(1) Write a java program thats reads a series of numbers from a file (input.txt), determines the higest number in the series, calculates the sum of natural numbers up to 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.

Solve:

```
import java.io.*;
import java.util.*;
public class NumberProcessor{
    public static void main(String[] args){
        try{
            Scanner scanner = new scanner(new File("input.txt"));
            String line = scanner.nextLine();
            String[] numbers = line.split(",");
            scanner.close();
```

```
List<Integer> numList = new ArrayList<>();
int highest = Integer.MIN_VALUE;

for(String numStr : numbers){
    int sum = Integer.parseInt(numStr.trim());
    numList.add(sum);
    If (num > highest) {
            highest = num; } }

int sumOfHighest = 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:" + SumOfHigest);
writer.close();

System.out.println("Processing complete. Check output.txt for
results."); } catch (IOException e) {
    System.err.println("Error : " + e.getMessage()); }
    catch (NumberFormatException e) {
        System.err.println("Invalid number format in inputfile.");
    } }
}
```

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

⇒

Static Fields & Methods: Static fields and methods belongs to the class itself, not any individual instance of the class. This means that there is only one copy of the static variable or method, shared by all instances of that class.

Final Fields & Methods: Final is a keyword that ensures that a variable or method cannot be modified or overridden after it is initialized or defined. The method body is defined once and for all.

Difference between Static and Final Fieds & Methods:

IT-23009

Feature	Static	Final
Belongs to	The class, not the instance.	The class, depending on context
~~Access~~ Usage	Used of class level data or methods shared across instances.	Used to make fields immutable or method prevent overriding.
Classes	Can make nested classes	Cannot be inherited
Fields	Shared by all instances	Value cannot be changed after initializa
Store	store in method area	Stored in Heap

code:

```
class Example{
    static int count =10;
    static void incrementCount(){
        count++;}}

public class Main{
    public static void main(String[] args){
        Example obj =new Example();
obj.incrementCount();
System.out.println(obj.count);

Example.incrementCount();
System.Out.println(Exampl.count);}}
```

1) First block (via object):

- We create and object [obj] of the Example class
- ~~Access~~ Java allows accessing static members through an instance.

3. Write a Java program to find all factorians numbers within a given range. A number is called 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, and print all factorion numbers within the range.

⇒ Code:

```
import java.util.scanner;
public class factorion{
    private static final int[] fact = new int[10];
    private static void precompute (){
        fact[0] =1;
        for (int i=1 ; i<10; i++){
            fact[i]= i * fact[i-1];
        }
    }
    private static boolean isFactorion (int n){
        int sum =0, temp =n;
        while (temp> 0) {
            sum += fact [temp%10];
            tem /= 10;}
        return sum == n ; }
```

```
publie static void main (string [] args) {
  Seanner se = new Seanner (System.in);
  System.out.print ("Enter lower bound : ");
  int low = se.next Int ();
  System.out.print ("Enter upper bound : ");
  int high = se.next Int ();
  preecompute ();
  System.out.println ("faetorion numbers:");
  boolean found = false;
  for (int i = low ; i <= high; i++) {
     if (isFaetorsion (i)) {
        System.out.print (i + " ");
        found = true; } }
  if (! found) { System.out.println ("None"); }
  se.close (); }
  }
```

Sample Input : Enter lower bound : 1
Enter upper bound : 100000

Sample Output : Faetorion numbers : 1, 2, 145, 40585

4. Distingwish the differenees among class, loeal and instanee variables what is sifgnifieanee of this keyword. [IT-23009]

⇒

1. Class variables:

Defination: Class variable are deelared with the statie keyword inside a class but outside any methods or construetors.

Seope : Shared aeaross all objeets of the elass.

Memory Alloeation: Stored in statie memory.

Default Value: Automatieally initialized with default values.

Aeeess : Can be aeeessed using either the elass name or an objeet.

Purpose : Useful when data needs to be shared among all instenees of the eolass.

2. Loeal variables:

Defination : Loeal variables are deelared inside a method, construetor or bloek of eode.

Seope : Limited to the method or bloek where they are deelared.

Memory Alloeation: stored in staek memory.

Default Value: Not initialized automatieally-must be explieitly assigned a value before use.

Aeeess : Can only be aeessed within the method or bloek where they are deelared.

Purpose: Used for temporary storage of values only during method exeeution.

3. Instance variables: [IT-23009]

Defination : Instance variables are declared inside a class but outside any methods and they do not have the static keyword.

Scope : Belongs to individual object, meaning each instance has its own separate copy

Memory Allocation: Stored heap memory, within the object.

Default value: Automatically initialized with default values.

Access: Can be accessed using an object reference.

Purpose: Holds unique attributes for each object, making objects independent of one other.

Significance of this keyword:

In Java this keyword reference to the current object of a class. It helps distinguishing instance variables from local variables, calling methods or constructors within the same class and enabling method chaining.

1. Referring to instance variables: When a local variable have the same name `this` is used to refer to the instance variable explicitly.
2. Calling another constructor: The `this()` Keyword is used to call another constructor within the same class, reducing redundant initialization code
3. Calling a method from the same class: You can use `this` to explicitly call another method within the same class.
4. Returning the current object: A method can return `this` allowing the method chaining.
5. Passing the current object as an argument: You can pass `this` as an argument to a method or constructor.

Answer of the question no-5 [IT-23009]

5.

Java Program:

```
import java.util.scanner;XXX
publie class ArraySum{
    publie statie int CalSum(int[]arr){
        int sum=0;
        for(int num:arr){
            sum+=num;
        }
        return sum;
    }
publie statie void main(string[]args){
    int[]numbers={5,10,15,20,25};
        int total=CalSum(numbers);

    System.out.println("Sum of array elements: "+total);
    }
}
```

Output: Sum of array elements: X5

Answer of the question no-6 [IT-23009]

6.

Ans: **Access Modifier:** An access modifier in java is a keyword that controls the visibility and accessibility of classes, methods, and variables in a program. Access modifiers help in data hiding, security and encapsulation by restricting access to certain parts of the code.

Comparison the accessibility of public, private and protected modifier.

Modifier	Accessibility Scope
Public	Accessible everywhere in the program, including different classes, packages and subclasses.
Private	Accessible only within the same class. Not visible to other classes even within the same package.
Protected	Accessible within the same package and in subclasses, even if they are in different packages.
Default (No modifier)	Accessible only within the same package but not in subclasse outside the package.

Describing different

1. Final variables (constant variables) [IT-23009]

i) Declared using the `final` keyword.

ii) Once assigned, the value cannot be changed.

iii) Can be, local, instance or static.

iv) Used for storing constants values that should not change.

Example: `PI` in a `Math` class.

2. Transient variables:

i) Declared using the `transient` keyword.

ii) Skipped during serialization (not saved in files or streams).

iii) Used when a variable should not be stored for security or optimization reasons.

Example: A `password` field in a `User` class that should not be saved

3. Volatile variables:

i) Declared using `volatile` keyword.

ii) Ensures visibility of changes across multiple threads in multithreading

iii) Prevents the compiler from caching the variables, forcing it to read the latest value from memory

Example: A shared variable `status` used in a multithreaded program.

Answer of the question no-7 [IT-23009]

Ans:

Java program:

```
import java.util.Scanner;
public class QuadRoot {
    public static void main (String[] args) {
        Scanner se= new Scanner (System.in);
        System.out.print ("Enter coefficients a, b, and c:");
        int a =se.nextInt(), b=se.nextInt(), c=se.nextInt();

        double d= b*b - 4*a*c;
        if (d<0){System.out.println ("No real roots.");}
        else { double r1=(-b + Math.sqrt(d)) /(2*a);
               double r2 =(-b- Math.sqrt(d)) /(2*a);

               double minRoot = Double.MAX_VALUE;
               if (r1>0) minRoot =Math.min (minRoot, r1);
               if(r2>0) minRoot = Math.min (minRoot, r2);

System.out.println (minRoot== Double.MAX_VALUE ? "No positive roots."
: "The smallest positive root is: " + minRoot); }
        sc.close();
    }
```

Answer of the question no-8 [IT-23005]

Ans: Write program to determine letters, whitespaces and digits.

Java Program:

```
import java.util.Scanner;
public class CharChecker{
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter a string: ");
    String str = sc.nextLine();
    int letters=0, digits=0, spaces=0;
    for(char ch : str.toCharArray()){
      if(Character.isLetter(ch)) letters++;
      else if(Character.isDigit(ch)) digits++;
      else if(Character.isWhitespace(ch)) spaces++;
    }
      System.out.println("Letters: " + letters);
      System.out.println("Digits: " + digits);
      System.out.println("Whitespaces: " + spaces);
      sc.close();
  }
}
```

[IT-23002]

Example run:

Input:
Enter a string: Hello 123 World!

Output:
Letters: 10
Digits: 3
Whitespaces: 2

In Java, an array can be passed as an argument to a method just like any other variable.

Example:

```
public class ArrayPass{
    static int sumArray(int[] arr){
        int sum = 0;
        for (int num : arr){
            sum += num; }
        return sum;}

public static void main(String[] args){
    int[] numbers = {1,2,3,4,5};
    int total = sumArray(numbers);
    System.out.println("Sum of array elements: " + total);

    }
}
```

Output: Sum of array elements: 15

i) Arrays are passed by reference to methods in Java,
ii) The method `sumArray(int[] arr)` takes an array parameter,
iii) We call `sumArray(numbers)` from `main()`, passing the array `numbe`

Answer of the question no-9 [IT-23009]

Ans:

Method Overriding:

Method overriding is a feature in Java that allows a subclass to provide a new implementation for a method that is already define in its superclass. How its works in inheritance —

i) When a subclass overrides a method, 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 behavior for subclass objects while maintaining a consistent interface.

What happens when a subclass overrides a Method —

1. Subclass method executes, replacing the superclass method.
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.

[IT-2300]

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) Visibility Restriction – Cannot reduce access (e.g: `public` → `private`)

ii) Exception Limitation – Cannot throw broader exceptions.

iii) Final & static methods – `final` methods can't be overriden; static method are hidden, not overriden.

2. Issues with constructors:

i) Constructors cannot be overridden 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 inserts a default constructor call, which may cause an error if the superclass lacks a no argument constructor.

Answer of the question no-10 [IT-23009]

Ans: Difference between Static and Non-static members in java —

Feature	Static Members	Non-Static Members
Defination	Belong to the class and are shared among all objects	Belong to the individual objects, each instance has its own copy.
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 memory.
Invocation	Can be called without creating an object.	Requires object creation before calling.
Usage	Used for constants, utility methods and shared properties	Used for Object specific behaviour and instance data.
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 employee IDs, car numbers, phone numbers. etc. They vary for each object and are not shared.

Check if a number or string is palindrome— [IT-2300J]

Program:

```
import java.util.Scanner;
public class PalindromeCheck {
    static boolean Pal(String s){
        return s.equalsIgnoreCase(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.print("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) ?" is":"is not")+ "a palindrome.");
        sc.close(); }
```

Answer of the question no-11 [IT-23009]

Class Abstraction:

Class abstraction is the process of simplifying complex reality by modeling only the essential attributes and behaviours of an object while hidding unnecessary details. It focuses on "what" an object does rather than "how" it does it.

Example: A `vehicle` class may define as abstract method start(), but each subclass (eg: car, bike) provides it's own implementation.

Class Encapsulation:

Encapsulation is the building of data (attributes) and methods (behaviors) that operate on that data within a single unit. It also involves controlling the access to the internal data of an object, preventing direct modification from the outside. This is often achived through access modifiers like private, protected and public

Example: A `BankAccount` class has a `balance` variable marked `private` and users can access it only through `getBalance()` and `deposit()` methods. These methods can include logic to validate the operations and maintain the integrity of the data.

Difference between abstract Class and Interface — [IT23009]

Abstract Class	Interface
1. Abstract class can have both abstract and concrete methods.	1. Interface contains only abstrac methods.
2. Can have instance variables with any access modifier	2. Can have only public, static, final constants (no instance variables).
3. Can have constructors	3. Cannot have constructors.
4. Can include both implemented and non-implemented methods.	4. All methods are public and abstract by default.
5. Used when classes share common behavior and need some abstraction.	5. Used when different class mus follow a common contract but sh no implementation.
6. Example: `abstract class Animal{}`	6. Example: `Interface Animal{}`

Answer of the question no -12 [IT-2300J]

Java program:

```
import java.util.Scanner;

class BaseClass{
  void printResult(String msg, Object result){
     System.out.println(msg +result);
    }
  }

class SumClass extends BaseClass {
  double sumSeries(){
       double sum =0.0;
       for(double i= 1.0; i>=0.1; i-=0.1){
          sum +=i;
       }
       return sum;
    }
}
class DivisorMultipleClass extends BaseClass{
   int gcd(int a, int b){
       while(b!=0){
       int temp =b;
       b=a%b
       a=temp; }
   return a;
}
```

```
class NumberConversionClass extends BaseClass{
    string toBinary(int num){
    return Integer.toBinaryString(num);
}
String toHex(int num){
    return Integer.toHexString(num).toUpperCase();
}
String toOctal(int num){
    return Integer.toOctalString(num);
}
}
class CustomPrintClass{
    void pr(String msg)
    System.Out.println(">>"+msg);
}
}
publie class MainClass{
    publie static void main(String[] args){
    Scanner se =new Scanner(System.in);
    SumClass sumobj =new SumClass();
    DivisorMultipleClass dmObj = new DivisorMultipleClass();
    NumberConversionClass noObj = new NumberConversionClass();
    CustomPrintClass printObj = new CustomPrintClass();
```

[IT23006]

[IT23009]

```
double sum = sumObj.sumSeries();
sumObj.PrintResult("Sum of series:", sum);

System.out.print("Enter two numbers for GCD & LCM:");
in num1 = se.nextInt(), num2 = se.nextInt();
sumObj.printResult("GCD:", dmObj.gcd(num1, num2));
sumObj.printResult("LCM:", dmObj.lcm(num1, num2));

System.out.print("Enter a decimal number for conversion:");
int num = sc.nextInt();
printObj.pr("Binary:" neObj.toBinary(num));
printObj.pr("Hexadecimal:" neObj.toHexa(num));
printObj.pr("Octal:" neObj.toOctal(num));

se.close();
}
}
```

Answer of the question no-13 [IT-23005]

Java program:

```
import java.util.Date;
  class Geometricobjeect{
     string color;
     boolean filled;
     Date dateCreated =new Date();

     Geometricobjeect(String color, boolean filled){
          this.color= color;
          this.filled = filled; }

     publie string toString(){
       return "Color:" + color +", filled:"+filled+", Created:"+dateCreat
     }
}
class Circle extends Geometricobjeect{
   double radius;
   Circle(double radius, String color, boolean filled){
     super(color, filled);
     this.radius =radius; }

   publie string toString(){
   return "Circle->"+super.toString()+", Radius:"+radius+", Area
   +(Math.PI * radius * radius);}
}
```

[IT-2300]

```
class Rectangle extends GeometricObject {
    double width, height;

    Rectangle(double width, double height, String color, boolean filled) {
        super(color, filled);
        this.width = width;
        this.height = height; }

    public string toString() {
        return "Rectangle -> " + super.toString() + ", width: " + width + ",
        Height: " + height + ", Area: " + (width * height); }
}

public class Main {
    public static void main (String[] args) {
        System.out.println (new Circle (5, "Red", true));
        System.out.println (new Rectangle (4, 7, "Blue", false));
    }
}
```

Answer of the question no-14 [JT-23005]

Significance of BigInteger in java:

BigInteger is used to handle arbitarily large integers that primitive types like int and long cannot store.

I) Supports mathematical operations by providing like `add()`, `divide()` `mod()` `pow()`, which primitive types don't offer,

II) Used in cryptography where large numbers are essential for encryption algorithms like RSA.

Java program to compute factorial by BigIntegers:

```
import java.math.BigInteger;
  public class FactorialBigInteger{
     public static BigInteger factorial (int n){
     BigInteger fact = BigInteger.ONE;
     for(int i=2; i<=n; i++){
       fact =fact.multiply(BigInteger.valueOf(i)); }
     }
     return fact;
  public static void main(String[] args){
       int num =50;
       System.out.println(num+"! is\n"+factorial(num)); }
}
```

Answer of the question no-15 [IT-23005]

Abstract class:

I) It's have both abstract methods and concrete methods.

II) Can have instance variables with any access modifier.

III) Can have constructors.

IV) A class can extend only one abstract class.

V) Methods can have any access modifier.

VI) Used when a class needs to share common behavior and state among related classes.

Interface:

I) It's can have only abstract methods.

II) Can only have public static final constants.

III) A class can implement multiple interfaces

IV) Can not have constructors.

V) Methods are public by default.

VI) Used when defining a contract that multiple classes must follow, regardless of their position in the class hierarchy.

When to use an abstract class over an interface —

I) Use an abstract class when subclass need default behavior,

II) If maintaining instance variables pefer an abstract class.

III) When enforcing a strict parent-child relationship, an abstract class is ideal.

IV) Use abstract class if subclasses share object initialization logic.

☐ Yes, a class in java can implement multiple interfaces. This is a key advantage of interfaces over abstract classes, as java does not support multiple inheritance with classes.

• two interface define the same method signature:

```
interface A{
    void display();
}
interface B{
    void display();
}
class C implements A, B {
        public void display(){
        System.out.println("Resolved display method in class C");}
}
```

Answer of the question no-16 [IT-23009]

Polymorphism in Java

Polymorphism is a ability of a single interface to represent multiple forms. In Java, polymorphism is achieved through method overriding (runtime polymorphism) and method overloading (compile-time polymorphism).

Dynamic method dispatch:

Dynamic method dispatch refers to the process where method calls are resolved at runtime instead of compile time. When a subclass overrides a method from its superclass and is referenced through a parent type, java determines the method to execute based on a actual object type no the referenced type.

Example: Polymorphism with inheritance and method overriding :—

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

IT-3300J

```
class Cat extends Animal{
    void makesound(){
    System.out.println("cat meows");}
}
publie class Main{
    publie statie void main(String[] args){
    Animal myAnimal;
    myAnimal = new Dog();
    myAnimal.makesound();

    myAnimal = new Cat();
    myAnimal.makesound();}
}
```

Impact of polymorphism on performance

Runtime Overhead: Method calls are resolve at runtime, adding slight per formence cost.

Cache Ineffieieney: Frequent palymorphie calls ean reduee CPU instruction caching.

JIT Optimization: Jvm may inline frequently used methods, reducing overhead.

Trade-offs: Polymorphism vs specific method calls IT23009

Polymorphism	Direct methods calls
1. High flexibility (work with the different class.	1. Low
2. Easy to extends new behaviors,	2. Harden
3. Slightly slower due to runtime lookup.	3. Faster due to direct execution,
4. Improves readability for high-level design.	4. Can become flattened with many conditions.

Answer of the question no 17 [IT23009]

- ArrayList is backed by a dynamic array. This means elements are stored contiguously in memory.
- LinkedList is a doubly linked list. Each element stores its data and pointers to the previous and next node in the sequence

Difference:

Operation	ArrayList	LinkedList
Access	O(1) - Direct access via index	O(n) - Sequential traverse ne
Insertion (add at end)	O(1) - Amortized	O(1) - Direct tail insertion
Insertion (add at index)	O(n) - shifting element	O(n) - Traversing to the inde
Deletion (remove at index)	O(n) - "	O(n) - Traversing to the inde
Deletion (remove first/last)	O(n) - "	O(1) - Direct head/tail remo

When to use consideration for large datasets: [IT23009]

1. ArrayList:

I) Frequently random access: If any application frequently needs to access elements at arbitrary position then arrayList is the better choice.

II) Predictable size: If we know the size of our list in advance then it's simpliers to be implementation.

2. LinkedList

I) Frequently insertions and deletion: It can do easily and where in large dataset it is common.

II) Implementing queues or stack: LinkedList is a natural choice for implementing queues or stacks.

Thread safety using synchronized [IT-23009]

The `synchronized` keyword ensures only one thread can access a block at a time.

Example:

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

Dead lock example and solution:

```
class Resource{
  synchronized void methodA(Resource r){
  System.out.println(Thread.currentThread().getName()+"locked A");
  synchronized(r){System.out.println(Thread.currentThread().
   getName()+"locked B");}
}
}
```

Issue: Thread 1 locks A and waits B, while Thread 2 locks B and waits for A → Infinite waiting!

[IT23009]

Avoid deadlock:

1. Always lock resources in the same order.
2. Use `tryLock()` from `ReentrantLock` to avoid waiting.

→

```
import java.util.concurrents.locks.*;
  class SafeResources{
     private final Lock lock =new ReentrantLock();

  boolean tryMethod (SafeResource other){
    if(!lock.tryLock()) return false;
    try{
       return other.lock.tryLock();}
       finally{
         lock.unlock();
         if (other.lock.tryLock()) other.lock.unlock();
        }
    }
  }
```

Answer of the question no-20 [IT23009]

Exception Handling in Java:

Exception handling in java ensures a program runs smoothly even unexpected error occur. It uses try-catch-finally, throw and throws to manage exceptions.

1. Checked exceptions:

i) Must be handled using try-catch or declared using throws

ii) Example: IOException, SQLException

code:

```
try{
    FileReader file = new FileReader ("file.txt");}
    catch (IOExceptione){
        System.out.println ("File not found");}
```

2. Unchecked exceptions:

i) Inherit from runtime exception.

ii) Occur at runtime and don't need explicit handling.

iii) Example: ArithmeticException

```
int a=10/0;
```

Creating and throwing custom exceptions: [IT23009]

```
class CustomException extends Exception{
    public CustomException (String mssg){super(msg);}
}
public class Main{
    static void validate (int age) throws CustomException{
    if(age<18) throw new customException("Age must be 18+");
}
public static void main(String[] args)
    try{validate(16);}
    catch (CustomException e){
    System.out.println(e.getMessage());}
    }
}
```

Answer of the question no 21 [IT-23005]

- Interface can have default methods, allowing method implements inside intefaces. This makes interfaces more flexible but still distinct from abstract classes.

- Impact of Default methods on interface usage —

1) Interfaces can now provide behavior, reducing boilerplate code.
11) Unlike abstract class, interfaces cannot have instance variables or constructors.

Handling Conflicting Default methods—

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

    class C implements A, B{
        public void show (){
            System.out.println("C"); } // Must
```

Answer of the question no-22 [IT23009]

Difference between Hash Map, Tree Map and Linked Hash Map in Java:-

Feature	Hash Map	Tree Map	Linked Hash Map
Structure	Hash Table	Red-Black tree	Hash table + Doubl Linked listed,
Ordering	No order	Sorted	insertion
Time Complexity	O(1)	O(logn)	O(1)
Allows Null	Only one null key	No	Only one null ke
Iteration order	Unpredictable	Sorted	Insertion order,
Best use case	Fast access when order doesn't matter	Sorted data	Predictable iterati order

When to use each:

I) Hash map → Best for lookups when ordering isn't needed,

II) Tree map → Needed when key must be sorted,

III) Linked HM → Use when maintaining insertion order,

Ordering difference:

Hash Maps → does not guarantee any order,

Tree map → keeps elements sorted by key.

Answer of the question no 23 [IT-23005]

Static Binding: (Compile time binding)

I) Happens at compile time.

II) Used for static, private and final methods, as well as method overloading.

III) Faster since method resolution occurs at compilation,

Dynamic Binding: (Runtime Binding)

I) Happens at runtime using method overriding.

II) Used for non-static methods in an inheritance hierarchy.

III) Enables polymorphism, allowing method calls to be resolved based on the objects runtime type.

Performance & method resulation impact:

→ Static binding is fasten because it resolved at compile time

→ Dynamic Binding is flexible but slightly slower due to runtime,

→ Use static methods for utility functions where overriding isn't needed

→ Use dynamic binding for polymorphism, allowing extensibility in OOP design.

Answer of the question no-24 [IT-23009]

Advantage of using ExecutorService Over manually Managing Threads –

1. Thread pooling → Reuses threads, reducing overhead.
2. Better Resource manggement → Avoids exessive thread creation.
3. Flexible task submission → Supports runnable, Callble and Future.
4. Graceful shutdown → Allows controlled thread termination,
5. Concurrency control → Handles multiple tasks efficiently.

Why use Callable instead of Runnable –

I) Returns a result (Future<V>)

II) Can throw checked exceptions,

III) Supports asynchronous processing

Submit() vs execute():

execute() → Runnable, void return type, uncaugh exceptions terminate thread and use case in Fire and forgot tasks,

submit() → Runnable/Callable, Future<V> return type, exception can be retrive using future.get() and use case when result are needed.

Runnable & callable; [IT-23005]

```
import java.util.concurrent.*;
public class ExecutorExample {
    public static void main(String[] args) throws Exception {
    ExecutorService executor = Executors.newFixedThreadPool(3);

    Callable<Integer> task = () -> {
        TimeUnit.SECONDS.sleep(2);
        return 42; };

    Future<Integer> result = executor.submit(task);
    System.out.println("Result:" + result.get());
    executor.shutdown();
    }
}
```

Answer of the question no-25 [IT23009]

Key issue in Exception Handling –

I) Checked vs Unchecked exceptions → Use checked exceptions for recoverable errors,

II) Proper use of try-catch-finally → Ensure resource cleanup,

III) Meaningful exception Message → Helps debugging.

IV) Avoid Swallowing exceptions → Always log or rethrow them,

V) Use custom exceptions when needed → Provides clarity in error handling.

Java Program (circle with Exception Handling):

Answer of the question no-31 [IT-2300J]

Java program to print the current date and time;

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateTime{
    public static void main(String[] args){

    LocalDateTime now=LocalDateTime.now();
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern
                                  ("yyyy-MM-dd HH:mm:ss");

    String formattedDateTime=now.format(formatter);

    System.out.println("Current Date and Time:"+formatted
                       DateTime);

    }
}
```

Answer of the question no - 32 [IT-23002]

Java Program (Counter):

```
class CounterClass {
    private static int instanceCount = 0;

    public CounterClass() {
        instanceCount++;
        If (instanceCount > 50) {
            instanceCount = 0; }
    }
    public static in getInstanceCount(){
        return instanceCount; }

    public static void main(String[] args){
        for(int i = 1; i <= 55; i++){
            new CounterClass();
System.out.println("Object" + i + ", Instance count: " +
                                    getInstanceCount()); }
    }
}
```

Answer of the question no-33 [IT-23009]

Java Program: (Extreme Finder)

```
public class ExtremeFinder {
    public static int findExtreme (String type, int... nums) {
    int res = nums[0];
    for (int n : nums)
        res = type.equals ("smallest") ? Math.min (res, n) :
                                         Math.max (res, n)

    return res;
    }

    public static void main(String[] args) {
    System.out.println (findExtreme ("Smallest", 5, 2, 9, 1));
    System.out.println (findExtreme ("largest", 8, 3, 10, 4));
    }
}
```

Answer of the question no-34 [IT-2300J]

```
Public static void main (String [] args) {
String s1 = "This is IET 2107 Java";
String s2 = new String ("This is IET 2107 Java");
String s3 = "This is IET 2107 Java";
System.out.println (s1.equals(s2)); // true - compares content, which is identical.
System.out.println(s1 == s2); // false - different object in memory
System.out.println(s1 == s3); // true - both refer to the same string literal in the string pool
```

Answer of the question no-37 [IT-23005]

Solution 1: (Override show() in z and explicitly call a specific interface method):

```
public class z implements x, Y {
    public void show(){
        x.super.show(); }
    public static void main(String[] args){
        z obj = new z();
        obj.show(); }
}
```

Solution 2: (Provide a new implementation in Z):

```
public class z implements x, Y {
    public void show(){
        System.out.println("z's own show method"); }

    public static void main(String [] args){
        z obj = new z();
        obj.show(); }
}
```

Answer of the question no-40 [IT23009]

Arithmetic operations:

```
import java.util.Scanner;

public class ArithmeticOperations {
    public static void main (String[] args) {
    Scanner sc = new Scanner (System.in);
    try {
        System.out.println ("Enter two numbers : ");
        int a = sc.nextInt(), b = sc.nextInt();

        System.out.println ("Sum : " + (a+b));
        System.out.println ("Difference : " + (a-b));
        System.out.println ("Product : " + (a*b));
        System.out.println ("Quotient : " + (b!=0 ? (a/b) :
                            "Undefined (division by zero)"));}

        catch (Exception e){
        System.out.println ("Error: Invalid input!");}

        finally {
            sc.close();}

    }
}
```