Final Examp Preparation

IT-23038

Answer to the question no. (1)

Code:

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class SumOfNaturalNumbers{
    public static void main(String[] args){
try{
    Scanner se = new Scanner(new File("input.txt"));
    String line = se.nextLine();
String[] numbers = line.split(";");
    int maxNumber = Integer.MIN_VALUE;
for(String number : numbers){
    int currentNumber = Integer.parseInt(number.trim());
      if(currentNumber > maxNumber){
          maxNumber = currentNumber;
        }
    }
long sum = (long) maxNumber * (maxNumber+1) / 2;
```

```
PrintWriter writer = new PrintWriter("output.txt");
    writer.println(sum);
    writer.close();
System.out.println("Sum of natural numbers up to" + maxNumber +
    "is" + sum);
} catch (FileNotFoundException e){
    System.out.println("File not found:" + e.getMessage());
    }
  }
}
```

Answer to the question no. (2)

Key Differences:

1. Static:
   i) Shared by all instances of the class.
   ii) Accessed via class name or object.

2. Final:
   i) Cannot be modified or overriden.
   ii) Must be initialized once.

Example:

```
class MyClass{
    static int staticField = 10;
  static void staticMethod(){
      System.out.println("Static method");
    }
}
public class Main{
    public static void main(String[] args){
      MyClass obj = new MyClass();
       obj.staticMethod();
   System.out.println(obj.staticfield);
    }
}
```

Answer to the question no. (3)

Code:

```
import java.util.Scanner;
public class FactorialNumbers{
      public static int factorial(int n){
       if(n==0||n==1){
          return 1;
       }
    int result = 1;
```

```
    for(int i = 2; i<= n; i++){
        result *= i;
    }
    return result;
}
public static boolean isFactorion(int number){
    int originalNumber = number;
    int sum = 0;
    while(number!=0){
        int digit = number%10;
        sum += factorial(digit);
        number /= 10;
    }
    return sum == originalNumber;
}
public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
System.out.println("Enter lowerbound");
    int lowerbound = sc.nextInt();
System.out.println("Enter upperbound");
    int upperbound = sc.nextInt();
System.out.println("Factorion numbers in the given range:");
    for(int i = lowerbound; i<= upperbound; i++){
        if(isFactorion(i)){
        System.out.print(i+" ");
        }
    }
}
```

Answer to the question no. (9)

Differences among Class, Local and Instance Variable:

1. Class Variables:
   i) Declared with static keyword.
   ii) Shared among all instances of the class.
   iii) Exist as long as the class is loaded into memory
   iv) Can be accessed using the class name or an instance of the class.

2. Instance Variables:
   i) Belong to individual instances of a class.
   ii) Declared without the static keyword.
   iii) Each object of the class gets its own copy of the instance variables.
   iv) Initialized to default values if not explicitly initialized.

3. Local Variables:
   i) Declared inside methods, constructors or blocks.
   ii) Limited to the scope of the method or block they are declared in.
   iii) Must be explicitly initialized before use.
   iv) Do not have default values.

④ Sigficance of (this) keyword:

i) Refers to the current instance of the class.

ii) Used to call the current class's constructor.

iii) Can be used to pass the current object as a parameter to the other methods or constructors.

Answer to the question no. (5)

```
public class ArraySum {
    public static int sumOfArray(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 result = sumOfArray (numbers);
   System.out.println ("Sum of array elements:" + result);
    }

}
```

Answer to the question no. (6)

Access Modifiers:

Access modifiers control the visibility of classes, methods and variables in Java. They define where these elements can be accessed from in the program.

Different Types of Access Modifiers:

1. Public - Accessible from any other class.
   Ex: public class MyClass { }
2. Private - Accessible only within the class it is declared in
   Ex: private int age;
3. Protected - Accessible within the same package and by subclasses.
4. Ex: protected int salary;
4. Default - Accessible only within the same package.
   Ex: int id;

Answer to the question no. (7)

Code:

```
import java.util.Scanner;
public class QESolver{
    public static void main(String[] args){
Scanner sc = new Scanner(System.in);
System.out.print("Enter coefficients a, b, c:");
  int a = sc.nextInt(); int b = sc.nextInt();
  int c = sc.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*a);
 double root2 = (-b - Math.sqrt(discriminant))/(2*a);
double smallestPositiveRoot = Double.MAX_VALUE;
  if(root1>0 && root1<smallestPositiveRoot)
      smallestPositiveRoot = root1;
  if(root2>0 && root2<smallestPositiveRoot)
      smallestPositiveRoot = root2;
```

```
    if (smallestPositiveRoot == Double.MAX_VALUE){
      System.out.println("No real roots");
    }
    else{
      System.out.println("Smallest positive root: "+ smallestPositiveRoot);
    }
  }
}
```

Answer to the question no. (8)

Here is the Java program that determines the count of letters, whitespaces and digits in a given string.

```
import java.util.Scanner;
public class CharacterCounter{
    public static void main(String[] args){
Scanner sc = new Scanner(System.in);
System.out.print("Enter a string");
    String input = sc.nextLine();
 int letters = 0, whitespaces = 0, digit = 0;
for(int i = 0; i<input.length(); i++){
    char ch = input.charAt(i);
```

```
if(Character.isLetter(ch)){
    letters++;
}
else if(Character.isWhitespace(i)){
    whitespaces++;
}
else if(Character.isdigit(ch)){
    digits++;
}
}
System.out.println("Number of letters:"+letters);
System.out.println("Number of whitespaces:"+whitespaces);
System.out.println("Number of digits:"+digits);
```

4. An example of passing an array to a function:

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

IT-23038

Answers to the question no.(9)

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 it works in context of inheritance:-

i) When a subclass overrides a method, the subclass versio 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 consistance 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 overriden superclass method.

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

* Potential issue when overriding methods:

1. Visibility restriction - Cannot access (e.g. `public` → `Private`)
2. Exception Limitation - Cannot throw broader exceptions.
3. Final & Static methods - `Final` methods cannot be overriden, static methods are hidden not overriden.

* Issues with constructors:

1. Constructors cannot be overriden because they are not inherited.
2. `super()` must be used for superclass initialization.
   - If the superclass has a parameterized construct the subclass must explicitly call `super(arguments);`
   - If no explicit `super()` is used, Java inserts a default constructor call, which may cause a err if the superclass lacks a no argument constructo

IT-23038

Answers to the question no. (10)

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 behaviours and instance data.
Example	Static members are shared and constant for all. Examples include- company name, gratuti, domain extensions etc	Non-static members are uniq to each instance. examples include employee IDs, car- numbers, phone numbers etc.

⊞ Check if a number or string is palindrome:

Code:

```
import java.util.Scanner;
public class PalindromeCheck{
    static boolean Pal(String s){
    return s.equalsIgnoreCase(new StringBuilder(s).reverse()+
    }
 static boolean Pal(Int n){
     int rev=0,temp=n;
 while(temp>0){
    rev=rev*10+temp%10;
    temp=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 palindr
    System.out.print("Enter a string:");
        String s=sc.nextLine();
System.out.println(s+(Pal(s)?"is":"is not")+"a palindr
}
}
```

IT-3038

Answer to the question no. (1)

Class Abstraction:

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

Example: A `vehical` class may define as abstract method sfant(), but each subclass provides its own implementation

Class Encapsulation:

Encapsulation is the building of data and methods that operate on 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 achieved through access modifiers like private, public, protected.

Example: A `BankAccount` class has a `balance` variable marked `private` and users can access it only through `getbalance()` and `deposite()` methods. These methods can

include logic to validate the operations and maintain the integrity of the data.

Difference between abstract class and interface :-

Abstract class	Interface
1. Abstract class can have both abstract and concrete methods.	1. Interface contains only the abstracts methods.
2. Can have instance variables with any access modifiers.	2. Can have only public, static, final constants.
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 behaviours and need some abstraction.	5. Used when different class must follow a common contract but share no implementation.
6. Example :- abstract class Animal { }	6. Example :- Interface Animal { }

Answer to the question no. (12)

Code:

```
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 NumberConversion 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);
    }
  }
public class MainClass{
      public static void main(String args[]){
          Scanner sc = new Scanner(System.in);
SumClass sumobj = new SumClass();
DivisorMultipleClass dmobj = new DivisorMultipleClass();
NumberConversion nCObj = new NumberConversion();
CustomPrintClass printobj = new CustomPrintClass();
```

```
double sum = sumobj.sumSeries();
sumobj.PrintResult("Sum of series:", sum);
System.out.print("Enter two numbers for GCD & LCM:");
  int num1 = sc.nextInt(), num2 = sc.nextInt();
dmObj.printResult("GCD:", dmobj.gcd(num1, num2));
dmobj.printResult("LCM:", dmobj.lcm(num1, num2));
System.out.print("Enter a decimal number:");
  int num = sc.nextInt();
printobj.pr("Binary:" nc obj.toBinary(num));
printobj.pr("Hexadecimal:" ncobj.toHex(num));
printobj.pr("Octal:" ncobj.toOctal(num));
sc.close();
}
}
```

IT-23038

Answers to the question no. (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 demostrate their functionality. Below is the Java code that follows the UML scenario:

```
import java.util.Date;
class GeometricObject{
    private String colour;
    private boolean filled;
    private Date dateCreated;
  public GeometricObject(){
      this.colour = "white";
      this.filled = false;
      this.dateCreated = new Date();
    }
public GeometricObject(String colour, boolean filled){
      this.colour = colour;
      this.filled = filled;
      this.dateCreated = new Date();
    }
    public String getColour(){
      return colour;
    }
```

```
public void setcolour(String colour){
    this.colour = colour;
  }
public boolean isFilled(){
   return filled;
  }
public void setfield(boolean filled){
     this.filled = fielld;
   }
class circle extends GeometricObject{
     private double radious;
   public circle(){
       super();
    this.radious = 1.0;
  }
public Circle(double radious){
    super();
   this.radious = radious;
 }
public double getRadious(){
   return radious;
}
```

```
public void setRadious(double radious){
    this.radious = radious;
  }
public double getArea(){
    return Math.PI * radious * radious;
  }
public class Main{
    public static void main(String args[]){
Circle circle = new Circle(5.0, "blue", true);
  Circle.printCircle();
System.out.println("Area:" + circle.getArea());
System.out.println("Perimeter:" + Circle.getPerimeter());
System.out.println("Diameter:" + circle.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());
}
}
```

Answer to the question no. (14)

Significance of BigInteger:

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

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

```
import java.math.BigInteger;
import java.util.Scanner;
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 sc = new Scanner(System.in);
System.out.println("Enter an integer:");
        int num = sc.nextInt();
  if (num<0){
System.out.println("Factorial is not defined");
}
else{
BigInteger factorialResult = factorial(num);
System.out.println("Factorial of"+num+"is");
System.out.println(factorialResult);
}
sc.close();
}
}
```

IT-23038

Answers to the question no. (15)

Abstract classes:

→ Purpose: Provide a partial implementation and allow subclasses to complete it.
→ Fields: Can have instance variable.
→ Methods: Can have both abstract and concrete methods.
→ Inheritance: Single inheritance.
→ Use case: Preferred when you need to share code or maintain 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:

1. Shared code.
2. State management.
3. Constructor logic.

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();
    }
}
```

IT-23038

Answer to the question no.(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 overriden 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 banks");
    }
}

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();
    myDog.sound();
    myCat.sound();
    }
}
```

Answer to the question no. (17)

ArrayList and LinkedList:

In Java both ArrayList and LinkedList are part of the java.util package and implement the List interface, but they differ significantly in how they store and manage elements, leading to different performance characteristics for common operations.

1. Underlying Data Structure:-

* ArrayList:
  - Array-backed - It uses a dynamic array to store elements.
  - The array may need resizing when the list grows beyond its capacity.

* LinkedList:
  - Doubly Linked List - Each element is represented by a node containing a reference to both the previous and the next node, allowing traversal in both directions.

2. Time Complexities for Operations:-

Operation	ArrayList	LinkedList
Access (get)	O(1)	O(n)
Insertion (end)	O(1)	O(1)
Insertion (index)	O(n)	O(n)
Deletion (end)	O(1)	O(1)
Deletion (index)	O(n)	O(n)
Search (contains)	O(n)	O(n)

3. When to use Each:

* Use ArrayList when -
  - You need fast random access to elements.
  - The list mostly involves appending elements and the number of insertions/deletions in the middle is minimal.
  - Memory overhead is a concern.

* Use LinkedList when -
  - You need frequent insertions/deletions at the beggining or on middle of the list.
  - You are implementing a queue or stack.
  - You don't need frequent access.
  - Memory overhead is less of a concern.

Answer to the question no. (8)

⊞ Designing CustomRandomGenerator class:

```
public class CustomRandomGenerator{
    public int myRand (int i, int n){
    int[] random = new int [n+1];
long CurrentTime = System.currentTimeMillis();
    int t = 1;
for (int i=1; i<=n; i++){
    random [i] = t+=2;
    }
int ran = (int)(currentTime*random [i])%1000;
    }
}
import java.util.Scanner;
public class RandNumber{
    public static void main (String [] args){
Scanner obj = new Scanner (System.in);
    int n = obj.nextInt();
CustomRandomGenerator obj2 = new CustomRandomGenerator();
    for (int i = 1; i<=n; i++){
        int num = obj2.myRand(i, n);
      if(num<0)
    System.out.println(-1*num);
else
    System.out.println (num);
    }
}  }
```

Answer to the question no. (19)

Multithreading in Java:

Multithreading in Java allows multiple threads to run concurrently within a single program, enabling better utilization of system resources and improving performance for tasks that can be done in parallel. Java provides built-in support for multithreading using the Thread class and the Runnable interface.

1. Thread class vs Runnable Interface:

* Thread class:

i) The Thread class is part of java.lang package and represents an individual thread of execution.

ii) It is a concrete class that provides methods to create and manage threads such as start(), run(), sleep() etc.

iii) A class can extend Thread and override run() method to define the task for the thread.

```
class MyThread extends Thread {
    public void run(){
  System.out.println("Thread running");
    }
}
public class Main {
    public static void main(String[] args){
  MyThread thread = new MyThread();
    thread.start();
    }
}
```

* Runnable Interface:

i) The Runnable interface is a functional interface that defines a single method run().

ii) It allows classes to implement the Runnable interface instead of extending Thread, making it more flexible.

iii) By implementing Runnable, you can pass an instance of it to a Thread object to execute it.

```
class MyRunnable implements Runnable{
        public void run(){
    System.out.println("Thread running");
        }
}
public class Main{
    public static void main(String[] args){
  MyRunnable task = new MyRunnable();
   Thread thread = new Thread(task);
      thread.start();
    }
}
```

* Key Differences:

i) The thread class is used to create a new thread, while the Runnable interface is used to define the task that the thread should execute.

ii) Runnable is preferred if you need to extend another class. Since Java only supports single inheritance, while Threads requires extending the Thread class, preventing you from extending another class.

2. Potential issues with Multithreading:

i) Race condition: When two or more threads access shared data concurrently, and the final outcome depends on the non-deterministic order of execution.

ii) Deadlock: Occurs when two or more threads are blocked forever, waiting for each other to release resources.

iii) Starvation: A situation where a thread perpetually denied access to resources, often because other threads are monopolizing them.

iv) Livelock: Similar to deadlock, but instead of being stuck waiting, the threads continue to execute but make no real progress.

3. Thread Safety and the Synchronized Keyword:

The Synchronized keyword in Java is used to ensure that only one thread at a time can access a critical section of code that manipulates shared resources. It ensures that the code block or method it is applied to is thread-safe preventing race conditions.

Synchronized Methods: Using Synchronized keyword on a method ensures that only one thread can execute that method at any given time.

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

4. Deadlock Example and Prevention:

Deadlock occurs when two or more threads are waiting for each other to release a resource resulting in a circular dependency.

Deadlock Example:

```
class A {
  synchronized void methodA(B b){
System.out.println("Thread 1: Holding lock on A");
try{Thread.sleep(100);} catch{(InterruptedException e){}
    b.last();
}
```

```
    synchronized void last(){
     System.out.println("In A's last method");
      }
   }
class B{
     synchronized void methodB(A a){
System.out.println("Thread 2: Holding on B");
try{Thread.sleep(100);} catch{(InterruptedException e){}
  a.last();
}
synchronized void last(){
   System.out.println("In B's last method");
  }
}
public class DeadlockDemo{
       public static void main (String[] args){
    A a = new A();  B b = new B();
   Thread t1 = new Thread(()->a.methodA(b));
   Thread t2 = new Thread(()->b.methodB(a));
    t1.start();
    t2.start();
   }
}
```

Answer to the question no. (20)

Exception Handling Structure:

1. Use try, catch, finally blocks to handle exceptions.
2. try: Code that may throw an exception.
3. catch: Handles specific exceptions.
4. finally: Executes regardless of exception occurrence.

Checked vs Unchecked Exceptions:

1. Checked Exceptions: Checked at compile time (e.g. IOException)
   Must be handled or declared with throws.
2. Unchecked Exceptions: Runtime errors (e.g. NullPointerException)
   Not required to be handled.

Creating Custom Exceptions:

i) Extend Exception for checked exceptions.
ii) Extend RuntimeException for unchecked exceptions.

Throwing Exceptions:

i) Use throw to explicitly throw an exception.
ii) Use throws in method signature or propagate exceptions.

```
class MyException extends Exception{
    public MyException (String message){ super(message);}
}
public void myMethod() throws myException{
    throw new MyException ("Custom Exception");
}
```

4. Resource Cleanup:

i) finally block ensures resources are cleaned up, even if an exception occurs.

ii) Try-with-resources (Java 7+) automatically closes resources that implements AutoClosable:

```
try(FileReader fr = new FileReader("file.txt")){
} catch(IOException e){
    e.printStackTrace();
}
// no need for finally block.
```

Answer to the question no. (21)

Interfaces with Default Methods (Java 8+):

i) Interfaces can now have default methods with implementation since Java 8.

ii) This allows interfaces to evolve without breaking existing implementations.

iii) However, interfaces still cannot have state (fields) unlike abstract classes.

Abstract class vs Interfaces:

i) Abstract classes can have state (fields), constructors and fully implemented methods.

ii) Interfaces can only have constants, abstract methods and default/static methods (no state).

Can an abstract class Implement an Interface?

* Yes, an abstract class can implement an interface.
* If the interface provides a default method, the abstract class can:

1. Inherit the default method as-is.
2. Override the default method with its own implementation.
3. Redeclare the method as abstract, forcing subclasses

to implement it.

⊞ Conflicting Method Implementations:

i) If both the interface and the abstract class provide implementations for the same method:

• The abstract class's implementation takes precedence over the interface's default method.

ii) If a subclass inherits conflicting default methods from multiple interfaces, it must explicitly override the method to resolve the conflict.

Answer to the question no. (22)

⊞ HashMap:

• Internal Data Structure: Uses a hash table where elements are stored in key-value pairs.

• Time Complexities:

1. Insertion - O(1)
2. Lookup - O(1)
3. Deletion - O(1)

Ordering: Unordered. The elements have no predictable order, as they are based on the hash function of the keys.

TreeMap:

- Internal Data Structure - Uses a red black tree.
- Time Complexities:
  1. Insertion - O(log n)
  2. Lookup - O(log n)
  3. Deletion - O(log n)
- Ordering - Sorted by the natural ordering of the keys or by a comparator if provided. The keys are stored in ascending order.

LinkedHashMap:

- Internal Data Structure - Uses a hash table and a doubly linked list.
- Time Complexities:
  1. Insertion - O(1)
  2. Lookup - O(1)
  3. Deletion - O(1)
- Ordering - Maintains insertion order by default. If configured with access order, it maintains the order in which entries were last accessed.

Summary of Ordering in TreeMap vs HashMap:

1. TreeMap - Sorted order of keys.
2. HashMap - No specific order, it is unordered and based on the hash values of the keys.

When to Use Each:

i) HashMap - For fast, unordered key-value mappings.

ii) TreeMap - When you need the keys to be sorted or for range queries.

iii) LinkedHashMap - When you need to maintain the order of insertion or the order of access.

Answer to the question no. (23)

In Java, polymorphism refers to the ability of an object to take on many forms, allowing for method overloading and overriding. The concept ties into binding, which is how methods are resolved at runtime or compile time.

Static Binding:

Static binding occurs at compile time, meaning that the method called is resolved based on the reference type not the object type. This happens for methods that are not overriden, like private, final or static methods.

Example:

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

Dynamic Binding:

Dynamic binding occurs at runtime, meaning that method resolution is based on the object type not the reference type. This happens for overriden methods.

Example:

```
class Animal {
    void sound(){
System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal.{
    @override
    void sound(){
system.out.println("Dog barks");
    }
}
public class Test{
    public static void main (String[] args){
Animal obj = new Dog();
    obj.sound();
}   }
```

IT-23038

Answer to the question no.(24)

Advantages of ExecutorService over Manual Thread Management

1. Thread Pooling - ExecutorService uses a thread pool to a set of worker threads, reducing the overhead of creating new threads for each task.
2. Task Management - It allows easier management of tasks with less manual control.
3. Resource Efficiency - ExecutorService reuses threads reducing the number of threads created and improving performance.

submit() vs execute() in ExecutorService:

1. execute(): Takes a Runnable task and returns void. It is used for tasks that don't return a result or throw an exception.

2. submit(): Takes both Runnable and Callable tasks and returns a Future object, which allows you to track the result or exception of the task asynchronously.

Benefits of Callable over Runnable:

i) Return Value - Callable can return a result using call() method unlike Runnable which can't return any result.

ii) Exception Handling - Callable can throw exceptions which can be caught and handled by the Future object returned by submit().

Example:

```
import java.util.concurrent.*;
public class ExecutorServiceExample{
    public static void main(String[] args){
ExecutorService executor = Executors.newFixedThreadPool(3);
    executor.execute(()->{
        System.out.println("Task 1 : Runnable");
    });
  Callable<Integer> task = ()->{
        system.out.println("Task 2 : Callable");
        return 123;
    };
  Future<Integer> result = executor.submit(task);
try{ System.out.println("Task 2 Result:"+result.get()); }
catch(Exception e){
    e.printStackTrace();
```

IT-23038

Answer to the question no. (25)

We need to consider those facts when we are handling exception:

1. Identify the exception type (e.g., Arithmatic, NullPointer).
2. Ensure the program doesn't crash and provides meaningful feedback.
3. Use finally or try-with-resources to release resources like files or connections.
4. Log exceptions for debugging and monitoring.
5. Display clear error messages to user.

Java program that calculates the area of circle:

```
import java.util.Scanner;
class Circle {
    private double radious;
  public void setRadious(double radious){
      if(radious<0){
  throw new IllegalArgumentException("Radius can't be negativ
  }
  this.radious = radious;
}
public double getArea(){
    return Math.PI * radious * radious;
    }
}
```

```
public class CircleMain{
    public static void main(String[] args){
Circle circle = new Circle();
try{
    Scanner sc = new Scanner(System.in);
     double r = sc.nextDouble();
     circle.setRadious(r);
 System.out.println("Area:"+circle.getArea());
} catch(IllegalArgumentException e){
   System.out.println(e);
   }
  }
}
```

Answer to the question no.(26)

In Java, there are two ways to create a thread:

1. By extending the Thread class.
2. By implementing the Runnable interface.

Code for extending Thread class:

```
class MyThread extends Thread {
    @override
    public void run() {
        for (int i=1; i<=5; i++){
            System.out.println(i);
        }
    }
public class ThreadMain {
    public static void main (String[] args) {
        MyThread thr = new MyThread();
            thr.start();
        }
    }
```

code for Implementing the Runnable interface:

```
class MyThread2 implements Runnable {
    @override
    public void run(){
        for (int i = 1; i<=5; i++){
            System.out.println(i);
        }
    }
public class ThreadMain {
    public static void main (String[] args){
        MyThread2 obj = new MyThread2;
        Thread thr = new Thread(obj);
        thr.start();
    }
}
```

IT-23038

Answer to the question no. (7)

```
interface Edible{
    String HowToEat();
}
abstract class Animal{
    abstract String sound();
}
class Tiger extends Animal{
    @override
    public String sound(){
        return "Tiger roars";
    }
}
class Chiken extends Animal implements Edible{
    @override
    public String sound(){
        return "Chiken cluks";
    }
    @override
    public String HowToEat(){
        return "Fry it to make chiken curry";
    }
}
abstract class Fruit implements Edible{}
```

```
class Orange extends Fruit{
    @override
public String HowToEat(){
    return "Peel and eat";
    }
}
class Apple extends Fruit{
    @override
public String HowToEat(){
    return "Make apple juice or eat raw";
    }
}
public class InterfaceMain{
    public static void main(String[]args){
     Animal tiger = new Tiger();
     Chicken chiken = new Chicken();
System.out.println(tiger.sound());
System.out.println(chicken.sound());
system.out.println(chicken.HowToEat());

     Orange orange = new Orange();
     Apple apple = new Apple();
System.out.println(orange.HowToEat());
System.out.println(apple.HowToEat());
    }
}
```

IT-23038

Answer to the question no. (28)

Java's garbage collection automatically reclaims unused memory to avoid leaks, running in the background.

Types of Garbage Collectors:

1. Serial GC - Single-threaded, suitable for small apps, but can cause pauses.
2. Parallel GC - Multi-threaded, better for multi-core systems, still causes pauses.
3. CMS - Concurrent work to reduce pauses, but can lead to fragmentation.
4. G1 GC - Efficient for large apps, reduces pauses, but more complex and needs tuning.

Issues in Large Apps:

1. Stop-the world pauses - Can interrupt app threads, causing latency.
2. Memory fragmentation - Can reduce memory efficiency.
3. Tuning - Optimizing GC for large apps can be complex.

finalize() Method: finalize() allows cleanup before GC but is unreliable for memory management and can delay memory release.

IT-23038

Answer to the question no. 29)

```
import java.io.*;
import java.util.*;
public class SumNaturalNumbers{
    public static void main(String[]args)throws Exception{
    Scanner sc = new Scanner(System.in);
    Scanner sc = new Scanner(new File("input.txt"));
       int highest = Integer.MIN_VALUE;
    while(sc.hasNext()){
          int num = sc.nextInt();
       if(num>highest){
          highest = num;
       }
    }
    sc.close();
int sum = highest*(highest+1)/2;
 PrintWriter writer = new PrintWriter(new FileWriter(output.txt,
       writer.print(sum+",");
       writer.close();
    }
}
```

Answer to the question no. (30)

IT-23038

```
import java.util.Scanner;
public class DivisionReminder{
    public static void main(String[] args){
     Scanner sc = new Scanner(System.in);
System.out.print("Enter the size of the array1 greater than 20");
     int n = sc.nextInt();
     if (n<=20){
System.out.println("The size of the array1 should be greater than
20");
     return;
}
int[] array1 = new int[n];
System.out.println("Enter"+n+" elements for array1:");
    for(int i=0; i<n; i++){
       array1[i] = sc.nextInt();
    }
 int m= (int) Math.ceil((double)n/10);
     int[] array2 = new int[m];
 System.out.println("Enter:"+m+" elements for array2:");
       for(int i=0; i<m; i++){
          array2[i] = sc.nextInt();
       }
 System.out.println("Array1: ");
    for(int i : array1){
      System.out.print(i+" ");
  }
```

```
        System.out.println();
    System.out.println("Array2:");
        for(int i: array2){
     System.out.print(i + " ");
        }
     System.out.println();
    for(int i=0; i<array1.length; i++){
        for(int j=0; j<array2.length; j++){
          int quotient = array1[i]/array2[j];
          int reminder = array1[i]%.array2[j];
System.out.println("Element"+array1[i]+"divided by"+array2[j]+
                  "gives: Quotient(Ceil)="+quotient+"Reminder="+reminder);
                }
            }
        sc.close();

}
```

IT-23038

Answer to the question no. (31)

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class CurrentDateTime{
    public static void main(String[] args){
  SimpleDateFormat sdf = new SimpleDateFormat("yy-MM-dd, HH:
      mm:ss");
   Date now = new Date();
  System.out.println("Current date and time:"+sdf.format(now));
  }
}
```

Answer to the question no. (32)

```
public class CounterClass{
    private static int instanceCount = 0;
   public CounterClass(){
        instanceCount++;
   if (instanceCount > 50){
        instanceCount = 0;
   System.out.println("Object count exceeded 50");
   }
}
 public static int getInstanceCount(){
   return instanceCount;
```

```
public static void main (String [] args) {
    for (int i = 0; i<100; i++){
        new CounterClass ();
System.out.println("Instance count:"+CounterClass.getInstanceCount()
```

IT-23038

Answer to the question no. (33)

```
public class Main{
    public static void main (String[] args){
  int x = findExtreme("Smallest", 5, 2, 9, 1);
  int y = findExtreme("largest", 8, 3, 10, 4);
  System.out.println("Smallest:"+x);
  System.out.println("Largest:"+y);
}
public static int findExtreme(String type, int.... numbers){
      if(type.equals("Smallest")){
        int min = numbers[0];
      for(int num: numbers){
          if(num<min){
            min = num;
          }
      }
    return min;
  }
  else if(type.equals("largest)){
        int max = numbers[0];
      for(int num: numbers){
          if(num>max){
            max = num;
          }
      }
    return max;
  }
  return 0;
}
```

IT-23038

Answer to the question no. (34)

Output:

true // s1.equals(s2): content is same.
false // s1==s2 : different objects.
true // s1 = s3 : same object in memory.

Answer to the question no. (36)

```
interface Alpha{
    void method1();
    void method2();
}
interface Beta{
    void method3();
    void method4();
}
abstract class AbstractBase implements Alpha{
    abstract void method5();
}
class FinalClass extends AbstractBase implements Beta{
    @override
    public void method1(){
    System.out.println("Method 1 from Alpha");
}
    @override
    public void method2(){
    System.out.println("Method 2 from Alpha");
}
```

```
    @Override
    public void method3(){
    System.out.println("Method 3 from Beta");
  }
    @Override
  void method5(){
  System.out.println("Method 5 from AbstractBase");
  }
public class Main{
    public static void main(String[]args){
FinalClass finalClass = new FinalClass();
      finalClass.method1();
      finalClass.method2();
      finalClass.method3();
      finalClass.method4();
      finalClass.method5();
    }
}
```

IT-23038

Answer to the question no.(37)

Two ways to solve the problem:

1.
```
public class Z implements X, Y {
    @Override
  public void show() {
    System.out.println("Z's show");
     X.super.show();
     Y.super.show();
    }
}
```

2.
```
public class Z implements X, Y {
    @Override
   public void show(){
     X.super.show();
     Y.super.show();
    }
}
```

IT-23038

Answer to the question no. (38)

```
public class SigletonExample{
    private static SingletonExample instance;
private SingletonExample(){
}
public static SingletonExample getInstance(){
    if (instance == null){
     synchronized (SingletonExample.class){
       if (instance == null){
          instance = new SingletonExample();
        }
      }
    }
 return instance;
}
public void displayMesgage(){
   System.out.println("Singleton instance is working");
  }
}
```

IT-23038

Answer to the question no. (30)

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

class InsufficientFundsException extends Exception {
    public InsufficientFundsException (String message){
        super (message);
    }
}

class Wallet {
    private double balance;
public Wallet (double initialBalance){
    this.balance = initialBalance;
}

public void addFunds (double amount) throws InvalidAmountException{
  if (amount<0){
  throw new InvalidAmountException ("Amount cannot be negative");
  }
  balance+= amount;

}
```

```
public double getBalance(){
    return balance;
  }
}
public void spend(double amount) throws InvalidAmountException,
InsufficientFundsException{
  if(amount<0){
throw new InvalidAmountException("Amount cannot be negative");
  }
  if(balance<amount){
throw new InsufficientFundsException("Insufficient balance");
}
balance-=amount;
}
public double getBalance(){
    return balance;
  }
}
public class Main{
  public static void main(String[]args){
    Wallet wallet = new Wallet(100.0);
```

```
try{
  wallet.addFunds(-50);
} catch(InvalidAmountException e){
  System.out.println(e.getMessage());
}
try{
  wallet.spend(200);
} catch(InsufficientFundsException e){
System.out.println(e.getMessage());
}
try{
  wallet.spend(30);
System.out.println("Remaining balance:"+wallet.getBalance());
} catch(Exception e){
  System.out.println(e.getMessage());
  }
 }
}
```

IT-23038

Answer to the question no. (40)

```
public class Calculator {
    public static void main (String [] args) {
try {
    if(args.length != 2){
throw new IllegalArgumentException("Please provide two integers");
}
int num1 = Integer.parseInt(args [0]);
int num2 = Integer.parseInt (args [1]);
System.out.println ("Sum:" + (num1+num2));
System.out.println("Difference:"+(num1-num2));
System.out.println("Product:"+(num1*num2));
System.out.println("Quotient:"+(num1/num2));
} catch (NumberFormatException e){
  System.out.println("Invalid input");
} catch (ArithmaticException e){
  System.out.println("Invalid operation: Division by zero");
} catch (IllegalArgumentException e) {
   System.out.println(e.getMessage());
   }
}}
```