ASSIGNMENT-3(JAVA CORE MODULE)

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

Ans-

```
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 Rs." + amount + "
successful.");
    }

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

```
public class BankingSystem {
    public static void main(String[] args) {
        // Create a bank account
        BankAccount account = new BankAccount("Amit Ranjan",
1000.0);

        // Display account holder name and balance
        System.out.println("Account Holder Name: " +
account.getAccountHolderName());
        System.out.println("Balance: Rs." + account.getBalance());

        // Perform a deposit and display the updated balance
        account.deposit(500.0);
        System.out.println("Updated Balance: Rs." +
account.getBalance());

        // Perform a withdrawal and display the updated balance
        account.withdraw(200.0);
        System.out.println("Updated Balance: Rs." +
account.getBalance());

        // Attempt to withdraw more than the available balance
        account.withdraw(1500.0);
    }
}
```

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

Ans-

```
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 AnimalSounds {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Dog dog = new Dog();
        Cat cat = new Cat();

        animal.sound();  // Output: Animal makes a sound.
        dog.sound();     // Output: Dog barks.
        cat.sound();     // Output: Cat meows.
    }
}
```

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

Ans-
```
class Animal {

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

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

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

```
    }
}

public class RuntimePolymorphism {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();

        animal1.makeSound();  // Output: Dog barks.
        animal2.makeSound();  // Output: Cat meows.
    }
}
```

Q4. Write a program to show Compile time polymorphism in Java.

Ans-

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

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

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

    public static void main(String[] args) {
        CompileTimePolymorphism obj = new CompileTimePolymorphism();

        int sum1 = obj.add(5, 10);                  // Invokes the
add(int, int) method
        double sum2 = obj.add(2.5, 3.7);            // Invokes the
add(double, double) method
        String concatenatedStr = obj.add("Hello", " World!");  //
Invokes the add(String, String) method

        System.out.println("Sum of integers: " + sum1);
        System.out.println("Sum of doubles: " + sum2);
```

```
        System.out.println("Concatenated string: " +
concatenatedStr);
    }
}
```

Q5. Achieve loose coupling in Java by using OOPs Concept?

Ans- To achieve loose coupling in Java using object-oriented programming (OOP) concepts, you can employ several techniques. Here are a few common approaches:

1. Abstraction and Encapsulation:

- Encapsulate related data and behaviors within classes by using access modifiers (e.g., private, protected) to control access to class members.

- Expose only necessary methods and hide internal implementation details, promoting information hiding and reducing dependencies.

2. Inheritance and Polymorphism:

- Use inheritance to create a hierarchical structure of classes, where derived classes inherit properties and behaviors from base classes.

- Leverage polymorphism to allow objects of different classes to be treated interchangeably through method overriding and method overloading.

3. Dependency Inversion Principle (DIP):

- Program to interfaces or abstract classes rather than concrete implementations.

- Declare variables, method parameters, and return types using interfaces or abstract classes, allowing for flexibility and easy substitution of implementations.

4. Dependency Injection (DI):

- Use dependency injection frameworks or manual dependency injection to decouple the creation and management of object dependencies.

- Inject dependencies through constructor injection, setter injection, or method injection, reducing direct dependencies and promoting easier testing and maintainability.

5. Interface Segregation Principle (ISP):

- Design interfaces that are specific to the requirements of clients, avoiding bloated and unnecessary interfaces.

- Split large interfaces into smaller, cohesive interfaces to minimize dependencies and ensure that clients only depend on what they actually need.

6. Favor Composition over Inheritance:

- Use composition to assemble objects by combining multiple classes rather than relying solely on inheritance.

- Encapsulate related objects as member variables within a class, reducing the tight coupling between classes and allowing for more flexibility and modularity.

By applying these principles and techniques, you can achieve loose coupling in your Java code. Loose coupling promotes flexibility, modifiability, and easier maintenance of code by reducing dependencies and promoting better encapsulation and abstraction.

Q6. What is the benefit of encapsulation in Java?

Ans- Encapsulation is one of the fundamental principles of object-oriented programming (OOP) and plays a crucial role in Java. It refers to the bundling of data (attributes) and methods (behavior) within a class, and controlling access to them through access modifiers. Encapsulation offers several benefits in Java:

1. Data Hiding: Encapsulation helps hide the internal implementation details of a class and allows access to data only through well-defined methods (getters and setters). It prevents direct access to the class's internal state, ensuring data integrity and security.

2. Information Hiding: By encapsulating data and exposing only necessary methods, encapsulation promotes information hiding. It allows the class to define a public interface that hides internal complexities, reducing complexity and making the code easier to understand and maintain.

3. Modularity and Maintainability: Encapsulation promotes modularity by organizing related data and behavior into a single unit (class). This makes it easier to manage and maintain the codebase, as changes to the internal implementation of a class do not affect other parts of the program that use the class's public interface.

4. Code Flexibility: Encapsulation provides the ability to modify the internal implementation of a class without affecting other parts of the program that use the class. By maintaining a consistent public interface, you can update or optimize the class's internals while ensuring compatibility with existing code.

5. Code Reusability: Encapsulation facilitates code reuse by creating self-contained objects. These objects can be used in various contexts, promoting code modularity and reducing duplication.

6. Data Validation and Control: Encapsulation allows for the enforcement of data validation rules and control over how data is accessed and modified. By encapsulating data within methods, you can apply validation checks, perform calculations, or trigger other actions whenever data is accessed or modified.

7. Access Control: Encapsulation enables the use of access modifiers (e.g., private, protected, public) to control the visibility and accessibility of attributes and methods. This helps enforce data encapsulation, ensuring that only authorized code can access or modify the encapsulated data.

Overall, encapsulation in Java brings many advantages, including enhanced data protection, improved code organization and maintenance, code flexibility, and code reuse. It fosters better design practices, encapsulates complexity, and promotes more robust and maintainable software development.

Q7. Is Java a 100% Object Oriented Programming Language? If now why?

Ans- Java is often considered a predominantly object-oriented programming (OOP) language, but it is not 100% object-oriented. The main reason is that Java supports both object-oriented programming and procedural programming paradigms.

While Java embraces core OOP concepts such as encapsulation, inheritance, polymorphism, and abstraction, there are a few aspects that deviate from pure OOP principles:

1. Primitive Data Types: Java includes primitive data types like `int`, `boolean`, `double`, etc., which are not objects and do not have associated methods or properties. These primitive types are not treated as objects in the same way as user-defined classes.

2. Static Members and Methods: Java allows the declaration of static variables and methods that belong to a class rather than instances of the class. Static members are not associated with specific objects and can be accessed without creating an instance. This concept goes against the idea of pure OOP, where everything is an object.

3. Procedural Syntax: Java supports procedural programming syntax as well. It includes control structures like loops (`for`, `while`), conditional statements (`if`, `switch`), and functions outside of class definitions. These procedural elements provide a more procedural programming style alongside OOP.

Despite these deviations, Java is still primarily considered an object-oriented programming language due to its strong emphasis on object-oriented concepts and its ability to model complex systems using classes and objects. The majority of Java programming involves designing and implementing classes, encapsulating data, and utilizing inheritance and polymorphism.

It's worth noting that some programming languages, such as Smalltalk or Eiffel, adhere more strictly to the principles of pure OOP, where everything is treated as an object and there are no primitive types or procedural elements. Java, on the other hand, strikes a balance between object-oriented and procedural programming paradigms, making it a versatile language that can accommodate different programming styles.

Q8. What are the advantages of abstraction in Java?

Q9. What is an abstraction explain with an example.?

Ans- Abstraction is a fundamental concept in Java and object-oriented programming (OOP) in general. It involves simplifying complex systems by modeling them at a higher level of abstraction. Here are some advantages of abstraction in Java:

1. Simplified Complex Systems: Abstraction allows you to focus on essential features and behaviors while hiding unnecessary details. It simplifies the complexity of a system by providing a higher-level view, making it easier to understand and work with.

2. Code Reusability: By creating abstract classes and interfaces, you can define common behaviors and attributes that can be inherited or implemented by multiple classes. This promotes code reuse and reduces duplication, saving development time and effort.

3. Modular and Maintainable Code: Abstraction promotes modular programming by breaking down a system into smaller, manageable units. Each module represents a distinct abstraction, making it easier to develop, test, and maintain individual components without affecting the entire system.

4. Encapsulation of Implementation: Abstraction allows you to hide the internal implementation details of a class or module. By exposing only the necessary interfaces and hiding the underlying implementation, you can protect sensitive information and maintain data integrity.

5. Flexibility and Extensibility: Abstraction provides a framework for defining contracts, specifications, and interfaces. This enables you to introduce new implementations or extend existing ones without affecting the code that relies on the abstracted interface. It promotes flexibility and allows for easy modification and enhancement of the system.

6. Polymorphism: Abstraction plays a vital role in achieving polymorphism, one of the core principles of OOP. By using abstract classes and interfaces, you can define common behaviors that can be implemented differently by different classes. Polymorphism allows objects of different classes to be treated interchangeably, enhancing code flexibility and facilitating code reuse.

7. Improved Collaboration: Abstraction enables better collaboration among developers by providing clear contracts and interfaces. It allows teams to work on different components independently as long as they adhere to the defined abstractions. This promotes parallel development and seamless integration of modules.

8. Maintenance and Debugging: Abstraction facilitates code maintenance and debugging by isolating and encapsulating specific functionalities. Errors or updates in one module can be addressed without affecting other parts of the system. It also enhances code traceability and readability, making it easier to locate and fix issues.

Abstraction is a powerful concept that enables developers to manage complexity, promote code reuse, and design robust and scalable systems. By abstracting away unnecessary details and focusing

on essential characteristics, abstraction provides a high-level perspective and a solid foundation for building modular, maintainable, and extensible Java applications.

Q10. What is the final class in Java?

Ans- In Java, the `final` keyword has multiple usages, one of which is the `final` class. A final class is a class that cannot be subclassed or extended by any other class. Once a class is declared as final, it cannot be inherited by other classes. Here are some key points about final classes:

1. Preventing Inheritance: By declaring a class as final, you explicitly indicate that it should not have any subclasses. It serves as a way to prevent further extension and ensure that the class's behavior remains unchanged.

2. Design Integrity: Final classes are often used when the design or functionality of a class is considered complete and should not be modified or overridden by subclasses. It helps maintain the integrity and consistency of the class's behavior.

3. Efficiency and Security: Final classes can provide certain optimizations and security benefits. Since a final class cannot be subclassed, the compiler can make certain assumptions and perform optimizations during compilation. Additionally, final classes can enhance security by preventing potential vulnerabilities that could arise from modifying or extending the class.
