

Behavioral Questions

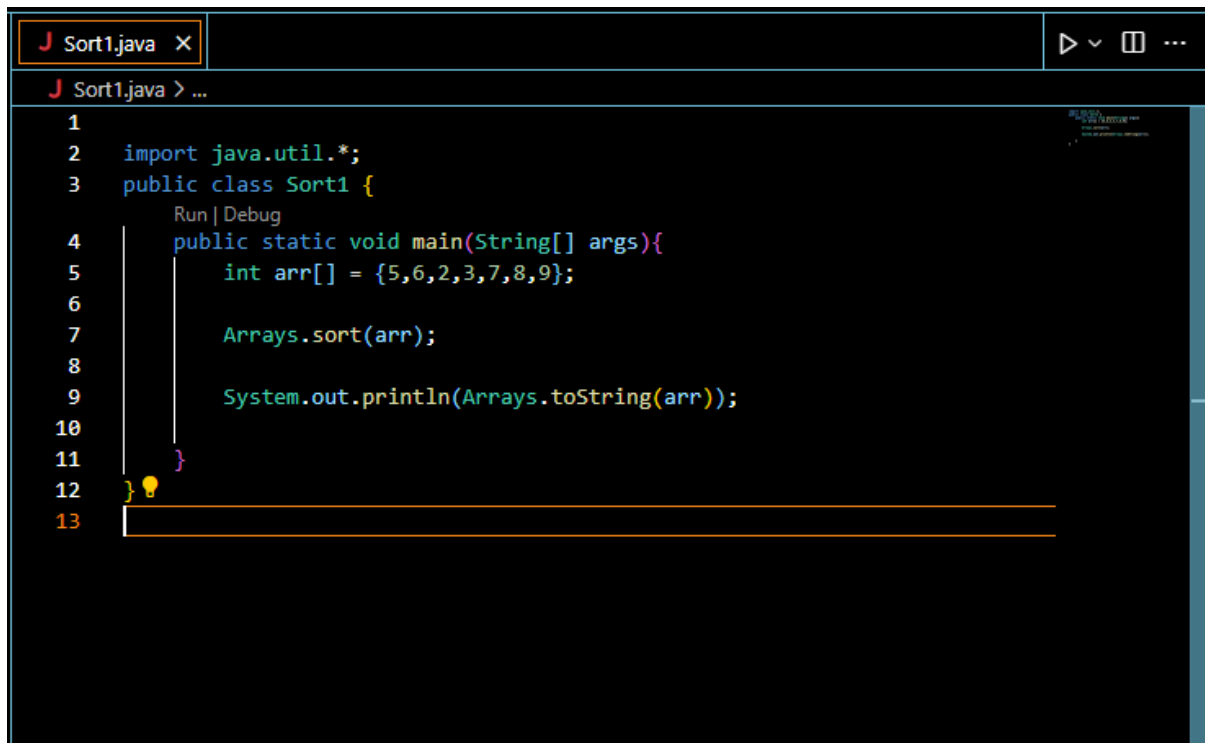
Question	Frequency
Do you use social media? What is the use of it?	1
Tell us about your hometown.	1
Why is your CGPA not very high?	1
How will you contribute to Cloud Kaptan Consultancy Services?	1
Speak about yourself non-stop	2
Tell me about yourself.	5
Family Background.	2
Why do you want to join CloudKaptan and not any other MNC?	3
Past interviews you have given and why were you not selected?	1

Technical Questions (Java/OOPs/DSA)

Question	Frequency
Basics of Java Programming Language	2
Sorting of array in descending order, print 2nd smallest number	1
Difference between bubble sort and selection sort	1
Types of constructors and demonstrate using code	1
Difference between print and println in Java	1
Pillars of OOPs, explain each part	4
Search array element using recursion	1
What is abstraction? How can we implement it?	2
Reverse a string	1
Check if a matrix is an identity matrix	1
Check if two strings are anagrams	1
Code to compress string (e.g., AAABBBBBBCCDDDDDD -> 3A5B2C6D)	1
How many types of inheritance are there in Java?	1

Question	Frequency
Why does Java not support multiple inheritance?	2
What is polymorphism?	2
Abstract class vs interface	2
Exception handling in Java and its hierarchy	3
Explain final, finally, finalize()	2
Code for String Palindrome	1
Static vs constant	2
JDK-JRE-JVM Comparison	1
Code to find the nth node in a linked list	1
Code to rotate a matrix by 180° and print the diagonal sum	1
Sort array in ascending order up to an index, then descending	1
Magic Number problem	1
GCD calculation	1
Print pattern	1
Write the fibonacci series code	1
Compare C vs Java	1
Tic Tac Toe by JAVA	1
Interface Usage Custom exception Integer to roman and vice versa Multithreading and Lifecycle of Threads	

- Sorting in Ascending.

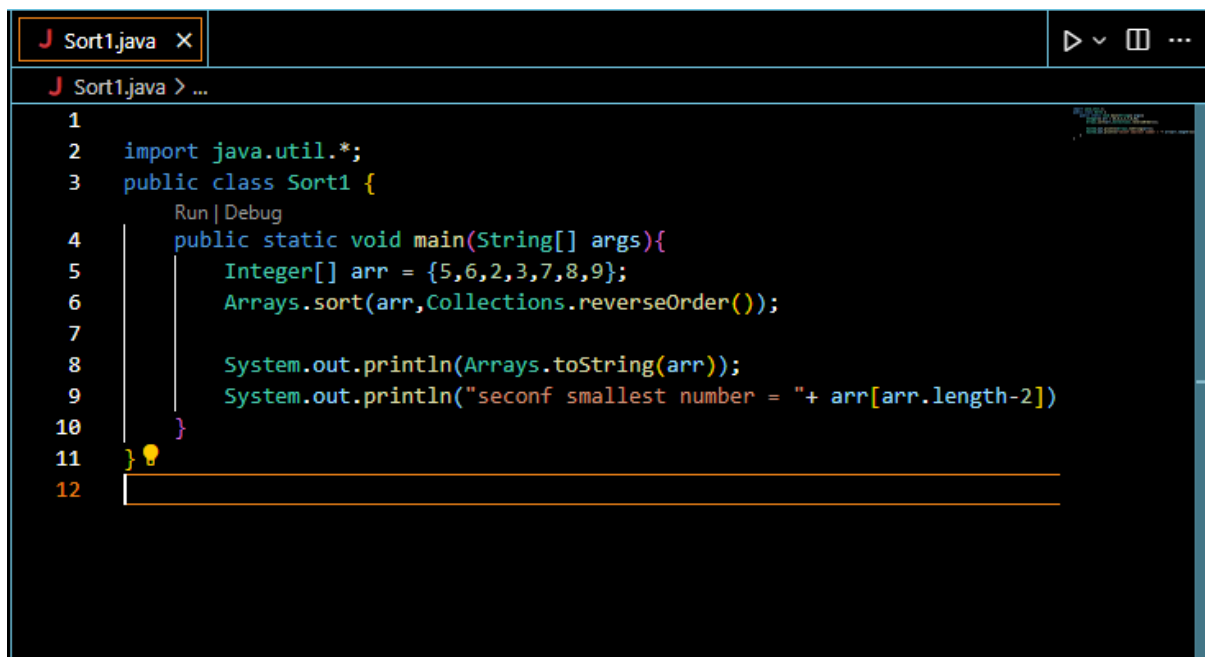


The screenshot shows an IDE window titled 'Sort1.java'. The code is as follows:

```
1
2 import java.util.*;
3 public class Sort1 {
4     public static void main(String[] args){
5         int arr[] = {5,6,2,3,7,8,9};
6
7         Arrays.sort(arr);
8
9         System.out.println(Arrays.toString(arr));
10    }
11 }
12
13
```

Line 12 has a yellow lightbulb icon, indicating a suggestion or error.

- Sorting in descending

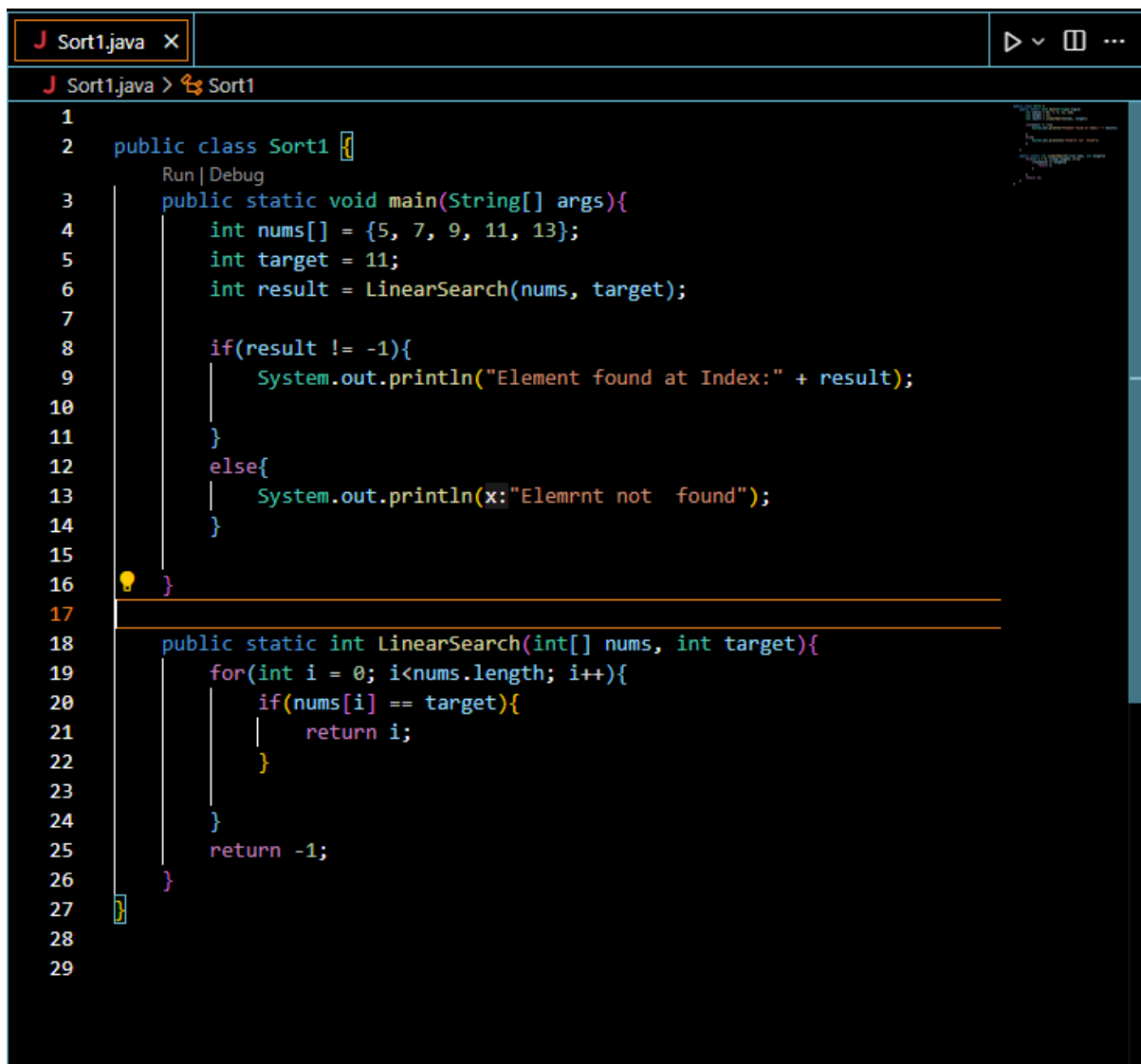


The screenshot shows an IDE window titled 'Sort1.java'. The code is as follows:

```
1
2 import java.util.*;
3 public class Sort1 {
4     public static void main(String[] args){
5         Integer[] arr = {5,6,2,3,7,8,9};
6         Arrays.sort(arr, Collections.reverseOrder());
7
8         System.out.println(Arrays.toString(arr));
9         System.out.println("seconf smallest number = "+ arr[arr.length-2])
10    }
11 }
12
```

Line 11 has a yellow lightbulb icon, indicating a suggestion or error.

Linear Search:



The image shows a screenshot of a Java IDE with a dark theme. The top bar shows a tab for 'Sort1.java' and a toolbar with icons for running, debugging, and other actions. The main editor area displays the following Java code:

```
1
2 public class Sort1 {
3     public static void main(String[] args){
4         int nums[] = {5, 7, 9, 11, 13};
5         int target = 11;
6         int result = LinearSearch(nums, target);
7
8         if(result != -1){
9             System.out.println("Element found at Index:" + result);
10        }
11        else{
12            System.out.println("Element not found");
13        }
14    }
15
16    public static int LinearSearch(int[] nums, int target){
17        for(int i = 0; i<nums.length; i++){
18            if(nums[i] == target){
19                return i;
20            }
21        }
22        return -1;
23    }
24 }
25
26
27
28
29
```

The code implements a linear search algorithm. The `main` method initializes an array `nums` with values {5, 7, 9, 11, 13} and a `target` value of 11. It calls the `LinearSearch` method, which iterates through the array. If the target is found, it returns the index; otherwise, it returns -1. The `main` method then prints the result.

Binary Search

```

    }
    return -1;
}

public static int BinarySearch(int[] nums, int target){
    int left = 0;
    int right = nums.length - 1;
    while(left<=right){
        int mid = (left+right)/2;

        if(nums[mid] == target){
            return mid;
        }
        else if(nums[mid]< target){
            left = mid+1;
        }
        else{
            right = mid-1;
        }
    }
    return -1;
}
}

```

Selection sort:

```

J Sort1.java X
J Sort1.java > Sort1 > main(String[])
1 import java.util.*;
2 public class Sort1 {
    Run | Debug
3     public static void main(String[] args) {
4         int arr[] = {1, 2, 1, 13, 0, 7, 3, 1};
5
6         // Perform selection sort
7         for (int i = 0; i < arr.length - 1; i++) {
8             int minIndex = i;
9             for (int j = i + 1; j < arr.length; j++) {
10                if (arr[j] < arr[minIndex]) {
11                    minIndex = j;
12                }
13            }
14
15            // Swap the found minimum element with the first element
16            int temp = arr[minIndex];
17            arr[minIndex] = arr[i];
18            arr[i] = temp;
19        }
20
21        // Print the sorted array
22
23        System.out.println(Arrays.toString(arr));
24    }
25 }
26

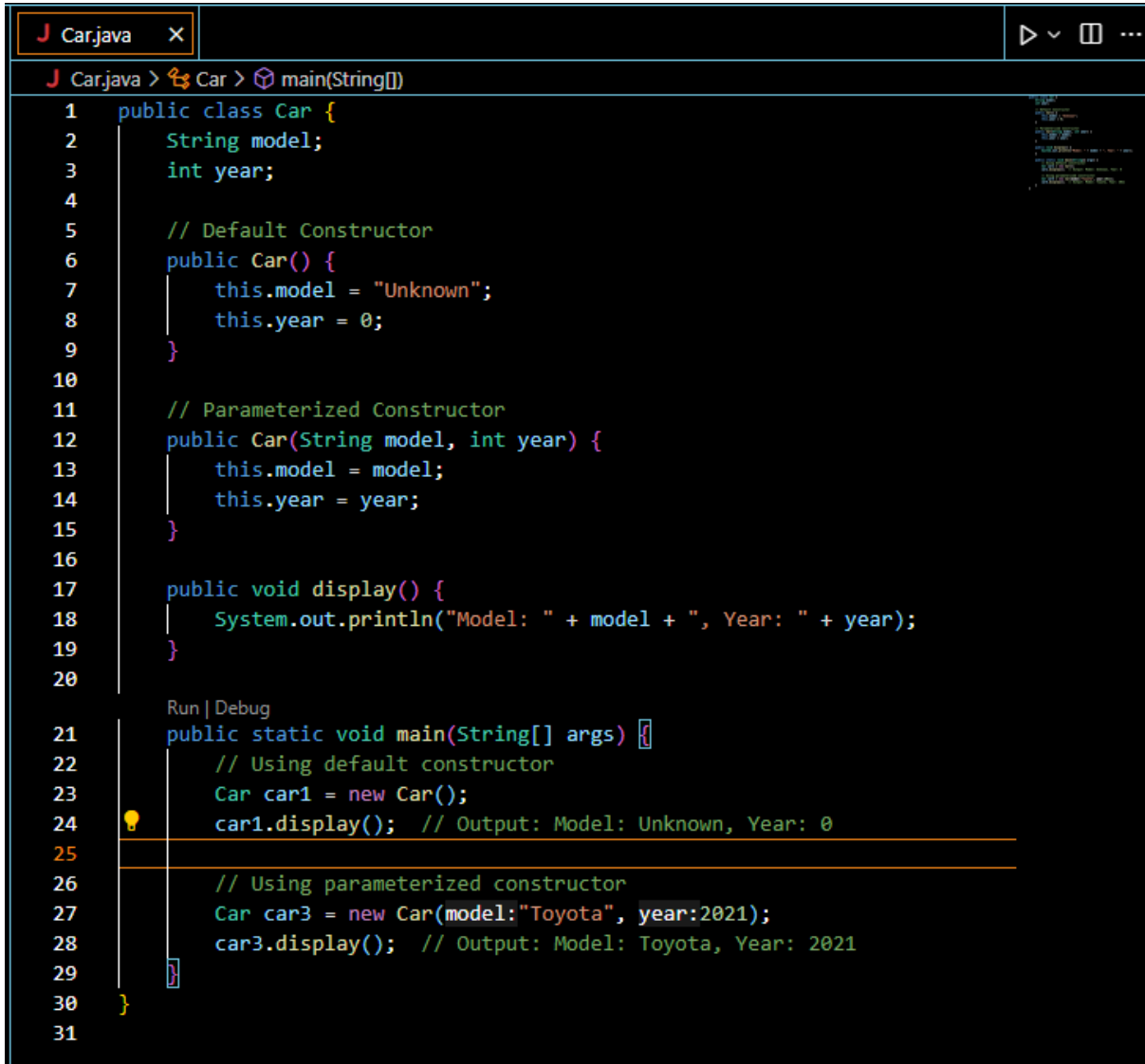
```

Types of Constructors:

🔗 **Default Constructor:** A no-argument constructor automatically provided by Java if no other constructor is defined.

🔗 **No-Argument Constructor:** A user-defined constructor with no parameters.

🔗 **Parameterized Constructor:** A constructor with parameters, allowing the initialization of objects with specific values.



```
J Car.java x
J Car.java > Car > main(String[])
1 public class Car {
2     String model;
3     int year;
4
5     // Default Constructor
6     public Car() {
7         this.model = "Unknown";
8         this.year = 0;
9     }
10
11    // Parameterized Constructor
12    public Car(String model, int year) {
13        this.model = model;
14        this.year = year;
15    }
16
17    public void display() {
18        System.out.println("Model: " + model + ", Year: " + year);
19    }
20
21    Run | Debug
22    public static void main(String[] args) {
23        // Using default constructor
24        Car car1 = new Car();
25        car1.display(); // Output: Model: Unknown, Year: 0
26
27        // Using parameterized constructor
28        Car car3 = new Car(model:"Toyota", year:2021);
29        car3.display(); // Output: Model: Toyota, Year: 2021
30    }
31 }
```

The four pillars of Object-Oriented Programming (OOP) in Java are:

1. **Encapsulation:** Encapsulation involves bundling the data (variables) and methods that operate on the data into a single unit or class. It also restricts access to some of the object's components to protect the object's integrity. Example: Using private fields and public getter/setter methods.
2. **Inheritance:** Inheritance allows a new class (subclass) to inherit attributes and methods from an existing class (superclass), promoting code reusability. Example: class Dog extends Animal inherits properties from Animal.

3. **Polymorphism:** Polymorphism allows objects to be treated as instances of their parent class, with the ability to override methods to provide specific behavior. Example: Method overriding and overloading.
4. **Abstraction:** Abstraction hides complex implementation details and shows only essential features. This can be achieved through abstract classes or interfaces. Example: Defining an interface `Animal` with abstract methods that different animal classes implement.

These pillars work together to create flexible, maintainable, and reusable code in Java.

```
java Copy code

class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers (method overloading)
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to multiply two integers
    public int multiply(int a, int b) {
        return a * b;
    }

    // Method to multiply three integers (method overloading)
    public int multiply(int a, int b, int c) {
        return a * b * c;
    }
}
```

```
java Copy code

class AdvancedCalculator extends Calculator {
    // Overriding the add method to add two integers and a constant value
    @Override
    public int add(int a, int b) {
        return a + b + 10; // Adds 10 to the sum
    }

    // Method Overloading example
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum of 2 and 3: " + calc.add(2, 3));
        System.out.println("Sum of 2, 3, and 4: " + calc.add(2, 3, 4));
        System.out.println("Product of 2 and 3: " + calc.multiply(2, 3));
        System.out.println("Product of 2, 3, and 4: " + calc.multiply(2, 3, 4));


        AdvancedCalculator advCalc = new AdvancedCalculator();
        System.out.println("Sum of 2 and 3 (with override): " + advCalc.add(2, 3));
        System.out.println("Sum of 2.5 and 3.5: " + advCalc.add(2.5, 3.5)); // Overloaded
    }
}
```

Method overloading vs overriding;

Recursion search in an array;

Here's how you can search for an element in an array using recursion in Java:

java

 Copy code

```
public class RecursiveSearch {
    public static void main(String[] args) {
        int arr[] = {1, 3, 5, 7, 9, 11};
        int target = 7;
        int index = search(arr, target, 0);

        if (index != -1) {
            System.out.println("Element found at index: " + index);
        } else {
            System.out.println("Element not found");
        }
    }

    // Recursive method to search for an element in an array
    public static int search(int[] arr, int target, int currentIndex) {
        // Base case: reached the end of the array
        if (currentIndex == arr.length) {
            return -1;
        }

        // Check if the current element is the target
        if (arr[currentIndex] == target) {
            return currentIndex;
        }

        // Recur for the next element in the array
        return search(arr, target, currentIndex + 1);
    }
}
```




Abstraction in Java is a fundamental concept in object-oriented programming (OOP). It involves hiding the complex implementation details of a class and exposing only the essential features to the user. This allows you to focus on what an object does rather than how it does it.

Implementation of Abstraction:

1. **Abstract Class:** A class that cannot be instantiated on its own and may contain abstract methods (methods without implementation) that must be implemented by its subclasses.
2. **Interface:** A completely abstract class that defines methods that must be implemented by any class that implements the interface.

Using an Abstract Class:

java

 Copy code

```
abstract class Animal {
    abstract void sound(); // abstract method


    void sleep() { // non-abstract method
        System.out.println("The animal is sleeping.");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound(); // Calls the implemented method
        dog.sleep(); // Calls the inherited non-abstract method
    }
}
```

Using an Interface:

java

 Copy code

```
interface Animal {
    void sound(); // abstract method
}

class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();
    }
}
```

Key Points:

- **Abstract classes** can have both abstract and non-abstract methods, while **interfaces** can only have abstract methods (though Java 8+ allows default methods with implementation in interfaces).
- Abstract classes are used when there is a common base class, but you don't want to allow creating objects of that base class.
- Interfaces are used to implement multiple inheritance and define a contract that the implementing classes must follow.

Abstract Class:


- **Methods:** Can have both abstract (no body) and concrete (with body) methods.
- **State:** Can have instance variables.
- **Inheritance:** Supports single inheritance (a class can inherit from one abstract class).
- **Accessibility:** Can have constructors, fields, and methods with any visibility (public, protected, private).
- **Use Case:** Best used when you need a base class with shared code for related classes.

Interface:


- **Methods:** Only abstract methods (until Java 8+ which allows default methods with bodies).
- **State:** Cannot have instance variables, only constants (final static fields).
- **Inheritance:** Supports multiple inheritance (a class can implement multiple interfaces).
- **Accessibility:** Methods are implicitly public and abstract, fields are public, static, and final.
- **Use Case:** Best used for defining a contract that multiple classes can implement, even if they are unrelated.

Reverse a string keeping the Spaces at place

java

 Copy code

```
public class ReverseStringWithSpaces {  
    public static void main(String[] args) {  
        String input = "a b c d e";  
        char[] arr = input.toCharArray();  
        int left = 0, right = arr.length - 1;  
  
        while (left < right) {  
            // Skip spaces from the left  
            if (arr[left] == ' ') {  
                left++;  
                continue;  
            }  
            // Skip spaces from the right  
            if (arr[right] == ' ') {  
                right--;  
                continue;  
            }  
            // Swap the characters  
            char temp = arr[left];  
            arr[left] = arr[right];  
            arr[right] = temp;  
  
            left++;  
            right--;  
        }  
  
        String result = new String(arr);  
        System.out.println(result);  
    }  
}
```



Identity Matrix or not?


A screenshot of a Java IDE window titled 'IdentityMatrixCheck.java'. The code defines a class 'IdentityMatrixCheck' with a 'main' method and a 'checkIdentityMatrix' static method. The 'main' method initializes a 3x3 matrix with values {1, 0, 0}, {0, 1, 0}, and {0, 0, 1}. It then calls 'checkIdentityMatrix' and prints the result. The 'checkIdentityMatrix' method first checks if the matrix is square (rows == cols). If not, it returns false. If square, it iterates through each element. For diagonal elements (i == j), it checks if the value is 1. For off-diagonal elements (i != j), it checks if the value is 0. If any check fails, it returns false; otherwise, it returns true. The code is syntactically correct, but there is a small error in the 'main' method's closing brace on line 40, which is highlighted with a yellow lightbulb icon.

```
1 public class IdentityMatrixCheck {
2     public static void main(String[] args) {
3         int matrix[][] = {
4             {1, 0, 0},
5             {0, 1, 0},
6             {0, 0, 1}
7         };
8
9         boolean isIdentity = checkIdentityMatrix(matrix);
10
11         if (isIdentity) {
12             System.out.println("The matrix is an identity matrix.");
13         } else {
14             System.out.println("The matrix is not an identity matrix.");
15         }
16     }
17
18     public static boolean checkIdentityMatrix(int[][] matrix) {
19         int rows = matrix.length;
20         int cols = matrix[0].length;
21
22         // Check if it's a square matrix
23         if (rows != cols) {
24             return false;
25         }
26
27         // Check for identity matrix properties
28         for (int i = 0; i < rows; i++) {
29             for (int j = 0; j < cols; j++) {
30                 if (i == j && matrix[i][j] != 1) {
31                     return false;
32                 } else if (i != j && matrix[i][j] != 0) {
33                     return false;
34                 }
35             }
36         }
37
38         return true;
39     }
40 }
41
```

String anagrams or not?:

anagrams are words or phrases formed by rearranging the letters of another word or phrase, typically using all the original letters exactly once. For example, the word "listen" can be rearranged to form the word "silent." Both words use the same letters, but in a different order, making them anagrams of each other.

java

 Copy code

```
import java.util.Arrays;

public class AnagramCheck {
    public static void main(String[] args) {
        String str1 = "Listen";
        String str2 = "Silent";

        if (areAnagrams(str1, str2)) {
            System.out.println("The strings are anagrams.");
        } else {
            System.out.println("The strings are not anagrams.");
        }
    }

    public static boolean areAnagrams(String str1, String str2) {
        // Remove spaces and convert to lowercase
        str1 = str1.replaceAll("\\s", "").toLowerCase();
        str2 = str2.replaceAll("\\s", "").toLowerCase();

        // Convert strings to char arrays
        char[] arr1 = str1.toCharArray();
        char[] arr2 = str2.toCharArray();


        // Sort both arrays
        Arrays.sort(arr1);
        Arrays.sort(arr2);

        // Compare sorted arrays
        return Arrays.equals(arr1, arr2);
    }
}
```

Compress string

Here's a Java code to compress a string as described:

java

 Copy code

```
public class StringCompressor {
    public static void main(String[] args) {
        String input = "AAABBBBCCDDDDDD";
        String compressed = compressString(input);
        System.out.println(compressed);
    }

    public static String compressString(String str) {
        StringBuilder compressed = new StringBuilder();
        int count = 1;

        for (int i = 0; i < str.length(); i++) {
            if (i + 1 < str.length() && str.charAt(i) == str.charAt(i + 1)) {
                count++;
            } else {
                compressed.append(count > 1 ? count : "").append(str.charAt(i));
                count = 1;
            }
        }

        return compressed.toString();
    }
}
```

Why Java Does Not Support Multiple Inheritance

Java does not support multiple inheritance (where a class inherits from more than one class) due to several reasons:

1. Diamond Problem:

- When a class inherits from multiple classes, there is ambiguity about which method implementation should be used if the parent classes have methods with the same signature. This problem is known as the "Diamond Problem."

2. Simplicity and Design:

- Avoiding multiple inheritance simplifies the language design and reduces complexity in understanding and maintaining the code.


3. Interface Solution:

- Java allows multiple inheritance of types through interfaces. A class can implement multiple interfaces, which provides the flexibility of multiple inheritance without the associated complexity. Interfaces only declare methods but do not provide implementations, thereby avoiding the diamond problem.

1. Single Inheritance:

- **Definition:** A class inherits from only one superclass.
- **Example:**

java

 Copy code

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}


class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); // Inherited method
        myDog.bark(); // Child class method
    }
}
```

2. Multilevel Inheritance:

- **Definition:** A class inherits from another class which is also inherited by a third class, forming a chain.
- **Example:**

java

 Copy code

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Mammal extends Animal {
    void breathe() {
        System.out.println("This mammal breathes air.");
    }
}

class Dog extends Mammal {
    void bark() {
        System.out.println("The dog barks.");
    }
}


public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();    // Inherited from Animal
        myDog.breathe(); // Inherited from Mammal
        myDog.bark();   // Child class method
    }
}
```



8. Hierarchical Inheritance:

- **Definition:** Multiple classes inherit from a single superclass.
- **Example:**

java

 Copy code

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");
    }
}


public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        Cat myCat = new Cat();

        myDog.eat(); // Inherited from Animal
        myDog.bark(); // Dog class method
        ↓
        myCat.eat(); // Inherited from Animal
```

4. Interface Inheritance:

- **Definition:** A class implements an interface, inheriting its abstract methods.
- **Example:**

java

 Copy code

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

class Dog implements Animal {
    public void eat() {
        System.out.println("The dog eats.");
    }

    void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); // Implemented method from interface
        myDog.bark(); // Dog class method
    }
}
```

Types of Polymorphism

1. Compile-Time Polymorphism (Static Polymorphism):

- **Definition:** Achieved through method overloading, where multiple methods in a class have the same name but different parameters (different type or number of arguments).
- **Example:**

```
java Copy code  
  
class Printer {  
    void print(int num) {  
        System.out.println("Printing number: " + num);  
    }  
  
    void print(String text) {  
        System.out.println("Printing text: " + text);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Printer printer = new Printer();  
        printer.print(10);           // Calls the print method with int parameter  
        printer.print("Hello");     // Calls the print method with String parameter  
    }  
}
```


Run-Time Polymorphism (Dynamic Polymorphism):

- **Definition:** Achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass. The method that gets executed is determined at runtime based on the object's actual type.

Key Points of Polymorphism

- **Flexibility:** Polymorphism allows you to write more flexible and reusable code. You can use the same method name to perform different tasks based on the object type or method parameters.
- **Method Overloading vs. Overriding:** Overloading is a compile-time concept, while overriding is a runtime concept. Overloading allows methods with the same name but different parameters in the same class, whereas overriding allows a subclass to provide a specific implementation of a method defined in its superclass.
- **Dynamic Method Dispatch:** In dynamic polymorphism, the method that gets called is determined at runtime based on the object type, not the reference type.

java

 Copy code

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

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

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


public class Main {
    public static void main(String[] args) {
        Animal myAnimal;

        myAnimal = new Dog();
        myAnimal.makeSound(); // Output: Dog barks

        myAnimal = new Cat();
        myAnimal.makeSound(); // Output: Cat meows
    }
}
```

Exception handling:

java


 Copy code

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("This will always execute.");
        }
    }
}
```

Custom Exception:

Example

java

 Copy code

```
// Custom Exception Class
class MyCustomException extends Exception {
    // Constructor
    public MyCustomException(String message) {
        super(message);
    }
}

// Class with Method Throwing Custom Exception
public class CustomExceptionDemo {
    public void checkValue(int value) throws MyCustomException {
        if (value < 0) {
            throw new MyCustomException("Value cannot be negative!");
        }
    }

    public static void main(String[] args) {
        CustomExceptionDemo demo = new CustomExceptionDemo();
        try {
            demo.checkValue(-5); // This will throw MyCustomException
        } catch (MyCustomException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

Explain final, finally, finalize()

chat gpt:

Palindrome Check Code


java Copy code

```
public class PalindromeChecker {

    // Method to check if a string is a palindrome
    public static boolean isPalindrome(String str) {
        int left = 0;
        int right = str.length() - 1;

        while (left < right) {
            // Compare characters from the start and end of the string
            if (str.charAt(left) != str.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }


    public static void main(String[] args) {
        String input = "madam"; // Example input
        // Convert input to lowercase to handle case sensitivity
        if (isPalindrome(input.toLowerCase())) {
            System.out.println(input + " is a palindrome.");
        } else {
            System.out.println(input + " is not a palindrome.");
        }
    }
}
```



Custom exeption:

Step 1: Create a Custom Exception


java

 Copy code

```
// Define a custom exception class
class DivisionByZeroException extends Exception {
    // Constructor that accepts a message
    public DivisionByZeroException(String message) {
        super(message);
    }
}
```

Step 2: Use the Custom Exception in Your Code

java

 Copy code

```
public class Ex {
    public static void main(String[] args) {
        int i = 0;
        int j = 0;

        try {
            if (j == 0) {
                throw new DivisionByZeroException("Cannot divide by zero!"); // Throw custom exception
            }
            int k = i / j;
        } catch (DivisionByZeroException e) { // Catch the custom exception
            System.out.println("Custom Exception: " + e.getMessage());
        }
    }
}
```



Final finally and finalize();

1. `final`


- **Definition:**

- `final` is a keyword used to declare constants, prevent inheritance, and prevent method overriding.

- **Usage:**

- **Final Variables:** If you declare a variable as `final`, you cannot change its value once it is assigned.


java

 Copy code

```
final int MAX = 100;  
// MAX cannot be reassigned
```

- **Final Methods:** A `final` method cannot be overridden by subclasses.


java

 Copy code

```
class Parent {  
    public final void show() {  
        System.out.println("This is a final method.");  
    }  
}
```

- **Final Classes:** A `final` class cannot be subclassed.

java

 Copy code


```
final class MathUtils {  
    // Class content  
}  
// You cannot extend MathUtils
```



2. `finally`

- **Definition:**
 - `finally` is a block used in exception handling that allows you to execute code after a `try` block, regardless of whether an exception is thrown or not.
- **Usage:**
 - The `finally` block is always executed after the `try` block (and associated `catch` blocks, if any), whether an exception occurs or not.
 - It is typically used to release resources like closing files, releasing database connections, etc.

java


 Copy code

```
try {  
    // Code that might throw an exception  
} catch (Exception e) {  
    // Handling exception  
} finally {  
    // Code that will always execute  
}
```

3. `finalize`

- **Definition:**
 - `finalize` is a method in Java, defined in the `Object` class, that is called by the garbage collector before an object is removed from memory.
- **Usage:**
 - The `finalize` method is intended to allow an object to clean up resources before it is garbage collected. However, its use is generally discouraged because it's unpredictable when, or even if, it will be called.
 - With the introduction of the `try-with-resources` statement and other mechanisms, the need for `finalize` has greatly diminished.

java


 Copy code

```
protected void finalize() {  
    // Cleanup code  
}
```

- **Note:** Starting from Java 9, the `finalize` method is deprecated due to performance and reliability issues. It's recommended to use other resource management techniques instead.

Fibonacci Series Using Iteration:

java

 Copy code

```
public class Fibonacci {
    public static void main(String[] args) {
        int n = 10; // Number of terms in the Fibonacci series
        int first = 0, second = 1;

        System.out.println("Fibonacci Series up to " + n + " terms:");


        for (int i = 1; i <= n; ++i) {
            System.out.print(first + " ");

            // Compute the next term
            int next = first + second;
            first = second;
            second = next;
        }
    }
}
```

Fibonacci Series Using Recursion:

Here's an alternative version using recursion:

java

 Copy code

```
public class FibonacciRecursion {

    public static void main(String[] args) {
        int n = 10; // Number of terms in the Fibonacci series

        System.out.println("Fibonacci Series up to " + n + " terms:");

        for (int i = 0; i < n; i++) {
            System.out.print(fibonacci(i) + " ");
        }
    }

    // Recursive method to calculate Fibonacci numbers
    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
        } else {
            return fibonacci(n - 1) + fibonacci(n - 2);
        }
    }
}
```

```
J Ex.java J LinkedListDemo.java X
J LinkedListDemo.java > ...
1 class Node {
2     int data;
3     Node next;
4
5     public Node(int data) {
6         this.data = data;
7         this.next = null;
8     }
9 }
10
11 class LinkedList {
12     Node head;
13
14     // Method to add a new node at the end of the list
15     public void append(int data) {
16         if (head == null) {
17             head = new Node(data);
18             return;
19         }
20         Node current = head;
21         while (current.next != null) {
22             current = current.next;
23         }
24         current.next = new Node(data);
25     }
26
27     // Method to print the linked list
28     public void printList() {
29         Node current = head;
30         while (current != null) {
31             System.out.print(current.data + " -> ");
32             current = current.next;
33         }
34         System.out.println("null");
35     }
36
37     // Method to reverse the linked list
38     public void reverse() {
39         Node previous = null;
40         Node current = head;
41         Node next = null;
42         while (current != null) {
43             next = current.next; // Store next node
44             current.next = previous; // Reverse current node's pointer
45             previous = current; // Move pointers one position ahead
46             current = next;
47         }
48         head = previous;
```

```
J Ex.java J LinkedListDemo.java X
J LinkedListDemo.java > ...
11 class LinkedList {
49 }
50
51 // Method to find the element at a specific index
52 public int findElementAt(int index) {
53     Node current = head;
54     int count = 0;
55
56     while (current != null) {
57         if (count == index) {
58             return current.data;
59         }
60         count++;
61         current = current.next;
62     }
63
64     throw new IndexOutOfBoundsException("Index " + index + " out of bounds.");
65 }
66 }
67 }
```

```
Ex.java  LinkedListDemo.java x
LinkedListDemo.java > ...
67
68 public class LinkedListDemo {
69     Run | Debug
70     public static void main(String[] args) {
71         LinkedList list = new LinkedList();
72
73         // Append elements to the linked list
74         list.append(data:10);
75         list.append(data:20);
76         list.append(data:30);
77         list.append(data:40);
78         list.append(data:50);
79
80         // Print the original list
81         System.out.println(x:"Original Linked List:");
82         list.printList();
83
84         // Reverse the linked list
85         list.reverse();
86         System.out.println(x:"Reversed Linked List:");
87         list.printList();
88
89         // Find element at a specific index
90         int index = 2;
91         try {
92             int element = list.findElementAt(index);
93             System.out.println("Element at index " + index + " is: " + element);
94         } catch (IndexOutOfBoundsException e) {
95             System.out.println(e.getMessage());
96         }
97     }
98 }
```