NAME: Md. Naiem

ID: IT-23025

1)

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

public class NumberProcessing {
    public static void main(String[] args){
        String inputFileName= "input.txt";
        String outputFileName= "output.txt";
List<Integer>numbers= new ArrayList<>();
    try(Scanner scanner= new Scanner(new File(inputFileName))) {
        if(scanner.hasNextLine()){
            String line= scanner.nextLine();
            String[] newstrings= line.split(",");
            for(String numstr : numstrings) {
                numbers.add(Integer.parseInt(numstr.trim()));
            }
        }
    } catch(FileNotFoundException e) {
        System.out.println("Error. The file "+inputFileName+" was not found.");
        return;
    }
```

```
catch (NumberFormatException e) {
System.out.println("Error: The file ~~+~~ one or more not valid integers");
  return;
}
int highestNumber = collections.max(numbers);
int sum = highestNumber * (highestNumber+1)/2;
try (PrintWriter writer = new PrintWriter(new FileWriter
                                (outputFileName))) {
  writer.println(highestNumber + ", " + sum);
} catch (IoException e) {
    System.out.println("Error: There was an issue..output file");
  }
  System.out.println("Processing complete " + outputFileName);
 }
}
```

NAME: Md. Naiem

IT : IT-23025

2|

In Java, the keywords static and final are used to define fields and methods with different characteristics

1: static Fields and methods:

Static Fields: A static fields is associated with the class itself, rather than with instances (objects) of the class. There is only one copy of the static field shared by all instances of the class.

- static fields are accessed using the class name.
- A static field can be modified, unless it is also final.

Static method: A static method belongs to the class and can be called without creating an instance of the class. static methods can only directly access fields and methods.

static methods are typically used for utility functions:

NAME: Md. Naiem

ID: IT-23025

Example:

```
class MyClass {
    static int count=0;
    static void increment() {
        count++;
    }
}
public class Test {
public static void main(String[] args) {
MyClass.increment();
System.out.println(MyClass.count);
}
}
```

2. Final Fields and Methods.

Final Fields: A final field can only be assigned a value once, either during it's initialization or within a constructor (if the field is an instance field). One assigned, it's value can not be changed.

- A final field can be used for constants.
- Final fields can be either instance-level or static

Final methods: A final method can not be overridden by any class.

NAME: Md. Naiem
ID: TT-23025

- This is useful when you want to make sure that the behaviour of the method remains the same in all subclasses.

key difference:

Feature	Static fields/methods	Final fields/method
Association	Belongs to the class, not an instance	can be instance-level or class level
Access	can be accessed via the class name or an object	cannot be modified once initialized
modification	~~can be accessed~~ Static fields can be changed, except if they are final	~~can not be~~ Final fields can't be reassigned after initialization
inheritance	Static methods are inherited but can not be overridden	Final method's can not be overridden.

What will happen if access a static field or method with an object:

It will work, but it's not recommended because it may lead to confusion, as it might seem like accessing instance-specific data, even though the field or method is shared across all instances of the class.

- The compiler can access the static members using the object reference, but using the class name is the cleaner and recommended way

Example of incorrect but compilable usage:

```
MyClass obj = new MyClass();
obj.increment();
```

correct way:

```
MyClass.increment();
```

NAME: Md. Naiem
ID : IT-23025

[3]

```
import .java.util.Scanner;
public class FactorialFinder{
 Private static final int[] factorials= new int[10];
 Private static void PrecomputeFactorials() {
    factorials[0]=1;
     for(int i=0; i<10;i++){
        factorials[i]= i*factorials[i-1];
        }
 }
Private static boolean isFactorial(int num) {
 int sum=0, temp=num;
 While(temp>0){
 int digit= temp%10;
 sum+= Factorials[digit];
 temp/= 10;
 }
 return sum== num;
 }
```

```
Public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
  System.out.print("Enter the lower bound of the range: ");
  int lowerBound = scanner.nextInt();
  System.out.print("Enter the upper bound range) ");
  int upperBound = scanner.nextInt();
  scanner.close();
  PerComputeFactorials();
  System.out.println("Factorial num in range");
  boolean found = false;
for(int i = lowerBound; i <= upperBound; i++){
  if(isFactorion)) {
    System.out.print(i + " ");
    found = true;
  }
}
  if(!found) {
  system.out.println("None found");
  }
}
}
```

NAME: Md. Naiem

ID : IT-23025

4| The differences among class, local and instance variable:

Feature	class variable (static)	instance variable	Local variable
Declared in	class, outside methods	class, outside method	inside methods or blocks
keyword	static	No keyword	No keyword
scope	shared across all objects	Exist per object	only in the method/block
lifetime	Exists as long as class is loaded	Exists as long as object exists	until the method ends
memory	stored in the class area	stored in the heap (inside the object)	stored in the stack
Accessed via	class name or object (className.var)	only through an object	only inside the method/ block.

Significance of this keyword in Java:

1. Referring of instance variable
2. Invoking instance methods
3. Returning the current class instance
4. Passing this an argument
5. Using this to call another constructor
6. Shared across all instance (class variable)

Md. Naiem

IT-23025

5

```
Public class Arraysum{

Public static int calculateSum(int[] arr){

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

Md. Naiem
IT-23025

// 6 //

Access modifiers in Java are keywords that define the scope and visibility of classes, methods, and variables. They control how other parts of the program can access a particular class, method, or variable.

Java has four access modifiers:

1. Public - Accessible from anywhere
2. Private - Accessible only with the same class.
3. Protected - Accessible within the same package and subclasses.
4. default (no modifier) - Accessible only within the same package.

* Comparison

modifier	same class	same package subclass	subclass (diff. package)	outside package
public	yes	yes	yes	yes
private	yes	no	no	no
protected	yes	yes	yes (only in subclass)	no
default	yes	yes	no	no

There are three main types of variables:

1. Instance variable (object-level)
2. Static variable (class-level)
3. Local variables (method-level)

Instance variable:

- Declare inside a class, but outside any method.
- Each object gets it's own copy of the variable.

Example:

```
class Person {
    String name;
    public static void main(String[] args) {
    Person P1 = new Person();
    P1.name = "Alice";
    Person P2 = new Person();
    P2.name = "Bob";
    System.out.println(P1.name);
    System.out.println(P2.name);
    }
}
```

Static variable

- Declare using static variable
- shared among all objects of the class.

Example:

```
class car{
  static int wheels=4;
  public static void main(String[] args){
    car c1= new car();
    car c2= new car();
  System.out.println(c1.wheels); //output 4
  System.out.println(c2.wheels); //output 4
  c1.wheels=6; //changing static variable.
  System.out.println(c2.wheels); //output 6
  }
}
```

Md. Naiem
IT-23025

Local variable: (method level)

- Declare inside a method
- only available inside the method

Example

```
class Example{
 void show(){
     int num=10;
  System.out.println(num);
 }
 Public static void main(String[]args){
      Example obj = new Example()
      obj.show();
 } }
```

Final variables (constant values)

- Declare using final keyword.
- Cannot be changed after assignment

```
class FinalExample{

 final int MAX_AGE =100;
  public static void main (String [] args){
    FinalExample obj=new FinalExample();
    System.out.println(obj.MAX_AGE);
  }
}
```

Md. Naiem
11-23025

7

```
import java.util.Scanner;
public class QuadraticEquation {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
 System.out.Print("Enter coefficients a, b, and c);
 int a= scanner.nextInt();
int b = scanner.nextInt();
int c = scanner.nextInt();
double discriminant= b*b-4*a*c;
if(discriminant<0) {
 System.out.println("no real roots");
  } else {
  double root1=(-b+Math.sqrt(discriminant)) /(2.0*a);
  double root2=(-b-Math.sqrt(discriminant) /2.0*a);
 double smallestRoot=Double.MAX.VALUE;
 if(root1>0) smallestRoot= root1;
 if(root2>0 && root2<smallestRoot) smallestRoot= root2;
 if(smallestRoot!=Double.MAX_VALUE) {
   System.out.println("the smallest positive roots: "+smallestRoot);
} else {System.out.println("no positive roots"); }
   } scanner.close();
  }
```

Md. Naiem
IT-23025

8

Determine Letters, Whitespace and Digits :

```
import java.util.Scanner;
Public class CharacterChecker {
 public static void main (String [] args) {
    Scanner scanner= new scanner(System.in);
 System.out.Print("Enter a character: ");
  char ch= scanner.next().charAt(0);
 if(character.isLetter(ch)) {
   system.out.println(ch+ "is a letter");
    }
 else if (character.isDigit(ch)) {
    System.out.println(ch+ "is a Digit");
    }
 else if(Character.isWhitespace(ch)) {
    system.out.println("ch is a whitespace character");
       }
 else {
     system.out.println(ch+ "is a special character");
    }
   scanner.close();
  }
}
```

How to pass an Array to a function

```
Public class A {
    Public static void PrintArray(int [] numbers){
    System.out.Print("Array elements: ");
    for(int num: numbers) {
     system.out.Print(num+ " ");
    }
    System.out.Println();
  }
Public static void main(String[] args) {
    int[] myArray= {2, 3, 2};
    PrintArray(myArray);
  }
}
```

MD. Naiem
IT23025

Q1

solve

Method Overridding in Java

Method overridding ~~helps~~ happens when a subclass provides a new version of a method that already exists in its Superclass. The method in the Subclass must have:

- the same name as the Superclass method.
- the same parameters.
- the same returntype.

How does it work?

When you call a method on a object of the subclass, Java will execute the overridden method in the sub-class, not the one in the Superclass. This allows Java to support runtime polymorphism.

Example:

```
class Animal {
    void sound () {
    System. out. println ("Animal makes a sound");
    }
}
```

```
class Dog extends Animal {
  Void Sound () {
    System. out. println (" Dog barks ");
  }
}
public class Main { }
  public static void main (String [] args) {
    Animal mypet = new Dog ();
    mypet.sound ();
  }
}
```

Using

☐ ^ Super to call the Superclass method

The Super keyword allows the Subclass to call a method from the Superclass. This is useful if you want to add to the original method instead of completely replacing it.

Example:

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

potential issue when overriding methods.

1. Constructor cannot be overridden

- Constructor are not inherited, so they cannot be overridden.
- If a super class only has a parameterized constructor, the subclass must call it explicity using super().

Example

```
Class Animal {
Animal (String name) {
System. out. Println ("Animel :" +name);
}
}
Class Dog extends Animal {
    Dog ( ) {
  Super (" Buddy ");
  System. out. Println (" Dog is created ");
}
}  public class Main {
public static void main(String[] args) {
```

```
    Dog myDog = new Dog();
}
}
```

2. Final methods cannot be overridden

- If a method is marked final in the ~~subclass~~ Super class, it cannot be changed in the subclass.

Example:

```
class Animal {
    final void sleep() {
        System.out.println("Animal sleeps");
    }
}
    class Dog extends Animal {

    }
```

3] Static methods are not overridden

- static method belong to the class, not the instance, so they do not support overriding.

Example

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

Access Modifiers Rule

- Overridden method cannot have a more restrictive access level than the original.

Example:

```
Class Animal {
  Protected void eat ( ) { }
}
class Dog extends Animal {
   public void eat() { }
}
```

Md. Naiem
11-23025

10

Key differences..

Feature	static members	Non-static members
Belongs to	class	object
Accessed by	class name	object
Memory Allocation	once for all object	separate for each object
Example	static int x,	int x

Example static variable and method

```
class StaticExample {
  static int c = 0;
  static void show(){
   System.out.println("count: "+c);
  }
Public static void main (String[] args) {
     c = 10;
   StaticExample.show();
  }
}
```

Example: Non static variable and method.

```
class NonStat {
    int age = 25;
    void showAge() {
        System.out.println("Age: " + age);
    }
public static void main(String[] args) {
    NonStat obj = new NonStat();
    obj.showAge();
    }
}
```

Pallindrome checker

```
import java.util.Scanner;
public class PallindromeChecker{
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a number:");
        String input = scanner.nextInt();
```

```
if(isPallindrome(input)) {
  System.out.println("It is Pallindrome.");
   }else{
   System.out.println("not Pallindrome:");
   }
   scanner close();
  }
public static boolean isPallindrome(String str){
  int left=0, right=str.length()-1;
  while(left<right){
       if(str.charAt(left)!=str.charAt(right)){
       return false;
      }left++;
       right--;
     }return true;
    }
  }
```

11|

MD. Naiem
IT23025

Abstraction

Abstraction is a concept in object-oriented Programming (OOP) that hides the implementation details and only shows the essential features of an object. It is achieved using abstract classes and interfaces.

Example of Abstraction in Java

```
abstract class vehicle {
  abstract void start();
  void fuel() {
   System.out.println("Fueling the vehicle...");
  }
}
class car extends vehicle {
  void start() {
  System.out.println("Car is starting with a key...");
  }
}
public class Abstraction_Example {
  public static void main(String[] args) {
  vehicle mycar = new car();
```

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

2| Encapsulation

Encapsulation is the process of binding data and methods together with in a class and restricting direct access to some details. It is implemented using private access modifiers and getter & setter methods.

Example:

```
class BankAccount {
    private double balance;
    private void deposit(double amount) {
        if(amount>0){
            balance += amount;
        }
    }public double getBalance() {
        return balance;
    }
} public class EncapsulationExample {
    public static void main(String[] args) {
```

```
BankAccount account = new BankAccount();

account.deposit(1000);
System.out.println("Balance: " account.getBalance());
}
}
```

Differences Between Abstract class and interface

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

12|

MD. Naiem
IT23025

Solve

```
Class Baseclass{
  Void PrintResult(String result) {
  System.out.println(result);
  }
}
Class Sumclass extends Baseclass {
  Void computeSum() {
    double sum=0;
    for(double i=1.0 i>, 0.1; i-= 0.1) {
        sum+=i;
    }
     PrintResult("Sum of the series: "+sum);
    }
  }
  Class DivisionMultipleClass extends Baseclass {
   int gcd(int a, int b) {
     while(b!=0) {
      int temp=b;
       b= a%b;
```

```
        a = temp;
    }
    return a;
}
int lcm(int a, int b) {
    return (a*b) / gcd(a,b);
}
void computeGCDLCM(int a, int b) {
    PrintResult("GCD of " + a + " and " + b + " is: " + gcd(a,
PrintResult("lcm of " + a + " and " + b + " is: " + lcm(a,b))
    }
}
Class NumberConversionClass extends BaseClass{
    void convertNumber(int num) {
    ~~Print result~~
    PrintResult("Decimal: " + num);
    PrintResult("Binary: " + Integer.toBinaryString(num));
    PrintResult("Octal: " + Integer.toOctalString(num));
    PrintResult("Hexadecimal: " + Integer.toHexString(num));
    }
}
```

```
Class CustomPrintClass extends BaseClass {
 Void Pr(String message) {
   PrintResult("[Custom Print]"+message);
 }
}

Public class MainClass{
 Public Static Void main(String[] args) {
  SumClass SumObj = new SumClass();
  SumObj.computeSum();
  DivisorMultipleClass divMulObj = new DivisorMultipleClass();
   divMulObj.computeGCDLCM(24,36);
  Number_ConversionClass numConvObj = new NumberConversionClass()
  numConObj.convertNumber(255);
 CustomPrintClass customPrintObj = new CustomPrintClass();
  customPrintObj.Pr("This is a formatted message.");
   }
 }
```

MD Naiem
IT23025

13] Java Program that implements the UML - diagram :

Solve

```
import java.util.Date;
Class GeometricObject {
 Private string color;
 Private boolean filled;
 Private Data date created;

 Public GeometricObject () {
 this.color = "white";
 this.filled = false;
 this.dateCreated = new Date ();
 }
 Public GeometricObject (String color, boolean filled) {
  this.color = color;
  this.filled = filled;
  this.dateCreated = new Date ();
 }
```

```
Public String getColor() {
    return color;
}
Public void setColor(String color) {
    this.color = color;
}
Public boolean isFilled() {
    return filled;
}
Public void setFilled(boolean filled) {
    this.filled = filled;
}
Public Date getDateCreated() {
    return dateCreated;
}
@Override
Public String toString() {
return "Color: " + color + ", " Filled: " + filled + ",
" Created on: " + dateCreated;
}
}
```

```
class circle extends Geometric_object {
    Private double radius;
  Public circle () {
     this.radius= 1.0;
  }
 public circle (double radious, String color, boolean filled) {
   super (color, filled);
     this. radius = radius;
  }
   public double getRadious () {
        return radius;
   }
  public void setRadious (double radious) {
      this. radious = radious;
  }
   public double getArea() {
    return Math. PI * radius * radius;
   }
  Public double getperimeter() {
   return 2* Math. PI * radius;
}
```

```
public double getDiameter() {
    return 2 * radius;
}
public void printCircle() {
    System.out.println("Circle: radius = " + radius);
  }
}

Class Rectangle extends GeometricObject {
    Private double width;
    Private double height;
  public Rectangle() {
      this.width = 1.0;
      this.height = 1.0;
    }
    public Rectangle(double width, double height) {
      this.width = width;
      this.height = height;
    }
```

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

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

```
Public Class TestGeometricObjects {
Public static void main (string [] args) {
Circle circle = new circle (5.0, "Red", true);
System.out.Println ("Circle :");
System.out.Println ("Radiy: " + circle.getRadiy ());
System.out.Println ("Area: " + circle.getArea ());
System.out.Println ("Perimeter: " + Circle.getPerimeter ());
System.out.Println ("Diameter: " + circle.getDiameter ());
System.out.println (circle.tostring ());

System.out.Println ("\n Rectangle:");
Rectangle rectangle = new Rectangle (4.0, 7.0, "Blue", false);
System.out.Println ("Width:" + rectangle.getWidth ());
System.out.Println ("Height:" + rectangle.getHeight ());
System.out.println ("Area:" + rectangle.getArea ());
System.out.println ("Perimeter:" + rectangle.getPerimeter ());
System.out.println (rectangle.toString ());
}
}
```

MD. Naiem
I123025

14] Significance of BigInteger

In Java, BigInteger is a class in the Java.math pakage used to handle arbitrarily large integers. It is significant because:

1. Handles large Numbers: Primitive data types like int, and long have limits (int up to $2^{31}-1$ and long up to $2^{63}-1$). BigInteger can store much larger values.

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

3. Useful in Cyptography: It is widely used in encryption algorithms where large numbers are required.

4. Immutable: Like, String, BigInteger objects are immutable, meaning operations create new objects instead of modifying the existing ones.

Java Program for Factorial using BigInteger

```
import java.math.BigInteger;
import java.util.Scanner;
public class FactorialBigInteger{
public static BigInteger factorial(int num){
BigInteger fact = BigInteger.ONE;
for(int i=2; i<=num; i++){
  fact = fact.multiply(BigInteger.valueOf(i));
}
return fact;
}
public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.Print("Enter a number: ");
    int number = Scanner.nextInt();
    BigInteger result = factorial(number);
    System.out.println("Factorial of " + number + "is: "
                       + result);
    scanner.close();
  }
}
```

MD. Naiem

IT23025

15| Comparison of Abstraction Classes and Interfaces in Java.

Abstract Class	Interface
1. A class that cannot be instantiated and may have both abstract and concreate methods.	1. A collection of abstract methods and default/static methods with no instence fields.
2. Can have abstract, concrete, static and final methods.	2. Can have abstract methods, default methods, static methods, but no constructor.
3. Can have instance variable with any access modifier.	3. cannot have constructor
4. A class can extend only one abstract class.	4. A class can implement multiple interfaces.
5. Can have any access modifier for methods and variables.	5. All methods are implicit public.

When to use an Abstract Class over an Interface

1. **Shared state**: If multiple related classes required common fields or methods with implementations, an abstract class is Preferred.

2. **Partial implementation**: If you want to provide some common functionality to subclasses while enforcing the implementation of specefic methods, use an abstract class.

3. **Performance considerations**: Calling methods from an abstract class is slightly faster than calling interface methods due to virtual table lookup optimizations.

4. **Version Compatibility**: Abstract classes Provide more flexibility in modifying the base class without beaking existing implementations.

Can a class implement Multiple Interfaces in Java?

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

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

Implication using multiple Interfaces

- **Code Resuability & Flexibility**: Since a class can implement multiple interfaces, it allows for greater modularity.
- **Conflict Resolution**: If two interfaces have methods with the same signature but different default implementations.

Code

```
interface X {
    default void show() {
        System.out.println("X's show");
    }
}
```

```
interface Y {
  default void show() {
  system.out.println("Y's show");
  }
}
Class MyClass implements X, Y{
   public void show () {
  System.out.println ("Resolved show method");
  }
}
```

MD Naiem
IT23025

16| Polymorphism in Java

Polymorphism in Java refers to the ability of an object to take multiple forms. It allows a single interface to be used for diffrent types, enabling code flexibility and resuability. polymorphism can be achived through method overriding and method overloading, but the most common from in Java is runtime polymorphism.

Dynamic method Dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime rather than at compile time. This allows Java to determine the actual method to be executed based on the runtime type of the object, not the reference type.

Example: polymorphism using Inheritance and method overriding :

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

```
class cat extends Animal {
  @Override
  void make sound ( ) {
    System.out.println ("cat meows");
  }
}
public class polymorphismExample {
  public static void main (String[] args) {
      Animal myAnimal;
      myAnimal = new Dog ( );
      myAnimal.makesound ( );
      myAnimal = new cat();
      myAnimal.makeSound ( );
    }
  }
```

Trade-offs Between using polymorphism and specific method calls

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

MD. Naiem
IT23025

17| Differencess Between Arraylist and Linkedlist in Java.

Arraylist	Linked List
1. O(1) (Direct indexing)	1. O(n) (sequential traversal
2. Amortized O(1) (Occasional resize)	2. O(1) (Direct append)
3. O(n) (Shifting required)	3. O(1) (If node reference is known, otherwise O(n) for traversal)
4. Less overhead (array only)	4. more overhead (extra node references)
5. search contains O(n)	5. search contains O(n)

⊞ When to use which ?

- Use Arraylist when :
  - You need fast random accesss (O(1) lookup).
  - ~~Interei~~ Insertions / delations infrequent.
  - Memory efficiency is a priority.

- Use Linked list when:

- Frequent insertions/deletions in the middle are required.
- Sequential access is preferred over random accesss.
- Memory overhead is not concern.

18] Random Generator

```
import java.util.Random;
import java.util.Arrays;

public class CustomRandomGenerator {
 Private static final int[] SEED_ARRAY={3,7,11,
 17,23};
 public static int myRand(int n){
   long timestamp=System.currentTimeMillis();
   int seed = SEED_ARRAY[(int)(timestamp % SEED_ARRAY.
              length)];
     return (int)((timestamp*seed)%n);
 }
```

```
public static int[] myRand (int n, int count) {
int[] randomNumbers = new int[count];
  for( int i =0; i<count ; i++) {
     randomNumbers[i] =myRand(n);
  }
  return randomNumbers;
}
  public static void main (string [] args) {
  System. out. Println ("Single random number: " +myRand(100)
  System. out. Println ("Multiple random numbers: " Arrays.tos
   (myRand (100, 5)));
}
}
```

MD. Naiem
IT23025

19| Multithreading in Java

Multithreading in java allows multiple threads to run concurrently, improving performance and responsiveness. Java provides built-in support through the Thread class and Runnable interface.

Thread class vs Runable Interface

1. Thread class - Extending Thread requires overriding the run() method but prevents extending other classes.
2. Runable Interface - Implementing Runnable allows better flexibility, enabling the class to extend other classes.

Example:

```
Class MyThread extends Thread {
 Public void run () { system.out. Println ("Thread
 using Thread class"); }
} }
```

```
class MyRunnable implements Runnable {
  public void run ( ) { system.out.Println ("Thread using
  Runnable interface") ; }
}
public class ThreadExample {
  public static void main (String [] args) {
    new MyThread ( ) . start ( ) ;
    new Thread (new MyRunnable ( )) . start ( ) ;
  }
}
```

potential Issue in Multithreading

- Race condition - Multiple threads modifying data - simultaneously, causing inconsistencies.
- Deadlocks - Two or more theads waiting for each other to release resources.
- Starvation - Low-Priority threads not getting CPU time.
- Livelock - Threads continously responding to each other without making progress.

Using Synchronised for Thread Safety

The Synchronized keyword Prevents multiple threads from accessing ~~strict~~ critical code simultaneously.

Example

```
class SharedResource {
  Private int count = 0;
  public Synchronized void increment() {count++;}
  public Synchronized int getcount() {return count;}
}
```

- Deadlock Scenario:

```
class Resource {
void methodA (Resource r) {
Synchronized (this) {
System.out.Println(Thread.currentThread().getName()+
" locked Resource 1");
Synchronized (r) {
```

```
        System.out.println(Thread.currentThread().getName() +
    " locked Resource 2");
    }
}
    }
}
```

Prevention:

- Lock Ordering - Always acquire locks in a fixed order.
- Try-Lock Mechanism - Use tryLock() from ReentrantLock instead of Synchronized.

Using ReentrantLock:

```
import java.util.concurrent.locks.*;

class SafeResource {
    Private final Lock lock1 = new ReentrantLock();
    Private final Lock lock2 = new ReentrantLock();
    Void safeMethod() {
     if (lock1.tryLock()) {
try {system.out.println("safe execution without deadlock.");}
```

```
        finally { lock2.unlock(); }
    }
    finally {lock1.unlock(); }
    }
  }
}
```

20|

MD. NAIEM
IT23025

Exception Handling in Java (short overview)

Exception handling in Java ensures that runtime errors donot disrupt program flow. It is managed using try, catch, finally, throw, and throws keywords.

⊞ Checked vs unchecked Exceptions

- Checked Exceptions: Checked at compile-time; must be handled using try-catch or declared with throws.
- Unchecked Exceptions: Occur at runtime; typically caused by Programming errors.

Creating & Throwing Custom Exceptions

- Extend Exception for checked exceptions or RuntimeException for unchecked exceptions.
- Use throw to explicitly throw an exception.

```
class CustomException extends Exception {
    public CustomException (String message)
    {
      Super (message); }
    }
throw new CustomException ("custom checked
    exception ");
```

Role of throw and throws

- throw : used to manualy throw an exception.
- throws : Declares exceptions a method might throw.

```
public void method () throws IOExceptions {
    throw new IOException ("Error occurred");
}
```

MD. Naiem
IT23025

~~Preventing~~

Preventing Resource Leaks

- Use finally Block: Ensures resources are closed.
- Try-with-Resources (AutoCloseable): Java 7+ automatically closes resources.

```
try (FileInputStream file = new FileInputStream("file.txt"))
{
} catch (IOException e) {
    e.printStackTrace();
}
```

MD. Naiem
IT-23025

21.

Interfaces in Java can have default methods staring from Java 8, and this feacher introduces some interesting distinctions when compared to abstract classes's.

Interfaces with defaut Methods :

- Interfaces can have both abstract methods and default methods.
- Default methods allow you to add functionally to an interface without breaking existing implementations.
- Since interfaces can have no state, they can only provide behavior or constants.

Abstract classes:

- Abstract classes can hold both state (instance variable) and behaviour (method).
- They can have constructors, non-static fields, and concrete methods.

Conflicting Method Implementations between abstract classes and Interfaces:

If both an abstract class and interface provide conflicting method implementations, the concrete class must explicity resolve the conflict by overriding the method.

example code :

```
interface A {
    default void hello() {
        System.out.println("Hellow from A");
    }
}

abstract class B implements A {
    @override
    public void hello() {
        System.out.println("Hellow from B");
    }
}

class C extends B {
    @override
    public void hello() {
        System.out.Println("Hellow from C");
    }
}
```

Naiem

(22) IT-23025

Feature	Hashmap	Tree map	LinkedHashMap
Internal Structure	Hash table	Red black tree	Hash Table + Doubly Linked List
Order of Elements	NO order guaranteed	sorted	Insertion order
Insertion time	O(1)	O(log n)	O(1)
Deletion time	O(1)	O(log n)	O(1)
When to use	Fast access with no order needed	when you need sorted keys or range queries	When you need insertion/access order

How does Tree map differ from HashMap in ordering

- Hasmap does not maintain any order of its keys The order in which you iterate over the keys or entries can be arbitrary.
- Treemap, on the other hand maintains a sorted order of the key based on their natural ordering or a comparator, meaning the elements will be traversed in ascending order by default.

NAME: Md. Naiem

Naiem
IT-23025

23

Static Binding and Dynamic Binding in Java and their Relation to polymorphism.

Static Binding : Occurs when the method call is resloved at compile-time. This happens when the method being invoked is determined based on the type of the reference. Static methods, private method, and final methods are examples of methods that are statically bound.

Dynamic Binding : Occurs when the method call is resolved at runtime, based on the actual object type, not the reference type. This is a key feature of runtime polymorphism. Methods overridden in subclasses are dynamically bound.

Naiem
IT-23025

Static binding Example :

a. animal.sound();

b. animal.bark();

24

Advantages of ExecutorService over Manual Thread Management:

1. Thread Pool management.

2. Automatice thread reuse.

3. Task scheduling.

4. Graceful Shutdown.

5. Error Handling and Monitoring.

Benefits of using callable over runnable.

Runnable:

*. runnable tasks are designed to perform an actio but do not return any result. It is deal when you simply want to excute some code without needing any feedback.

*. Runnable cannot throws checked exceptions directly; they must be wrapped in a try-catch block.

Callable:

*. Callable is similar Runnable but can return aresult. This is usefull when you need the outcome of the task or need to compute a value.

*. Callable can throw checked exceptions, which means you can propogate exceptions from the task directly.

Differences between submit() and excute() Methods.

⊛ excute method :

*. This method is used to submit a runnable task to the ExecutorService. The execute() method doesn't return any result any or feedback. it only signals that the task the task has been submitted for execution.

*. It doesn't provide a way to haddle exceptions thrown by the task.

submit method:

*. The submit method is more versatile than execute.

It allows you to submit both runnable and callable tasks to the ExecutorService.

- If the task throws an exception, it can be retrieved through the future object.

Naiem
IT-23041

25

code:

```
Class NegativeRadiusException extends Exception{
    public NegativeRadiusException(String message){
     super(message);
    }
}
public class circle{
        private double radious;
public void setRadius(double radius)throws NegativeRadius
                                        Exception{
if(radius<0){
throw new NegativeRadiusException("Radius cannot be
                                    negative");
}
  this.radius = radius;
}
public static void main(string[]args){
   circle circle = new circle(){
```

```
try{
    circle.setRadius(-5);
    System.out.println("Area of circle: "+ circle.calculate
                                    Area(); }
catch(NegativeRadiusException e){
  System.out.println("Error", + e.getmessage());
}

try {
    Circle.setRadious(10);
    System.out.println("Area of circle: " + circle.
                                    calculateArea());
}
catch(NegativeRadiousException e) {
    System.out.println("Error:+ e.getmesage
    }
  }
}
```

Naiem
IT-23025

26. There are two primary ways to create a thread:

1. By implementing the runnable interface.
2. By extending the thread. class.

Creating a thread using the runnable interface:

```
class MyRunnable implements Runnable {
 @Override
 public void run(){
 System.out.println("Thread is running Runnable
                    Interface!");
 }
}
public class RunnableExample{
public static void main(String[] args){
 MyRunnable myRunnable = new MyRunnabe();
 Thread thread = new Thread(myRunnable);
 thread.start();
 }
}
```

Creating a thread by extending the thread class:

```
class MyThread extends Thread {
@Override
public void run() {
  System.out.println("Thread is running by
                      extending Thread class!");
}
}
public class ThreadExample {
public static void main(String[] args) {
    MyThread mythread = new MyThread();
    myThread.start();
  }
}
```

Naiem
IT-23025

28||

Java's garbage collection (GC) is reponsible for automatically managing memory by reclaiming the memory that is no longer in use. Java uses automatic memory management, which helps developers focus on logic without having to manually allocate memory. The goal is to find and remove objects that are no loger referred by the program, freeing up memory to be reused.

phases of Garbage Collection:

1. Mark Phase: The GC identifies all objects that are reachable
2. Sweep phase: The GC removes objects that are not reachable (no references).

3. Compact Phase: The GC may move objects in memory to eliminate gaps, allowing more efficient memory usage.

Naiem
IT-23025

29

Code:

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

Public class HighestvalueAndSum{
    public static void main (String[] args){
        String inputFile = "input.txt";
        String outputFile = "output.txt";
int highestValue = Integer.MIN_VALUE;
    int sum = 0;
try(Scanner scanner = new Scanner(new File
                    inputFile))){
while (scaner.hasNextInt()){
int number = scanner.nextInt();
        sum += number;
if(number > highestValue){
        highestValue = number;
    }
}
```

NAME: Md. Naiem

```
catch(FileNotFoundException e){
System.out.Println("The input file was not found.");
  return;
}

try(PrintWriter writer = new PrintWriter(new File(outputFile))
{ writer.Println("Highest Value : " + highestValue);
  Writer.Println("Sum : " + sum);
system.out.println("Results written to " + outputFile);
}
 catch(FileNotFoundException e){
  System.out.println("Error writing to the
                      output file.");
  }
}
}
```

Naiem
IT-23025

30.11

CODE:

```
import java.utill.Scanner;
import java.utill.Arrays;

Public class ArrayDivision{
 public static void main(String[]args){
    Scanner sc = new Scanner(system.in);
     system.out.print("Enter the size of the first
        array (n>20): ");
    int n= sc.nextInt();

 if(n<=20){
   System.out.println("n must greater than 20");
   return;
 }

   int m = n/10;
   int [] array1 =new int[n];
   int array2 =new int[m];
   Systemout.Println("Enter elements of the first
      array(size "+n+ "): ");
   for(int i=0; i<n; i++){
```

Naiem
IT-23025

```
        array1[i] = sc.nextInt();
    }

    System.out.println("Result of dividing the first array
    by the second array:");
    for(int i=0; i<m; i++) {
    double divisionResult = (double) array1[i]/array2[i];
    int divisor = (int) Math.ceil(divisionResult);
    int remainder = array1[i] % array2[i];
    ~~int divisor = (int) Math.ceil~~
    System.out.println("For index "+i+": ");
    System.out.print("Divisor(roundup): "+divisor);
    System.out.println("Remainder: "+remainder);
    }
    sc.close();
    }
```

IT23025

31

current Datetime:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
Public class currentDateTime{
    public static void main (String[] args) {
    LocalDateTime currentDateTime = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-mm-
                                                    HH:mm:ss
String formattedDateTime = currentDaTime.format(formatter)
System.out.println("Current Date and Time: "+formattedDateTim
}.
```

MD. Naiem
IT230[redacted]25

32

```
Public class CounterClass {
    Privale static int instanceCount = 0;
    Privale static final int MAX_COUNT = 50;
    Public CounterClass() {
     instanceCount++;
     if(instanceCount > MAX_count) {
       instance Count = 0;
     }
    }
public static int getInstanceCount() {
       return instanceCount;
    }
 Public static void main(String[] args) {
 for(int i=0; i<55; i++) {
 new CounterClass();
 System.out.Println("Instance Count: "+CounterClass.getInstanceCou
                                                        ());
   }
 }
}
```

MD. Naiem
IT23025

33]

```
Public class ExtremeFinder {

 Public static int findExtreme(String type, int numbers) {

    if(numbers.length==0){

    throw new IllegalArgumentException("At least one number
                                        must be provided");

    }

    int extreme=numbers[0];

    for(int num:numbers) {
    if(type.equalsIgnoreCase("smallest")) {
     if(num<extreme) {
     extreme=num;
    }
    else if(type.equalsIgnoreCase("largest")) {
      if(num>extreme) {
      extreme=num;
      }
    else {
    throw new IllegalArgumentException("Invalid typ. Use 'smallest
                                        or 'largest'");
} }
return extreme; }
```

```
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);
}
}
```

34|

```
/*The .equals() method checks whether the content
of s1 and s2 are the same. since both contain
"This is ICT 2107 Java", it returns true.*/
true
// The == operator checks whether s1 and s2 refer to the same
memory location. They are different objects,
false
//they point to the same memory location in string pool
true
```

MD. Naiem
IT23025

35, 36]

```
interface Alpha {
    void methodA();
    void methodB();
}
interface Beta {
    void methodC();
    void methodD();
}

abstract class AbstractBase implements Alpha {
    public void methodA() {
        System.out.println("Method A implemented in AbstractBase.");
    }
    public abstract void methodE();
}

class FinalClass extends AbstractBase implements Beta {
    public void methodB() {
        System.out.println("method B implemented FinalClass.");
    }
```

```
Public void methodC() {
  System.out.Println("method c implemented in Finalclass.");
 }

Public void methodD() {
 System.out.Println("Method D implemented in FinalClass.");
 }
Public void methodE(){
 System.out.Println("method E implemented in FinalClass.");
 }
Public class Main {
Public static void main(String[] args) {
FinalClass obj=new FinalClass();
obj.methodA();
obj.methodB();
obj.methodC();
obj.methodD();
obj.methodE();
 }
}
```

MD. Naiem
IT23025

37]

Issue with the code

The class Z implements both interfaces X and Y, but both interfaces contain a default method show()

This results in a "duplicate default method" conflict, leading a compilation error because Java does not know which show() method to inherit

Solution 1: Override the show() method in class Z

```
Public class Z implements X,Y{
@Override
Public void show(){
X.super.show(); //Explicitly calling show() from inherit X
}
Public static void main(String[] args){
  Z obj = new Z();
  Obj.show();
 }
}
```

Solution 2

```
Public class Z implements X,Y {
@Override
Public Void show() {
System.out.Println("Z's own show method");
}
Public static void main(String[] args){
Z obj = new Z();
obj.show(); //calls overridden show() from Z
}
}
```

MD. Naiem
IT23025

38)

```
Class singletonExample{

  Private static singletonExample instance;

  Private singletonExample(){
   System.out.println("singletonExample instance created.");
  }
 Public static synchronized singletonExample getInstance(){
  if(instance == null){
  instance= new singltonExample();
  }
  return instance;
}
Public void showMessage(){
   System.out.println("Hello from singletonExample");
   }
 }
Public class SingletonDemo{
 Public static void main(String[]args){
singletonExample obj1 = singletonExample.getInstance();
singletonExample obj2 = singletonExample.getInstance();
obj1.showMessage();
if(obj1 == obj2){ system.out.println("Both are same instance");
else{ system.out.println("Different instance");
```

MD. Naiem
IT23025

Q1

```
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 InvalidAmountExce
{
    if(amount<0){
    throw new InvalidAmountException("Amount can't be negat
}
```

MD. Naiem
IT23025

99)

```
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 InvalidAmountExce
{
    if(amount<0){
    throw new InvalidAmountException("Amount can't be neg
}
```

```
balance += amount;
System.out.println("Added $"+amount+" to wallet. New balance:
                                        $"+balance);
    }
Public void spend(double amount) throws InvalidAmountExcept
InsufficientFundsException {
        if(amount<0) {
        throw new InvalidAmountException("Amount can't be negative
        }
        if(amount>0) {
    throw new InsufficientFundsException("Insufficient balance");
    }
    balance-= amount;
    System.out.Println("spent $"+amount+" from wallet. Newbalance:
                                        $"+balance);
    }
    Public double getbalance(){
    return balance;
}
```

```
Public class WalletTest {
Public static void main (String[] args) {
Wallet mywallet = new Wallet(100);
try {
    mywallet.addfunds(50);
    mywallet.spend(30);
    myWallet.addFunds(-20);
    } catch (InvalidAmountException/InsufficientFundsException e)

      System.out.Println("Exception caugh: "+e.getMessage()

    }
try {
  myWallet.spend(200);
   } catch(InvalidAmountException/InsufficientFundsException e)

    System.out.Println("Exception caught: "+e.getMessage()

   }
}
}
```

MD. Naiem
IT23025

4o

```
Public class ArithmeticOperation{
Public static void main(String[] args){
 if(args.length != 2) {
 System.out.println("Error: Please provide exactly two integers");
  return;
  }
try{
    int num1 = Integer.parseInt(args[0]);
    int num2 = Integer.parseInt(args[1]);
    int sum = num1+num2;
    int difference = num1-num2;
    int product = num1*num2;
   double quotient = (double)num1/num2;
 System.out.println("sum "+sum);
 System.out.println("difference"+ difference);
system.out.println("product "+ product);
system.out.println("quotient "+quotient); }
catch(NumberFormatException e) {
System.out.println("Error: Both args must integers.");
} catch(ArithmeticException e) {
  system.out.println("Error: Divison by zero is not allowed.");
  }
```