1/ Write a Java program that reads a series of numbers from a file (input.txt), determines the highest number in the series, calculates the sum of natural numbers upto that highest number, and writes the result to another file (output.txt). Use Scanner to read from the file and Printwritter to write to the file. Assume the numbers in the input file separated by commas.

Sample Input.txt

10, 55, 1000..... 100

Sample Output.txt

55, 1540, 500500..... 5050

Soln

```
import java.io.*;
import java.util.*;
public class SumNaturalNumbers {
  public static void main(String[] args) {

    String inputFile = "input.txt";
    String outputFile = "output.txt";
    try (Scanner scanner = new Scanner(new File(inputFile));
    PrintWriter writer = new PrintWriter(new File(outputFile))) {
```

```
// Read numbers from file
scanner.usePelimiter(",");
List<Integer> numbers = new Arraylist<>();
while(scanner.hasNext()){
  if(scanner.hasNextINT()){
  numbers.add(scanner.nextINT());
  } else {
     scanner.next();
  }
}
// Process numbers and Write to Output file
for(int num : numbers){
  int sum = SumNaturalNumbers(num);
  Writer.Print(num + "," + sum + "\n");
}
  System.out.Println("Processing compleate.Cheak output.txt for
                      results.");
      System.err.Println("Error: File not found.");
    }
  }// Function to compute Sum of first N natural numbers
    Private static int SumNaturalNumbers(int n){
       return n*(n+1)/2;
    }
```

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

Solution

Differences Between static and final :

1. static modifier :

- Purpose : The static modifier means that the field or method belongs to the class, rather than to any specific instance of the class.
- Fields : A static field is shared by all instances of the class. It is not tied to any particular instance, and it can be accessed using the class name, or via an object reference (though it's generally best practice to access it via the class).
- Methods : A static method can be ~~allowed~~ called without creating an instance of the class. It can only directly access other static members of the class, not instance members.

2. Final modifier :

- Purpose : The final modifier is used to indecate that the field or method cannot be changed once assigned or overhidden.
- Fields : A final field must assigned exactly once, either - during ~~defation~~ declaration or in the constructor.
- Methods : A final method cannot be overridden by Subclass
- classes : A final class cannot be subclassed.

Example of Static and Final Fields

```
Public class Example {
    Static int staticfield = 10;

    final int finalField ;
    public Example (int value) {
        this. finalFied = value;
    }
    }
```

- key take ways :
  - Static members belong to the class, and final members cannot be changed or overhidden.

3] Write a Java Program to find all factorion numbers within given range. A number is called a factorion if the sum of the factorials of its digits equals the number itself. The program should take user input for the lower and upper bounds of the range and print all factorion numbers with in that range.

Code

```
import java.util.Scanner;
public class FactorionFinder {
Private static final int [] FACTORIALS =new int[10]

static {
FACTORIALS[0] = 1;
int fact =1;
for(int i= 1; i<10; i++) {
    fact *= i;
    FACTORIALS[i] = fact;
  }
}
```

```
Private static int sumOfDigitFactorials (int num) {
  int sum = 0, temp = num;
  while (temp > 0) {
     int digit = temp % 10;
     Sum + = FACTORIALS [digit];
     temp /= 10;
  }
   return sum;
}
Private static boolean isFactorion (int num) {
     return num == SumOfDigitFactorials (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 of the
                        range: ");
     int upperBound = scanner.nextInt ();
```

```
System.out.print("Factorion numbers in the range:");
boolean found = false;
for (int i = lowerBound ; i <= upperBound ; i++) {
 if (isFactorion(i)) {
  System.out.print(i + " ");
      found = true;
  }
 }
 if (!found) {
   system.out.println("None");
  }
   Scanner.close();
   }
  }
```

4| Distinguish the differences among class, local and instance variables what is significant of this keyword?

1. Class Variables

- Defination : Also known as static variables, class variables are declared with the static keyword with in a class.

- Scope & Lifetime:

- They belong to the class itself rather than to any Particul instance.
- They are created when the class is loaded and exist fo as long as the class is loaded in memory.
- Every instance of the class shares the same class variable.

2. Instance Variables:

Instance variables are declared in a class but outside any method, and they are not maked as static.

Scope & Lifetime

- They belong to an individual object of the class.
- Each time a new object is created, a new copy of the instance variable allocated.

- They exist as long as the object exists.

3. Local Variables :

- Local variables are declared inside a method.
  - They exist only with in the method where they are declared.
  - Their lifetime is limited to the execution of that method.
  - They are not visible outside the method and must be explicitly initialized before use.

Significance of this keyword

- this keyword is a reference to the current object — the instance on which the method or constructor is being called.
- It is especially useful for distinguish between instance variables and parameters or local variables that have the same name.

5] Write a Java Program that defines a method to calculate the sum of all elements in an integer array. The method should take an integer array as a parameter and return the sum. Demonstrate this method by passing an array of integers from the main method.

Solve

```
public class ArraySumCalculator {
    public static int calculateSum(int[] numbers) {
        int sum = 0;
        for(int num : numbers) {
            sum += num;
        }
        return sum;
    }
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        int sum = calculateSum(numbers);
        System.out.println("Sum of array elements: " + sum);
    }
}
```

6| What is called Access Modifier? Compare the accessibility of public, Private and Protected modifier. Describe diffrent types of variable in java with example.

Access Modifier:

Access Modifier in java define the scope and visibility of variables, methods, and classes. They control how other parts of a Program can interact with the members of a class.

Comparison of Access Modifiers

Modifier	Accessibility within class	Accessibility within package	Accessibility in subclass	Accessibility outside package
public	yes	yes	Yes	Yes
Private	yes	No	No	No
Protected	yes	yes	Yes	No
Default	yes	yes	No	No

Types of Variables in Java

Java has three main type variables:

1] Local Variables

- Declared inside a method, constructor, or block.
- Scope is limited to the block where they are defined.
- No default value, must be initialized before use.

Example:

```
public class Example {
    public void display() {
        int x = 10;
        System.out.println("Local Variable: " + x);
    }
}
```

2] Instance Variables

- Declared inside a class but outside any method.
- Associated an instance of the class.
- Default values.
    - int → 0, double → 0.0, boolean → false, String → null.

Example:

```
public class Example{
    int num=28;
    public void show() {
    system.out.println("Instance variable:"+num);
  }
}
```

3| Static variable

- Declared with the static keyward inside a class.
- shared among all objects of the class.
- Memory allocated only once at class loading time.

Example:

```
public class Example {
static int count=0 // static variable
public void increment() {
    count++;
}
public static void main (String[] args) {
Example obj1=new Example();
Example obj2=new Example();
```

```
Obj1.increment();
Obj2.increment();

System.out.println("Static Variable:" + Example.count);

}
}
```

7| Write a Java Program quradratic equation.

Solve

```
import java.util.Scanner;
public class QuadraticRootFinder {
public static void main (String[] args) {
Scanner scanner = new Scanner(System.in);
System.out.println("Enter co-efficients a, b, and c:");
int a = scanner.nextInt();
int b = scanner.nextInt();
int c = scanner.nextInt();

double discriment = b*b-4*a*c;
if (discriment <0) {
System.out.println("No real roots.");
}
```

```
else {
double root1 = (-b + Math.sqrt(discriment)) / (2.0*a);
double root2 = (-b+ Math.sqrt(discriment)) / (2.0*a);

if (root1>0 && root2>0) {
System.out.Println("The Smallest positive root is:
" + Math.min(root1, root2));
} else if (root1>0){
System.out.Println("The Smallest positive root is:
" +root1);
} else if (root2>0){
System.out.Println("The Smallest positive root is:"
+ root2);
} else {
system.out.println("No positive roots.");
}
}
scanner.close();
}
}
```

8| Write a Program character type Detection

code

```
import java.util.Scanner;

public class Characterchaecker {
  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.");
  } elseif (character.iswhitespace(ch)) {
     System.out.println("It is whitespacecharacter.");
  }else{
       System.out.println("ch+" is a special character.");
     }
     Scanner close();
  }
```

Passing an Array to a Function in Java

In Java, you can pass an array to a function just like any other variable. Here's an example:

Code

```
public Class ArrayExample {
Public static void PrintArray (int[] arr) {
System.out.println ("Array elements:");
for (int num: arr) {
  System. out. Println (num + " ");
}
  System.out. println ( );
}
  public static void main (string [ ] args) {
  int [ ] numbers = {1, 2, 3, 4, 5}
  PrintArray (numbers);
  }
}
```

9|

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 ("Animal: " +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 ~~superclass~~ 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 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() { }
}
```

10| Differentiate Between static and Non-static members.

solve static member

- Declared using the static keyword.
- Belong to the class rather than an instance of the class.
- shared among all instances of the class.
- canbe accessed using the Class name.
- cannot access non-static members directly.

Example:

```
Class staticExample {
    static int count = 0;
    static void showCount () {
        System.out.println ("Count: " + count);
    }
}
    public class main {
    public class static void main (string[] args) {
    Static Example.count = 5;
    Static Example.showCount ();
}}
```

2| Non-static Members

- Do not use the static keyword.
- Belong to an instance of the class.
- Each instance has its own copy of the non-static members.
- can access both static and non-static members.

Example of Non-static Member:

```
class NonstaticExample {
int value;
void display() {
System.out.println("value:" + value);
}
}
public class main {
public static void main (String[] args) {
NonstaticExample obj1 = new NonStaticExample();
obj1.value = 10;
obj1.display();
NonstaticExample obj2 = new NonstaticExample();
obj2.value =20;
obj2.display();
}
}
```

Palindrome Checker Program

```
import java.util.Scanner;
public class PalindromeChecker {
  public static boolean ispalindrome(int num) {
  int original=num, reverse=0, remainder;
   while (num>0) {
      remainder=num%10;
      reverse = reverse*10 + remainder;
      num /=10;
   }
    return original == reverse;
  }
  public static boolean ispalindrome(String str) {
   int left=0, right=str.length()-1;
   while (left < right) {
   if (str.charAt(left) != str.charAt(right)) {
        return false;
      }
      left++;
      right--;
   }
```

```
return true;
}
Public static void main (String [] args) {
Scanner scanner = new scanner (System.in);
System.out.print(" Enter a number or a String: ");
if (scanner.hasNextInt()) {
    Int num = scanner.nextInt();
  if (ispalindrome(num)) {
   System.out.println (num + " is a palindrome.");
  } else {
   string str = scanner.next();
   if (is palindrome (str)) {
   System.out.println ("\" + str + "\" is a palindrome.");
  } else {
    System.out.println ("\" + str "\" is not palindrome");
   }
  }
    scanner.close();
 }
}
```

11

Abstraction

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

Example of Abstraction in Java

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

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

2| Encapsulation

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

Example:

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

```
BankAccount account = new BankAccount();

account.deposit(1000);

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

Differences Between Abstract class and interface

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

12|

Solve

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

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

```
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.");
  }
}
```

13] Java Program that implements the UML - diagram :

Solve

```
import java.utill.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 radius, String color, boolean filled).
   super (color, filled);
     this. radius = radius;
  }
   public double getRadius ( ) {
        return radius;
   }
  public void setRadius (double radius) {
      this. radius = radius;
  }
   public double getArea( ) {
    return Math. PI * radius * radius;
   }
  Public double getperimeter ( ) {
   return 2 * Math. PI * radius;
}
```

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

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

```
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 ("Radius:" + circle.getRadius ());
System.out.Println ("Area:" + circle.getArea ());
System.out.Println ("Perimeter:" + Circle.getPerimeter ());
System.out.println ("Diameter:" + circle.getDiameter ());
System.out.println (circle.tostring ());

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

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 upto $2^{31}-1$ and long up to $2^{63}-1$). BigInteger can store much largen 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();
  }
}
```

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 - mutiple interfaces.
5. Can have any access modifier for methods and variables.	5. All methods are implicitly public.

When to use an Abstract class over an Interface

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

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

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

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

Can a class implement Multiple Interfaces in Java?

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

```
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");
    }
}
```

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 flexib and resuability. polymorphism can be achived through method overriding and method overloading, but the mos 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 makeSound() {
    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.

17| Differences Between Arraylist and LinkedList in Java.

Arraylist	LinkedList
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 references)
5. Search contains O(n)	5. Search contains O(n)

When to use which?

- Use Arraylist when:
- You need fast random access (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 accesses.
- 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_A
              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
  System.out.Println ("Multiple random numbers: " Arrays.
   (myRand (100, 5)));
}
}
```

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 oth to release resources.
- Starvation - Low-Priority threads not getting CPU ti
- Livelock - Threads continuously responding to each other without making progress.

Using Synchronised for Thread Safety

The Synchronized keyword Prevents multiple threads from accessing ~~erict~~ 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 ReentrantLo
  instead of Synchronized.

Using ReentrantLock:

```
import java.utill.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 dead
```

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

20|

Exception Handling in Java (short overview)

Exception handling in Java ensures that runtime errors do not 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 IOException {
    throw new IOException ("Error occurred");
}
```

~~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();
  }
```

(21)

Interfaces in Java can have default methods staring from Java 8, and this feaehor 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");
    }
}
```

(22)

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(logn)	O(1)
Deletion time	O(1)	O(logn)	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 Treemap differ from HashMap in ordering

- Hashmap 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.

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.

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.