IT-23013

Answer to the Q NO-1

```
import. java.io *;
import java.util.*;
public class Numberprocessor{
   public static void main(string[ ] args){
      string inputfile = "input.txt";
      string output = "output.txt";
    try{
Scanner scanner = new Scanner(new File(inputfile));
scanner.use delimeter(",");
List <Integer> number = new Arraylist<>();
while (scanner.hasnext()){
   if(scanner.hasnext()){
      numbers.add(scanner.nextint());
   } else{
         scanner.next();
   }} scanner.close();
```

```
if (numbers.isEmpty()){
    System.out.println("No valid Numbers
    found in input file"); }
    return; }
    int highestNumber = collections.Max(Numbers);
// Prepare result
List<String> results = new ArrayList<>();
for (int num: numbers){
    int sum=(num * (num+1)) /2;
    results.add(num+" , " +sum); }
results.add("Highest Number: "+ highestNumber);
PrintWriter writer= new PrintWriter(New File(
outputfile));
for (string results : results) {
    writer.println(result); }
```

```
writer.close();
        System.out.println("Processing complete. check"
        +outputfile+" for results.");}
        catch(FileNotFoundException e){
            System.out.println("Error: Input file not found");
        }
    }
```

IT23013

Difference Between static and final Fields and methods in java:

Feature	static	Final
Defination	Belongs to the class, not objects.	Makes a variable constant or a method non-overridable.
Usage with variable	Shared across all objects (only one copy exists).	Prevents modification (for variable)
usage with methods	Can be called without creating object	can be called creating object
usage with clases	Not applicable	Prevents inheritance if used with a class
Access Modifier Restrictions	can be public, private, protected or default	Can be public, Private, protected or default.
Memory Allocation	Stored in Method Area (common for all objects)	Same as regular variables but values are immutable.

IT-23013

If I use an object instead of the class name to access static members:

- It works but shows a warning:

Static member should be accessed in a static way.

why → Because static member belong to the class not the object.

IT23013

A Java program to find all factorion numbers within a given range:

Code:

```
import java.util.Scanner;
public class FactorionFinder{
private static final int[] FACTORIALS = new int[10];
static{
FACTORIALS[0]=1;
for(int i=1; i<10; i++){
FACTORIALS[i]= i * FACTORIALS[i-1];
}
}
private static boolean isFactorion (int num){
int sum=0, temp= num;
while(temp>0){
int digit = temp % 10;
sum+ = FACTORIALS[digit];
temp/=10; }
return sum=num; }
```

IT23013

```
public static void main(String[] args){
Scanner scanner = new Scanner(System.in);

System.out.print("Enter the lower bound of
    the range: ");
int lower = scanner.nextInt();
System.out.print("Enter the upper bound
    of the range: ");
int upper = scanner.nextInt();
scanner.close();
System.out.println("Factorion numbers in the
    range:");
boolean found = false;
for(int i = lower; i<=upper; i++){
if(isFactorion(i)){
    System.out.print(i+" ");
    found= true; } }
```

```
if (!found){
    System.out.println("None found in the
given range.");}}}
```

Answer to the Q No-4

Types of Variable	Defination	scope	Lifetime	Accessed By
class variable	A variable that is shared among all instances of a class	class level	Existance as long as the ~~object~~ class is loaded in memory	Accessed using the class or an instance
Instance variable	A variable that is unique to each intance of a class	object level	Exists as long as the object exists	Accessed using self inside class method.
Local variable	A variable declared inside a method or function	Method or function scope	Exists only while the method/function is running	can only be accessed within the function/ method where it is defined.

IT23013

Significance of this keyword in Java:

1. Referencing to instance Variables:
   - use when local and istance variable have the same name to differentiate them.
2. calling other constructor:
   - this() can be used to call another constructor in the same class.
3. Returning the current Object:
   - used to return the current object from a method.
4. Passing the current Object as a Argument.

IT23013

Calculate the sum of all elements in an integer array and demostrates its use in the main method.

```
public class ArrayExample{
    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 = {10, 20, 30, 40, 50};
        int sum = calculatesum(numbers);
        System.out.println("Sum: "+sum);
    } }
```

Access Modifier:

Access modifiers in Java define the visibility and accessibility of classes, methods and variables. They control which parts of a program can access specific members of a class.

Comparison of Access Modifies in Java

Modifier	class	same Package	Different Package (sub class)	Different Package (Non-subclass)
public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
default	✓	✓	✗	✗
Private	✓	✗	✗	✗

- Public → Accessible everywhere.
- Protected → Accessible within the same package and subclass in other package.
- Default → Accessible only within the same package.
- Private → Accessible only within the same class.

⑦ Find the smallest Positive root:

```
import java.util.Scanner;
public class QuadraticEquationSolver {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter coefficients a, b, 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(discriminant))/(2.0*a);
    double root2 = (-b+Math.sqrt.sqrt(discriminant))/(2.0*a)
```

IT23013

(8) Java program that determines whether a character is a letter, a whitespace or a digit.

```
public class characterChecke{
    public static void main(String[] args){
        char[] characters={'A',' ','5','z','\n','9',
            'G','\n'};
        checkcharacters(characters);
    }
    public static void checkCharacters(char[] arr){
        for(char c: arr){
            if(Character.isLetter(c)){
                System.out.println("'"+c+"' is a letter.");}
            else if(Character.isWhitespace(c)){
                System.out.println("'"+c+"' is a whitespace
                    character.");}
            else if(Character.isDigit(c)){
                System.out.println("'"+c+"' is a Digit.");}
```

System.out.println(" :" +c + " is a special character.");

}

}

}

How to pass an Array to a Function in Java?

To pass an array to a function in Java:

- Define the function with a parameter of type array (char[] arr in this case).
- Call the function and pass the array as an argument.

IT23013

⑨ How Method Overriding works in inheritance:

- A subclass provides a new implementation for a method that exists in its superclass.
- The overridden method must have the same method signature (name, return type and parameters)
- The subclass method cannot have a weaker access modifier.
- Overriding occurs with instance methods, not static methods.

What Happens When a subclass Overrides a Method?

- When an overridden method is called on a subclass object, the subclass version executes, not the superclass version.
- This enables dynamic method dispatch.
- The JVM determines the method to call based on the actual object type, not the reference type.

Using the super keyword:

The super keyword helps:

1. calling the overridden method from the superclass
2. calling the superclass constructor.

Potential Issues when Overriding methods:

1. Access Modifier restrictions
2. Return type issues.
3. Exception Handing Restrictions.
4. Static methods cannot Be Overridden.
5. Final methods cannot be overridden.

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

Feature	Static Members	Non-static Members
Destination	Belong to the class rather than any object	Belong to individual objects of the class
Access	can be accessed using class name	Required an instance of the class to be accessed
Memory Allocation	Memory is allocated only one	Memory is allocated each time an object is created
Modification	Changes made to static members affect all objects	Changes made to non-static members are specific to the individual object
calling context	Can be called without creating an object.	Needs an object to be called.
Example	static int count;	int age.

IT23013

Palindrome cheaker in Java:

```
import java.util.Scanner;
public class palindrome.cheaker{
  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;
  }
static boolean isPalindrome(int num){
int original=num, reverse=0, digit;
while (num>0){
  digit=num%10;
```

```
        reverse = reverse * 10 + digit;
        num /= 10;
    }
    return original == reverse;
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter a string: ");
    String str = scanner.nextLine();
    if (isPalindrome(str)) {
        System.out.println("\"" + str + "\" is a palindrome");
    } else {
        System.out.println("\"" + str + "\" is not a palindrome");
    }
    System.out.print("Enter a number: ");
    int num = scanner.nextInt();
```

IT23013

```
if (isPalindrome(num)){
    System.out.println(num+" is a palindrom
    }
else{
    system.out.println(num+" is not a palindrom
    }
    scanner.close();
}
}
```

IT23013

11|

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();
```

```
        myCar.start();
        myCar.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());
}
}
```

Differences Between Abstract Class and Interface

Abstract Class	Interface
1. A class that contains abstract and con-create methods	1. A collection of abstract methods that must be implemented by a class.
2. Can have both abstract and con-create methods	2. Only abstract methods.
3. Can have instance-variable	3. Can only have public static final variables.
4. Can have construction	4. Cannot have construction
5. A class can extend only one abstract class.	5. A class can implement interfaces.
6. Used when classes share common behaviour with some default implementation.	6. Used when unrelated classes need to follow the same contract.

12|

Solve

```
Class Baseclass{
  Void PrintResult(String result) {
  System.out.println(result);
  }
}
Class Sumclass extends Baseclass {
  Void computeSum() {
    double sum=0;
    for( double i=1.0 i>, 0.1 ; i-= 0.1) {
        Sum+=i ;
    }
     PrintResult("Sum of the Series: " +sum);
    }
  }
  Class DivisionMultipleClass extends Baseclass {
   int gcd(int a, int b) {
      while ( b!=0 ) {
       int temp =b ;
       b= a%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) {
~~Print result~~
PrintResult("Decimal: "+num);
PrintResult("Binary :" + Integer.toBinaryString(num));
PrintResult("Octal: " + Integer.toOctalString(num));
PrintResult("Hexadecimal: " + Integer.toHexString(num));
}
}
```

IT23013

```
Class Customprint-Class extends Baseclass {
 Void Pr (String message) {
    PrintResult ("[Custom Print]" +message);
 }
}

public class Main class{
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();
 numConObj.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 Data date created;
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;
}
}
```

IT23013

```
class circle extends Geometric_object {
    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;
  }
```

IT23012

```
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 setHeight(double height) {
        this.height = height;
    }
    public double getArea() {
        return width * height;
} public double getperimeter() {
    return 2 * (width + height);
}
```

IT 23013

```
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 ("\n 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());
}
}
```

IT23013

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 have limits (int up to $2^{31}-1$ and long up to $2^{63}-1$). BigInteger can store much larger values.

2. Supports Arithmatic Operations: It provides method for addition, subtraction, multiplication, division and modular operations.

3. Useful in Cyptography: 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 one.

IT23013

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();
}
}
```

IT23013

15] Comparison of ~~Compar~~ Abstraction Classes and Interfaces in Java.

Abstract Class	Interface
1. A class that cannot be instantiated and may have both abstract and concreate methods.	1. A collection of abstract methods and default/static methods with no instence 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 mutiple interfaces.
5. Can have any access modifier for methods and variables.	5. All methods are implicitly 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 specefic methods, use an abstract clas.

3. Performance conjderations: 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 beaking existing implementations.

☐ Can a class implement Multiple Interfaces in Java?

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

IT23013

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

Implication using multiple Interface

- Code Resuability & 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");
    }
}
```

IT23013

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");
  }
}
```

IT23013

```
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.makesound ( );
     myAnimal = new cat();
     myAnimal.makeSound ( );
    }
  }
```

IT23013

Trade-offs Between using polymorphism and specific method calls

Using polymorphism	Using specific method calls
1. High code is more resuable and adaptable	1. Low code is tightly coupled.
2. Slightly slower due to dynamic method lookup.	2. Faster due to direct method calls
3. Easier to maintain and extend.	3. Harder to maintain.
4. Slightly higher due to vtable usage.	4. Lower since method calls are direct.