

1) Write a Java program that reads a series of numbers from a file (input.txt), determines the highest number in the series, calculates the sum of natural numbers upto that highest number, and writes the result to another file (output.txt). Use Scanner to read from the file and PrintWriter to write to the file. Assume the numbers in the input file separated by commas.

Sample Input.txt

10, 55, 1000..... 100

Sample Output.txt

55, 1540, 500500..... 5050

Soln

```
import java.io.*;
import java.util.*;

public class SumNaturalNumbers {
    public static void main(String [] args) {
        String inputFile = "input.txt";
        String outputFile = "Output.txt";
        try (Scanner scanner = new Scanner(new File(inputFile));
             PrintWriter writer = new PrintWriter(new File(outputFile))) {
```

// Read numbers from file

```
scanner.useDelimiter(",");
```

```
list<Integer> numbers = new ArrayList<>();
```

```
while (scanner.hasNext()) {
```

```
if (scanner.hasNextInt()) {
```

```
numbers.add (scanner.nextInt());
```

```
} else {
```

```
scanner.next();
```

```
}
```

```
}
```

// Process numbers and Write to Output file

```
for (int num : numbers) {
```

```
int sum = sumNaturalNumbers(num);
```

```
writer.print (num + ", " + sum + "\n");
```

```
}
```

System.out.println ("Processing complete. Check output.txt for results.");

System.err.println ("Error: File not found.");

// Function to Compute Sum of first N natural number

```
private static int sumNaturalNumbers (int n) {
```

```
return n * (n+1) / 2;
```

```
,
```

Q1 What are the differences you have ever found between static and final fields and methods? Exemplify what will happen if you try to access static method/field with the object instead of class name.

Solution

Q1 Differences Between static and final :

1. static modifier :

- Purpose : The static modifier means that the field or method belongs to the class, rather than to any specific instance of the class.
- Fields : A static field is shared by all instances of the class. It is not tied to any particular instance, and can be accessed using the class name, or via an object reference (though it's generally best practice to access it via the class).
- Methods : A static method can be called without creating an instance of the class. It can only directly access other static members of the class, not instance members.

2. Final modifier :

- Purpose : The final modifier is used to indicate that the field or method cannot be changed once assigned or overridden.
- fields : A final field must be assigned exactly once, either during declaration or in the constructor.
- methods : A final method cannot be overridden by subclasses.
- classes : A final class cannot be subclassed.

Example of static and final fields

```
public class Example {
```

```
    static int staticfield = 10;
```

```
    final int finalField;
```

```
    public Example(int value) {
```

```
        this.finalField = value;
```

```
}
```

```
}
```

- key takeaways :

- static members belong to the class, and final members cannot be changed or overridden.

3] Write a java Program to find all factorion numbers within given range. A number is called a factorion if, the sum of the factorials of its digits equals the number itself. The program should take user input for the lower and upper bounds of the range and print all factorion numbers within that range.

Code

```
import java.util.Scanner;
```

```
public class FactorionFinder {
```

```
    private static final int[] FACTORIALS = new int[10]
```

```
    static {
```

```
        FACTORIALS[0] = 1;
```

```
        int fact = 1;
```

```
        for (int i = 1; i < 10; i++) {
```

```
            fact *= i;
```

```
            FACTORIALS[i] = fact;
```

```
}
```

```
}
```

```
private static int sumOfDigitFactorials(int num) {  
    int sum = 0, temp = num;  
    while (temp > 0) {  
        int digit = temp % 10;  
        sum += FACTORIALS[digit];  
        temp /= 10;  
    }  
    return sum;  
}
```

```
private static boolean is Factorion (int num) {  
    return num == sumOfDigitFactorials(num);  
}  
  
public static void main (String [] args) {  
    Scanner scanner = new Scanner (System.in);  
    System.out.print ("Enter the lower bound of the  
                      range: ");  
    int lowerBound = scanner.nextInt();  
    System.out.print ("Enter the upper bound of the  
                      range: ");  
    int upperbound = scanner.nextInt();
```

```
System.out.print("Factorion numbers in the range:  
boolean found = false;  
for (int i = lowerBound; i <= upperBound; i++) {  
    if (isFactorion(i)) {  
        System.out.print(i + " ");  
        found = true;  
    }  
}  
if (!found) {  
    System.out.println("None");  
}  
Scanner.close();
```

4) Distinguish the differences among class, local and instance variables what is significant of this keyword?

1. Class Variables

- Definition : Also known as static variables, class variables are declared with the static keyword within a class.
- Scope & Lifetime :
- They belong to the class itself rather than to any particular instance.
- They are created when the class is loaded and exist for as long as the class is loaded in memory.
- Every instance of the class shares the same class variable.

2. Instance Variables :

Instance variables are declared in a class but outside any method, and they are not marked as static.

Scope & Lifetime

- They belong to an individual object of the class.
- Each time a new object is created, a new copy of the instance variable allocated.

- They exist as long as the object exists.

? borrowed with no modifications in value addition is costed

3. Local Variables :

- Local variables are declared inside a method.

? added side condition states as return value : initialized

- They exist only within the method where they are declared.

- Their lifetime is limited to the execution of that method.

- They are not visible outside the method and must be explicitly initialized before use.

- They are not visible outside the method and must be explicitly initialized before use.

4. Significance of this keyword

- This keyword is a reference to the current object - the instance on which the method or constructor is being called.

- It is especially useful for distinguish between instance variables and parameters or local variables that have the same name.

- It is especially useful for distinguish between instance variables and parameters or local variables that have the same name.

- It is especially useful for distinguish between instance variables and parameters or local variables that have the same name.

51 Write a Java Program that defines a method to calculate the sum of all elements in an integer array. The method should take an integer array as a parameter and return the sum. Demonstrate this method by passing an array of integers from the main method.

Solve

```
public class ArraySumCalculator {  
    public static int calculateSum(int[] numbers) {  
        int sum = 0;  
        for (int num : numbers) {  
            sum += num;  
        }  
        return sum;  
    }  
    public static void main(String[] args) {  
        int[] numbers = {1, 2, 3, 4, 5};  
        int sum = calculateSum(numbers);  
        System.out.println("Sum of array elements: " + sum)  
    }  
}
```

Q6 What is called Access Modifiers? Compare the accessibility of public, private and protected modifier.

Describe different types of variable in java with example.

Access Modifiers:

Access Modifiers in java define the scope and visibility of variables, methods, and classes. They control how other parts of a program can interact with the members of a class.

Comparison of Access modifiers

Modifier	Accessibility within class	Accessibility within package	Accessibility in subclases	Accessibility outside package
public	yes	yes	Yes	yes
private	yes	No	No	No
protected	yes	yes	Yes	No
Default	yes	yes	No	No

Types of Variables in Java

Java has three main type variables:

1) Local Variables

- Declared inside a method, constructor, or block.
- Scope is limited to the block where they are defined.
- No default value, must be initialized before use.

Example:

```
public class Example {
```

```
    public void display() {
```

```
        int x = 10;
```

```
        System.out.println("Local Variable: " + x);
```

```
}
```

2) Instance Variables

- Declared inside a class but outside any method.
- Associated an instance of the class.
- Default values.
 - int → 0, double → 0.0, boolean → false, String → null.

Example:

```
public class Example {
```

```
    int num = 28;
```

```
    public void show() {
```

```
        System.out.println("Instance variable: " + num);
```

```
}
```

3) Static Variable

- Declared with the static keyword inside a class.
- Shared among all objects of the class.
- Memory: allocated only once at class loading time.

Example:

```
public class Example {
```

```
    static int count = 0; // static variable
```

```
    public void increment() {
```

```
        count++;
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        Example obj1 = new Example();
```

```
        Example obj2 = new Example();
```

```
obj1. increment();
```

```
obj2. increment();
```

```
System.out.println("Static Variable: " + Example.count);
```

```
}
```

```
} (position 88 character) if
```

```
:21 too many fall-throughs (11) (line 9, column 2)
```

Q1 Write a Java Program Quadratic equation.

Solve

```
import java.util.Scanner;
```

```
public class QuadraticRootFinder {
```

```
public static void main (String [] args) {
```

```
Scanner scanner = new Scanner (System.in);
```

```
System.out.println ("Enter co-efficients a, b, and c:");
```

```
int a = scanner.nextInt();
```

```
int b = scanner.nextInt();
```

```
int c = scanner.nextInt();
```

```
double discriminant = b*b - 4*a*c;
```

```
if (discriminant < 0) {
```

```
System.out.println ("No real roots.");
```

```
}
```

```
else {  
    double root1 = (-b + Math.sqrt(discrimant)) / (2.0 * a);  
    double root2 = (-b - Math.sqrt(discrimant)) / (2.0 * a);
```

```
    if (root1 >= 0 && root2 >= 0) {
```

System.out.println ("The smallest positive root is :
" + Math.min (root1, root2));

```
} else if (root1 > 0) {
```

System.out.println ("The smallest positive root is
" + root1);

```
} else if (root2 > 0) {
```

System.out.println ("The smallest positive root is :
" + root2);

```
} else {
```

System.out.println ("No positive roots.");

```
}
```

```
}
```

```
scanner.close();
```

```
}
```

```
1
```

IT23019

8| Write a Program character type Detection

code

```
import java.util.Scanner;

public class CharacterChecker {
    public static void main (String [] args) {
        Scanner scanner = new Scanner (System.in);
        System.out.print ("Enter a character: ");
        char ch = scanner.next ().charAt (0);

        if (Character.isLetter (ch)) {
            System.out.println (ch + " is a letter.");
        } else if (Character.isDigit (ch)) {
            System.out.println (ch + " is a digit.");
        } else if (Character.isWhitespace (ch)) {
            System.out.println ("It is whitespace character.");
        } else {
            System.out.println ("ch + " is a special character.");
        }
        scanner.close ();
    }
}
```

Passing an Array to a Function in Java

In Java, you can pass an array to a function just like any other variable. Here's an example:

Code

```
public class ArrayExample {
    public static void PrintArray (int [] arr) {
        System.out.println ("Array elements : ");
        for (int num : arr) {
            System.out.print (num + " ");
        }
        System.out.println ();
    }
    public static void main (String [] args) {
        int [] numbers = {1, 2, 3, 4, 5};
        PrintArray (numbers);
    }
}
```

Gupinath

IT23019

Q1

Solve

Method overriding in Java

Method overriding happens when a subclass provides a new version of a method that already exists in its SuperClass. The method in the Subclass must have:

- The same name as the SuperClass method.
- The same parameters.
- The same return type.

How Does it work?

When you call a method on a object of the subclass, Java will execute the overridden method in the subclass, not the one in the Superclass. This allows Java to support runtime polymorphism.

Example:

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

```
class Dog extends Animal {  
    void sound() {  
        System.out.println("Dog barks");  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        Animal mypet = new Dog();  
        mypet.sound();  
    }  
}
```

Super to call the Superclass' method

The **super** keyword allows the subclass to call a method from the superclass. This is useful if you want to add to the original method instead of completely replacing it.

Example:

IT23019

Class Animal {

void sound () {

System.out.println ("Animal makes a sound");

}

Class Dog extends Animal {

void sound () {

super.sound ();

System.out.println ("Dog barks");

}

public class Main {

public static void main (String [] args) {

Dog myDog = new Dog();

myDog.sound ();

}

potential issue when overriding methods.

1. constructor cannot be overridden

- constructors are not inherited, so they cannot be overridden.

- If a super class only has a parameterized constructor, the subclass must call it explicitly using Super().

Example

```
Class Animal {  
    Animal (String name) {  
        System.out.println ("Animal: " + name);  
    }  
}  
  
Class Dog extends Animal {  
    Dog () {  
        Super ("Buddy");  
        System.out.println ("Dog is created");  
    }  
}  
  
{ public class Main {  
    public static void main (String [] args) {
```

```
Dog myDog = new Dog();
```

```
}
```

2. Final methods cannot be overridden

- If a method is marked final in the ~~superclass~~ Super class, it cannot be changed in the subclass.

Example:

```
Class Animal {
```

```
    final void sleep() {
```

```
        System.out.println("Animal sleeps");
```

```
}
```

```
Class Dog extends Animal {
```

```
}
```

3) Static methods are not overridden

- static method belong to the class, not the instance, so they do not support overriding.

Example

```
class Animal {  
    static void info() {  
        System.out.println("Animal class info");  
    }  
}  
class Dog extends Animal {  
    static void info() {  
        System.out.println("Dog class info");  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        Animal obj = new Dog();  
        obj.info();  
    }  
}
```

Access Modifiers Rule

- Overridden method cannot have a more restrictive access level than the original.

Example:

Class Animal {

protected void eat () { }

}

class Dog extends Animal {

public void eat () { }

}

10 Differentiate Between static and Non-static members.

solve static member

- Declared using the static keyword.
- Belong to the class rather than an instance of the class.
- Shared among all instances of the class.
- Can be accessed using the class name.
- Cannot access non-static members directly.

Example:

```
Class staticExample {  
    static int count = 0;  
    static void showCount() {  
        System.out.println ("Count: " + count);  
    }  
}  
public class Main {  
    public class static void main (String [] args) {  
        static Example.count = 5 ;  
        static Example.showCount();  
    }  
}
```

2) Non-static Members

- Do not use the static keyword.
- Belong to an instance of the class.
- Each instance has its own copy of the non-static members.
- Can access both static and non-static members.

Example of Non-static Member:

```
class NonStaticExample {  
    int value;  
    void display() {  
        System.out.println ("Value: " + value);  
    }  
}  
  
public class Main {  
    public static void main (String [] args) {  
        NonStaticExample obj1 = new NonStaticExample ();  
        obj1.value = 10;  
        obj1.display ();  
        NonStaticExample obj2 = new NonStaticExample ();  
        obj2.value = 20;  
        obj2.display ();  
    }  
}
```

Palindrome Checker Program

```
import java.util.Scanner;
```

```
public class PalindromeChecker {
```

```
    public static boolean isPalindrome(int num) {
```

```
        int original = num, reverse = 0, remainder;
```

```
        while (num > 0) {
```

```
            remainder = num % 10;
```

```
            reverse = reverse * 10 + remainder;
```

```
            num /= 10;
```

```
}
```

```
        return original == reverse;
```

```
}
```

```
    public static boolean isPalindrome(String str) {
```

```
        int left = 0, right = str.length() - 1;
```

```
        while (left < right) {
```

```
            if (str.charAt(left) != str.charAt(right)) {
```

```
                return false;
```

```
}
```

```
            left++;
```

```
            right--;
```

```
    return true;
```

```
}
```

```
public static void main (String [] args) {
```

```
Scanner scanner = new Scanner (System.in);
```

```
System.out.print ("Enter a number or a string : ");
```

```
if (scanner.hasNextInt ()) {
```

```
int num = scanner.nextInt ();
```

```
if (isPalindrome (num)) {
```

```
System.out.println (num + " is a palindrome.");
```

```
} else {
```

```
String str = scanner.next ();
```

```
if (isPalindrome (str)) {
```

```
System.out.println ("\\" + str + "\\" is a palindrome.");
```

```
} else {
```

```
System.out.println ("\\" + str + "\\" is not palindrome.)
```

```
}
```

```
scanner.close ();
```

```
}
```

```
}
```

Abstraction

Abstraction is a concept in Object-Oriented Programming (OOP) that hides the implementation details and only shows the essential features of an object. It is achieved using abstract classes and interfaces.

Example of Abstraction in Java

```
abstract class Vehicle {
    abstract void start();
    void fuel() {
        System.out.println("Fueling the vehicle...");
    }
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car is starting with a key.");
    }
}
```

public class Abstraction_Example

```
public static void main(String[] args) {
    Vehicle myCar = new Car();
```

```
my car.start();  
my car.fuel();  
}  
}
```

2/ Encapsulation

Encapsulation is the process of binding data and methods together with in a class and restricting direct access to some details. It is implemented using private access modifiers and getter & setter methods.

Example:

```
class BankAccount {  
    private double balance;  
    private void deposit(double amount) {  
        if(amount > 0) {  
            balance += amount;  
        }  
    }  
    public double getBalance() {  
        return balance;  
    }  
}  
public class EncapsulationExample {  
    public static void main(String[] args) {
```

```

BankAccount account = new BankAccount();
account.deposit(1000);
System.out.println("Balance: " + account.getBalance());
}

```

} ~~bottom line of code to 220079 with mistakes~~
 Differences Between Abstract Class and
 interface ~~at 22008 feasible from notes~~

Abstract Class	Interface
<ol style="list-style-type: none"> 1. A class that contains abstract and concrete methods. 2. Can have both abstract and concrete methods. 3. Can have instance variables. 4. Can have constructor. 5. A class can extend only one abstract class. 6. Used when classes share common behaviour with some default implementation. 	<ol style="list-style-type: none"> 1. A collection of abstract methods that must be implemented by a class. 2. Only abstract methods. 3. Can only have public static final variables. 4. Cannot have constructor. 5. A class can implement interfaces. 6. Used when unrelated classes need to follow the same contract.

Gupinath

IT23019

121

Solve

```
Class BaseClass {  
    void PrintResult (String result) {  
        System.out.println (result);  
    }  
}  
  
Class SumClass extends BaseClass {  
    void computeSum () {  
        double sum = 0; // "base" part "to MD") + 1000  
        for (double i = 1.0; i > 0.1; i -= 0.1) {  
            sum += i;  
        }  
        PrintResult ("Sum of the Series: " + sum);  
    }  
}  
  
Class DivisorMultipleClass extends BaseClass {  
    int gcd (int a, int b) { // "part" + 1000  
        while (b != 0) {  
            int temp = b; // "part" + 1000  
            b = a % b;  
            a = temp;  
        }  
        return b;  
    }  
}
```

```

a = temp;
}
return a;
}
int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}
void computeGCDLCM (int a, int b) {
    PrintResult ("GCD of " + a + " and " + b + " is: " + gcd(a, b));
    PrintResult ("LCM of " + a + " and " + b + " is: " + lcm(a, b));
}
}

class NumberConversionClass extends BaseClass {
    void ConvertNumber (int num) {
        PrintResult ();
        PrintResult ("Decimal: " + num);
        PrintResult ("Binary: " + Integer.toBinaryString(num));
        PrintResult ("Octal: " + Integer.toOctalString(num));
        PrintResult ("Hexadecimal: " + Integer.toHexString(num));
    }
}

```

```
Class CustomPrintClass extends BaseClass {  
    void pr(String message) {  
        PrintResult ("[Custom Print] " + message);  
    }  
}  
  
public class MainClass {  
    public static void main (String [] args) {  
        SumClass sumObj = new SumClass ();  
        sumObj.computeSum ();  
        DivisorMultipleClass divMulObj = new DivisorMultipleClass ();  
        divMulObj.computeGCDLCM (24, 36);  
        Number_ConversionClass numConvObj = new NumberConversionClass ();  
        numConvObj.convertNumber (255);  
        CustomPrintClass customPrintObj = new CustomPrintClass ();  
        customPrintObj.pr ("This is a formatted message.");  
    }  
}
```

13] Java Program that implements the UML -
diagram :

Solve

```
import java.util.Date;
```

```
class GeometricObject {
```

```
    private String color;
```

```
    private boolean filled;
```

```
    private Date dateCreated;
```

```
    public GeometricObject() {
```

```
        this.color = "white";
```

```
        this.filled = false;
```

```
        this.dateCreated = new Date();
```

```
}
```

```
    public GeometricObject(String color, boolean filled) {
```

```
        this.color = color;
```

```
        this.filled = filled;
```

```
        this.dateCreated = new Date();
```

```
}
```

```
public String getColor() {  
    return color;  
}  
  
public void setColor(String color) {  
    this.color = color;  
}  
  
public boolean isFilled() {  
    return filled;  
}  
  
public void setFilled(boolean filled) {  
    this.filled = filled;  
}  
  
public Date getDateCreated() {  
    return dateCreated;  
}  
  
@Override  
public String toString() {  
    return "Color :" + color + ", Filled :" + filled + ",  
    \"Created On :\" + dateCreated;  
}  
}
```

Ocupinath
IT23019

class circle extends GeometricObject {
 private double radius;

 public Circle () {
 this.radius = 1.0;
 }

 public Circle (double radius, String color, boolean filled) {
 super (color, filled);
 this.radius = radius;
 }

 public double getRadius () {
 return radius;
 }

 public void setRadius (double radius) {
 this.radius = radius;
 }

 public double getArea () {
 return Math.PI * radius * radius;
 }

 public double getPerimeter () {
 return 2 * Math.PI * radius;
 }
}

```
public double getDiameter() {  
    return 2 * radius;  
}  
  
public void PrintCircle() {  
    System.out.println("Circle: radius = " + radius);  
}  
  
class Rectangle extends GeometricObject {  
    private double width;  
    private double height;  
  
    public Rectangle() {  
        this.width = 1.0;  
        this.height = 1.0;  
    }  
  
    public Rectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
}
```

```
public Rectangle(double width, double height, String color,  
boolean filled) {  
    Super(color, filled);  
    this.width = width;  
    this.height = height;  
}  
  
public double getWidth() {  
    return width;  
}  
  
public void setWidth(double width) {  
    this.width = width;  
}  
  
public double getHeight() {  
    return height;  
}  
  
public double getArea() {  
    return width * height;  
}  
  
public double getPerimeter() {  
    return 2 * (width + height);  
}  
}
```

Public class TestGeometricObjects {

public static void main (String [] args) {

Circle circle = new Circle(5.0, "Red", true);

System.out.println ("Circle:");

System.out.println ("Radius: " + circle.getRadius());

System.out.println ("Area: " + circle.getArea());

System.out.println ("Perimeter: " + circle.getPerimeter());

System.out.println ("Diameter: " + circle.getDiameter());

System.out.println (circle.toString());

System.out.println ("In Rectangle:");

Rectangle rectangle = new Rectangle(4.0, 7.0, "Blue", false);

System.out.println ("Width: " + rectangle.getWidth());

System.out.println ("Height: " + rectangle.getHeight());

System.out.println ("Area: " + rectangle.getArea());

System.out.println ("Perimeter: " + rectangle.getPerimeter());

System.out.println (rectangle.toString());

}

}

14] Significance of BigInteger

In Java, BigInteger is a class in the java.math package used to handle arbitrarily large integers. It is significant because:

1. Handles Large Numbers: Primitive data types like int and long ~~can~~ have limits (int upto $2^{31}-1$ and long upto $2^{63}-1$). BigInteger can store much larger values.
2. Supports Arithmetic Operations: It provides methods for addition, subtraction, multiplication, division and modular operations.
3. Useful in Cryptography: It is widely used in encryption algorithms where large numbers are required.
4. Immutable: Like, String, BigInteger objects are immutable, meaning operations create new objects instead of modifying the existing ones.

Java Program for Factorial Using BigInteger

```
import java.math.BigInteger;
import java.util.Scanner;

public class FactorialBigInteger {
    public static BigInteger factorial (int num) {
        BigInteger fact = BigInteger.ONE;
        for(int i=2; i<=num; i++) {
            fact = fact.multiply(BigInteger.valueOf(i));
        }
        return fact;
    }

    public static void main (String [] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        BigInteger result = factorial(number);
        System.out.println("Factorial of " + number + " is: "
                           + result);
        scanner.close();
    }
}
```

15 | Comparison of Abstraction Classes and interfaces in Java.

Abstract Class	Interface
1. A class that cannot be instantiated. It is a collection of abstract methods and may have both abstract and concrete methods.	methods and default/static methods with no instance fields.
2. Can have abstract, concrete, static and final methods.	2. Can have abstract methods, default methods, static methods but no constructor.
3. Can have instance variable with any access modifier.	3. Cannot have constructor.
4. A class can extend only one abstract class.	4. A class can implement multiple interfaces.
5. Can have any access modifier for methods and variables.	5. All methods are implemented public.

- When to use an Abstract Class over an Interface
1. Shared state: If multiple related classes required common fields or methods with implementations, an abstract class is preferred.
 2. Partial implementation: If you want to provide some common functionality to subclasses while enforcing the implementation of specific methods, use an abstract class.
 3. Performance considerations: Calling methods from an abstract class is slightly faster than calling interface methods due to virtual table lookup optimizations.
 4. Version Compatibility: Abstract classes provide more flexibility in modifying the base class without breaking existing implementations.

Can a class implement Multiple Interfaces in Java?

```
interface A {  
    void methodA();  
}  
interface B {  
    void methodB();  
}
```

```
class MyClass implements A, B {  
    public void methodA() {  
        System.out.println("Method A Implementation");  
    }  
    public void methodB() {  
        System.out.println("Method B Implementation");  
    }  
}
```

④ Implications using multiple Interfaces

- Code Reusability & Flexibility: Since a class can implement multiple interfaces, it allows for greater modularity.
- Conflict Resolution: If two interfaces have methods with the same signature but different default implementations.

Code
interface X {
 default void show() {
 System.out.println("X's show");
 }
}

```
interface Y {  
    default void show() {  
        System.out.println("Y's.show");  
    }  
}  
  
class MyClass implements X, Y {  
    public void show() {  
        System.out.println("Resolved show method");  
    }  
}
```

Gupinath
ITR3019

16| Polymorphism in Java

Polymorphism in Java refers to the ability of an object to take multiple forms. It allows a single interface to be used for different types, enabling code flexibility and reusability. Polymorphism can be achieved through method overriding and method overloading, but the most common form in Java is runtime polymorphism.

Dynamic method Dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime rather than at compile time. This allows Java to determine the actual method to be executed based on the runtime type of the object, not the reference type.

Example: Polymorphism using Inheritance and 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");  
    }  
}
```

Animal
Dog

Class Cat extends Animal {

@Override

```
void make sound () {  
    System.out.println ("cat meows");  
}
```

}

public class PolymorphismExample {

```
public static void main (String [] args) {
```

```
    Animal myAnimal;
```

```
    myAnimal = new Dog();
```

```
    myAnimal.make sound();
```

```
    myAnimal = new Cat();
```

```
    myAnimal.make sound();
```

```
}
```

```
}
```

□ Trade-offs Between using polymorphism and specific method calls

using polymorphism	Using Specific method calls
<ol style="list-style-type: none">1. High code is more reusable and adaptable2. Slightly slower due to dynamic method lookup.3. Easier to maintain and extend.4. Slightly higher due to vtable usage.	<ol style="list-style-type: none">1. Low code is tightly coupled.2. Faster due to direct method calls.3. Harder to maintain.4. Lower since method calls are direct.

17L Differences Between ArrayList and LinkedList in Java.

ArrayList	LinkedList
1. $O(1)$ (Direct indexing)	1. $O(n)$ (sequential traversal)
2. Amortized $O(1)$ (Occasional resize)	2. $O(1)$ (Direct append)
3. $O(n)$ (shifting required)	3. $O(1)$ (If node reference is known, otherwise $O(n)$)
4. Less overhead (array only)	4. More overhead (extra node references)
5. Search contains $O(n)$	5. Search contains $O(n)$

When to use which?

- Use ArrayList when:
 - You need fast random access ($O(1)$ lookup).
 - Infrequent insertions/deletions.
 - Memory efficiency is a priority.

- Use Linked list when: IT23019

- Frequent insertions / deletions in the middle are required.

- Sequential access is preferred over random access.

- Memory Overhead is not a concern.

18] Random Generator

```
import java.util.Random;
```

```
import java.util.Arrays;
```

```
public class CustomRandomGenerator {
```

```
    private static final int[] SEED_ARRAY = {3, 7, 11,
```

```
    17, 23};
```

```
    public static int myRand(int n) {
```

```
        long timestamp = System.currentTimeMillis();
```

```
        int seed = SEED_ARRAY[(int)(timestamp % SEED_ARRAY.length)];
```

```
        return (int)((timestamp * seed) % n);
```

```
}
```

```
public static int[] myRand (int n, int count) {  
    int[] randomNumbers = new int [count];  
    for (int i = 0; i < count; i++) {  
        randomNumbers[i] = myRand(n);  
    }  
    return randomNumbers;  
}
```

```
public static void main (String [] args) {  
    System.out.println ("Single random number: " + myRand(100));  
    System.out.println ("Multiple random numbers: " + Arrays.toString  
        (myRand (100, 5)));  
}
```

```
}
```

19] Multithreading in Java

Multithreading in java allows multiple threads to run concurrently, improving performance and responsiveness. Java provides built-in support - through the Thread class and Runnable interface.

Thread class vs Runnable Interface

1. Thread Class - Extending Thread requires overriding the run() method but prevents extending other classes.
2. Runnable Interface - Implementing Runnable allows better flexibility, enabling the class to extend other classes.

Example:

```
Class MyThread extends Thread {  
    public void run () { system.out.println ("thread  
using Thread class"); }  
}
```

```
class MyRunnable implements Runnable {  
    public void run () { System.out.println ("Thread using  
    Runnable interface"); }  
}  
  
public class ThreadExample {  
    public static void main (String [] args) {  
        new MyThread ().start ();  
        new Thread (new MyRunnable ()).start ();  
    }  
}
```

Potential Issue in Multithreading

- Race condition - Multiple threads modifying data simultaneously, causing inconsistencies.
- Deadlocks - Two or more threads waiting for each other to release resources.
- Starvation - Low-priority threads not getting CPU time.
- Livelock - Threads continuously responding to each other without making progress.

using Synchronised for Thread Safety

The synchronized keyword prevents multiple threads from accessing ~~exact~~ critical code simultaneously.

Example

```
class SharedResource {  
    private int count = 0;  
    public synchronized void increment() { count++; }  
    public synchronized int getCount() { return count; }  
}
```

Deadlock Scenario:

```
class Resource {  
    void methodA(Resource r) {  
        synchronized (this) {  
            System.out.println(Thread.currentThread().getName() +  
                " locked resource 1");  
            synchronized (r) {  
                System.out.println(Thread.currentThread().getName() +  
                    " locked resource 2");  
            }  
        }  
    }  
}
```

Gupinath
IT23019

```
System.out.println(Thread.currentThread().getName() +  
    " locked Resource 2");  
}  
}  
}
```

Prevention:

- Lock Ordering - Always acquire locks in a fixed order.
- Try-Lock Mechanism - Use `tryLock()` from `ReentrantLock` instead of `Synchronized`.

Using ReentrantLock:

```
import java.util.concurrent.locks.*;  
  
class SafeResource {  
    private final Lock lock1 = new ReentrantLock();  
    private final Lock lock2 = new ReentrantLock();  
  
    void safeMethod() {  
        if (lock1.tryLock()) {  
            try {  
                System.out.println("Safe execution without deadlock.");  
            } finally {  
                lock1.unlock();  
            }  
        }  
    }  
}
```

Audiobook
Series II

```
    + (Comments. ( ) book->commenting) reading too much
        finally { lock2.unlock(); }
    }
    finally { lock1.unlock(); }
}
}


```

• writing exit from school determine - always - first - look
decide where went () school see - remember do first
• circumstances to begin

• Recitation

(* - 2001 , - 2002 , - 2003 , - 2004 , - 2005 , - 2006)

{ () school room = school and wait when
; () school room = school and school returning
 } () bottom floor then
 } (() school & class) fit
 } (* . audiobook reading with other children - but another) fit

Exception Handling in Java (short overview)

Exception handling in Java ensures that runtime errors do not disrupt program flow. It is managed using try, catch, finally, throw, and throws key words.

Checked vs unchecked Exceptions

- Checked Exceptions: Checked at compile-time; must be handled using try-catch or declared with throws.
- Unchecked Exceptions: Occur at runtime; typically caused by programming errors.

Creating & Throwing Custom Exceptions

- Extend Exception for checked exceptions or RuntimeException for unchecked exceptions.
- Use throw to explicitly throw an exception.

class CustomException extends Exception {

 public CustomException (String message)

{

 Super (message);

}

throw new CustomException ("Custom checked exception");

Role of throw and throws

- throw : Used to manually throw an exception.
- throws : Declares exceptions a method might throw.

public void method() throws IOException {

 throw new IOException ("Error occurred");

}

~~Preventing~~

Gupinath
IT23019

Preventing Resource Leaks

- Use finally Block: Ensures resources are closed.
- Try-with-Resources (AutoCloseable): Java 7+ automatically closes resources.

```
try (FileInputStream file = new FileInputStream("file.txt"))  
{  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Interfaces in Java can have default methods starting from Java 8, and this feature introduces some interesting distinctions when compared to abstract classes.

Interfaces with default Methods:

- Interfaces can have both abstract methods and default methods.
- Default methods allow you to add functionality to an interface without breaking existing implementations.
- Since interfaces can have no state, they can only provide behavior or constants.

Abstract classes:

Q1

- Abstract classes can hold both state (instance variable) and behaviour (method).
- They can have constructors, non-static fields, and concrete methods.

Conflicting Method Implementations between abstract classes and Interfaces:

If both an abstract class and interface provide conflicting method implementations, the concrete class must explicitly resolve the conflict by overriding the method.

Example code :

```
interface A {
```

```
    default void hello() {
```

```
        System.out.println("Hello from A");
```

```
}
```

```
abstract class B implements A {
```

```
    @Override
```

```
    public void hello() {
```

```
        System.out.println("Hello from B");
```

```
}
```

```
class C extends B {
```

```
    @Override
```

```
    public void hello() {
```

```
        System.out.println("Hello from C");
```

```
}
```

```
}
```

Feature	Hashmap	Tree map	LinkedHashMap
Internal Structure	Hash table	Red black tree	Hash Table + Doubly linked List
Order of Elements	No order guaranteed	Sorted	Insertion order
Insertion time	$O(1)$	$O(\log n)$	$O(1)$
Deletion time	$O(1)$	$O(\log n)$	$O(1)$
When to use	Fast access with no order needed	When you need sorted keys or range queries	When you need insertion / access order

How does TreeMap differ from HashMap in ordering

- Hasmap does not maintain any order of its keys
The order in which you iterate over the keys or entries can be arbitrary.
- TreeMap, on the other hand maintains a sorted order of the key based on their natural ordering or a comparator, meaning the elements will be traversed in ascending order by default.

Q: What is the difference between TreeMap and HashSet?
Ans: Both are sorted collections but Hashset is not ordered while TreeMap is ordered.
Both are implemented with tree, so they provide O(log n) time complexity for insertion, deletion and search operations.

K Gopinath

23

IT-23019

Static Binding and Dynamic Binding in Java and their relation to polymorphism.

Static Binding : Occurs when the method call is resolved at compile-time. This happens when the method being invoked is determined based on the type of the reference. Static methods, private method, and final methods are examples of methods that are statically bound.

Dynamic Binding : Occurs when the method call is resolved at runtime, based on the actual object type, not the reference type. This is a key feature of runtime polymorphism. Methods overridden in subclasses are dynamically bound.

Static binding Example :

a. animal.sound();

b. animal.bark();

24

Advantages of ExecutorService over Manual Thread Management:

1. Thread Pool management.

2. Automatic thread reuse.

3. Task scheduling.

4. Graceful shutdown.

5. Errors Handling and Monitoring.

Benefits of using callable over runnable.

Runnable:

- * runnable tasks are designed to perform an action but do not return any result. It is used when you simply want to execute some code without needing any feedback.
- * Runnable cannot throw checked exceptions directly; they must be wrapped in a try-catch block.

Callable:

- * Callable is similar to Runnable but can return a result. This is useful when you need the outcome of the task or need to compute a value.

* Callable can throw checked exceptions, which means you can propagate exceptions from the task directly.

→ makes task different from all earlier threads till . *

Differences between submit() and execute() Methods.

① execute method :

* This method is used to submit a runnable task to the ExecutorService. The execute() method

doesn't return any result or feedback.

it only signals that the task has been submitted for execution.

* It doesn't provide a way to handle exceptions thrown by the task.

submit method

it performs background work and adds it to
the list of waiting tasks along with log

* The submit method is more versatile than execute

It allows you to submit both runnable and callable
tasks to the Executor Service.

• If the task throws an exception, it can be retrieved
through the future object

↳ after throwing an exception, feedback
method will return Future object which contains

exception object

↳ if we want to know about the exception

↳ handle and print

85

code:

```
Class NegativeRadiusException extends Exception{  
    public NegativeRadiusException(String message){  
        super(message);  
    }  
    public class circle{  
        private double radius;  
        public void setRadius(double radius) throws NegativeRadius  
            Exception {  
            if(radius<0){  
                throw new NegativeRadiusException("Radius cannot be  
                negative");  
            }  
            this.radius = radius;  
        }  
        public static void main(String[] args){  
            circle circle = new circle();  
        }  
    }
```

```
try {  
    circle.setRadius(-5);  
    System.out.println("Area of circle: " + circle.calculateArea());  
}  
catch(NegativeRadiusException e){  
    System.out.println("Error: " + e.getMessage());  
}  
  
try {  
    circle.setRadius(10);  
    System.out.println("Area of circle: " + circle.calculateArea());  
}  
catch(NegativeRadiusException e){  
    System.out.println("Error: " + e.getMessage());  
}  
}
```

Gupinath

IT-23019

26. There are two primary ways to create a thread:

1. By implementing the runnable interface.

2. By extending the thread class.

Creating a thread using the runnable interface:

class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("Thread is running Runnable
Interface!");

}

public class RunnableExample {

public static void main(String[] args) {

MyRunnable myRunnable = new MyRunnable();

Thread thread = new Thread(myRunnable);

thread.start();

}

}

Creating a thread by extending the Thread class:

```
class MyThread extends Thread {
```

@Override

```
public void run() {
```

```
    System.out.println("Thread is running by
```

extending Thread class!");

```
}
```

```
public class ThreadExample {
```

```
    public static void main(String[] args) {
```

```
        MyThread myThread = new MyThread();
```

```
        myThread.start();
```

```
}
```

```
}
```

28/1

Java's garbage collection (GC) is responsible for automatically managing memory by reclaiming the memory that is no longer in use. Java uses automatic memory management, which helps developers focus on logic without having to manually allocate memory. The goal is to find and remove objects that are no longer referred by the program, freeing up memory to be reused.

phases of Garbage Collection :

1. Mark Phase : The GC identifies all objects that are reachable.
2. Sweep phase : The GC removes objects that are not reachable (no references).

3. Compact Phase: The GC may move objects in memory to eliminate gaps, allowing more efficient memory usage.

After each phase, the program continues with the standard algorithm of initial detection and no search for objects. This is because the objects have been moved to a new location, so they are no longer found by the search algorithm.

The process continues until all objects have been found.

When the search algorithm finds an object, it marks it as found and continues to search for the next object.

Once all objects have been found, the search algorithm ends.

Step 4: De-allocate objects: Once all objects have been found, the search algorithm ends.

Step 5: De-allocate objects:

Step 6: De-allocate objects:

Step 7: De-allocate objects:

Step 8: De-allocate objects:

29/1

Code:

```
import java.io.*;
import java.util.Scanner;

public class HighestvalueAndSum {
    public static void main (String [] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";
        int highestValue = Integer.MIN_VALUE;
        int sum = 0;
        try (Scanner scanner = new Scanner (new File
                (inputFile))) {
            while (scanner.hasNextInt ()) {
                int number = scanner.nextInt ();
                sum += number;
                if (number > highestValue) {
                    highestValue = number;
                }
            }
        }
    }
}
```

```
catch(FileNotFoundException e){  
    System.out.println("The input file was not found.");  
    return;  
}  
  
try(PrintWriter writer = new PrintWriter(new File(outputFile)))  
{  
    writer.println("Highest Value : " + highestValue);  
    writer.println("Sum : " + sum);  
    System.out.println("Results written to " + outputFile);  
}  
catch(FileNotFoundException e){  
    System.out.println("Error writing to the  
    output file.");  
}  
}
```

Gupinath
IT-23019

30.11

CODE:

```
import java.util.Scanner;  
import java.util.Arrays;  
  
Public class ArrayDivision {  
    public static void main(String []args) {  
        Scanner sc = new Scanner(system.in);  
        system.out.print("Enter the size of the first  
array (n>20): ");  
        int n = sc.nextIn();  
  
        if(n<=20){  
            system.out.println("n must greater than 20");  
            return ;  
        }  
  
        int m = n/10;  
        int [] array1 = new int [n];  
        int array2 = new int [m];  
  
        system.out.println("Enter elements of the first  
array (size "+n+"): ");  
        for(int i=0; i<n; i++) {
```

Gupinath
IT-23019

```
array1[i] = sc.nextInt();  
}
```

```
System.out.println("Result of dividing the first array  
by the second array:");
```

```
for(int i=0; i<m; i++) {  
    double divisionResult = (double) array1[i]/array2[i];  
    int divisor = (int) Math.ceil(divisionResult);  
    int remainder = array1[i] % array2[i];  
    int divisor = (int) Math.floor
```

```
    System.out.println("For index " + i + ": ");
```

```
    System.out.print("Divisor (round up): " + divisor);
```

```
    System.out.print("Remainder: " + remainder);
```

```
}
```

```
sc.close();
```

```
}
```

Gupinath
T123019

31

current Datetime:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class currentDateTime {
    public static void main (String [ ] args) {
        LocalDateTime currentDateTime = LocalDateTime.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern ("yyyy-mm-dd
HH:mm:ss");
    }
}
```

```
String formattedDateTime = currentDateTime.format (formatter);
```

```
System.out.println ("Current Date and Time: " + formattedDateTime);
```

```
}
```

Gupinath
JTR3019

32

Public class CounterClass {

private static int instanceCount = 0;

private static final int MAX_COUNT = 50;

public CounterClass() {

instanceCount++;

} (if (instanceCount > MAX_COUNT) {

instanceCount = 0;

}

public static int getInstanceCount() {

return instanceCount;

}

public static void main(String[] args) {

for (int i=0; i<55; i++) {

new CounterClass();

System.out.println("Instance Count: "+CounterClass.getInstanceCount());

}

{

{

33

Public class ExtremeFinder {

public static int findExtreme(String type, int[] numbers) {

if (numbers.length == 0) {

throw new IllegalArgumentException("At least one number

must be provided");

{

int extreme = numbers[0];

for (int num : numbers) {

if (type.equalsIgnoreCase("smallest")) {

if (num < extreme) {

extreme = num;

}

else if (type.equalsIgnoreCase("largest")) {

if (num > extreme) {

extreme = num;

}

else {

throw new IllegalArgumentException("Invalid typ. Use 'smallest'"

or "largest.");

}

Gupinath
I123019

L8P

```
public static void main(String[] args) {  
    int x = findExtreme("smallest", 5, 2, 9, 1);  
    int y = findExtreme("largest", 8, 3, 10, 9);  
    System.out.println("smallest: " + x);  
    System.out.println("Largest: " + y);  
}
```

341

// The .equals() method checks whether the content of s₁ and s₂ are the same. Since both contain "This is IET 2107 Java", it returns true.

true

// The == operator checks whether s₁ and s₂ refer to the same memory location. They are different objects, false

// they point to the same memory location in string pool

true

Gupinath

ITR23019

35, 36]

interface Alpha {

 void methodA();

 void methodB();

}

interface Beta {

 void methodC();

 void methodD();

}

abstract class AbstractBase implements Alpha {

 public void methodA() {

 System.out.println("Method A implemented in AbstractBase.");

 }

 public abstract void methodE();

}

class FinalClass extends AbstractBase implements Beta {

 public void methodB() {

 System.out.println("Method B implemented FinalClass.");

}

```
public void methodC() {
```

```
    System.out.println("Method C implemented in FinalClass.");
```

```
}
```

```
public void methodD() {
```

```
    System.out.println("Method D implemented in FinalClass.");
```

```
}
```

```
public void methodE() {
```

```
    System.out.println("Method E implemented in FinalClass.");
```

```
}
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        FinalClass obj = new FinalClass();
```

```
        obj.methodA();
```

```
        obj.methodB();
```

```
        obj.methodC();
```

```
        obj.methodD();
```

```
        obj.methodE();
```

```
}
```

```
}
```

37

Issue with the Code

The class Z implements both interfaces X and Y, but both interfaces contain a default method show().

This results in a "duplicate default method" conflict, leading to a compilation error because Java does not know which show() method to inherit.

Solution 1: Override the show() method in class Z

```
public class Z implements X,Y {  
    @Override  
    public void show() {  
        X.super.show(); //Explicitly calling show() from interface X  
    }  
    public static void main(String[] args) {  
        Z obj = new Z();  
        obj.show();  
    }  
}
```

Solution 2

```
public class Z implements X Y {  
    @Override  
    public void show() {  
        System.out.println("Z's own show method");  
    }  
  
    public static void main(String[] args) {  
        Z obj = new Z();  
        obj.show(); // calls overridden show() from Z  
    }  
}
```

381

Class singletonExample {

private static singletonExample instance;

private singletonExample() {

System.out.println("singletonExample instance created.");
}

public static synchronized singletonExample getInstance() {

if(instance == null) {

instance = new singletonExample();

}

return instance;

}

public void showMessage() {

System.out.println("Hello from singletonExample!");

{

}

public class SingletonDemo {

public static void main(String[] args) {

singletonExample obj1 = singletonExample.getInstance();

singletonExample obj2 = singletonExample.getInstance();

obj1.showMessage();

if(obj1 == obj2) { System.out.println("Both are same instance"); }

else { System.out.println("Different instance"); }

```
public class WalletTest {  
    public static void main (String [] args) {  
        wallet mywallet = new wallet(100);  
        try {  
            mywallet.addfunds(50);  
            mywallet.spend(30);  
            mywallet.addfunds(-20);  
        } catch (InvalidAmountException | InsufficientFundsException e)  
        {  
            System.out.println("Exception caught: "+e.getMessage());  
        }  
        try {  
            mywallet.spend(200);  
        } catch (InvalidAmountException | InsufficientFundsException e)  
        {  
            System.out.println("Exception caught: "+e.getMessage());  
        }  
    }  
}
```

Grupinath

IT23019

40t

```
public class ArithmeticOperation {
    public static void main(String[] args) {
        if(args.length != 2) {
            System.out.println("Error: please provide exactly two integers");
            return;
        }
        try {
            int num1 = Integer.parseInt(args[0]);
            int num2 = Integer.parseInt(args[1]);
            int sum = num1 + num2;
            int difference = num1 - num2;
            int product = num1 * num2;
            double quotient = (double)num1 / num2;
            System.out.println("sum " + sum);
            System.out.println("difference " + difference);
            System.out.println("product " + product);
            System.out.println("quotient " + quotient);
        } catch(NumberFormatException e) {
            System.out.println("Error: Both args must integers.");
        } catch(ArithmaticException e) {
            System.out.println("Error: Division by zero is not allowed.");
        }
    }
}
```