Session 4

Object-Oriented Programming Part 2

- Inheritance
- Method Overriding and Overloading
- Abstract Classes and Interfaces
- Packages and Import Statements
- Multiple Choice Questions (MCQs)
- Coding Questions

Class

A class is a blueprint for creating objects (instances).

It defines the properties (attributes) and behaviors (methods) that objects of the class will have.

Example: A "Car" class may have attributes like "color" and "speed", and behaviors like "accelerate" and "brake".

Abstraction

Abstraction is the process of hiding complex implementation details and showing only essential features of an object.

It focuses on what an object does rather than how it does it.

Example: A "Vehicle" class may have a method "start" without revealing the internal combustion engine mechanism.

Encapsulation

Encapsulation is the bundling of data (attributes) and methods (behavior) that operate on the data into a single unit, i.e., a class.

It restricts access to the internal state of an object and protects it from outside interference.

Example: Making attributes private and providing public getter and setter methods to access and modify the attributes.

Object:

An object is an instance of a class.

It represents a real-world entity and encapsulates data (attributes) and behavior (methods).

Example: An object "myCar" created from the "Car" class represents a specific car with its color, speed, and actions.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass.

It enables flexibility and extensibility by allowing methods to behave differently based on the object they are invoked on.

Example: A "Shape" superclass may have a method "draw", and its subclasses like "Circle" and "Square" can override the "draw" method to implement their specific drawing behavior.

Inheritance

Inheritance is a mechanism where a new class (subclass) is created based on an existing class (superclass).

It allows the subclass to inherit properties and behaviors from the superclass, promoting code reuse and establishing an "is-a" relationship.

Example: A "Car" class inheriting from a "Vehicle" class to reuse common vehicle properties and behaviors.

The six pillars of Object-Oriented Programming (OOP) are

Explanation of Inheritance and Its Benefits

Inheritance is a fundamental concept in object-oriented programming (OOP) where a new class, known as a subclass or derived class, is created based on an existing class, known as a superclass or base class. The subclass inherits the properties and behaviors (attributes and methods) of the superclass, allowing it to reuse and extend the functionality defined in the superclass.

Benefits of Inheritance

Code Reusability:

Inheritance promotes code reuse by allowing subclasses to inherit and use the properties and methods of the superclass. This reduces redundancy and makes the code more efficient and maintainable.

Enhanced Modularity:

Inheritance facilitates the creation of modular and organized code by promoting the separation of concerns. Common functionalities can be grouped in a superclass, while specific functionalities can be implemented in subclasses, resulting in a more organized and modular codebase.

Promotes Extensibility:

Inheritance enables the creation of specialized classes that extend or enhance the functionality of the superclass. Subclasses can add new methods, properties, or behaviors without modifying the superclass, thus promoting extensibility and flexibility in the design.

Facilitates Polymorphism:

Inheritance is closely related to polymorphism, another important concept in OOP. Polymorphism allows objects of different classes to be treated as objects of a common superclass. Inheritance provides the foundation for achieving polymorphism, enabling objects to exhibit different behaviors based on their specific class implementations.

Types of Inheritance:

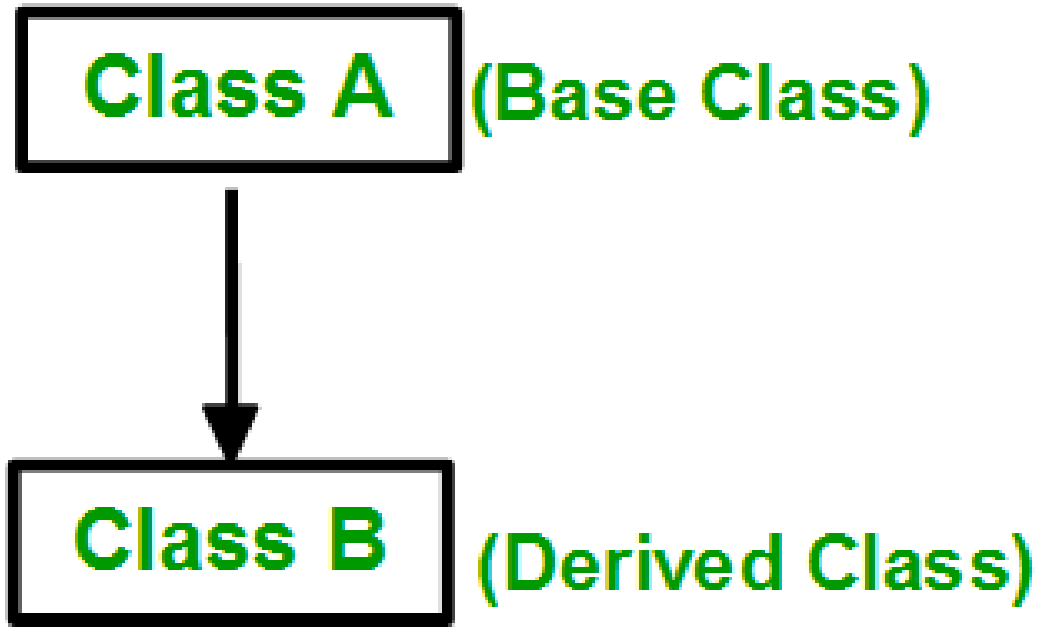
1. Single Inheritance: In single inheritance, a subclass inherits from only one superclass. This is the simplest form of inheritance, where a subclass extends a single parent class.

2.Multi-level Inheritance: In multi-level inheritance, a subclass inherits from another subclass, forming a chain of inheritance. This allows for the creation of a hierarchy of classes, with each level adding additional features or behavior.

3.Hierarchical Inheritance: In hierarchical inheritance, multiple subclasses inherit from the same superclass. This creates a branching structure where different classes share a common ancestor.

Multiple inheritance is not supported in Java to avoid complexities such as the diamond problem, method name conflicts, and ambiguity in code. Java favors composition over inheritance and achieves multiple inheritance of behavior through interfaces.

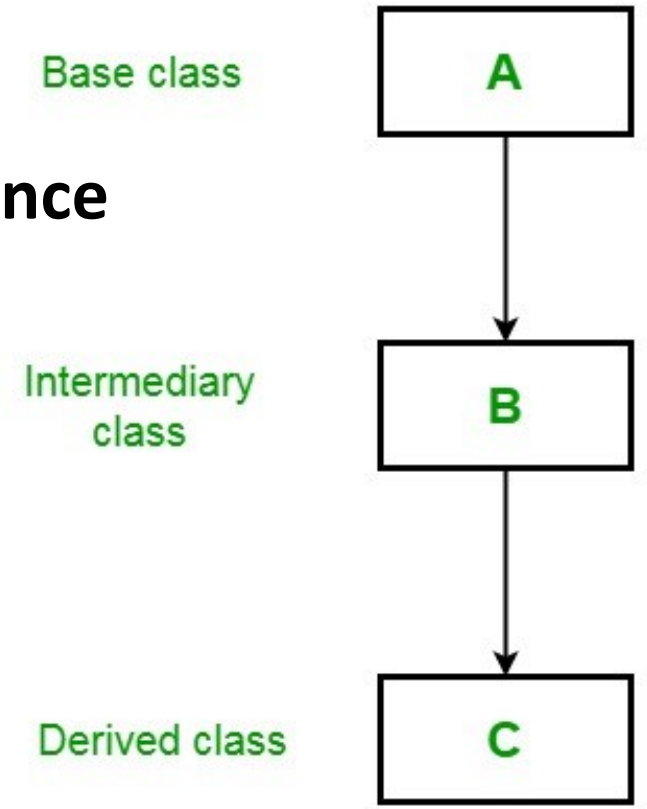
```
1. //Single Level Inheritance

1. // Parent class
2. class Animal {
3.     void eat() {
4.         System.out.println("Animal is eating...");
5.     }
6. }

7. // Child class inheriting from Animal
8. class Dog extends Animal {
9.     void bark() {
10.        System.out.println("Dog is barking...");
11.    }
12. }

13. // Main class
14. public class Main {
15.     public static void main(String[] args) {
16.         Dog dog = new Dog();
17.         dog.eat();  // Inherited method from Animal class
18.         dog.bark(); // Method from Dog class
19.     }
20. }
```



Single Level Inheritance

```
1. //Multi-level Inheritance**:

2. // Parent class
3. class Animal {
4.     void eat() {
5.         System.out.println("Animal is eating...");
6.     }
7. }

8. // Child class inheriting from Animal
9. class Dog extends Animal {
10.     void bark() {
11.         System.out.println("Dog is barking...");
12.     }
13. }

14. // Grandchild class inheriting from Dog
15. class Puppy extends Dog {
16.     void wagTail() {
17.         System.out.println("Puppy is wagging its
    tail...");
18.     }
19. }
```

```
20. // Main class
21. public class Main {
22. public static void main(String[] args) {
23. Puppy puppy = new Puppy();
24. puppy.eat();    // Inherited method from Animal class
25. puppy.bark();   // Inherited method from Dog class
26. puppy.wagTail();// Method from Puppy class
27. }
28. }
```

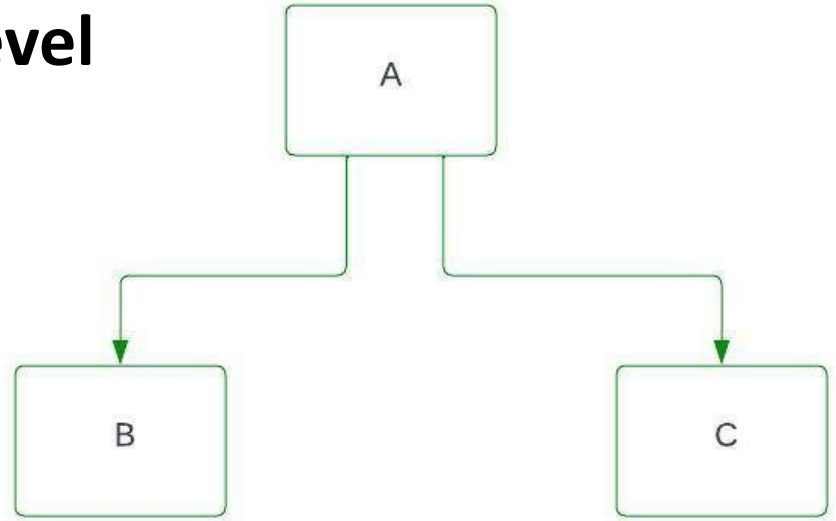


```java
//Hierarchical Inheritance

// Parent class
class Animal {
    void eat() {
        System.out.println("Animal is eating...");
    }
}

// Child class inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking...");
    }
}

// Another child class inheriting from Animal
class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing...");
    }
}
```

```java
// Main class
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method from Animal class
        dog.bark(); // Method from Dog class

        Cat cat = new Cat();
        cat.eat();  // Inherited method from Animal class
        cat.meow(); // Method from Cat class
    }
}
```

Hierarchical Level Inheritance



```
1.  //Example Code Snippet

2.      class Animal {
3.          void makeSound() {
4.              System.out.println("Animal makes a sound");
5.          }
6.      }
7.
8.      class Dog extends Animal {
9.          @Override
10.         void makeSound() {
11.             System.out.println("Dog barks");
12.         }
13.     }
```

Method Overriding

- **Method Overriding**

- Explanation
- - Method overriding is a feature in object-oriented programming where a subclass provides a specific implementation of a method that is already provided by its superclass.

- - It allows a subclass to provide its own implementation of a method that is already defined in its superclass.

- - Method overriding is essential for achieving runtime polymorphism in Java.

```
1.  //Example Code Snippet

2.      class Calculator {
3.          int add(int a, int b) {
4.              return a + b;
5.          }
6.
7.          double add(double a, double b) {
8.              return a + b;
9.          }
10.     }
```

Method Overloading

Explanation

- Method overloading is a feature in Java where multiple methods can have the same name but different parameters.
- It allows different methods to perform similar tasks with varying input parameters.
- Method overloading enhances code readability and reduces the complexity of method names.

Abstract Classes and Interfaces

Abstract Classes:

Explanation

- Abstract classes are classes that cannot be instantiated on their own and may contain abstract methods.
- They serve as blueprints for other classes and are often used to define common behavior for subclasses.
- Abstract classes are crucial for achieving abstraction, where implementation details are hidden from the user.

```
//Example Code Snippet:

1.  abstract class Shape {
2.      abstract void draw();
3.  }
4.
5.  class Circle extends Shape {
6.      void draw() {
7.          System.out.println("Drawing a circle");
8.      }
9.  }
```

Interfaces

Explanation

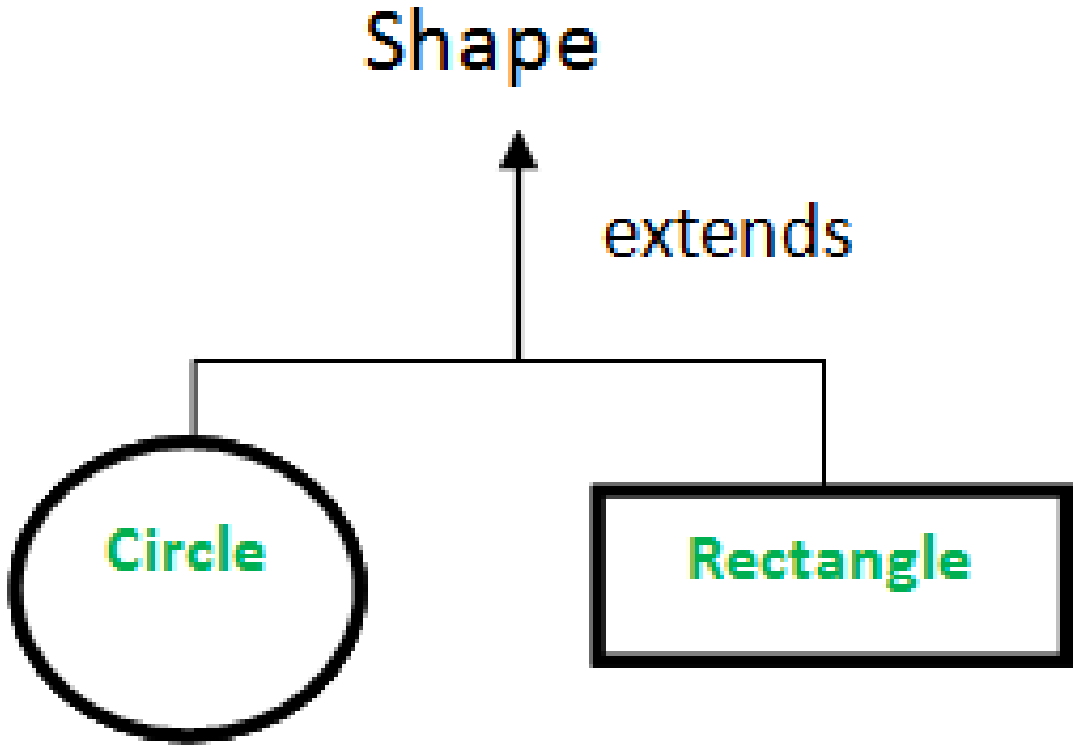
- Interfaces in Java are similar to abstract classes but cannot contain method implementations.

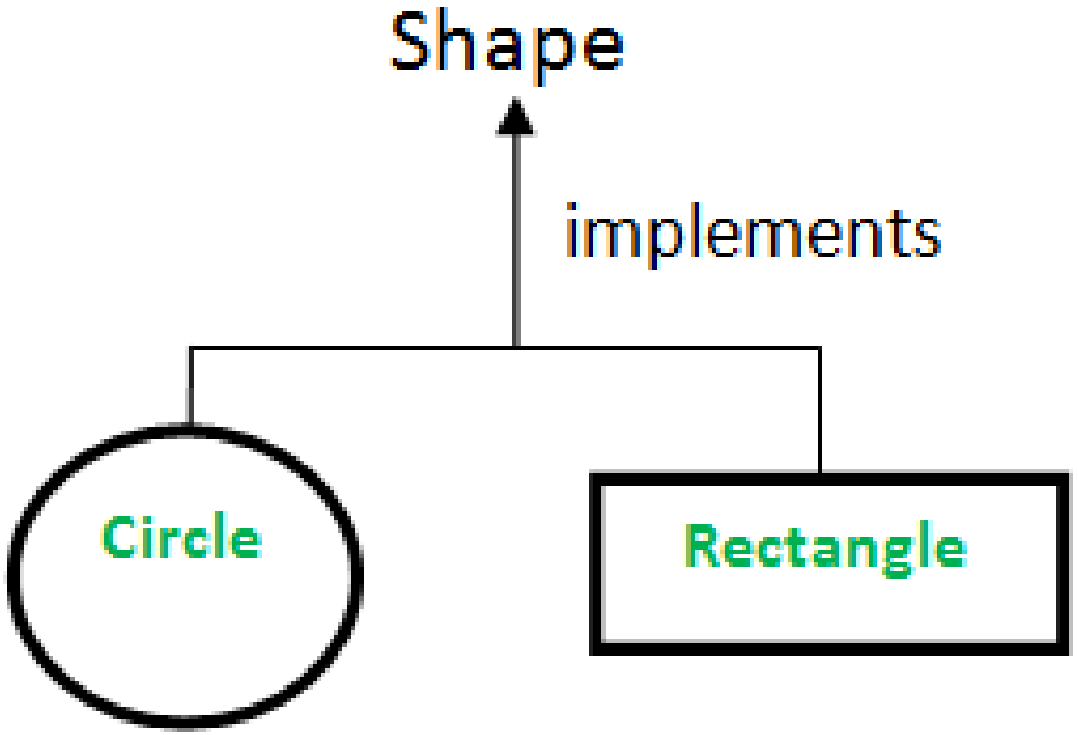
- They define a contract for classes that implement them, specifying methods that the implementing class must provide.

- Interfaces are used to achieve multiple inheritance in Java, as a class can implement multiple interfaces.

```
1.  //Example Code Snippet

2.      interface Drawable {
3.          void draw();
4.      }
5.
6.      class Circle implements Drawable {
7.          public void draw() {
8.              System.out.println("Drawing a circle");
9.          }
10.     }
```

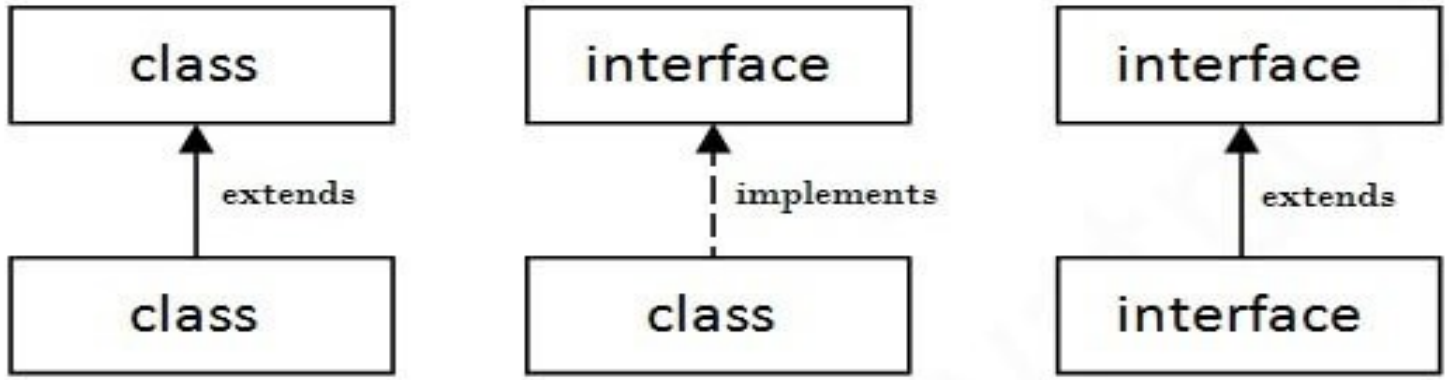




- Abstract classes and interfaces are powerful tools in Java for achieving abstraction and defining contracts for classes. While abstract classes provide partial implementation and can have constructors, interfaces provide full abstraction and support multiple inheritance.

Relation between Class and Interface

- class implement Interface
- Interface never extends a class



Abstract class	Interface
Abstract keyword	Interface keyword
Extend another class , implements multiple interface	Extend only interface
Members of class can be private, public etc..	Members of a interface are by default public
Extends	Implements
Doesn’t support multiple inheritance	It supports multiple inheritance
public abstract class Shape{ public abstract void draw();}	public interface Drawable{ void draw();}

Packages and Import Statements

Packages

Explanation

- Packages are containers for organizing Java classes into namespaces.
- They help in organizing and structuring code by grouping related classes together.
- Packages prevent naming conflicts and make it easier to locate and manage classes.

Creation and Usage

- To create a package, use the `package` keyword followed by the package name at the beginning of the Java file.
- To use classes from a package, import them using the `import` statement.

Import Statements

Explanation

- Import statements are used to access classes and interfaces from other packages.
- They allow you to use classes without specifying their fully qualified names.
- Import statements can be specific to individual classes or can use wildcard (*) to import all classes from a package.

Usage

- Single import: `import packageName.className;`
- Wildcard import: `import packageName.*;`

```
1. //Package Creation
2.
3.     // File: MyClass.java
4.     package com.example; // Define
   package
5.
6.     public class MyClass {
7.         // Class implementation
8.     }
```

```
1. //Package Usage

2.     // File: Main.java
3.     import com.example.MyClass; // Import specific class
4.
5.     public class Main {
6.         public static void main(String[] args) {
7.             MyClass obj = new MyClass(); // Use class from
   package
8.         }
9.     }
```

Packages and import statements play a crucial role in organizing code, preventing naming conflicts, and facilitating code reuse in Java projects.

Basic Interview Questions (MCQs)

1. What is the purpose of an abstract class in Java?
   - a) To prevent instantiation of the class
   - b) To define a blueprint for other classes
   - c) To provide implementation for all methods
   - d) To hide implementation details of a class

2. In Java, can abstract classes have constructors?
   - a) Yes, but only default constructors
   - b) No, abstract classes cannot have constructors
   - c) Yes, and they can have both default and parameterized constructors
   - d) Yes, but only parameterized constructors

3. What is the keyword used to declare an abstract class in Java?
   - a) abstract
   - b) class
   - c) interface
   - d) final

4. Which of the following is true about abstract methods?
   - a) They must have a body
   - b) They can be declared in concrete classes
   - c) They are implicitly final
   - d) They must be implemented by concrete subclasses

5. What is the purpose of method overloading in Java?
   - a) To provide different implementations of a method
   - b) To allow a method to have multiple return types
   - c) To define multiple methods with the same name but different parameters
   - d) To prevent method overriding

Basic Interview Questions (MCQs)

6. Can method overloading be achieved by changing the return type of methods?
   - a) Yes
   - b) No
   - c) Only if the return type is void
   - d) Only if the return type is a primitive type

7. Which of the following is true about method overriding?
   - a) The overridden method must have a different name from the superclass method
   - b) The overridden method can have a different return type from the superclass method
   - c) The overridden method must have a different signature from the superclass method
   - d) The overridden method must be declared as static

8. In Java, can a subclass call the superclass method that it has overridden?
   - a) Yes, using the super keyword
   - b) No, it leads to a compilation error
   - c) Yes, by directly calling the superclass method
   - d) No, it leads to a runtime exception

9. What is the primary purpose of interfaces in Java?
   - a) To provide default implementations of methods
   - b) To define a blueprint for classes
   - c) To achieve multiple inheritance
   - d) To define contracts for classes that implement them

10. Can interfaces have fields (variables)?
   - a) Yes, but they must be declared as static and final
   - b) No, interfaces cannot have fields
   - c) Yes, they can have any type of fields
   - d) Yes, but they must be declared as private

Question 1:

Implement an abstract class called `Shape` with an abstract method `calculateArea()`. Then, create two subclasses `Rectangle` and `Circle` that extend the `Shape` class. Provide implementations for the `calculateArea()` method in both subclasses.

```
1.  // Shape.java
2.  abstract class Shape {
3.      abstract double calculateArea();
4.  }

5.  // Rectangle.java
6.  class Rectangle extends Shape {
7.      private double length;
8.      private double width;

9.      public Rectangle(double length, double width) {
10.         this.length = length;
11.         this.width = width;
12.     }
13.     @Override
14.     double calculateArea() {
15.         return length * width;
16.     }
17. }

18. // Circle.java
19. class Circle extends Shape {
20.     private double radius;

21.     public Circle(double radius) {
22.         this.radius = radius;
23.     }

24.     @Override
25.     double calculateArea() {
26.         return Math.PI * radius * radius;
27.     }
28. }
```

Coding Questions

Question 2:

Create a Java class called `Calculator` with overloaded methods for addition and subtraction. The class should have methods `add(int a, int b)`, `add(double a, double b)`, `subtract(int a, int b)`, and `subtract(double a, double b)`. Ensure proper handling of integer and floating-point numbers.

```
1.  // Calculator.java
2.  public class Calculator {
3.      public int add(int a, int b) {
4.          return a + b;
5.      }

6.      public double add(double a, double b) {
7.          return a + b;
8.      }

9.      public int subtract(int a, int b) {
10.         return a - b;
11.     }

12.     public double subtract(double a, double b) {
13.         return a - b;
14.     }
15. }
```

Coding Questions

Problem 1: Abstract Class and Method Overriding

Create an abstract class `Animal` with an abstract method `makeSound()`. Implement two subclasses `Dog` and `Cat` that extend the `Animal` class and provide their own implementations of the `makeSound()` method. Test the classes by creating instances of `Dog` and `Cat` and calling the `makeSound()` method.

````
Solution:

```java
// Animal.java
abstract class Animal {
    abstract void makeSound();
}

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

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

// Main.java
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();

        dog.makeSound(); // Output: Dog barks
        cat.makeSound(); // Output: Cat meows
    }
}
```
```

Problem 2: Method Overloading

Create a class `Calculator` with overloaded methods `add()` and `subtract()` that accept both integer and double parameters. Test the class by performing addition and subtraction operations with different types of operands.

Solution:

```java
// Calculator.java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

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

    public int subtract(int a, int b) {
        return a - b;
    }
    public double subtract(double a, double b) {
        return a - b;
    }
}
```

```java
// Main.java (Test Cases)
public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        // Addition
        System.out.println(calculator.add(5, 3)); // Output: 8
        System.out.println(calculator.add(2.5, 3.5)); // Output: 6.0

        // Subtraction
        System.out.println(calculator.subtract(10, 3)); // Output: 7
        System.out.println(calculator.subtract(5.5, 3.5)); // Output: 2.0
    }
}
```

Problem 3: Interfaces and Method Overriding

Create an interface `Shape` with a method `calculateArea()`. Implement two classes `Rectangle` and `Circle` that implement the `Shape` interface and provide their own implementations of the `calculateArea()` method. Test the classes by calculating the areas of a rectangle and a circle.

```
1. //Solution:
2. // Shape.java
3. interface Shape {
4.     double calculateArea();
5. }

6. // Rectangle.java
7. class Rectangle implements
   Shape {
8.     private double length;
9.     private double width;

10.    public Rectangle(double
   length, double width) {
11.        this.length = length;
12.        this.width = width;
13.    }
```

```
14.    @Override
15.    public double calculateArea() {
16.        return length * width;
17.    }
18. }
19. // Circle.java
20. class Circle implements Shape {
21.    private double radius;

22.    public Circle(double radius) {
23.        this.radius = radius;
24.    }
25.    @Override
26.    public double calculateArea() {
27.        return Math.PI * radius *
   radius;
28.    }
29. }
```

```
30. // Main.java (Test Cases)
31. public class Main {
32.    public static void main(String[] args) {
33.        Shape rectangle = new Rectangle(5, 3);
34.        Shape circle = new Circle(5);
35.
36.        // Calculate area
37.        System.out.println("Area of rectangle: "
   + rectangle.calculateArea()); // Output: 15.0
38.        System.out.println("Area of circle: " +
   circle.calculateArea()); // Output:
   approximately 78.54
39.    }
40. }
```

Thank You

By AMA