Object-Oriented Programming (OOP) Questions

1. What is the difference between Abstraction and Encapsulation in OOP? Can you give examples from your projects?

- **Answer**: **Abstraction** is the concept of hiding the complex implementation details and showing only the essential features of an object. For example, in one of my projects, I used an abstract class `Payment` with an abstract method `processPayment()` to define a general concept of payment without detailing how it is processed. Different subclasses like `CreditCardPayment` and `PaypalPayment` provided specific implementations.

2. Encapsulation, on the other hand, is the technique of bundling data and methods that operate on the data within a single unit or class and restricting access to some of the object's components. In the `Student Management System` project, I encapsulated the student details by making the fields `private` and providing `public` getter and setter methods to control access and modification.

3. Can you explain the concept of Inheritance and how you've used it in your code?

- **Answer**: **Inheritance** is an OOP concept where a new class is created from an existing class, inheriting fields and methods of the existing class. It promotes code reusability and establishes a natural hierarchy between classes. In my project, `CRUD Implementation using Angular`, I had a base class `User` with common properties like `id`, `name`, and `email`. From this class, I derived two subclasses, `AdminUser` and `RegularUser`, each with additional specific attributes and methods.

4. What is Polymorphism and how is it achieved in Java? Provide examples from your experience.

- **Answer**: **Polymorphism** is an OOP principle that allows one interface to be used for a general class of actions, with specific actions determined by the exact nature of the situation. In Java, polymorphism is mainly achieved through method overloading (compile-time polymorphism) and method overriding (runtime polymorphism). In my project, I used method overriding in the `Student Management System` to provide different implementations of a method `generateReport()` in `MonthlyReport` and `AnnualReport` classes.

5. What are Design Patterns and which ones have you used in your projects?

Design Patterns are general, reusable solutions to common problems in software design. They provide a standardized approach to solving design challenges, allowing developers to create more maintainable, flexible, and scalable code. Design patterns can be categorized into three main types:

Some Design Patterns Might Have Used

1. **Singleton Pattern**: Used in your Spring Boot projects to ensure a single instance of a service or component (like a database connection or configuration class).

2. **Factory Pattern**: Useful in creating objects without specifying the exact class of object that will be created, which can be seen in projects that involve complex object creation, such as different types of request handlers in REST APIs.
3. **Observer Pattern**: Often used in Angular applications, especially when dealing with event handling or state management, like updating a component when a service emits a change.
4. **MVC (Model-View-Controller) Pattern**: This is a structural pattern that is widely used in web applications, including those you build with Spring Boot and Angular. It separates the application into three interconnected components, improving modularity and scalability.
5. **Decorator Pattern**: In projects involving REST APIs, this pattern can be used to add functionality to objects dynamically. This could be seen in middleware implementations or request/response modifications.
6. **Proxy Pattern**: Useful in situations where you need to control access to an object, such as in a caching mechanism for a database in a Spring Boot application.

6. Explain the concept of Interfaces in Java. How do they differ from Abstract Classes?

- **Answer**: **Interfaces** in Java are abstract types that allow the declaration of methods that one or more classes are expected to implement. Unlike abstract classes, interfaces cannot have any method implementations; they are purely a contract that defines a set of methods. In contrast, abstract classes can have both complete (concrete) methods and incomplete (abstract) methods. I have used interfaces in my projects to define methods that different classes must implement, ensuring consistent behavior across different modules.

1. Interfaces: An interface in Java is a reference type that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors, and all the methods are implicitly abstract (unless they are default or static).

Here's an example of an interface:

```
interface Animal {
    void eat();
    void sleep();
}
```

2. Abstract Classes: An abstract class is a class that cannot be instantiated on its own and can have both abstract methods (methods without a body) and concrete methods (methods with a body). Abstract classes are used to provide a base for subclasses to extend and can contain fields, constructors, and any access modifiers.

Here's an example of an abstract class:

```
abstract class Bird {
    // Abstract method (does not have a body)
    abstract void fly();

    // Concrete method
    void chirp() {
        System.out.println("Bird is chirping");
    }
}
```

Example Combining Both:

Let's consider an example where we have an interface `Animal` and an abstract class `Bird`. A concrete class `Sparrow` implements the `Animal` interface and extends the `Bird` abstract class:

```
// Interface
interface Animal {
    void eat();
    void sleep();
}

// Abstract class
abstract class Bird implements Animal {
    abstract void fly();

    // Concrete method
    @Override
    public void sleep() {
        System.out.println("Bird is sleeping");
    }
}

// Concrete class
class Sparrow extends Bird {
    @Override
    public void eat() {
        System.out.println("Sparrow is eating");
    }

    @Override
    void fly() {
        System.out.println("Sparrow is flying");
    }

    // Optional: Override the inherited method from Bird (sleep)
    @Override
    public void sleep() {
        System.out.println("Sparrow is sleeping");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Sparrow sparrow = new Sparrow();
        sparrow.eat();    // Output: Sparrow is eating
        sparrow.sleep();  // Output: Sparrow is sleeping
        sparrow.fly();    // Output: Sparrow is flying
        sparrow.sleep();  // Output:  Sparrow is sleeping
    }
}
```

Key Points:

- **Interfaces** define methods that must be implemented by any class that implements the interface.
- **Abstract classes** provide a partial implementation that can be extended by subclasses, which may override or use the provided methods.

- A class like `Sparrow` can implement multiple interfaces but can extend only one class (abstract or concrete).

-- >> *If you want to call the `sleep` method specifically from the `Bird` class rather than the overridden `sleep` method in the `Sparrow` class, you will need to make a small change to the `Sparrow` class to call the parent class's `sleep` method.*

Here's how you can modify the `Sparrow` class to call the `sleep` method from the `Bird` class:

```
// Concrete class
class Sparrow extends Bird {
    @Override
    public void eat() {
        System.out.println("Sparrow is eating");
    }

    @Override
    void fly() {
        System.out.println("Sparrow is flying");
    }

    // Override the inherited method from Bird (sleep)
    @Override
    public void sleep() {
        // Calling the Bird class sleep method
        super.sleep(); // This will call Bird's sleep method
        System.out.println("Sparrow is sleeping");
    }
}

public class Main {
    public static void main(String[] args) {
        Sparrow sparrow = new Sparrow();
        sparrow.eat();    // Output: Sparrow is eating
        sparrow.sleep();  // Output: Bird is sleeping
                          //         Sparrow is sleeping
        sparrow.fly();    // Output: Sparrow is flying
        sparrow.chirp();  // Output: Bird is chirping
    }
}
```

`Explanation:`

- The `super.sleep()` call inside the `Sparrow` class's `sleep()` method is used to invoke the `sleep()` method of the `Bird` class. This ensures that the `Bird` class's `sleep` method is executed first, printing "Bird is sleeping".
- After that, "Sparrow is sleeping" is printed, demonstrating that both the parent class (`Bird`) and child class (`Sparrow`) methods are being called.

>>>**In Java, the `super` keyword** is used in a subclass to refer to its immediate superclass (parent class). It allows you to access methods, constructors, and fields of the superclass that are overridden or hidden by the subclass.

Here are the main scenarios where `super` is used:

1. Calling the Superclass Constructor:

When you want to call the constructor of the superclass from a subclass, you use `super()`. This is often done to initialize the superclass part of the object.

- **Example:**

```
class Animal {
    Animal() {
        System.out.println("Animal constructor called");
    }
}

class Dog extends Animal {
    Dog() {
        super(); // Calls the constructor of Animal
        System.out.println("Dog constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        // Output:
        // Animal constructor called
        // Dog constructor called
    }
}
```

2. Calling the Superclass Method:

When a method in the superclass is overridden in the subclass, but you still want to call the superclass version of the method from the subclass, you use `super.methodName()`.

- **Example:**

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

class Dog extends Animal {
    @Override
    void sound() {
        super.sound(); // Calls the sound method of Animal
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
        // Output:
        // Animal makes a sound
        // Dog barks
    }
}
```

3. Accessing Superclass Fields:

If a field in the subclass hides a field with the same name in the superclass, you can use `super.fieldName` to access the field in the superclass.

- **Example:**

```
class Animal {
    String type = "Animal";
}

class Dog extends Animal {
    String type = "Dog";

    void printType() {
        System.out.println(type); // Accesses Dog's field
        System.out.println(super.type); // Accesses Animal's field
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.printType();
        // Output:
        // Dog
        // Animal
    }
}
```

Summary:

- **`super()`**: Used to call the constructor of the superclass.
- **`super.methodName()`**: Used to call a superclass method that has been overridden in the subclass.
- **`super.fieldName`**: Used to access a field of the superclass when there is a field with the same name in the subclass.

Additional Points About Superclasses and `super` Keyword

1. **Constructor Chaining**:
    - In Java, the constructor of every subclass implicitly calls the no-argument constructor of its superclass if no constructor is explicitly called using `super()`. This is known as constructor chaining.
    - If a superclass does not have a no-argument constructor, and you have a parameterized constructor in the subclass, you **must** explicitly call one of the superclass's constructors using `super(parameters...)`.

    Example:

```
class Animal {
    Animal(String name) {
        System.out.println("Animal constructor called with name: " +
name);
    }
}
```

```
class Dog extends Animal {
    Dog() {
        super("Buddy");  // Explicitly calling the superclass constructor
with parameters
        System.out.println("Dog constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        // Output:
        // Animal constructor called with name: Buddy
        // Dog constructor called
    }
}
```

2. **Using `super` in Nested Classes**:

    - In the case of nested or inner classes, `super` can refer to the instance of the enclosing class or a superclass of the enclosing class. This can be useful when accessing members of the outer class from within a nested class.

    Example:

```
class Outer {
    void show() {
        System.out.println("Outer class method");
    }

    class Inner extends Outer {
        void show() {
            super.show();  // Calls Outer class's show() method
            System.out.println("Inner class method");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Outer.Inner inner = new Outer().new Inner();
        inner.show();
        // Output:
        // Outer class method
        // Inner class method
    }
}
```

3. **`super` with Multiple Inheritance Through Interfaces**:

    - Although Java does not support multiple inheritance with classes, it allows a class to implement multiple interfaces. If two interfaces contain methods with the same signature, `super` can be used to differentiate which interface's default method to call.
    - You use `InterfaceName.super.methodName()` to specify which interface's default method to invoke if they have conflicting methods.

    Example:

```
interface A {
```

```
    default void print() {
        System.out.println("Interface A");
    }
}

interface B {
    default void print() {
        System.out.println("Interface B");
    }
}

class C implements A, B {
    @Override
    public void print() {
        A.super.print();  // Calls A's default print() method
        B.super.print();  // Calls B's default print() method
        System.out.println("Class C");
    }
}

public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.print();
        // Output:
        // Interface A
        // Interface B
        // Class C
    }
}
```

4. **Restrictions on `super`**:
    - `super` must be the first statement in a constructor if it is used. You cannot use `super` after any other statement in a constructor.
    - You cannot use `super` in static contexts because `super` relates to an instance of a class, and static methods/fields do not belong to any instance.

>> **The `static` keyword in Java** is used for memory management and is one of the fundamental keywords in the language. It can be applied to variables, methods, blocks, and nested classes. The `static` keyword indicates that the particular member belongs to the class itself rather than to instances of the class (objects). This means that the static member is shared across all instances of the class and does not need an instance to be accessed.

Let's dive deeper into each usage of the `static` keyword:

1. Static Variables (Class Variables)

A static variable is shared among all instances of a class. It is initialized only once at the start of the execution, and all instances of the class share the same static variable.

- **Characteristics:**
    - Static variables are also known as **class variables** because they belong to the class, not any specific instance (object) of the class.
    - Static variables are initialized only once, at the start of the execution. These are allocated memory in the static memory area.

    - Even if you create multiple objects of the class, the static variable will have only one copy.
    - They can be accessed directly using the class name without needing an instance of the class.
- **Example:**

```
class Counter {
    static int count = 0; // Static variable

    Counter() {
        count++; // Incrementing the static variable count
        System.out.println(count);
    }
}

public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter(); // Output: 1
        Counter c2 = new Counter(); // Output: 2
        Counter c3 = new Counter(); // Output: 3
    }
}
```

2. Static Methods

A static method belongs to the class rather than any specific instance. It can be called without creating an instance of the class. Static methods can access static data directly and can change the value of it.

- **Characteristics:**
    - Static methods **cannot** access instance variables and instance methods directly; they can only access static variables and static methods.
    - Static methods can be called using the class name without creating an instance of the class.
    - The `this` and `super` keywords cannot be used in static methods because `this` and `super` are associated with an instance of the class.
- **Example:**

```
class MathUtils {
    static int square(int x) { // Static method
        return x * x;
    }
}

public class Main {
    public static void main(String[] args) {
        int result = MathUtils.square(5); // Calling static method without
creating an object
        System.out.println(result); // Output: 25
    }
}
```

3. **Static Blocks**

Static blocks are used for static initializations of a class. This code inside the static block is executed once, when the class is first loaded into memory. Static blocks are useful for initializing static variables or executing code that needs to run once, such as loading a native library.

- **Characteristics:**
    - Static blocks are executed when the class is loaded into memory, and only once.
    - You can have multiple static blocks in a class, and they will execute in the order in which they are defined.
- **Example:**

```
class StaticBlockDemo {
    static int data;

    static { // First static block
        data = 50;
        System.out.println("Static block 1 executed: " + data);
    }

    static { // Second static block
        data += 10;
        System.out.println("Static block 2 executed: " + data);
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Main method executed");
        StaticBlockDemo obj = new StaticBlockDemo(); // Creating an object to
trigger class loading
    }
}
```

Output:

```
Static block 1 executed: 50
Static block 2 executed: 60
Main method executed
```

4. Static Classes (Nested Static Classes)

In Java, you cannot declare a top-level class as static. However, you can declare a nested class as static. A static nested class does not have access to the instance variables and methods of the outer class. It can access static data members of the outer class.

- **Characteristics:**
    - Static nested classes are associated with their outer class and can be accessed without creating an instance of the outer class.
    - Static nested classes can access all static members (methods and fields) of the outer class, but **cannot** access non-static members directly.
- **Example:**

```
class OuterClass {
    static int data = 10;
```

```
    static class NestedStaticClass {
        void display() {
            System.out.println("Data from outer class: " + data); // Accessing
static data from outer class
        }
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass.NestedStaticClass nested = new
OuterClass.NestedStaticClass(); // No need to create an instance of OuterClass
        nested.display(); // Output: Data from outer class: 10
    }
}
```

***** Summary of the `this` keyword in Java, covering its main uses with brief examples:***

1. Referring to Instance Variables

The `this` keyword differentiates instance variables from parameters with the same name.

Example:

```
class Person {
    private String name;

    public Person(String name) {
        this.name = name; // 'this.name' refers to the instance variable
    }
}
```

2. Invoking Current Class Methods

`this` can be used to call another method of the current class.

Example:

```
class Calculator {
    public void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
        this.display();
    }

    public void display() {
        System.out.println("Operation completed.");
    }
}
```

3. Invoking Current Class Constructors

`this()` can be used to call another constructor in the same class (constructor chaining).

Example:

```
class Box {
    int width, height, depth;

    public Box(int width, int height) {
        this(width, height, 10); // Calls the three-parameter constructor
```

```
    }

    public Box(int width, int height, int depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
}
```

4. Returning the Current Class Instance

`this` can return the current instance, enabling method chaining.

Example:

```
class Builder {
    private StringBuilder sb = new StringBuilder();

    public Builder addString(String str) {
        sb.append(str);
        return this; // Returns the current instance
    }

    public String build() {
        return sb.toString();
    }
}

public class Main {

      public static void main(String[] args) {

             Builder builder = new Builder();

            // Method chaining: multiple methods called in a single statement

            String result =
      builder.addString("Hello").addString("World").build();

            System.out.println(result); // Output: Hello World

      }

}
```

5. Passing the Current Object as an Argument

`this` can be used to pass the current object as an argument to a method or constructor.

Example:

```
class Data {
    private String content;
    public void process() {
        Processor.process(this); // Passing the current object to the process
method
    }
}

class Processor {
```

```
    public static void process(Data data) {
        System.out.println("Processing: " + data);
    }
}
```

6. Accessing Members of the Outer Class from Inner Class

In an inner class, `this` can refer to the outer class object.

Example:

```
class Outer {
    String name = "Outer";

    class Inner {
        String name = "Inner";

        public void printNames() {
            System.out.println(name); // Inner's name
            System.out.println(Outer.this.name); // Outer’s name
        }
    }
}
```

Summary Points

- **`this`**: Refers to the current object.
- **`this.variable`**: Distinguishes instance variables from local variables.
- **`this.method()`**: Calls a method of the current object.
- **`this()`**: Calls another constructor in the same class.
- **`return this`**: Returns the current object instance for chaining.
- **`this` as a method parameter**: Passes the current object to another method.
- **`OuterClassName.this`**: Used inside an inner class to refer to the outer class object.

Constructors in Java

A **constructor** in Java is a special method that is called when an object is instantiated. It is used to initialize the state of an object. Constructors have the same name as the class and do not have a return type, not even `void`.

Types of Constructors

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor** (Java does not have a built-in copy constructor, but one can be created manually)

Let's go through each of these with brief examples.

1. Default Constructor

A **default constructor** is automatically provided by Java if no constructors are defined in the class. It initializes the object with default values.

Example:

```
class Student {
    String name;
    int age;

    // Default constructor provided by Java
    Student() {
        this.name = "Unknown";
        this.age = 0;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student1 = new Student();  // Default constructor is called
        student1.display();  // Output: Name: Unknown, Age: 0
    }
}
```

2. Parameterized Constructor

A **parameterized constructor** is defined with parameters, allowing you to initialize the object with specific values.

Example:

```
class Student {
    String name;
    int age;

    // Parameterized constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student2 = new Student("Alice", 20);  // Parameterized
constructor is called
        student2.display();  // Output: Name: Alice, Age: 20
    }
}
```

3. Copy Constructor

A **copy constructor** creates a new object as a copy of an existing object. Java does not provide a built-in copy constructor, but you can define one manually.

Example:

```
class Student {
    String name;
    int age;

    // Parameterized constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Copy constructor
    Student(Student other) {
        this.name = other.name;
        this.age = other.age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student student2 = new Student("Alice", 20);  // Parameterized
constructor
        Student student3 = new Student(student2);     // Copy constructor
        student3.display();  // Output: Name: Alice, Age: 20
    }
}
```

Key Points About Constructors

- **No return type**: Constructors do not have a return type, not even `void`.
- **Same name as the class**: The constructor's name must match the class name.
- **Called automatically**: Constructors are automatically called when an object is created.
- **Cannot be abstract, static, final, or synchronized**: Constructors are meant to initialize objects, so these keywords are not applicable.

Access Specifiers in Java

Access specifiers (or access modifiers) in Java define the visibility or scope of a class, its constructors, fields, and methods. Java provides four access specifiers:

1. **Private** (`private`)
2. **Default** (no keyword, also known as package-private)
3. **Protected** (`protected`)
4. **Public** (`public`)

1. Private Access Specifier (`private`)

- **Scope**: The member is accessible only within the class in which it is declared.
- **Use Case**: Restricting access to the internal details of a class to ensure data encapsulation and hiding.

Example:

```
class Person {
    private String name;  // 'name' is private to the 'Person' class

    private void display() {  // 'display' method is private to the 'Person'
class
        System.out.println("Name: " + name);
    }

    public void setName(String name) {  // Public setter to allow controlled
access
        this.name = name;
    }

    public String getName() {  // Public getter to allow controlled access
        return name;
    }
}
```

- In the above example, the `name` variable and `display()` method are private and cannot be accessed outside the `Person` class. Public methods `setName()` and `getName()` provide controlled access to the `name` variable.

2. Default Access Specifier (Package-Private)

- **Scope**: The member is accessible only within the same package. There is no specific keyword for default access; it is simply the absence of any access specifier.
- **Use Case**: Allows access to class members only within the same package, useful for creating a group of classes that work closely together.

Example:

```
class Animal {
    String type;  // Default access specifier

    void displayType() {  // Default access specifier
        System.out.println("Type: " + type);
    }
}
```

- In the above example, the `type` variable and `displayType()` method have default access. They can be accessed only by classes within the same package as `Animal`.

3. Protected Access Specifier (`protected`)

- **Scope**: The member is accessible within the same package and by subclasses in different packages.
- **Use Case**: Allows controlled access to class members by subclasses, enabling inheritance.

Example:

```
package animals;

public class Animal {
    protected String species;  // 'species' is protected

    protected void displaySpecies() {  // 'displaySpecies' is protected
        System.out.println("Species: " + species);
    }
}

package animals.mammals;

import animals.Animal;

public class Dog extends Animal {
    public void showDetails() {
        species = "Canine";  // Accessing protected member from superclass
        displaySpecies();    // Accessing protected method from superclass
    }
}
```

- In this example, the `species` variable and `displaySpecies()` method in the `Animal` class are `protected`. They can be accessed by the `Dog` class, which is in a different package but is a subclass of `Animal`.

4. Public Access Specifier (`public`)

- **Scope**: The member is accessible from any other class, in any package.
- **Use Case**: Allows maximum visibility and accessibility, often used for classes and methods intended for broad use.

Example:

```
public class Car {
    public String model;  // 'model' is public

    public void displayModel() {  // 'displayModel' is public
        System.out.println("Model: " + model);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.model = "Toyota";  // Accessing public variable
        car.displayModel();    // Accessing public method
    }
}
```

- In this example, the `model` variable and `displayModel()` method are `public`, allowing them to be accessed from any class, in any package.

Differences Between `final`, `finally`, and `finalize`

1. `final`

- **Purpose**:

    - **Variables**: Used to define constants. Once assigned, the value cannot be changed.
    - **Methods**: Prevents method overriding in subclasses.
    - **Classes**: Prevents inheritance. No other class can extend a `final` class.
- **Usage**:
    - **Variable Example**:

```
final int MAX_SPEED = 120;  // Cannot change MAX_SPEED later
```

    - **Method Example**:

```
class Parent {
    final void show() {
        System.out.println("Final method in Parent class.");
    }
}

class Child extends Parent {
    // void show() { }  // Error: Cannot override final method
}
```

    - **Class Example**:

```
final class FinalClass { }

// class AnotherClass extends FinalClass { }  // Error: Cannot
inherit from final class
```

2. `finally`

- **Purpose**: Ensures that a block of code is executed after a `try` block, regardless of whether an exception is thrown or not. Typically used for cleaning up resources like closing files or database connections.
- **Usage**:
    - **Example**:

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            int result = 10 / 0;  // Throws ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        } finally {
            System.out.println("This block is always executed.");
        }
    }
}
// Output:
// Inside try block
// Exception caught: java.lang.ArithmeticException: / by zero
// This block is always executed.
```

3. `finalize`

- **Purpose**: Allows an object to perform cleanup operations before it is garbage collected. The `finalize` method is called by the garbage collector before the object is removed from

memory. However, its use is discouraged because it does not guarantee when or if it will be executed, and it can lead to performance issues.

- **Usage**:
  - **Example**:

```
class Resource {
    @Override
    protected void finalize() throws Throwable {
        try {
            System.out.println("Cleaning up resources...");
        } finally {
            super.finalize();
        }
    }
}

public class FinalizeExample {
    public static void main(String[] args) {
        Resource resource = new Resource();
        resource = null; // Suggests to garbage collector that
resource can be collected
        System.gc();     // Requests garbage collection
    }
}
// Output: Cleaning up resources...
```