1. 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) {

if (amount > 0) {

balance += amount;

System.out.println("Deposit of $" + amount + " successful.");

} else {

System.out.println("Invalid deposit amount.");

}

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

System.out.println("Withdrawal of $" + amount + " successful.");

} else {

System.out.println("Invalid withdrawal amount or insufficient balance.");

}

}

}

public class BankingSystem {

public static void main(String[] args) {

// Create a bank account for a customer

BankAccount account = new BankAccount("John Doe", 1000.0);

// Get account information

String accountHolderName = account.getAccountHolderName();

double balance = account.getBalance();

// Display account information

System.out.println("Account Holder Name: " + accountHolderName);

System.out.println("Current Balance: $" + balance);

// Perform some transactions

account.deposit(500.0);

account.withdraw(200.0);

account.withdraw(1500.0);

account.deposit(-100.0);

// Display updated balance

balance = account.getBalance();

System.out.println("Updated Balance: $" + balance);

}

}

Output: Account Holder Name: John Doe

Current Balance: $1000.0

Deposit of $500.0 successful.

Withdrawal of $200.0 successful.

Invalid withdrawal amount or insufficient balance.

Invalid deposit amount.

Updated Balance: $1300.0

2. class ParentClass {

public void printMessage() {

System.out.println("This is the parent class.");

}

}

class ChildClass extends ParentClass {

@Override

public void printMessage() {

System.out.println("This is the child class.");

}

}

public class InheritanceExample {

public static void main(String[] args) {

ParentClass parent = new ParentClass();

parent.printMessage(); // Output: "This is the parent class."

ChildClass child = new ChildClass();

child.printMessage(); // Output: "This is the child class."

ParentClass polymorphicChild = new ChildClass();

polymorphicChild.printMessage(); // Output: "This is the child class."

}

}

Output: This is the parent class.

This is the child class.

This is the child class.

3. class Animal {

public void makeSound() {

System.out.println("Animal is making a sound.");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Dog is barking.");

}

}

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("Cat is meowing.");

}

}

public class RuntimePolymorphismExample {

public static void main(String[] args) {

Animal animal1 = new Dog();

animal1.makeSound(); // Output: "Dog is barking."

Animal animal2 = new Cat();

animal2.makeSound(); // Output: "Cat is meowing."

}

}

Output: Dog is barking.

Cat is meowing.

4. class MathOperations {

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 CompileTimePolymorphismExample {

public static void main(String[] args) {

MathOperations math = new MathOperations();

int sum1 = math.add(5, 10);

System.out.println("Sum 1: " + sum1); // Output: Sum 1: 15

int sum2 = math.add(2, 4, 6);

System.out.println("Sum 2: " + sum2); // Output: Sum 2: 12

double sum3 = math.add(2.5, 3.7);

System.out.println("Sum 3: " + sum3); // Output: Sum 3: 6.2

}

}

Output: Sum 1: 15

Sum 2: 12

Sum 3: 6.2

5. To achieve loose coupling in Java using object-oriented programming concepts, you can employ several techniques and principles. Here are some common practices that promote loose coupling:

1. Encapsulation: Encapsulate the implementation details of a class by providing well-defined interfaces and hiding internal implementation details. This allows other classes to interact with the object through the defined interfaces, without knowing the implementation details.

2. Abstraction: Use abstraction to define interfaces or abstract classes that provide a high-level overview of functionality, without exposing the implementation details. This allows classes to interact based on the defined abstractions, promoting loose coupling between them.

3. Dependency Injection (DI): Use dependency injection to provide dependencies to a class from external sources rather than creating them internally. By injecting dependencies through interfaces, you can easily swap different implementations without affecting the class that relies on them.

4. Inversion of Control (IoC): Apply the principle of inversion of control, where a higher-level component or framework controls the flow of execution and manages the dependencies of the components. This reduces the coupling between components, as they rely on abstractions and let the higher-level component handle the creation and management of dependencies.

5. Interface-based Programming: Program to interfaces rather than concrete implementations. This allows you to replace implementations easily without affecting the classes that depend on them. By depending on interfaces, you achieve loose coupling between components.

6. Dependency Abstraction: Avoid directly referring to concrete classes or specific implementation details in your code. Instead, depend on abstractions, interfaces, or abstract classes. This way, you can substitute different implementations without affecting the dependents.

7. Favor Composition over Inheritance: Use composition to combine multiple classes or components to achieve functionality rather than relying solely on inheritance. Composition allows you to assemble objects and behaviors dynamically, promoting loose coupling by avoiding tight relationships defined by class inheritance hierarchies.

By following these practices, you can achieve loose coupling in your Java code. Loose coupling enhances flexibility, maintainability, and testability by reducing dependencies and promoting modularity and extensibility within your application.

6. Encapsulation is a fundamental concept in object-oriented programming, and it provides several benefits in Java:

1. Data Hiding: Encapsulation allows you to hide the internal data and implementation details of a class. By declaring data members as private, you restrict direct access from outside the class. This ensures that the data can only be accessed and modified through defined methods (getters and setters), providing better control over the data and preventing unauthorized modifications.

2. Access Control: Encapsulation provides access modifiers (such as private, protected, and public) to control the visibility and accessibility of class members. By setting appropriate access modifiers, you can restrict or allow access to specific members based on your intended design. This enhances security and maintains the integrity of the class's internal state.

3. Code Flexibility and Maintainability: Encapsulation allows you to change the internal implementation of a class without affecting other parts of the program. By encapsulating the implementation details, you can modify or refactor the internal structure of the class without breaking the code that uses the class. This promotes code flexibility and makes maintenance and enhancements easier.

4. Code Reusability: Encapsulation promotes code reusability. By encapsulating related data and behavior into a single class, you can create reusable components. These components can be used in different parts of your program, reducing code duplication and improving productivity.

5. Enhanced Readability and Understandability: Encapsulation improves code readability and understandability by providing a clear and consistent interface for interacting with the class. By exposing only the necessary methods and hiding internal details, encapsulation makes it easier to comprehend and use the class. It also helps in reducing complexity and making the code more maintainable.

6. Encourages Good Programming Practices: Encapsulation enforces good programming practices such as information hiding, modularity, and separation of concerns. It promotes the principle of encapsulating related data and behavior into coherent units, making the code more organized and maintainable.

Overall, encapsulation in Java provides benefits such as data protection, controlled access, code flexibility, reusability, improved readability, and adherence to good programming practices. It is a crucial concept that helps in building robust and maintainable software systems.

7. No, Java is not considered a 100% pure object-oriented programming language. While Java is primarily object-oriented, it does incorporate some non-object-oriented elements as well. Here are a few reasons why Java is not classified as a pure object-oriented language:

1. Primitive Data Types: Java includes primitive data types such as `int`, `char`, `boolean`, etc., which are not objects. These primitive types do not have associated methods or inheritance hierarchies like objects do.

2. Static Members: Java allows the use of static methods and static variables. Static members are associated with the class itself rather than with specific instances of the class. They can be accessed without creating an instance of the class, which goes against the principles of object-oriented programming.

3. Procedural Syntax: Java supports procedural programming constructs such as loops (`for`, `while`), conditional statements (`if`, `else`), and methods that are not associated with objects. These procedural elements can be used alongside object-oriented programming constructs.

4. Java API: The Java API (Application Programming Interface) includes a wide range of classes and utility methods that are not purely object-oriented. For example, the `Math` class contains static methods that can be used directly without creating objects.

5. Lack of Multiple Inheritance: Java does not support multiple inheritance, which is a feature of some purely object-oriented languages. In Java, a class can inherit from only one superclass, restricting the use of multiple inheritance and favoring interface-based programming instead.

Despite these factors, Java is predominantly an object-oriented programming language. It follows object-oriented principles such as encapsulation, inheritance, polymorphism, and abstraction. Most code in Java is written in an object-oriented style, and the language provides robust support for object-oriented programming concepts.

8. Abstraction is an important concept in Java and provides several advantages:

1. Simplified Complexity: Abstraction allows you to simplify complex systems by focusing on essential aspects while hiding unnecessary details. It provides a high-level overview of the system's functionality, making it easier to understand and work with. By abstracting away implementation details, you can deal with the system in a more manageable and intuitive manner.

2. Modularity and Maintainability: Abstraction promotes modularity by dividing a system into separate components with well-defined interfaces. Each component can be implemented independently, allowing for easier maintenance and updates. Modularity also facilitates code reusability, as abstract components can be used in different contexts without requiring modifications to the underlying implementation.

3. Code Reusability: Abstraction facilitates code reusability, as abstract classes and interfaces can serve as blueprints for creating multiple derived classes. By defining common methods and properties in an abstract class or interface, you can create reusable components that can be extended or implemented by different classes. This promotes efficient and DRY (Don't Repeat Yourself) coding practices.

4. Encapsulation of Complexity: Abstraction encapsulates the complexity of a system, providing a simplified interface for interacting with the underlying implementation. By encapsulating implementation details within abstract classes or interfaces, you can shield the users of the abstraction from unnecessary complexity and provide them with a clear and concise set of methods and properties to work with.

5. Flexibility and Extensibility: Abstraction allows for flexibility and extensibility in software design. By programming to abstractions (interfaces or abstract classes) rather than concrete implementations, you can easily swap different implementations without affecting the code that depends on them. This flexibility enables you to incorporate changes or add new features to a system without disrupting existing code.

6. Polymorphism: Abstraction enables polymorphism, which is a core principle of object-oriented programming. By defining abstractions through interfaces or abstract classes, you can have multiple implementations of those abstractions. This allows for dynamic method binding at runtime, facilitating code reuse and promoting flexibility in system design.

Overall, abstraction in Java offers advantages such as simplifying complexity, promoting modularity and maintainability, facilitating code reusability, encapsulating complexity, providing flexibility and extensibility, and enabling polymorphism. It is a powerful concept that helps in designing scalable, maintainable, and robust software systems.

9. Abstraction is a concept in object-oriented programming that allows you to create abstract representations of real-world objects or systems. It focuses on defining essential characteristics and behaviors while hiding unnecessary implementation details. Here's an example to help explain abstraction:

Example: Vehicle Abstraction

abstract class Vehicle {

private String registrationNumber;

private String brand;

public Vehicle(String registrationNumber, String brand) {

this.registrationNumber = registrationNumber;

this.brand = brand;

}

public abstract void start();

public abstract void stop();

public String getRegistrationNumber() {

return registrationNumber;

}

public String getBrand() {

return brand;

}

}

class Car extends Vehicle {

public Car(String registrationNumber, String brand) {

super(registrationNumber, brand);

}

@Override

public void start() {

System.out.println("Car started.");

}

@Override

public void stop() {

System.out.println("Car stopped.");

}

}

class Motorcycle extends Vehicle {

public Motorcycle(String registrationNumber, String brand) {

super(registrationNumber, brand);

}

@Override

public void start() {

System.out.println("Motorcycle started.");

}

@Override

public void stop() {

System.out.println("Motorcycle stopped.");

}

}

public class AbstractionExample {

public static void main(String[] args) {

Vehicle car = new Car("ABC123", "Toyota");

car.start(); // Output: Car started.

car.stop(); // Output: Car stopped.

Vehicle motorcycle = new Motorcycle("XYZ789", "Honda");

motorcycle.start(); // Output: Motorcycle started.

motorcycle.stop(); // Output: Motorcycle stopped.

}

}

Output: Car started.

Car stopped.

Motorcycle started.

Motorcycle stopped.

10. In Java, the `final` keyword can be applied to classes, methods, and variables. When applied to a class, it signifies that the class cannot be subclassed, meaning it cannot be extended by other classes.

Here are the key characteristics and implications of a final class in Java:

1. Inheritance Restriction: A final class cannot be subclassed. It prevents other classes from extending it and inheriting its members (methods and variables). This restriction is in place to ensure that the final class's implementation is not modified or overridden by other classes.

2. Preventing Method Overriding: When a class is marked as final, it also implies that its methods cannot be overridden by subclasses. Any attempt to override a final method in a subclass will result in a compilation error.

3. Security and Integrity: By declaring a class as final, you can prevent sensitive or critical functionality from being altered or overridden by other classes. This helps in maintaining the security and integrity of the final class's implementation.

4. Performance Optimization: Marking a class as final can enable certain performance optimizations by the Java compiler and runtime. Since the class cannot be subclassed, the compiler can make certain assumptions and optimizations during compilation, leading to potential performance improvements.

5. Design and Stability: Final classes are often used for classes that have a well-defined purpose and implementation. By making a class final, you indicate that its design and implementation are considered complete and should not be modified further. This promotes stability and consistency in the codebase.

To declare a class as final in Java, simply use the `final` keyword before the class declaration:

```java

final class FinalClass {

// Class implementation

}

```

It's important to note that while a class can be declared as final, its variables and methods can still be marked with other modifiers like `public`, `private`, `protected`, etc. The finality applies only to the class itself and its ability to be subclassed or its methods to be overridden.