Q-1 IT-23032

```
import java.io.File
import java.io.FileNotFoundException
import java.io.PrintWriter
import java.util.Scanner
public class Main() {
    public static void main(String[] args){
try{ File inputFile = new File("input.txt");
    Scanner sc = new Scanner(inputfile);
    String[] numbers = sc.nextLine().split(",");
    sc.close();
    int maxi = -1;
for (String number : numbers){
    int n = Integer.parseInt(number.trim());
    if (n > maxi){
        maxi = n
    }
}
Int num = (maxi * (maxi+1))/2;
File outputFile = new File("Output.txt");
PrintWriter writer = new PrintWriter(outputFile);
writer.println("Highes number : "+maxi);
writer.println("num = "+num);
}
Catch (FileNotFoundException e) {
    system.out.println("File not Found"); }
```

IT-23032

Q-2|

⇒ Difference betn Static and Final method:

Static: Can be accessed via the class name without creating an instance - shared across all instances. it can be modified also.

Final: Can't be modified once initialized. Can't be overridden by subclass.

- It can be accessed static fields/methods using an object, but it's not the proper way.

code:

```
class myclass {
    static int x=7;
    static void display(){
        System.out.println("I am static");
    }}
public class main(){
    public static void main(String[] args){
    myclass obj = new myclass();
     obj.display();
     System.out.println(obj.x); //Not proper way
// proper way given below
System.out.println(myclass.x);
     myclass.display();
    }
}
```

IT-23032

Q-3

```
import.java.util.*;
public class Factorians{
  public static int fact(int n){
     int fact=1;
    for (int i=1; i<=n;i++) fact*=i;
    return fact;
  }
public static void digit(int l, int h){
  for (int i=l; i<=h; i++){
      int sum=0, temp=i;
    while (temp>0) {
       sum+=factorial(temp%10);
      temp/=10;
    }
   if(sum==i) system.out.println(num+" ");
}}
public static void main(String[] args){
   Scanner sc=new Scanner(system.in);
   int l=sc.nextInt();
   int h= sc.nextInt();
   digit(l,m);
   sc.close();
  }
}
```

IT-23032

Q-4| Differences among class, local and instance variables are given below.

1| Class variables: a) Declared with static
b) Shared by all objects of the class.
c) Accessed via class name or object.

2| Instance variable: a) Declared without static
b) Each object has it's own copy.
c) Accessed via an object reference

3| Local variables: a) Declared inside method/blocks
b) Limited to the block
c) Exists only during execution.

Significance of 'this' keyword:

It refers to the current instance of the class.
It is used to distinguish instance variables from local variables when they have the same name
It passes the current instance as an argument to a method or constructor.

IT-23032

Q-5]

```
public class SumArray {
    public static int sum(int[] array){
        int s=0;
        for(int num: array){
            s+= num;
        }
        return s;
    }
    public static void main(String[] args){
        int[] array = {1,2,3,4,5,6,7,8};
        int ans = sum(array);
        System.out.println(ans);
    }
}
```

Q-6]

Access modifiers determine the visibility of classes, methods, and variables in Java. The main access modifiers are public, private and protected.

Accessibility comparison:

public: Accessible from any other class.

private: Accessible only within the class it is declared.

protected: Accessible within the same package and subclass.

Types of variables in Java

IT-23032

① class variables declared with 'static', shared by all objects.

Code:
```
public class Exp{
    static int x=5;
}
```

② instance variable — declared inside a class but outside methods.

code:
```
public class Exp{
    int x=32;
}
```

③ Local variables: declared inside a method or block.

code:
```
public class Exp{
    public void ok(){
        int sk=32;
    }
}
```

Q-7
```
import java.util.*;
public class QuadRoot {
    public class void main(String[] args){
        Scanner sc = new Scanner(system.in);
System.out.println("Enter the coefficients of a,b,c");
int a = sc.nextInt();
int b = sc.nextInt();
int c= sc.nextInt();
double d = b*b-4*a*c;
if (d<0) { system.out.println("No real roots");
}
```

IT-23032

```
else {
    double r1 = (-b + Math.sqrt(d))/(2*a);
    double r2 = (-b - Math.sqrt(d))/(2*a);

if (r1 > 0 && r2 > 0) {
    System.out.println("The smallest positive root is: " + Math.min(r1, r2));
    }
else if(r1 > 0) {
        System.out.println("The smallest positive root is: " + r1);
        }
else if (r2 > 0) {
    System.out.println("The smallest positive root is " + r2);
    }
else System.out.println("No positive roots");
    }
    sc.close();
}
```

Q-8]

```
import java.util.Scanner;

public class Check {
    public class void main(String[] args) {
     Scanner sc = new Scanner(System.in);
    System.out.println("Enter a character:");
    Char ch = sc.next().charAt(0);
if (Character.isLetter(ch)) {
    System.out.println("it is a letter");
```

IT-23032

```
else if(Character.isDigit(ch)){
  system.out.println("It is a digit"); }
else if(Character.isWhitespace(ch)){
  system.out.println("It is a whitespace"); }
else { system.out.println("It is a special character"); }
  sc.close();
}
}
```

Passing an Array to a Function

```
public class Exp {
  public static void display(int[] array){
    for(int num: array){
      system.out.println(num+" ");
    }}
  public static void main(String[] args){
    int[] array = {1,2,3,4,5,6,7,8};
    display(array);
  }
}
```

Q-9| IT-23032

Method overriding in Java occurs when a subclass provides a new implementation of a method already defined in it's superclass using the same method signature. This enables runtime polymorphism, allowing the overridden method to be called based on the actual object type. Overriding rules include maintaining the same method signature, not reducing visibility and not overriding final methods. The 'super' keyword allows access to the superclass's method if needed. This concept enhances code reusability and enables subclass-specific behaviour while preserving the inheritance structure.

When a subclass overrides a method from it's superclass, the overridden method in the subclass is executed instead of the superclass version when called through a subclass instance. The 'super' keyword in Java is used to call the superclass method from a subclass when it has been overridden, allowing access to the parent class implementation. It is also used to call the superclass constructor to initialize inherited properties. When overriding methods, potential issues include losing access to the original behaviour unless explicitly called with 'super', reducing method visibility, which is not allowed, and attempting to override 'final' methods, which results.

in a compilation error.

In the case of constructor, Java automatically calls the default superclass constructor, but if the superclass lacks a no-argument constructor, the subclass must explicitly call a parameterized constructor using super.

Q-10]

Static Members	Non-static Members.
① Declared with 'static' keyword.	① Not declared with 'static'
② Shared by all objects of the class	② Each object has its own copy.
③ Can be accessed using the class name or object.	③ Can only be accessed using an object of the class.
④ initialized once and shared across instances.	④ Initialized for each instance.
⑤ Can have only static methods.	⑤ Can have instance methods.

Exp:
```
class Exp {
  static int x=32;
static void display() {
System.out.println("Sageeb");
  }
}
```

Exp:
```
class Exp2 {
  int y=31;
void display() {
System.out.println("Non-St");
  }
}
```

palindrom code: IT-23032

```
import java.io.*;
public class Palindrome {
    public static boolean isPalindrome(String s) {
        s = s.toLowerCase();
        String rev = "";
    for (int i = s.length()-1; i>=0; i--) {
        rev += s.charAt(i);
    }
    return s.equals(rev);
}
public void static main (String[] args) {
    String Scanner sc = new Scanner(System.in);
    String sk = sc.nextLine();
    boolean res = isPalindrome(sk);
    if (res) { system.out.println("it's a palindrome"); }
else { system.out.println("it's not a palindrome"); }
}
}
```

Q-11]

IT-23032

Abstraction: Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. In Java, abstraction is acheived using abstract class and interfaces.

Code:

```
abstract class Animal {
    abstract void sound();
}
class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}
public class main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        mydog.sound();
    }
}
```

Encapsulation: Encapsulation is the concept of bundling the data and the methods that operate on the data into a single unit (class).

Code:

```
class person {
    private String name;
    public string getName() {
        return name;
    }
```

IT-23032

```
public void setName(String name){
    this.name = name;
}
}

public class main{
    public static void main(String[] args){
    Person p1 = new Person();
        p1.setName("Sajeeb");
    System.out.println(p1.getName());
    }
}
```

Q-12]

```
import java.util.Scanner;
public class Mainclass {
    public static void display(String result) {
        System.out.println(result); }
    public static double CompuleSum() {
        double num=0;
        for(double i=1.0; i>=0; i-=0.1){
            num+=i;
        return num;
    }
    public static int gcd(int a, int b) {
            while(b!=0) {int temp=b; b=a%b; a=temp; }
        return a;
    }
```

IT-23032

```
public static int lcm(int a, int b) {
    return (a*b) / gcd(a,b); }
public static String toBinary(int number) {
    return Integer.toBinaryString(number); }
public static String toHex(int number) {
    return Integer.toHexString(number); }
public static String toOctal(int number) {
    return Integer.toOctalString(number); }
public static void pr(String msg) {
    System.out.printf("Formatted print: %s\n", msg); }
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    display("sum of the series " + computeSum());
System.out.println("Enter two numbers for GCD,LCM");
int a = sc.nextInt(); int b = sc.nextInt();
display("GCD " + gcd(a,b) + ", LCM " + lcm(a,b));
System.out.println("Enter a number for conversion");
    int x = sc.nextInt();
display("Binary: " + toBinary(x) + ", Hex : " + toHex(x) + "Octal: " + to
                                                    Octal(x));
    sc.nextLine();
    System.out.print("Enter a message: ");
        pr(sc.nextLine()); sc.close(); } }
```

Q-13 IT-23032

```
Import java.util.Date
Class GeometricObject {
    private String color; private boolean filled; private java.util.Date dateCreated
    public GeometricObject() {
        this.color = "white"; this.filled = false; this.dateCreated = new java.util.Date
    }
    public GeometricObject(String color, boolean filled) {
        this.color = color; this.filled = filled; this.dateCreated = new java.util.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; }
    public String toString() {
        return "color: " + color + ", filled " + filled + "; Created on " + dateCreated; }
}
Class Circle extends GeometricObject {
    private double radius;
    public Circle() { this.radius = 1.0; }
    public Circle(double radius) { this.radius = radius; }
    public Circle(double radius, String color, boolean filled) {
        super(color, filled); this.radius = radius; }
```

IT-23032

```
public doble 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 and diameter "+radius, getDiameter());
    }
}
Class Rectangle extends GeometricObject {
    private double width, 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, boleam filled) {
        super (color, filled);
        this.width = width; this.height = height; }
    public double getwidth () { return getwidth; }
    public void setwidth (double width) { this.width = width; }
    public double getheight () { return height; }
    public void setHeight (double height) { this.height = height; }
    public double getArea () { return height * width; }
    public double get perimeter () { return 2 * (width + height); }
    }
}
```

IT-23032

```
public class main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

System.out.println("Enter the radius, color, filled ?(true/false) of circle ");

 double radius = sc.nextDouble();
  String circleColor = sc.next();
   boolean Fill = sc.nextBoolean();
 Circle cc = new Circle(radius, circlecolor, fill);
  System.out.println ("circle Area "+ cc.getArea() +", perimeter: "+ cc.getPerimeter());

    cc.printCircle();

System.out.println("Enter the height and width and filled(true/false) of Recta");
   double width = sc.nextDouble(); double height = sc.nextDouble();
    boolean RecFill = sc.nextBoolean();
    String RColor = sc.next();
 Rectangle rr = new Rectangle(width, height, Recolor, RecFill);

System.out.println("Area "+ rr.getArea() + "perimeter :"+ rr.getPerimeter());

 sc.close();
    }
}
```

Q-14 IT-23032

The significance of 'BigInteger' in Java lies in its ability to handle very large integers, beyond the limits of primitive data types such as int and long. The BigInteger class, part of the java.math package, allows for mathematical operations on arbitrarily large integer with high precision, making it ideal for tasks such as calculating large factorials.

Code :

```
import java.math.BigInteger;
public class Factorial {
    Pub static BigInteger factorial (int n) {
        BigInteger result = BigInteger.ONE;
    for (int i=1; i<=n; i++) {
        result = result.multiply (BigInteger.valueOf(i));
        }
    return result;
    }
public static void main (String[] args) {
    Scanner se = new Scanner (System.in);
    int x = se.nextInt ();
    BigInteger ans = factorial(x);
System.out.println (ans);
} }
```

Q-15 IT-23032

Abstract class	Interfaces
① Provide a partial implementation and allows subclasses to complete.	① Define a contract that implementing classes must follow.
② Can have instance variables	② Only constant public, static, final var
③ Can have both abstract and concrete methods	③ can have abstract, default, static methods
④ single inheritence	④ multiple inheritance allows
⑤ prefered when need to share code or maintain state among subclass	⑤ Prefered when need to define a behaviour that can be implemented by unrelated classes.

When to use:

① When need to provide some shared code or state among related classes.

② when we have a common base implementation that multiple subclass should inherit.

Yes; A class in Java can implement multiple interfaces. This allows for a more flexible design, enabling a class to inherit behaviour and contracts from multiple sources

Implications of using multiple interface:

if two interfaces have methods with the same signature but different return types, it causes a compile-time error.

If two interfaces have methods with the same signature,

the implementing class must override the method to resolve ~~ab~~ ambiguity.

Q-16

Polymorphism:

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 acheived 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.

Exp:

```
class Animal {
    void sound() { system.out.println("Animal makes round"); }
}
class Dog extends Animal {
@override
 void round() { system.out.println("Dog Barks"); }
}
class cat extends Animal {
    @override
 void round() { system.out.println("Cat meows"); } }
```

IT-23032

```
public class Main{
    public static void main(String[] args){
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();
        myAnimal.sound(); myDog.sound(); myCat.sound();
    }
}
```

polymorphism can impact performence due to the overhead of dynamic dispatch, where the method to call is determined at runtime. This introduces a slight performence cost compared to direct method calls, which are faster.

Trade-offs:

1) polymorphism offers flexibility, scability and easier maintenance but can be slower due to method lookup.
2) Specific method calls are more efficient in performance but reduce flexibility and make the code harder to maintain and extend.

Q-17 IT-23032

ArrayList and LinkedList are two commonly used implementations of the List interface in Java but they have different underlying data structures, which result in distinct performance characteristics.

ArrayList	LinkedList
1) Based on dynamic array.	1) Based on doubly linked list.
2) provides constant time random access for elements	2) provides linear-time access.

Time complexities:

operation	ArrayList	LinkedList
Access(get)	O(1)	O(n)
insertion(add)	O(n)*	O(1)**
Deletion(remove)	O(n)*	O(1)**
search(contains)	O(n)	O(n)

Prefer Arraylist: when

1) Frequent access to elements is needed O(1)
2) Insertion/deletion happen mostly at the end
3) Memory efficiency is important.

IT-23032

Prefer Linked list when

→ Frequent insertion/deletions at the beginning or middle
→ No need for fast random access

Impact on Large Datasets

ArrayList performs better for large datasets with frequent access but struggles with frequent insertions due to shifting. Linked List handles frequent modifications efficiently, but consume more memory and has slower access time.

Q-18

```
import java.util.*;
class CustomRandomGenerator {
    public int myRand(int i, int n) {
        int[] random = new int[n+1];
        long currentTime = System.currentTimeMillis();
        int t=1;
        for(int j=1; j<=n; j++) { random[j]=t; t+=2; }
        return Math.abs((int)(currentTime * random[i] % 1000)); }
    }
public class RandNumber {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

IT-23032

```
int n=obj.nextInt();
CustomRandomGenerator obj2=new CustomRandomGenerator();
for(int i=1; i<=n ;i++){
    System.out.println(obj.myRand(i,n));
}
obj.close();
}
}
```

Q-19

In Java, Multithreading is handled using Thread class and the Runnable interface. A thread is a lightweight process, and Java allows multiple threads to run concurrently within a program.

Thread class code:

```
class myThread extends Thread {
    public void run() {
        // code to execute
    }
}
```

Runnable interface code:

```
class MyRunnable implements Runnable {
    public void run() { // code here
    }
}
Thread t=new Thread(new MyRunnable());
t.start();
```

IT-23032

Threads are executed by calling the start(), which invokes the run method in a seperate clock.

Difference between Thread class and Runable interface

Thread class: Extends Thread and overrides the run() method. it restricts the class from extending other classes.

Runnable interface: Implements Runnable and passes it to a thread object, allowing flexibility for class inheritance.

Potential issues:

Race condition, deadlocks, starvation and thread interface become the issue.

'Synchronized' Ensures only one thread executes a critical section at a time. It can be applied to methods or blocks to prevent data inconsistency

Example of deadlock:

Two threads, thread1, thread2 hold locks on different resources and wait for each other, leading to a deadlock.

Code: Synchronized (lock1){ synchronized(lock2){ } }

Avoiding deadlock: ① Lock ordering: Always acquire locks in the same order

② Timeouts: use trylock() with timeouts to avoid indefinite blocking.

③ deadlock detection: Periodically check for deadlocks and handle them.

Q-20 IT-23032

In Java, exception handling is done using try, catch, throw, throws and finally blocks:

1) try: Code that might throw an exception
2) catch: Handles the exception if thrown
3) throw: used to explicitly throw an exception
4) throws: Declares exceptions that a method might throw.
5) finally: Executes code for cleanup, regardless of an exception.

```
try { //code here }
catch (Exception e) { //handle exception }
finally { //cleanup }
```

Difference betn checked and unchecked exceptions

→ Checked Exception: Must be either caught or declared using throws. Example: IOException

→ Unchecked Exception: Runtime exceptions that don't require declaration or handling. Exp: NullPointerException

Custom exceptions: Create a custom exception by extending Exception or RuntimeException. use throw to throw the custom exception.

code :

IT-23022

```
class CutomException extends Exception {
    public CustomExeception (String message) {
        super (message);
    }
}
```

throw : used to explicitly throw an exception.

throws : Declares the exception in the method signature to propagate further

To prevent resource leak :

1) Use finally block : Ensure resources are closed in finally, which runs regardless of exceptions.

```
try {
    // use resources
}
finally {//close resources }
```

2) use try-with-resources : Automatically closes resources that implement AutoCloseable after the try blocks

```
try (FileInputStream fis = new FileInputStream("file.txt")) {
    // work with file }
```

This prevents resource leaks even if an exception occurs.