Can We Overload main() Method in Java? Explain your answer with Example code.

Yes, we can overload the main() method in Java. A Java class can have any number of overloaded main() methods. But the very first thing JVM (Java Virtual Machine) seeks is the original main() method, i.e., public static void main(String[] args) to execute.

```
public class MethodOverloadExample1 {

    // Overloaded main() method

    public static void main() {

        System.out.println("Overloaded main() method called");

    }

    // Original main() method

    public static void main(String args[]) {

        System.out.println("Original main() method called");

    }

}
```

How to Overload main() Method in Java ?

Method overloading **is a feature in Java that allows a class to have more than one method with the same name as long as their** parameter declarations **are *different*, i.e., they differ in the number of parameters, parameter type, or both. One of the ways Java supports** polymorphism **is by method overloading.**

```
public class MethodOverloadExample2 {

        // Overloaded main() method which accepts Integer array as an argument.
        public static void main(Integer args[]) {
                System.out.println("Overloaded main() method called");
                System.out.println("Accepts Integer array as an argument.");
        }

        // Overloaded main method which accepts double value as an argument.
```

```
	public static void main(double args) {
		System.out.println("Overloaded main() method called");
		System.out.println("Accepts double value as an argument.");
	}

	// Original main() method which accepts String array as an argument.
	public static void main(String args[]) {
		System.out.println("Original main() method called");
		System.out.println("Accepts String array as an argument.");
	}
}
```

How We Can Invoke the Overloaded main() Method?

The **public static void main(String[] args)** is the entry point for the JVM to start its execution. So, to invoke the overloaded main() methods, we need to call it from the original main() method.

In the following example, there are three different main methods, and we'll invoke other overloaded main methods from the original one to execute.

```
public class MethodOverloadExample3 {

	// Overloaded main method which accepts two int values as arguments.
	// And prints the sum of two numbers
	public static void main(int num1, int num2) {
		System.out.println("Overloaded main() method 1 called");
		System.out.println(num1+" + "+num2+" = " + (num1 + num2));
	}

	// Overloaded main method which accepts String value as an argument.
	// And calls overloaded main method 1
	public static void main(String str) {
		System.out.println("Overloaded main() method 2 called");
		System.out.println(str);
		main(3,5);
	}

	// Original main() method which accepts String array as an argument.
	public static void main(String args[]) {

		System.out.println("Original main() method called");

		// Calls overloaded main method 2
		main("Hello");
	}
}
```

In java can you achieve runtime polymorphism by data members. Explain your answers with example code.

In Java, runtime polymorphism is primarily achieved through method overriding, not through data members. Polymorphism is a fundamental concept in object-oriented programming, and it allows objects of different classes to be treated as objects of a common superclass. This is typically done by overriding methods in the subclass to provide specialized behaviour while maintaining a common interface defined in the superclass.

Data members (fields or attributes) in Java do not directly participate in achieving runtime polymorphism because they are not overridden like methods. Fields are typically accessed directly using the object's reference, and the field accessed depends on the reference's type at compile-time, not runtime.

Here's a simple example to illustrate the concept of runtime polymorphism using method overriding:

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

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

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

public class PolymorphismExample {
   public static void main(String[] args) {
      Animal myAnimal1 = new Dog();
      Animal myAnimal2 = new Cat();

      myAnimal1.makeSound(); // Calls Dog's makeSound method
      myAnimal2.makeSound(); // Calls Cat's makeSound method
   }
}
```

Can there be an abstract method without an abstract class? Describe your answer.

Yes because, the methods in an interface are abstract . So the interface can be use to declare abstract method. We do not require abstract class. Other than interface, if we need to use abstract method then it is compulsory to declare the class as abstract.

Write three different ways to prevent the method from being overridden in java ?

Methods:

1. Using a static method: This is the first way of preventing method overriding in the child class. If you make any method static then it becomes a class method and not an object method and hence it is not allowed to be overridden as they are resolved at compilation time and overridden methods are resolved at runtime.

2. Using private access modifier: Making any method private reduces the scope of that method to class only which means absolutely no one outside the class can reference that method.

3. Using the final keyword method: The final way of preventing overriding is by using the final keyword in your method. The final keyword puts a stop to being an inheritance. Hence, if a method is made final it will be considered final implementation and no other class can override the behaviour.

Why We Use Constructor in Java ?

Constructor in java is used to create the instance of the class. Constructors are almost similar to methods except for two things - its name is the same as the class name and it has no return type.

Sometimes constructors are also referred to as special methods to initialize an object.

State true or false with reasons:

I) True. Abstraction and Encapsulation are complementary concepts. Through encapsulation only we are able to enclose the components of the object into a single unit and separate the private and public members. It is through abstraction that only the essential behaviours of the objects are made visible to the outside world.

II) False because private methods are not directly accessible in derived classes due to the principle of encapsulation.

III) True. In Java, methods declared within an interface are implicitly public, and they cannot have any other access modifier (protected, default, or private). This is a fundamental rule of Java interfaces, and it's defined in the Java language specification.

IV) True. A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass using ‘super’ keyword.

Note: Constructors are special methods used to initialize objects of a class.

1.State whether each of the following is *True or False*. If a statement is *false*, explain why.

(a) A Car class has an is-a relationship with the steeringwheel and Brakes classes.

(b) A has-a relationship is implemented via inheritance.

(c) Superclass constructors are not inherited by subclasses.

(d) When a subclass redefines a superclass method by using the same signature, the subclass is said to overload that superclass method.

Solutions:

(a) False. A Car class typically has a has-a relationship with the steering wheel and brakes classes, not an is-a relationship. In object-oriented programming, an is-a relationship usually implies inheritance, where a subclass is a specialized version of a superclass. In contrast, a has-a relationship implies that one class contains an instance of another class as one of its attributes or components, which is typically implemented using composition.

(b) False. A has-a relationship is typically implemented using composition, not inheritance. Composition means that one class contains an instance of another class as an attribute. Inheritance, on the other hand, is used to implement an is-a relationship, where a subclass inherits the properties and behaviours of a superclass.

(c) True. Superclass constructors are not inherited by subclasses in most object-oriented programming languages. Subclasses may have their own constructors, which can call the constructor of the superclass using the "super" keyword or equivalent, but the superclass constructors themselves are not automatically inherited.

(d) False. When a subclass redefines a superclass method using the same method name and signature, it is said to override the superclass method, not overload it. Overloading typically refers to defining multiple methods in the same class with the same name but different parameters (method overloading), or having operators with different implementations based on the data types of their operands (operator overloading). Overriding, on the other hand, is when a subclass provides its own implementation of a method that is already defined in the superclass with the same signature.

Elaborate with an appropriate example that how OOPS features abstraction and polymorphism can benefit from inheritance in Java:

(i) Software reuse

(ii) Software Extensibility

In Java, Object-Oriented Programming (OOP) features like abstraction and polymorphism can greatly benefit from inheritance, promoting software reuse and extensibility. Let's explore these two aspects in detail with appropriate examples:

i) Software Reuse:

Abstraction: Abstraction is the process of simplifying complex systems by modeling classes based on real-world entities and their behaviours. It allows developers to hide unnecessary details and focus on essential features. Inheritance supports abstraction by allowing you to create a base class (also known as a superclass) with common attributes and methods and then derive specialized classes (subclasses) from it.

Polymorphism: Polymorphism is the ability of objects of different classes to be treated as objects of a common superclass. This enables you to write more generic code that can work with objects of various types. In Java, polymorphism is often achieved through method overriding, where a subclass provides its own implementation of a method defined in the superclass.

Let's consider an example of a simple geometric shapes application where we want to calculate the area of different shapes, such as circles and rectangles.

```
// Abstract superclass representing a generic shape

abstract class Shape {

   abstract double calculateArea();

}


// Subclass Circle inheriting from Shape

class Circle extends Shape {
   private double radius;

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

   @Override
   double calculateArea() {
       return Math.PI * radius * radius;
   }
}

// Subclass Rectangle inheriting from Shape
class Rectangle extends Shape {
   private double width;
   private double height;

   public Rectangle(double width, double height) {
       this.width = width;
       this.height = height;
   }

   @Override
   double calculateArea() {
       return width * height;
   }
}

public class Main {
   public static void main(String[] args) {
       Shape circle = new Circle(5.0);
       Shape rectangle = new Rectangle(4.0, 6.0);


       // Calculate and print areas
       System.out.println("Circle Area: " + circle.calculateArea());
       System.out.println("Rectangle Area: " + rectangle.calculateArea());
   }
}
```

In this example, we use abstraction by creating an abstract Shape class with a method calculateArea(). We then use inheritance to create concrete subclasses (Circle and Rectangle) that provide their specific implementations of the calculateArea() method. The Shape reference in the Main class demonstrates polymorphism as it can represent both circles and rectangles. This promotes software reuse as we can easily add more shape types (e.g., triangles) without modifying the existing code.

ii) Software Extensibility:

Inheritance also enhances software extensibility by allowing you to add new functionality or attributes to existing classes without modifying their source code. You can create new subclasses that inherit from existing classes and extend their behaviour.

Continuing with the geometric shapes example, suppose we want to add color information to our shapes. Instead of modifying the Shape, Circle, or Rectangle classes, we can create a new subclass:

```
// Subclass ColoredCircle inheriting from Circle
class ColoredCircle extends Circle {
    private String color;


    public ColoredCircle(double radius, String color) {
        super(radius);
        this.color = color;
    }


    public String getColor() {
        return color;
    }
}
```

In this case, we've extended the functionality of the Circle class by adding a color attribute to the ColoredCircle subclass. This demonstrates software extensibility without modifying the existing codebase.

Define the Diamond inheritance problem with an example and a sample code. Explain how the multiple inheritance can be achieved in java.

Diamond problems in Java Programming Language may occur when a class(subclass) inherits from multiple classes with a common ancestor(superclass). ***There might be a chance that both parent classes have the same method with the same name and arguments.*** That causes ambiguity during runtime. It makes it difficult for the compiler to choose which method to call. ***Therefore Java does not allow Multiple inheritances to avoid Diamond Problem.***

Java does not allow Multiple Inheritance. For a demonstration of the Diamond Problem, we are assuming that Java allows Multiple Inheritance. In the below diagram, we make class A the superclass of class B and class C. Class B and Class C are a subclass of class A. Also, class B and class C are superclasses for class D. Class D is a common child of class B and class C. Because of this Diamond Like structure, we call this problem a Diamond problem.

```
class A {
    void foo() {
        System.out.println("A's foo");
    }
}


class B extends A {
    void foo() {
        System.out.println("B's foo");
    }
}
```

```
class C extends A {
    void foo() {
        System.out.println("C's foo");
    }
}

class D extends B, C {
// This is not valid in Java
    // ...
}
```

However, Java does not support multiple inheritance like this, and it will result in a compilation error.

In Java, to achieve a form of multiple inheritance, you can use interfaces. Java supports multiple inheritance of interfaces, allowing a class to implement multiple interfaces. Here's an example:

```
interface Interface1 {
    void method1();
}
interface Interface2 {
    void method2();
}
class MyClass implements Interface1, Interface2 {
    public void method1() {
        System.out.println("Implementation of method1");
    }
    public void method2() {
        System.out.println("Implementation of method2");
    }
}
```