1

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

public class NumberProcessor {
    public static void main(String[] args) {
        try {
            Scanner sc = new Scanner(new File("input.txt"));
            scanner.useDelimiter(",|\\s+");
            List<Integer> numbers = new ArrayList<>();
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                numbers.add(scanner.nextInt());
            } else {
                scanner.next();
            }
        }
        sc.close();
int highest = collections.max(numbers);
long sum = (long) highest * (highest + 1) / 2;
PrintWriter writer = new PrintWriter(new File("output.txt"))
writer.println("Highest number: " + highest);
writer.println("sum of natural numbers up to " +
                highest + ": " + sum);
```

```
writer.close();
System.out.println("Results written to output.txt");
}
catch (FileNotFoundException e) {
    System.err.println("File not found: " + e.getMessage());
}
}
}
```

[2] Differences between static and final:

Aspect	static	final
purpose	Belongs to the class, not to any specific instance.	Makes a field or meth immutable.
Fields	shared across all instances of the class. Only one copy exist.	Value cannot be chang once assigned.
methods	can be called without creating an instance of the class.	cannot be overriden in subclasses.
memory	Allocated memory once for the class.	Memory is allocated per instance.
Usage	Used for utility methods on constants shared across instances	Used to define constant on prevent overriding.

[3]

```
import java.util.Scanner;

public class FactorionNumbers {

    private static final int[] FACTORIALS = {
        1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880
    };

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the lower bound of the range:
        int lowerBound = sc.nextInt();
        System.out.print("Enter the upper bound of the range: ");
        int upperBound = sc.nextInt();

        System.out.println("Factorions numbers in the rang: ");

        for (int i = lowerBound; i <= upperBound; i++) {
            if (isFactorion(i)) {
                System.out.println(i + " ");
            }
        }
        sc.close();
    }

    private static boolean isFactorion(int number){
        int sum = 0;
        int originalNumber = number;
```

```
while (number > 0) {
    int digit = number % 10;
    sum += FACTORIALS [digits];
    number /= 10;
}

return sum == originalNumber;
}
}
```

IT23014

[4] Differences among class, local and instance variables.

Aspect	class variable	instance variable	local variables
Declaration	static keyword	No static keyword	inside a method
scope	Entire class	instance of the class	Limited to the bl
Lifetime	class loading to unloading	object creation to destruction.	Block entry t exit.
Memory Allocation	method area	Heap memory	stack memory
Default	Yes (e.g., 0, null)	Yes (e.g., 0, null)	No (must be initialized

5

```
public class ArraySumCalculator {
    public static void main(String[] args) {
        int numbers = {10, 20, 30, 40, 50};
        int sum = calculateSum(numbers);
System.out.println("Sum of array elements: " + s
}
public static int calculateSum(int[] array) {
    int sum = 0;
        for (int num : array) {
            sum += num;
        }
    return sum;
}
}
```

9

Method overriding: It is a feature in java that allows a subclass to provide a new implementation for a method tha is already define in its superclass. How it works in inheritanc

i) When a subclass overrides a methods, the subclass version of the method gets executed, even if the method called on a superclass reference holding a superclass object.

ii) This process is called runtime polymorphism, because the method call is resolved at runtime.

iii) Overriding enables customized behaviour for subclass object while maintaining a consistent interface.

What happens when a subclass overrides a method -

1. subclass method executes, replacing the superclass metho
2. Runtime polymorphism determines method execution at runtime.
3. Superclass method is hidden unless called using `super`
4. Overriding rules apply.
5. `super` can call the overridden superclass method

The `super` keyword is used to call an overridden method from the superclass. This allows the subclass to extend or modify the behavior without completely replacing it.

1. Potential issue when overriding methods:

i) Visible restriction - Cannot reduce access (e.g: `public` → `private`)

ii) Final and static methods - `final` methods can't be overriden

Static methods are hidden, not overriden.

2. Issues with constructors:

i) Constructors cannot be overriden because they are not inherited.

ii) `super()` must be used for superclass initialization:

- If the superclass has a parameterized constructor, the subclass must explicitly call, `super(arguments);`:
- If no explicit `super()` is used, Java insents a default constructor call, which may cause an error if the superc lacks a no argument constructor.

10 Difference between static and non-static member in Java -

Feauture	static members	non-static members
Defination	Belong to the class and are shared among all objects	Being to the individual objec each instance has its own c
Access	Accessed using class name of instance	Access only through an object of the class.
Memory Allocation	stored in the method area	stored in the heap area
Usage	used for constants, utility methods and shared properties	Used for object specific behavior and instance da
Invocation	can be called without creating an object.	Requines object creati before calling.
Example	Static members are shared and constant for all. examples include company name, gravity, national anthem, domain extensions like ".com". They apply to all, not just individuals	Non-static members are unique to each instance. examples include employe IDs, can numbers, phon number etc. They vary for each object and are not shared.

Check if a number or string is palindrome :

program:

```
import java.util.Scanner;
public class Palindromcheck {
    Static boolean Pal(String s) {
        return s.equalIsIgnoreCase(new StringBuilder(s).reverse().
                                                    toString());
    }
Static boolean Pal (Int n) {
        int rev=0 , temp=n;
        while (temp >0) {
            rev = rev * 10 + temp%10;
            temp /= 10;
    } return n= rev;}
public static void main (String[] args){
    Scanner sc = new Scanner(System.in);
System.out.println("Enter a number: ");
int n= sc.nextInt();
System.out.println(n+ Pal(n) ? " is : is not") + "a palindrome");
System.out.print ("Enter a string : ");
String s= sc.next();
System.out.println (s+ (Pal(s) ? "s: is not ") + "a pallindrome");
sc.close();}
}
```

11)

Class abstraction is a concept in object-oriented programming (OOP) that focuses on hiding the complex implementation details and showing only the essential features of an object. It allows you to create a simplified model of a real-world entity by focusing on what the object does rather than how it does it.

Encapsulation is the mechanism of bundling the data (attributes) and methods (functions) that operate on the data into a single unit, typically a class.

Example:

```
class Car {
    private String make;
    private String model;
    private int year;
    private int mileage;

    public Car (String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.mileage = 0;
    }
```

```
public void drive (int miles) {
        this.mileage += miles;
        System.out.println("Driving " + miles + " miles. Total
                                mileage: " + this.mileage);
    }

    public int getMileage() {
            return this.mileage;
        }
    }

public class Main {
        public static void main(String[] args) {
            Car myCar = new Car("Toyota", "Corolla", 2024);
            myCar.drive(100);
            System.out.println("Toyota mileage: " +
                                    myCar.getMileage());
        }
}
```

Explanation:

- Abstraction: The drive method abstracts the internal logic of updating mileage. This user doesn't need to know how it works; they just call the method.
- Encapsulation: The Car class encapsulates the attributes (make, model, year, mileage) and provides public methods (drive, getMileage) to interact with them.

Key differences between Abstract class and Interface in Java.

Feature	Abstract class	Interface
methods	can have both abstract and concrete methods.	can only have abstrac methods.
variables	can have instance variables	can only have public static final constants
multiple inheritance	A class can extend only one abstract class.	A class can implemen multiple interfaces.
constructor	can have a constructor	cannot have a constru
Purpose	Used for partial abstraction (shared implementation).	Used for full abstraction (contract)

12

```
class BaseClass {
    public void printResult(String operation, String result) {
        System.out.println(operation + ": " + result);
    }
}

class SumClass extends BaseClass {
    public void computeSum() {
        double sum = 0.0;
        for(double i = 1 ; i >= 0.1 ; i -= 0.1) {
            sum += i;
        }
        printResult("Sum of the series", String.valueOf(sum));
    }
}

class DivisorMultipleClass extends BaseClass {
    private int gcd(int a, int b) {
        if (b == 0) {
            return a;
        }
        return gcd(b, a%b);
    }
}
```

```
    private int lcm(int a, int b) {
        return (a*b) / gcd(a,b);
    }

public void computeGCDAndLCM(int num1, int num2) {
    int gcd = gcd(num1, num2);
    int lcm = lcm(num1, num2);
printResult("GCD of " + num1 + "and" + num2, String.valueOf(gcd));
printResult("LCM of" + num1 + "and" + num2, String.valueOf(lcm));
}
}
class NumberConversionClass extends BaseClass {
    public void convertNumber(int number) {
        String binary = Integer.toBinaryString(number);
        String hexadecimal = Integer.toHexString(number);
        String octal = Integer.toOctalString(number);

        printResult("Binary of " + number, binary);
        printResult("Hexadecimal of " + number, hexadecimal);
        printResult("Octal of " + number, octal);
    }
}
```

```
public class Mainclass {

    public static void main (String[] args) {

        Sumclass sumObj = new sumclass ();

        DivisonMultipleClass divisonMultipleObj = new DivisonMultipled
NumberConversionclass NumberconversionObj = new NumberConversionclas

        sumobj.computeSum ();

        divisonMultipleObj. computeGCDAndLCM (12,18);

        numberConversionObj. convertNumber (25);

    }

}
```

13 To complete the program based on the provided UML diagram, we need to implement the GeometricObject, Circle and Rectangle classes, along with a main class to demonstrate their functionality. Below is the Java code that follows the UML scenario:

```
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;
}

class Circle extends GeometricObject {
    private double radius;

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

    public Circle(double radius) {
        super();
        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 class main {
    public static void main (String[] args) {
        Cincle cincle = new Cincle (5.0, "blue", true);
        Cincle .printCincle ( );
System.out. println ("Areai " + cincle.getArea()};
System.out. println ("Perimeter: " + cincle.getPerimeter());
System.out.println ("Diameter: " + cincle. getDiameter());

    Rectangle rectangle = new Rectangle (4.0, 6.0, "red", false);
System.out.println(rectangle.toString());
System.out.println ("Area:" + rectangle.getArea());
System.out.println ("Perimeter: " + rectangle.getPerimeter()
}
}
```

14 The BigInteger class in Java is part of the java.math package and is used to handle arbitrarily large integers. Unlike primitive data types like int or long, which have fixed size limits (int can store up to $2^{31}-1$ and long up to $2^{63}-1$) BigInteger can store integers of virtually any size, limited only by the available memory. This makes it ideal for calculations involving very large numbers, such as factorials of large integers, cryptographic algorithms, or precise mathematical computations.

Below is a Java program that calculates the factorial of any integer using BigInteger:

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

public class FactorialCalculator {
    public static BigInteger factorial(int n) {
        BigInteger result = BigInteger.ONE;
        for (int i=2; i<=n; i++) {
        result = result.multiply(BigInteger.valueOf(i));
    }
    return result;
```

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

    Scanner scanner = new Scanner(System.in);

System.out.print("Enter an integer to compute its facto

    int number = scanner.nextInt();

        if (number < 0) {

            System.out.println("Factorial is not defined fo
                                    negative numbers");

        } else

        else {

            BigInteger factorialResult = factorial(number

        System.out.println("Factorial of " + number + "is

        System.out.println(factorialResult);

    }

    scanner.close();

}

}
```

15 Abstract classes:

- Purpose: Provide a partial implementation and allow sub-classes to complete it.
- Fields: can have instance variables.
- Methods: can have both abstract (no body) and concrete (with body) methods.
- Inheritance: single inheritage.
- Use case: Preferred when you need to share code or maintain state among subclasses.

Interfaces:

- Purpose: Define a contract that implementing classes must follow.
- Fields: Only constants (public static final)
- Methods: Abstract methods, default methods, static methods.
- Inheritance: Multiple inheritance.
- Use case: Preferred when you need to define a behaviour that can be implemented by unrelated classes

When to use Abstract classes over interfaces:

- ☐ Shared code
- ☐ State management
- ☐ Constructor logic.

And, yes, a class can implement multiple interfaces.

Example:

```
interface A {
    default void show() { System.out.println("A"); }
}

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

class C implements A, B {
    @Override
    public void show() {
        A.super.show();
    }
}
```

16 Polymorphism in Java

Polymorphism is the ability of an object to take on many forms. In Java, it allows a single method or class to operate on objects of different types. It is achieved through method overriding and method overloading.

☐ Dynamic method dispatch

It is the mechanism by which a call to an overridden method is resolved at runtime rather than compile time. It is the foundation of runtime polymorphism in Java.

Example of polymorphism using interface and method overriding

```
class Animal {
    void sound() {
        System.out.println("Animal make a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
```

```
class Cat extends Animal {
    @Override
        void sound(){
            System.out.println("Cat meows");
        }
}
public class Main {
    public static void main(String[] args){
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myAnimal.sound();
        ~~myAnimal.s~~
        myDog.sound();
        myCat.sound();
    }
}
```

Trade-offs:

- Flexibility
- Readability
- Performance.