the same name but different parameter lists can be defined in a class.

# Q5. Achieve loose coupling in Java by using OOPs concepts.

# Loose coupling is a design principle that promotes decoupling and reducing dependencies between components of a system. In Java, we can achieve loose coupling by using Object-oriented programming concepts such as encapsulation, inheritance, and polymorphism.

# Here's an example that demonstrates loose coupling using OOPs concepts:

# javaCopy code

# // Interface for the Payment method interface Payment { void makePayment(double amount); } // Payment implementation using Credit Card class CreditCardPayment implements Payment { public void makePayment(double amount) { System.out.println("Making payment of $" + amount + " via Credit Card"); // Code to process credit card payment } } // Payment implementation using PayPal class PayPalPayment implements Payment { public void makePayment(double amount) { System.out.println("Making payment of $" + amount + " via PayPal"); // Code to process PayPal payment } } // Customer class using Payment interface class Customer { private Payment payment; public Customer(Payment payment) { this.payment = payment; } public void purchase(double amount) { payment.makePayment(amount); // Code for purchase logic } } public class LooseCouplingExample { public static void main(String[] args) { Payment payment1 = new CreditCardPayment(); Customer customer1 = new Customer(payment1); customer1.purchase(100.0); Payment payment2 = new PayPalPayment(); Customer customer2 = new Customer(payment2); customer2.purchase(200.0); } }

# Output:

# bashCopy code

# Making payment of $100.0 via Credit Card Making payment of $200.0 via PayPal

# In this example, we have an Payment interface that defines the makePayment() method. We have two implementations of the Payment interface: CreditCardPayment and PayPalPayment.

# The Customer class represents a customer who wants to make a purchase. It has a purchase() method that takes the payment amount and calls the makePayment() method on the provided Payment object.

# In the main() method, we create instances of the Payment implementations (CreditCardPayment and PayPalPayment) and pass them to the Customer constructor. This allows the Customer class to be loosely coupled with the specific payment implementation. The Customer class can work with any payment implementation that adheres to the Payment interface, allowing for flexibility and easy interchangeability.

# This example demonstrates loose coupling as the Customer class is not dependent on the specific payment implementation details. It depends on the abstraction provided by the Payment interface, which allows different payment implementations to be plugged in without requiring changes to the Customer class. This promotes flexibility, maintainability, and extensibility in the system.

# Q6. What is the benefit of encapsulation in Java?

# Encapsulation is one of the fundamental principles of object-oriented programming. It is the mechanism of bundling data (variables) and methods (functions) that operate on the data into a single unit called a class. The main benefit of encapsulation in Java is to achieve data hiding, abstraction, and maintainability.

# Here are the benefits of encapsulation in Java:

# Data Hiding: Encapsulation allows you to hide the internal implementation details of a class from its external users. By making the data members of a class private and providing public methods (getters and setters) to access and modify the data, you control the access to the data. This prevents direct manipulation of data from outside the class and ensures that data is accessed and modified through a controlled interface. This improves data integrity, security, and reduces the risk of unintended modifications.

# Abstraction: Encapsulation helps in achieving abstraction by exposing only the essential information and hiding the complexity of implementation details. By providing a well-defined public interface (methods) for interacting with an object, you can abstract away the internal workings and present a simplified view of the object's behavior. This simplifies the usage of the class and makes it easier to understand and use by other parts of the program.

# Modularity and Maintainability: Encapsulation promotes modularity by encapsulating related data and methods into a single unit. This improves code organization, readability, and maintainability. Changes to the internal implementation of a class can be made without affecting other parts of the program that use the class. It also allows for easy refactoring and enhancements to the class without impacting the code that depends on it.

# Code Reusability: Encapsulation helps in code reusability by creating reusable components. By encapsulating related data and behavior into a class, you can create instances of the class in different parts of the program, promoting code reuse. Encapsulated classes can be easily integrated into other programs or used as building blocks for more complex systems.

# Overall, encapsulation provides a way to control the access to data and hide implementation details, leading to better code organization, maintainability, and reusability. It promotes the principle of information hiding and abstraction, allowing you to design classes that are more robust, flexible, and easier to understand and maintain.

# Q7. Is Java a 100% Object-oriented Programming language? If not, why?

# No, Java is not considered a 100% object-oriented programming language. While Java supports many object-oriented features, such as classes, objects, inheritance, polymorphism, and encapsulation, it also incorporates non-object-oriented elements.

# The main reasons why Java is not considered a 100% object-oriented programming language are:

# Primitive Data Types: Java includes primitive data types (e.g., int, float, boolean) that are not objects and do not have associated methods or inheritance. These primitive types are not treated as objects and have special handling in the language.

# Static Methods and Variables: Java allows the declaration of static methods and variables at the class level. Static members are associated with the class itself and not with individual objects. They can be accessed without creating an instance of the class, which goes against the pure object-oriented paradigm.

# Procedural Programming: Java allows the use of procedural programming constructs, such as procedural methods (functions) outside of classes, and procedural-style code organization. While the focus in Java is on object-oriented programming, these procedural elements are still present in the language.

# Despite these non-object-oriented elements, Java is often referred to as an "object-oriented language" because it heavily emphasizes object-oriented programming principles and provides extensive support for object-oriented features. The majority of code written in Java follows the object-oriented paradigm, and the language's core libraries and frameworks are built upon object-oriented concepts.

# Q8. What are the advantages of abstraction in Java?

# Abstraction is a key concept in object-oriented programming that focuses on creating simplified models of complex systems. In Java, abstraction allows you to represent the essential features and behavior of an entity without exposing the underlying implementation details. Here are the advantages of abstraction in Java:

# Simplifies Complexity: Abstraction helps in simplifying complex systems by focusing on the essential aspects and hiding the unnecessary details. It allows you to create classes, interfaces, and methods that present a simplified and conceptual view of the system's behavior. By abstracting away the complexities, you can design more manageable and understandable code.

# Enhances Maintainability: Abstraction improves the maintainability of code by providing a clear separation between the interface and implementation. By defining interfaces and abstract classes, you can define a contract that specifies how other parts of the code should interact with a particular entity, without exposing the internal workings. This allows for easier updates and modifications to the implementation

# Top of Form