KLS GOGTE INSTITUTE OF TECHNOLOGY, BELAGAVI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

Subject:

Object Oriented Programming with Java (22CS33)

TERMWORK 1

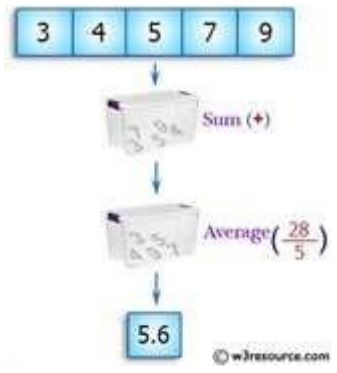
Tittle: Write a program to demonstrate the implementation of a 2-dimension array.

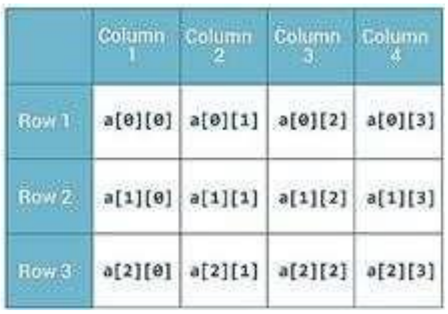
Problem Definition:

In a college, a professor wants to assess the performance of 5 students in Java subject where they have taken three internal assessments (IA1, IA2, and IA3). However, to be fair and to account for any poor performance in one of the tests due to unforeseen circumstances, the professor decides to consider only the best two out of the three assessments to compute the final average score for each student.The professor decides to develop a program that will accept the marks obtained by each student in these three assessments, calculate the total marks, and then defines a method to compute the average marks using the best two scores. This average will be used to determine the final internal assessment score that contributes to their overall grade in the subject.

Introduction:

Basics – for 5 subject marks example –





	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Here is how we can initialize a 2-dimensional array in Java.

Algorithm:

```
1. Repeat For I = 1 to M
    2. Repeat For J = 1 to N
        3. Create a 2 – dimensional array
    [End of Step 2 For Loop]
[End of Step 1 For Loop]
```

4. Exit

Implementation Steps:

Step 1: Create 2 – dimensional Array and store marks data and calculate total

Step 2: Compute average function

Step 3: Prompt the outputs to the screen – total and average – for 5 students and 3 subjects –

Code

```
import java.util.Scanner;
public class TW1 {
    public static void main(String[] args){
        int[][] marks=new int[5][3];
        int[] total = {0,0,0,0,0};
        int[] avg = new int[5];
        Scanner in = new Scanner(System.in);
        for(int i=0; i<5; i++){
            System.out.println("Enter the marks of student "+(i+1));
            for(int j=0; j<3; j++){
                marks[i][j]=in.nextInt();
                total[i]=total[i]+marks[i][j];
            }
        }
        for(int i=0;i<5;i++)
            avg[i] = computeAvg(marks[i][0], marks[i][1],marks[i][2]);

        System.out.println("Student results:");
        for(int i=0;i<5;i++){
            System.out.println("For student " +(i+1));
            System.out.println("Total marks:" +total[i]);
            System.out.println("Avreage marks:" +avg[i]);
        }
    }
    static int computeAvg(int m1,int m2,int m3){
        int min = m1;
        if(m2<min)
            min = m2;
        if(m3<min)
```

```
            min = m3;
        int total = (m1+m2+m3-min);
        return(int)Math.ceil(total/2.0);

    }
}
```

Output

```
Enter the marks of student 1

65 96 48

Enter the marks of student 2

65 39 87

Enter the marks of student 3

69 85 45

Enter the marks of student 4

63 64 25

Enter the marks of student 5

35 87 96

Student results:

For student 1

Total marks:209

Avreage marks:81

For student 2

Total marks:191

Avreage marks:76

For student 3

Total marks:199

Avreage marks:77

For student 4

Total marks:152

Avreage marks:64
```

```
For student 5
Total marks:218
Avreage marks:92
```

TERMWORK 2a

Title: Write a program to demonstrate the implementation of string handling.

Problem Definition: Imagine you are developing a feature for a word game that involves

matching words that can be rearranged into each other. The objective is to determine if two given words are anagrams. An anagram is defined as two strings that have the same characters in the same frequency, but possibly in a different order. For example, if a player inputs the words "listen" and "silent", or "stressed" and "desserts", your program should identify them as anagrams. As part of the game, your task is to write a Java program that checks whether two provided strings are anagrams of each other. The program will help players verify if one word can be rearranged to form the other.

Implement this feature by writing a Java method that takes two strings as input and returns whether they are anagrams. If the words "dusty" and "study" are inputted, your method should confirm that they are anagrams, since both contain the exact same characters with the same frequency.

Your solution should be case-insensitive, ensuring that "Creative" and "Reactive" are also

recognized as anagrams.

Note: Do not use the built-in method to sort the array elements.

Introduction:

Scanner Class in Java: Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.

- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort() ● To read strings, we use nextLine().
- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.

- The Scanner class reads an entire line and divides the line into tokens. Tokens are small elements that have some meaning to the Java compiler. For example, suppose there is an input string: How are you In this case, the scanner object will read the entire line and divides the string into tokens: “How”, “are” and “you”. The object then iterates over each token and reads each token using its different methods.

Java's Scanner String input method

1. Import java.util.*; to make Java's Scanner class available.
2. Use the new keyword to create an instance of the Scanner class.
3. Pass the static System.in object to the Scanner's constructor.
4. Use Scanner's next() method to take input one String at a time.

isLowerCase( ): The method determines whether the specified char value is lowercase.

Syntax : boolean isLowerCase(char ch) isUpperCase() Method: This method determines whether the specified char value is uppercase.

Syntax: boolean isUpperCase(char ch)

Algorithm:

```
1. Start
2. Input both the string literals
3. Convert the obtained strings to LowerCase.
4. Call check_anagrams( ) method by passing those strings
5. In check_anagrams() method,
     i.   Create character arrays of the given strings and store them in two
          variables
     ii.  Call sort method by passing each character array individually
     iii. Display the result
6. In sort( ) method
     i.   Sort the character array in ascending order by using any sorting
          technique
     ii.  Return the result in String
7. Stop
```

Code:

```
 import java.util.Scanner;
public class tw2a {
   @SuppressWarnings("empty-statement")
   public static void main(String args[])
   {
      String s1,s2;
      Scanner in=new Scanner(System.in);
            System.out.println("Enter the String #1 :");
            s1=in.next().toLowerCase();
            System.out.println("Enter the String #2 :");
            s2=in.next().toLowerCase();
            check_anagrams(s1,s2);
   }
   static void check_anagrams(String s1,String s2)
   {
      char c1[] = s1.toCharArray();
      char c2[] =s2.toCharArray();
     String s3 =sort(c1);
     String s4 =sort(c2);
     System.out.println(s1 +" and " +s2 + " are anagrams--->"+s3.equalsIgnoreCase(s4));
   }
   static String sort(char arr[])
   {
      for(int i= 0; i < arr.length; i++){
         for(int j=0; j<arr.length-1; j++)
         {
            if(arr[j]>arr[j+1])
            {
               char temp =arr[j];
               arr[j]=arr[j+1];
               arr[j+1]=temp;
            }
         }
      }
```

```
        return new String(arr);
    }
}
```

Output

```
Enter the String #1 :
creative
Enter the String #2 :
reactive
creative and reactive are anagrams--->true

Enter the String #1 :
stress
Enter the String #2 :
Dessert
stress and dessert are anagrams--->false
```

TERMWORK 2b

Tittle: Write a program to demonstrate the implementation of class and its member methods.

Problem Definition:

Imagine you're a software developer tasked with creating a program for a geometry tutoring app. The app helps students learn about different types of triangles and their properties by allowing them to input the lengths of the sides of a triangle and receiving feedback on the type of triangle it is (Equilateral, Isosceles, or Scalene) and its area.

Requirements:

To achieve this, you need to design a class named myTriangle that models a triangle using its three sides. The class should have the following functionalities:

a) Initialization of the Triangle:

The class should initialize a triangle object with three sides provided by the user. This will allow the app to model any triangle based on the sides' lengths.

b) Determine the Type of Triangle:

Implement a method that determines whether the triangle is Equilateral (all sides are equal), Isosceles (two sides are equal), or Scalene (all sides are different). This will help students learn about the different classifications of triangles based on their side lengths.

c) Compute and Return the Area:

Using the given formula, s = (a + b + c) / 2 and area = sqrt(s * (s - a) * (s - b) * (s - c)), implement a method that calculates and returns the area of the triangle. This feature will help students understand how the area of a triangle can be

Introduction:

A triangle is a three-sided polygon, which has three vertices. The three sides are connected with each other end to end at a point, which forms the angles of the triangle. The sum of all three angles of the triangle is equal to 180 degrees.

Types of Triangle:

- Scalene Triangles
- Isosceles triangles
- Equilateral triangles

If all the three sides are different in length, then its scalene triangle.

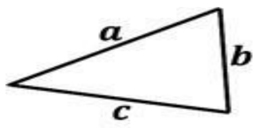
If any two sides are equal in length, then it is an isosceles triangle.

If all three sides are equal in length, then it is an equilateral triangle.

Area is the region occupied by it in a two-dimensional space. Area = ½ (Product of base and height of a triangle) Sum of the Lengths of Two Sides of a Triangle:

A polygon is a simple closed figure made up of only line segments. A triangle is the smallest polygon formed by joining three-line segments. These three-line segments are called the sides of the triangle. Any three-line segments cannot form a triangle. A triangle can be drawn only if the sum of any two sides of the triangle is greater than the third side. This is stated as the triangle inequality.

Triangle Inequality



$$a + b > c$$
$$a + c > b$$
$$b + c > a$$

Algorithm:

Step 1: Get inputs – sides of a triangle a, b, c
Step 2: Testing triangle existence – sum of two sides must be greater than the third side.
Step 3: Calculation area of a triangle, Area = ½ X Base X Height.

Code

```
import static java.lang.System.exit;
import java.util.Scanner;
class Triangle {
    double a,b,c;
    void getSides(){
        Scanner in=new Scanner(System.in);
        System.out.println("Enter 3 sides of a triangle:");
        a=in.nextDouble();
        b=in.nextDouble();
        c=in.nextDouble();
    }
    void checkTriangle(){
        if((a+b)>c && (b+c)>a && (a+c)>b){
            if(a==b && b==c && c==a)
                System.out.println("Triangle is equilateral");
            else if(a==b || b==c || c==a)
                System.out.println("Triangle is isosceles");

            else
                System.out.println("Triangle is scalene");
        }
        else{
            System.out.println("Triangle cannot be formed");
            exit(0);
        }
    }

    double computeArea(){
      double s=(a+b+c)/2;
      double area=Math.sqrt(s*(s-a)*(s-b)*(s-c));
      return area;
    }
}

public class tw2b {
    public static void main(String[] args){
        Triangle t=new Triangle();
        t.getSides();
        t.checkTriangle();
        if((t.computeArea())!=0){
            System.out.println("Area is "+t.computeArea());
        }

    }

}
```

Output

```
Enter 3 sides of a triangle:
2 2 2
Triangle is equilateral
Area is 1.7320508075688772

Enter 3 sides of a triangle:
2 2 1
Triangle is isosceles
Area is 0.9682458365518543

Enter 3 sides of a triangle:
2 3 4
Triangle is scalene
Area is 2.9047375096555625

Enter 3 sides of a triangle:
-1 2 -3
Triangle cannot be formed
```

TERMWORK 3

Title: Write a program to demonstrate the implementation of parameterized:
a. Methods. b. Constructor.

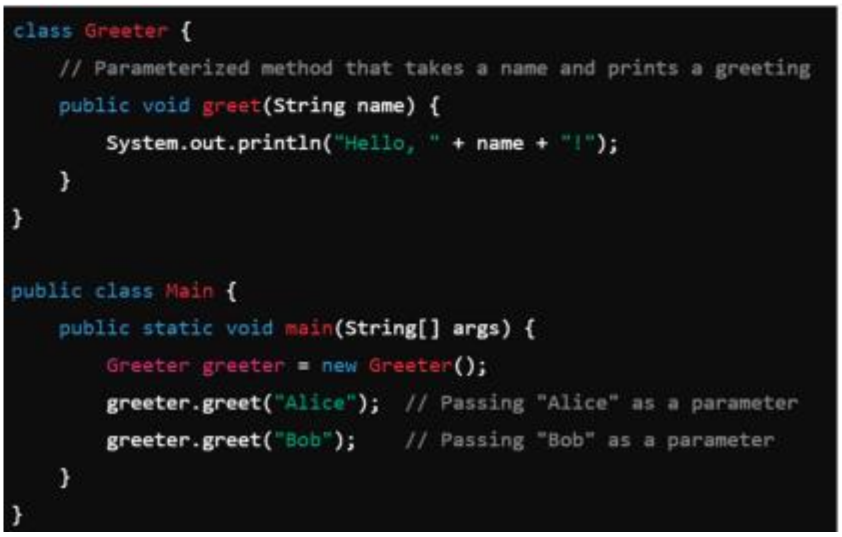
Problem Statement: A certain small bank intends to automate a few of its banking operations for its customers. Design a class by name mybankAccount to store the customer data having following details:
1.accountNumber 2. acctType 3. Name 4. Address 5. accountBalance

The class must have both default and parameterized constructors. Write appropriate method to compute interest accrued on accountBalance based on accountType and time in years. Assume 5% for S/B account 6.5% for RD account and 7.65 for FD account. Further, add two methods withdrawAmount/depositAmount with amount as input to withdraw and deposit respectively. The withdrawAmount method must report in-sufficient balance if accountBalance falls below Rs. 1000.

Introduction

Parameterized Methods:
A parameterized method accepts one or more parameters (arguments) that provide data to the method when it is called. The method can then use these parameters within its body. This concept allows for reusability of code by enabling methods to perform actions based on the input values passed.

Example:



```
class Greeter {
    // Parameterized method that takes a name and prints a greeting
    public void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }
}

public class Main {
    public static void main(String[] args) {
        Greeter greeter = new Greeter();
        greeter.greet("Alice");  // Passing "Alice" as a parameter
        greeter.greet("Bob");    // Passing "Bob" as a parameter
    }
}
```

Parameterized constructors:

A parameterized constructor is a constructor that accepts parameters to initialize an object’s attributes when it is created. This allows for initializing an object with specific values during the creation. It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

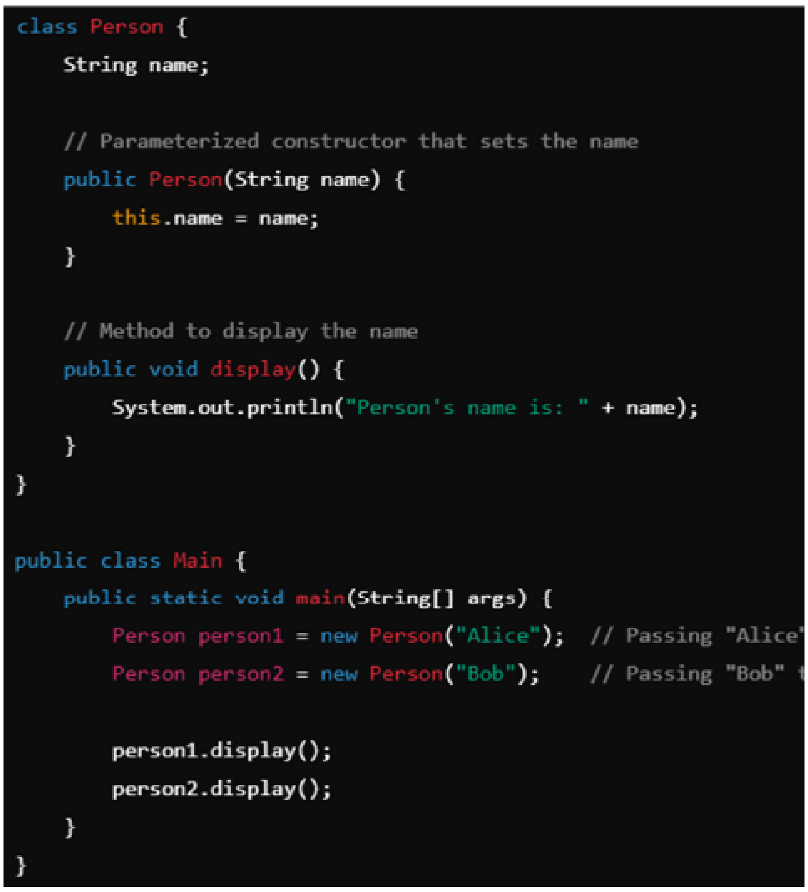
A constructor which has a specific number of parameters is called a parameterized constructor.

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

In the below example, we have created the constructor of Student class that have two parameters.

We can have any number of parameters in the constructor.

Example



```
class Person {
    String name;

    // Parameterized constructor that sets the name
    public Person(String name) {
        this.name = name;
    }

    // Method to display the name
    public void display() {
        System.out.println("Person's name is: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Person person1 = new Person("Alice");  // Passing "Alice"
        Person person2 = new Person("Bob");    // Passing "Bob" t

        person1.display();
        person2.display();
    }
}
```

Algorithm/Implementation Steps

1. Define the Class and Attributes

- Define a class named mybankAccount to store the details of a bank account.
- The class should include the following attributes:
- accountNumber (to store the account number)
- acctType (to store the account type like SB, RD, FD)
- name (to store the customer’s name)
- address (to store the customer’s address)
- accountBalance (to store the current balance in the account)

2. Create Default and Parameterized Constructors

- The default constructor should initialize the attributes to default values.
- The parameterized constructor should accept values for the account details and initialize the attributes.

3. Method to Compute Interest

- Create a method computeInterest that takes two arguments: years (the time in years for
- interest calculation) and accountType (the type of account).
- Use different interest rates based on the account type:
- 5% for SB (Savings Bank)
- 6.5% for RD (Recurring Deposit)
- 7.65% for FD (Fixed Deposit)
- The method will return the accrued interest on the accountBalance.

4. Methods for Deposit and Withdrawal

- depositAmount(double amount): Add the amount to the accountBalance.
- withdrawAmount(double amount): Subtract the amount from accountBalance and check whether the balance falls below Rs. 1000. If so, print "Insufficient balance."

Code:

```
import java.util.Scanner;
// Class representing a Bank Account
class BankAccount {
    // Instance variables
    private int accNumber;
    private String name;
    private String address;
    private String accType;
    private double accBal;
    private static int count = 0; // Static counter for account numbers
```

```
// Default constructor
public BankAccount() {
    accNumber = ++count; // Auto-generate account number
    Scanner in = new Scanner(System.in);

    System.out.println("Enter Customer Name: ");
    name = in.nextLine();

    System.out.println("Enter Address: ");
    address = in.nextLine();

    System.out.println("Enter Account Type (sb/rd/fd): ");
    accType = in.nextLine().toLowerCase();

    System.out.println("Enter Initial Account Balance: ");
    accBal = in.nextDouble();

    System.out.println("Account Created Successfully!\n");
}

// Parameterized constructor
public BankAccount(String name, String address, String accType, double accBal) {
    accNumber = ++count; // Auto-generate account number
    this.name = name;
    this.address = address;
    this.accType = accType.toLowerCase();
    this.accBal = accBal;
}

// Display account details
public void displayAccountDetails() {
    System.out.println("\nAccount Details:");
    System.out.println("Account Number: " + accNumber);
    System.out.println("Name: " + name);
    System.out.println("Address: " + address);
    System.out.println("Account Type: " + accType.toUpperCase());
    System.out.println("Account Balance: " + accBal);
}

// Compute and add interest based on account type and duration
public void computeInterest(int time) {
    double interestRate = 0;

    if (accType.equals("sb")) {
        interestRate = 0.05; // 5% annual interest for SB
    } else if (accType.equals("rd")) {
            interestRate = 0.063; // 6.3% annual interest for RD
        } else if (accType.equals("fd")) {
            interestRate = 0.0765; // 7.65% annual interest for FD
        } else {
            System.out.println("Invalid account type. Unable to compute interest.");
            return;
        }

        double interest = accBal * interestRate * time;
        accBal += interest; // Add interest to account balance
        System.out.printf("Interest Earned for %s account (%.2f%%): %.2f%n", accType.toUpperCase(),
interestRate * 100, interest);
    }
    // Deposit an amount into the account
    public void deposit(double amount) {
        accBal += amount;
        System.out.println("Amount Deposited Successfully. Updated Balance: " + accBal);
    }

    // Withdraw an amount from the account
    public void withdraw(double amount) {
        if ((accBal - amount) < 1000) {
            System.out.println("Insufficient Balance! Minimum balance of 1000 must be maintained.");
        } else {
            accBal -= amount;
            System.out.println("Amount Withdrawn Successfully. Updated Balance: " + accBal);
        }
    }
}
// Main class to test the BankAccount functionality
public class Tw3 {
    public static void main(String[] args) {
        // Creating accounts
        BankAccount b1 = new BankAccount();
        BankAccount b2 = new BankAccount("Ranjana", "XYZ Street", "fd", 20000);
        BankAccount b3 = new BankAccount();

        // Test operations on the first account
        b1.displayAccountDetails();
        b1.computeInterest(1); // Compute interest for 1 year
        b1.deposit(500);
        b1.withdraw(1500);

        // Test operations on the second account
        b2.displayAccountDetails();
        b2.computeInterest(1); // Compute interest for 1 year
        b2.deposit(10000);
        b2.withdraw(1000);

        // Test operations on the third account
        b3.displayAccountDetails();
        b3.computeInterest(1); // Compute interest for 1 year
        b3.deposit(2000);
        b3.withdraw(3000);
    }
}
```

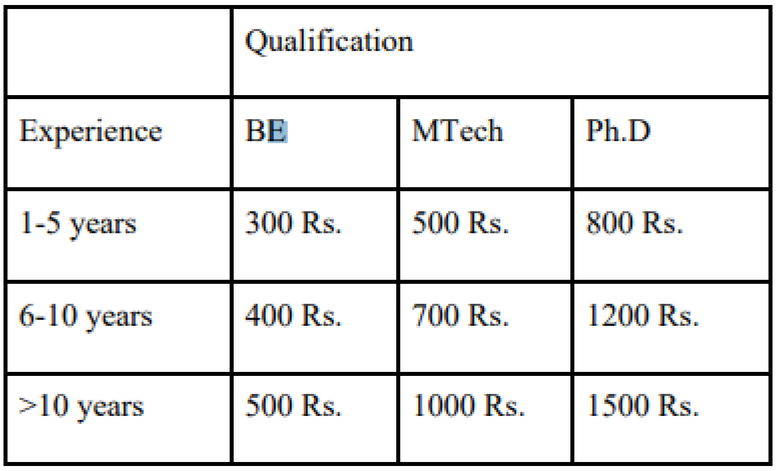
Output

```
Enter Customer Name:
abc
Enter Address:
bgm
Enter Account Type (sb/rd/fd):
sb
Enter Initial Account Balance:
2000
Account Created Successfully!

Enter Customer Name:
abc
Enter Address:
bgm
Enter Account Type (sb/rd/fd):
rd
Enter Initial Account Balance:
5000
Account Created Successfully!


Account Details:
Account Number: 1
Name: abc
Address: bgm
Account Type: SB
Account Balance: 2000.0
Interest Earned for SB account (5.00%): 100.00
Amount Deposited Successfully. Updated Balance: 2600.0
Amount Withdrawn Successfully. Updated Balance: 1100.0

Account Details:
Account Number: 2
Name: Ranjana
Address: XYZ Street
Account Type: FD
Account Balance: 20000.0
Interest Earned for FD account (7.65%): 1530.00
Amount Deposited Successfully. Updated Balance: 31530.0
Amount Withdrawn Successfully. Updated Balance: 30530.0

Account Details:
Account Number: 3
Name: abc
Address: bgm
Account Type: RD
Account Balance: 5000.0
Interest Earned for RD account (6.30%): 315.00
Amount Deposited Successfully. Updated Balance: 7315.0
Amount Withdrawn Successfully. Updated Balance: 4315.0
```

TERMWORK 4

Title: Write a program to demonstrate the implementation of inheritance.

Problem Statement:
A company has two types of employees – Full Time and Par time. The company records for each employee his/her name, age, address, salary and gender. Given the basic salary of the Full Time employee the components of his/her gross salary are: Dearness allowance – 75% of basic salary, HRA – 7.5% of basic salary, IT – 10% of basic. The salary of a Part time employee is dependent on the qualification, experience, number of working hours and the rate per hour, as below:



	Qualification		
Experience	BE	MTech	Ph.D
1-5 years	300 Rs.	500 Rs.	800 Rs.
6-10 years	400 Rs.	700 Rs.	1200 Rs.
>10 years	500 Rs.	1000 Rs.	1500 Rs.

Introduction:

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also. Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

Terms used in Inheritance

● Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

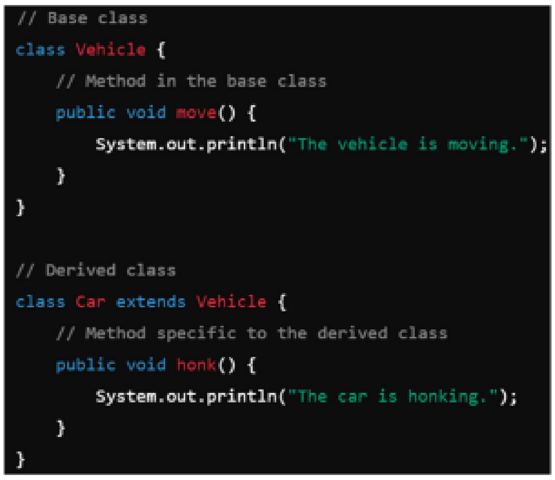
● Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

● Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

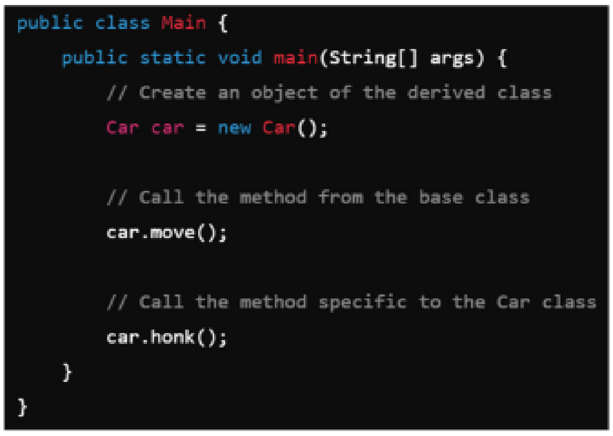
● Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

When a class inherits another class, it is known as a single inheritance. In the example given below,

Car class inherits the Vehicle class, so there is a single inheritance.



```java
// Base class
class Vehicle {
    // Method in the base class
    public void move() {
        System.out.println("The vehicle is moving.");
    }
}

// Derived class
class Car extends Vehicle {
    // Method specific to the derived class
    public void honk() {
        System.out.println("The car is honking.");
    }
}
```



```java
public class Main {
    public static void main(String[] args) {
        // Create an object of the derived class
        Car car = new Car();

        // Call the method from the base class
        car.move();

        // Call the method specific to the Car class
        car.honk();
    }
}
```

Algorithm /Implementation Steps:

Step 1: Define the Employee class

- Create a base class Employee with common attributes: name, age, address, gender, and salary.
- Define a method calculateSalary() as a placeholder to be overridden by derived lasses.

Step 2: Define the FullTimeEmployee class

- Derive a class FullTimeEmployee from Employee.
- Add an attribute basicSalary.
- Override calculateSalary() to compute gross salary using the given components: DA, HRA, IT.

Step 3: Define the PartTimeEmployee class

- Derive a class PartTimeEmployee from Employee.
- Add attributes qualification, experience, workingHours, and ratePerHour.
- Use conditional statements to determine ratePerHour based on qualification and experience.

- Calculate salary as workingHours * ratePerHour.

Step 4: Test the System

- Create instances of both FullTimeEmployee and PartTimeEmployee.
- Call calculateSalary() for each employee and display their details.

Code

```
class Employ{
   String name,address,gender;
   int age;
   double sal;
   Employ(String name,int age,String address,String gender){
      this.name=name;
      this.age=age;
      this.address=address;
      this.gender=gender;
   }

   void show(){
      System.out.println("Name:"+name);
      System.out.println("Age:"+age);
      System.out.println("Address:"+address);
      System.out.println("Gender:"+gender);
      System.out.println("Salay:"+sal);
   }
}
class FTEmploy extends Employ{
   int basSal;
   FTEmploy(String name,int age,String address,String gender,int basSal){
      super(name,age,address,gender);
      this.basSal=basSal;
   }
      void calSal(){
      sal=(basSal+basSal*0.75+basSal*0.075-basSal*0.1);
   }
}
class PTEmploy extends Employ{
   String qual;
   int exp,numHour;
   PTEmploy(String name,int age,String address,String gender,String qual,int exp,int
numHour){
      super(name,age,address,gender);
      this.qual=qual;
      this.exp=exp;
      this.numHour=numHour;
   }

   void calSal(){
      switch(qual){
          case "BE":
              if(exp<=5) sal=numHour*300;
              else if(exp<=10) sal=numHour*400;
              else sal=numHour*500;
              break;
          case "MTech":
              if(exp<=5) sal=numHour*500;
              else if(exp<=10) sal=numHour*700;
              else sal=numHour*1000;
              break;
          case "PhD":
              if(exp<=5) sal=numHour*800;
              else if(exp<=10) sal=numHour*1200;
              else sal=numHour*1500;
              break;
      }
   }
}
public class Tw4{
   public static void main(String []args){
      FTEmploy f1=new FTEmploy("Arun",25,"Vijayapur","Male",10000);
      f1.calSal();
      System.out.println("Details of Full time Employ");
      f1.show();
      PTEmploy e1=new PTEmploy("Rohit",30,"Belgum","Male","BE",6,10);
      e1.calSal();
      System.out.println("\nDetails of Part time Employ 1:");
      e1.show();
      PTEmploy e2=new PTEmploy("Rohini",26,"Mysore","Female","PhD",10,8);
      e2.calSal();
      System.out.println("\nDetails of Part time Employ 2:");
      e2.show();
   }
}
```

Output

```
Details of Full time Employ

Name:Arun

Age:25

Address:Vijayapur

Gender:Male

Salay:17250.0


Details of Part time Employ 1:

Name:Rohit

Age:30

Address:Belgum

Gender:Male

Salay:4000.0


Details of Part time Employ 2:

Name:Rohini

Age:26

Address:Mysore

Gender:Female

Salay:9600.0
```

TERMWORK 5a

Title: Write a program to demonstrate the implementation of method overloading.

Problem Definition:

You are developing a web browser application and need to manage the user's browsing history. The browser should be able to handle navigation through different web pages, which requires a stack to keep track of recently visited pages.

Requirements:

1. Initialization:

- **By Size**: Initialize a stack to hold a fixed number of recent pages (e.g., 50).
- **From Another Stack**: Duplicate the browsing history stack when a user logs in from a different device.
- **From Array**: Load the stack with previously saved browsing history from an array of URLs.

2. Operations:

- **Push**: Add a new URL to the stack whenever the user visits a new page.
- **Pop**: Navigate back to the most recent page (remove and return the top URL from the stack).
- **Peek**: Show the current page without removing it from the stack.

Introduction

Method overriding is one of the ways by which Java achieves Run Time Polymorphism. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed**.**

Usage of Java Method Overriding

Method overriding is used to provide the specific implementation of a method that is already provided by its superclass.

Method overriding is used for runtime polymorphism.

Method overriding allows subclasses to reuse and build upon the functionality provided by their superclass, reducing redundancy and promoting modular code design.

Subclasses can override methods to tailor them to their specific needs or to implement specialized behavior that is unique to the subclass.

Method overriding enables dynamic method dispatch, where the actual method implementation to be executed is determined at runtime based on the type of object, supporting flexibility and polymorphic behavior.

Rules for Java Method Overriding

Same Method Name: The overriding method in the subclass must have the same name as the method in the superclass that it is overriding.

Same Parameters: The overriding method must have the same number and types of parameters as the method in the superclass. This ensures compatibility and consistency with the method signature defined in the superclass.

IS-A Relationship (Inheritance): Method overriding requires an IS-A relationship between the subclass and the superclass. This means that the subclass must inherit from the superclass, either directly or indirectly, to override its methods.

Same Return Type or Covariant Return Type: The return type of the overriding method can be the same as the return type of the overridden method in the superclass, or it can be a subtype of the return type in the superclass. This is known as the covariant return type, introduced in Java 5.

Access Modifier Restrictions: The access modifier of the overriding method must be the same as or less restrictive than the access modifier of the overridden method in the superclass. Specifically, a method declared as public in the superclass can be overridden as public or protected but not as private. Similarly, a method declared as protected in the superclass can be overridden as protected or public but not as private. A method declared as default (package-private) in the superclass can be overridden with default, protected, or public, but not as private.

No Final Methods: Methods declared as final in the superclass cannot be overridden in the subclass. This is because final methods cannot be modified or extended.

No Static Methods: Static methods in Java are resolved at compile time and cannot be overridden. Instead, they are hidden in the subclass if a method with the same signature is defined in the subclass.

Example:

```
class Vehicle{

  void run(){System.out.println("Vehicle is running");}

}

//Creating a child class

class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Output:

```
Vehicle is running
```

Code:

```
class Stack{
    int[] ele;
    int top;

    void initStack(int size){
        ele=new int[size];
        top=-1;
    }

    void initStack(Stack another){
        ele=new int[another.ele.length];
        top=-1;
        for(int item:another.ele)
            push(item);
    }

    void initStack(int[] a){
        ele=new int[a.length];
        top=-1;
        for(int item:a)
            push(item);
    }

    void push(int item){
        if(top<ele.length){
            ele[++top]=item;
            System.out.println("Pushed element is "+item);
        }
        else
            System.out.println("Stack overflow");
    }

    int pop(){
        if(top==-1){
            System.out.println("Stack underflow");
            return -1;
        }
        else{
            int item=ele[top--];
            return item;
        }
    }
    int peek(){
        return ele[top];
    }
}
public class TW5a {
    public static void main(String[] args) {
        Stack s1=new Stack();
        Stack s2=new Stack();
        s1.initStack(5);
        s1.push(10);
        s1.push(20);
        s1.push(30);
        s1.push(40);
        s1.push(50);
        s2.initStack(s1);
        int[] array={1,2,3,4};
        Stack s3=new Stack();
        s3.initStack(array);
        System.out.println("Popped element in S1 object is "+s1.pop());
        System.out.println("Element on top of the stack of object s1 is "+s1.peek());
        System.out.println("Element on top of the stack of object s2 is "+s2.peek());

    }
}
```

Output

```
Pushed element is 10

Pushed element is 20

Pushed element is 30

Pushed element is 40

Pushed element is 50

Pushed element is 10

Pushed element is 20

Pushed element is 30

Pushed element is 40

Pushed element is 50

Pushed element is 1

Pushed element is 2

Pushed element is 3

Pushed element is 4

Popped element in S1 object is 50

Element on top of the stack of object s1 is 40

Element on top of the stack of object s2 is 50
```

TERMWORK 5b

Title: Write a program to demonstrate the implementation of method overriding

Problem Definition:

As a 3D modelling software developer that allows users to create and analyze various geometric shapes. One of the features of the software is to compute and display the properties of a cuboid, which is a rectangular prism with three dimensions: length, width, and height.

Requirements:

a) Rectangle Class: This class represents a 2D rectangle and includes methods to compute the area and perimeter of the rectangle.

b) Cuboid Class: This class is an extension of the Rectangle class to represent a 3D cuboid. It replaces the methods for computing area and perimeter to handle the surface area and perimeter of the cuboid. Additionally, it introduces a new method to compute the volume of the cuboid.

Functionality:

a) Compute Surface Area: Override the computeArea() method to calculate the surface area of the cuboid, which includes all six faces.

b) Compute Perimeter: Override the computePerimeter() method to calculate the perimeter of the cuboid's base (rectangle).

c) Compute Volume: Add a method to calculate the volume of the cuboid using the length, width, and height.

formula to find the surface area = 2(lw) + 2(hl) + 2(hw)

formula to find the perimeter = 2l + 2w

formula to find the volume = l x w x h

Introduction:

In object-oriented programming, method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. The method in the child class must have the same name, return type, and parameters as the method in the parent class.

Method overriding is used to achieve runtime polymorphism, where the type of object determines which version of the method will be called. It allows a subclass to modify the behaviour of a method to suit its own needs, while still maintaining the same interface as the superclass.

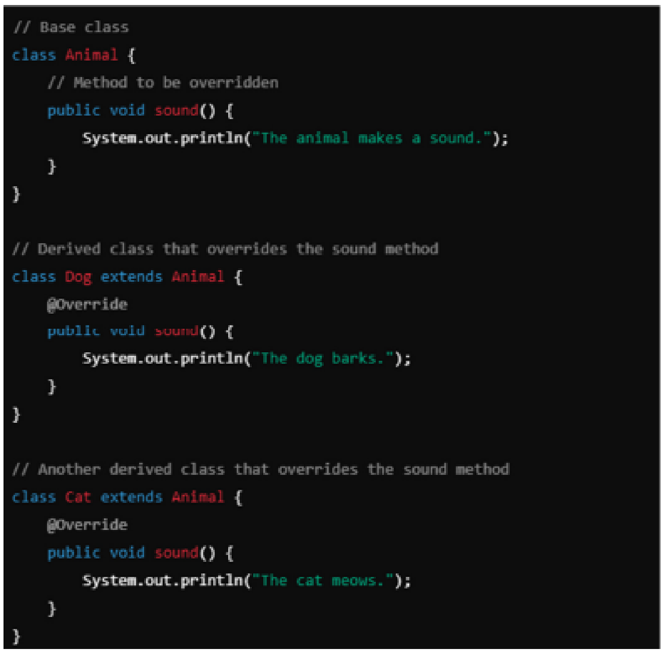
Key Points about Method Overriding:

1. Same Method Signature: The method in the child class must have the same name, return type, and parameter list as the method in the parent class.

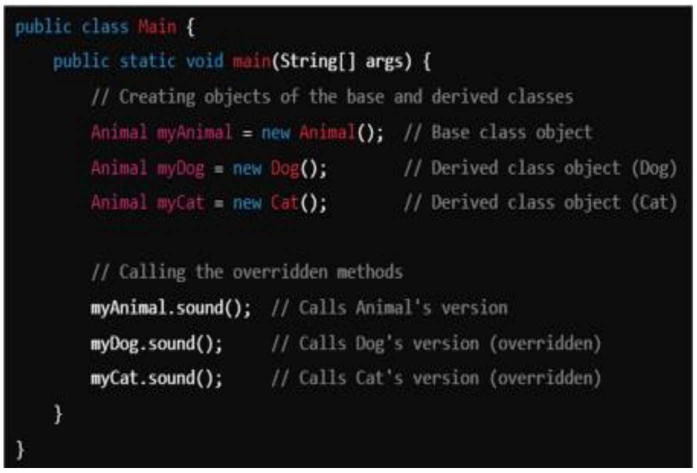
2. @Override Annotation (Optional): It's a good practice to use the @Override annotation to indicate that a method is overriding a superclass method.

3. Polymorphism: The object decides which method implementation to invoke at runtime, depending on its actual class.

Example:

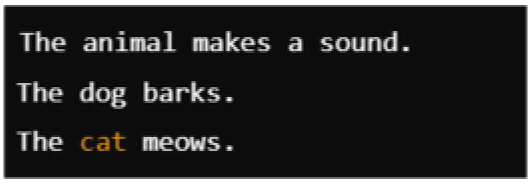


```java
// Base class
class Animal {
    // Method to be overridden
    public void sound() {
        System.out.println("The animal makes a sound.");
    }
}

// Derived class that overrides the sound method
class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("The dog barks.");
    }
}

// Another derived class that overrides the sound method
class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("The cat meows.");
    }
}
```



```java
public class Main {
    public static void main(String[] args) {
        // Creating objects of the base and derived classes
        Animal myAnimal = new Animal();  // Base class object
        Animal myDog = new Dog();        // Derived class object (Dog)
        Animal myCat = new Cat();        // Derived class object (Cat)

        // Calling the overridden methods
        myAnimal.sound(); // Calls Animal's version
        myDog.sound();    // Calls Dog's version (overridden)
        myCat.sound();    // Calls Cat's version (overridden)
    }
}
```

Output:



```
The animal makes a sound.
The dog barks.
The cat meows.
```

Code

```
class Rectangle {
    double length;
    double width;
    Rectangle() {
        length = 1.0;
        width = 1.0;
    }
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    double computeArea() {
        return length * width;
    }
    double computePerimeter() {
        return 2 * (length + width);
    }
}
class Cuboid extends Rectangle {
    double height;
    Cuboid() {
        super();
        height = 1.0;
    }
    Cuboid(double length, double width, double height) {
        super(length, width);
        this.height = height;
    }
```

```
  @Override
   double computeArea() {
       return 2 * ((length * width) + (width * height) + (length * height));
   }
  @Override
   double computePerimeter() {
       return 4 * (length + width + height);
   }
   double computeVolume() {
       return length * width * height;
   }
}
public class TW5b{
   public static void main(String[] args) {
       Rectangle r1 = new Rectangle();
       System.out.println("Rectangle 1:");
       System.out.println("Area:" + r1.computeArea());
       System.out.println("Perimeter:" + r1.computePerimeter());
       Rectangle r2 = new Rectangle(10,30);
       System.out.println("\nRectangle 2:");
       System.out.println("Area:" + r2.computeArea());
       System.out.println("Perimeter:" + r2.computePerimeter());
       Cuboid c1=new Cuboid();
       System.out.println("\nCuboid 1:");
       System.out.println("Area:" + c1.computeArea());
       System.out.println("Perimeter:" + c1.computePerimeter());
       System.out.println("Volume:" + c1.computeVolume());
       Cuboid c2=new Cuboid(10,30,40);
```

```
        System.out.println("\nCuboid 2:");
        System.out.println("Surface area:" + c2.computeArea());
        System.out.println("Perimeter:" + c2.computePerimeter());
        System.out.println("Volume:" + c2.computeVolume());
    }
}
```

Output

```
Rectangle 1:
Area:1.0
Perimeter:4.0

Rectangle 2:
Area:300.0
Perimeter:80.0

Cuboid 1:
Area:6.0
Perimeter:12.0
Volume:1.0

Cuboid 2:
Surface area:3800.0
Perimeter:320.0
Volume:12000.0
```

TERMWORK 6

Title: Write a program to demonstrate Run-time Polymorphism.

Problem definition: You are working on a fleet management system for a car rental service. The system needs to handle different types of vehicles and manage their operations. Your task is to implement a class hierarchy that allows the system to support various car brands and their unique starting and steering mechanisms. Additionally, the system should be able to handle multiple drivers interacting with different types of cars, demonstrating run-time polymorphism.

Requirements:

1. Abstract Class:

- Car: An abstract class representing a general car with attributes carName,
- chassiNum, and modelName. It should include two abstract methods:
- startCar(): Method to initiate the car's engine.
- operateSteering(): Method to control the car's steering mechanism.

2. Concrete Classes:

- MarutiCar: Inherits from Car and provides specific implementations for startCar() and operateSteering() relevant to Maruti cars.
- BmwCar: Inherits from Car and provides specific implementations for startCar() and operateSteering() relevant to BMW cars.

3. Driver Class:

- Driver: Represents a driver with attributes driverName, gender, and age. It includes

a method driveCar() that takes a Car abstract class reference and performs the

operations startCar() and operateSteering() to showcase run-time polymorphism.

4. Additional Functionality:

- Service Management: Implement functionality to log and track car services, such

as oil changes and tire rotations, based on car type and driver interactions.

Introduction:

Runtime polymorphism in Java is also popularly known as Dynamic Binding or Dynamic Method Dispatch. In object-oriented programming (OOP), run-time polymorphism is a concept where the method that gets executed is determined at run time rather than compile time. This is typically achieved through method overriding, which allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

Key points:

1. Method Overriding: The subclass defines a method with the same name, return type, and parameters as a method in its parent class.

2. Dynamic Method Dispatch: The actual method that gets called is resolved during runtime, based on the object type (not the reference type).

Here's a basic example in Java that demonstrates run-time polymorphism through method

Example:

```
class Parent {
   void show() { System.out.println("Parent's show()"); }
}
class Child extends Parent {
      @Override void show()
   {
      System.out.println("Child's show()");   }
}
class Main {
   public static void main(String[] args)
   {
      Parent obj1 = new Parent();
      obj1.show();
      Parent obj2 = new Child();
      obj2.show();
   }
```

}

- Upcasting happens when a variable of type Animal refers to the Dog object.
- You can call the sound() method, and it will run the Dog class version because Dog
- overrides it. This demonstrates polymorphism.
- You cannot call the wagTail() method because the reference type (Animal) does not know
- about this method. Only methods available in the Animal class are accessible when upcasting.

Algorithm / Implementation Steps:

1. Create Abstract Class Car:

- Attributes: carName, chassiNum, and modelName.
- Abstract methods: startCar() and operateSteering().

2. Create Concrete Classes MarutiCar and BmwCar:

- These classes will inherit from Car and implement their own versions of the startCar() and operateSteering() methods.

3. Create Driver Class:

- Attributes: driverName, gender, and age.
- Method driveCar(), that takes a Car reference which calls startCar() and operateSteering() to demonstrate run-time polymorphism.

4. Service Management:

- Create a class ServiceManagement to log and track services like oil changes and tire rotations. This class has a static method logService() that takes a Car object and logs services.

5. Main Class to Test the Implementation:

In the main class, create instances of different cars and drivers, then use the driveCar()

- method to demonstrate run-time polymorphism. You can also log services using
- ServiceManagement.

Code:

```
abstract class Car{
    String carName,modelName;
    int chassiNum;
    Car(String carName,int chassiNum,String modelName){
        this.carName=carName;
        this.chassiNum=chassiNum;
        this.modelName=modelName;
    }
    abstract void startCar();
    abstract void operateSteering();
    void display(){
        System.out.println("Car Name:"+carName);
        System.out.println("Chassi number:"+chassiNum);
        System.out.println("Model Name:"+modelName);
    }
}
class MarutiCar extends Car{
   MarutiCar(String carName, int chassiNum, String modelName) {
        super(carName, chassiNum, modelName);
    }
    void startCar(){
        System.out.println("Starting a Maruti car....");
    }
    void operateSteering(){
        System.out.println("This car is manually steered.....");
    }
}
```

```
class BmwCar extends Car{
    BmwCar(String carName, int chassiNum, String modelName) {
        super(carName, chassiNum, modelName);
    }
    void startCar(){
        System.out.println("Starting a BMW car....");
    }
    void operateSteering(){
        System.out.println("This car is automatically steered.....");
    }
}
class Driver{
    String name,gender;
    int age;
    Driver(String name,int age,String gender){
        this.name=name;
        this.age=age;
        this.gender=gender;
    }
    void driveCar(Car obj){
            System.out.println("Driver:"+name);
        System.out.println("Age:"+age);
        System.out.println("Gender:"+gender);
        obj.display();
        obj.startCar();
        obj.operateSteering();
    }
}
public class Tw6 {
    public static void main(String []args){
        MarutiCar m=new MarutiCar("Suzuki",1253,"A21s");
        BmwCar b=new BmwCar("BMW5",4596,"S5");
        Driver d1=new Driver("Vishal",25,"Male");
        d1.driveCar(m);
        System.out.println();
        Driver d2=new Driver("Priya",23,"Female");
        d2.driveCar(b);
    }
}
```

Output

```
Driver:Vishal
Age:25
Gender:Male
Car Name:Suzuki
Chassi number:1253
Model Name:A21s
Starting a Maruti car....
This car is manually steered.....

Driver:Priya
Age:23
Gender:Female
Car Name:BMW5
Chassi number:4596
Model Name:S5
Starting a BMW car....
This car is automatically steered.....
```

TERMWORK 7

Title: Write a program to demonstrate the implementation of interfaces.

Problem Definition:Write a Java application to implement the following UML diagram.

● PrimeTester class implements isPrime() method by iterating from 2 to n-1 for a given number

● ImprPrimeTester class implements isPrime() method by iterating from 2 to n/2

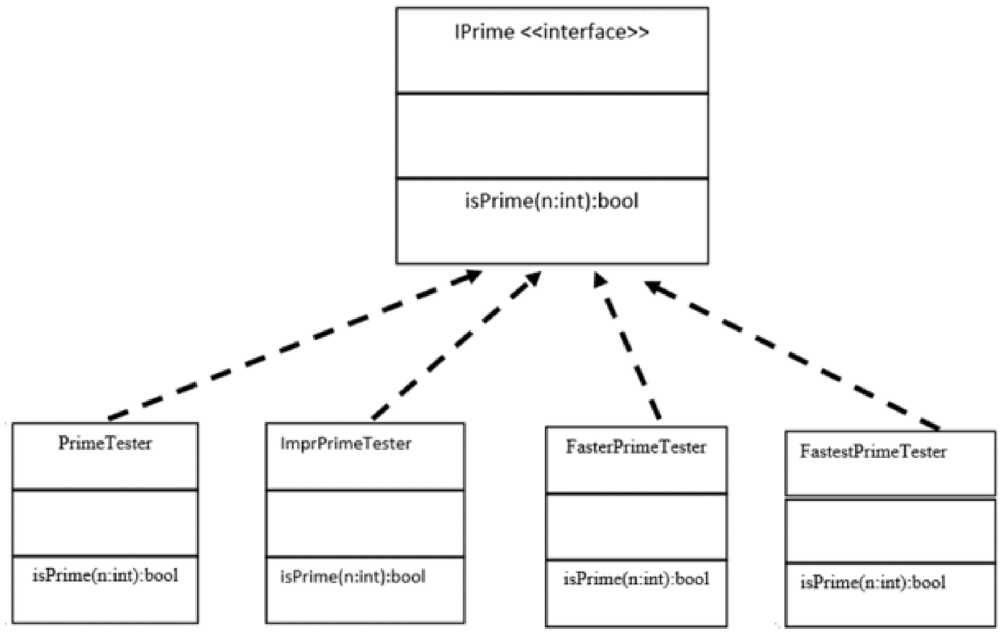
● FasterPrimeTester class implements isPrime() method by iterating from 2 to

● FastestPrimeTester class implements isPrime() method using Fermat's Little theorem.

○ Fermat's Little Theorem:

○ If n is a prime number, then for every a, 1 < a < n-1,

■ a n-1 % n = 1



Introduction:

An interface is an abstract "class" that is used to group related methods with "empty" bodies:

To access the interface methods, the interface must be "implemented" (kind of like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class.

Example:

The implements keyword is used to implement an interface.

It cannot be used to create objects (in the example above, it is not possible to create an “Animal” object in the MyMainClass).

Interface methods does not have a body – the body is provided by the “implement" class.

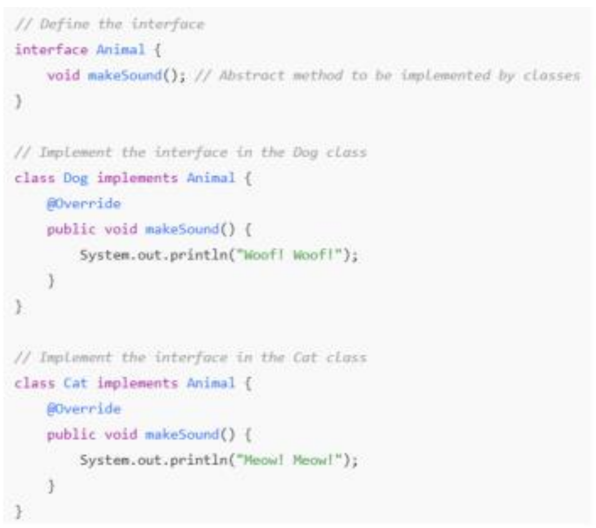
On implementation of an interface, you must override all of its methods.

Interface methods are by default abstract and public.

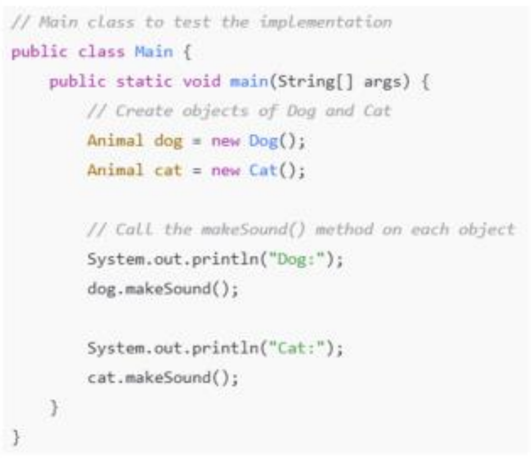
Interface attributes are by default public, static and final.

An interface cannot contain a constructor (as it cannot be used to create objects)

Ex:



```java
// Define the interface
interface Animal {
    void makeSound(); // Abstract method to be implemented by classes
}

// Implement the interface in the Dog class
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }
}

// Implement the interface in the Cat class
class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow! Meow!");
    }
}
```



```java
// Main class to test the implementation
public class Main {
    public static void main(String[] args) {
        // Create objects of Dog and Cat
        Animal dog = new Dog();
        Animal cat = new Cat();

        // Call the makeSound() method on each object
        System.out.println("Dog:");
        dog.makeSound();

        System.out.println("Cat:");
        cat.makeSound();
    }
}
```

Algorithm / Implementation Steps:

1. Define the Interface IPrime

Create the IPrime interface with a method declaration for isPrime.

2. Implement the PrimeTester Class

This class implements the IPrime interface and provides a basic method to test if a number is prime.

3. Implement the ImprPrimeTester Class

This class improves the basic prime testing method by reducing unnecessary iterations.

4. Implement the FasterPrimeTester Class

This class optimizes the method further using efficient algorithms like trial division.

5. Implement the FastestPrimeTester Class

This class uses an advanced algorithm like using Fermat’s Little theorem or probabilistic methods for testing primality.

6. Test the Implementation

Create a main class to test all the implementations.

Code:

```
interface IsPrime{
   boolean isPrime(int n);
}
class PrimeTester implements IsPrime{
   @Override
   public boolean isPrime(int n){
      boolean flag=true;
      for(int i=2; i<=n-1; i++){
         if((n%i)==0)
         {
            flag=false;
            break;
         }
      }
      return flag;
   }
}
```

```
class ImprPrimeTester implements IsPrime{
    @Override
    public boolean isPrime(int n){
        boolean flag=true;
        for(int i=2; i<=n/2; i++){
            if((n%i)==0)
            {
                flag=false;
                break;
            }
        }
        return flag;
    }
}
class FasterPrimeTester implements IsPrime{
    @Override
    public boolean isPrime(int n){
        boolean flag=true;
        for(int i=2; i<=Math.sqrt(n); i++){
            if((n%i)==0)
            {
                flag=false;
                break;
            }
        }
        return flag;
    }
}
class FastestPrimeTester implements IsPrime{
    @Override
    public boolean isPrime(int n){
        int a=2;
        return Math.pow(a,n-1)%n==1;
    }
}
public class TW7 {
    public static void main(String[] args) {
       PrimeTester p1=new PrimeTester();
       ImprPrimeTester p2=new ImprPrimeTester();
       FasterPrimeTester p3=new FasterPrimeTester();
       FastestPrimeTester p4=new FastestPrimeTester();
       System.out.println("32 is Prime? "+p1.isPrime(32));
       System.out.println("17 is Prime? "+p1.isPrime(17));
       System.out.println("32 is Prime? "+p2.isPrime(32));
       System.out.println("17 is Prime? "+p2.isPrime(17));
       System.out.println("32 is Prime? "+p3.isPrime(32));
       System.out.println("17 is Prime? "+p3.isPrime(17));
       System.out.println("32 is Prime? "+p4.isPrime(32));
       System.out.println("17 is Prime? "+p4.isPrime(17));

    }
}
```

Output

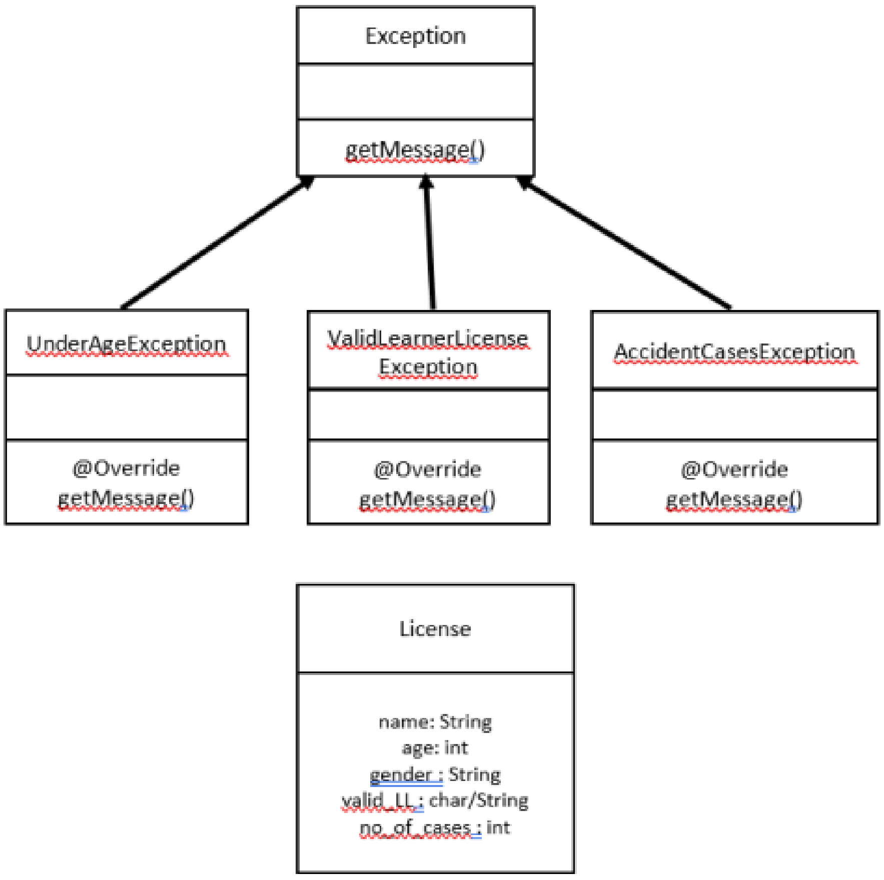
```
32 is Prime? false
17 is Prime? true
32 is Prime? false
17 is Prime? true
32 is Prime? false
17 is Prime? true
32 is Prime? False
17 is Prime? True
```

TERMWORK 8

Title: Write a program to demonstrate the implementation of customized exception handling.

Problem Definition: Assume that you have received a request from the transport authority for automating the task of issuing the permanent license for two wheelers. The mandatory condition to issue the license are: 1) the applicant must be over 18 years of age and 2) holder of a valid learner's license and 3) no accident cases in the last one year.

Write a Java program that reads user details as required (use the Scanner class). Create user defined exceptions to check for the three conditions imposed by the transport authority. Based on the inputs entered by the user, decide and display whether or not a license has to be issued or an error message as defined by the user exception.

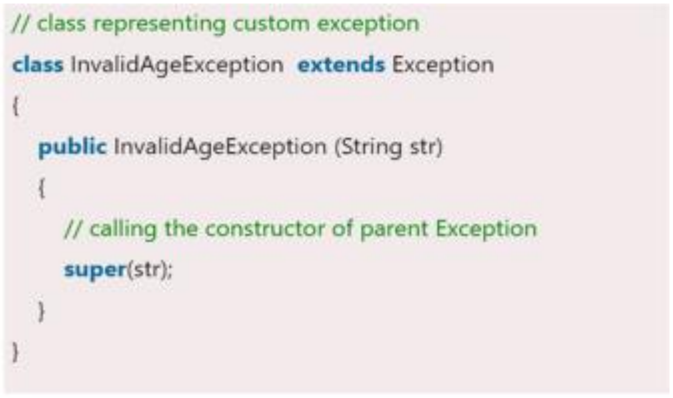


Introduction:

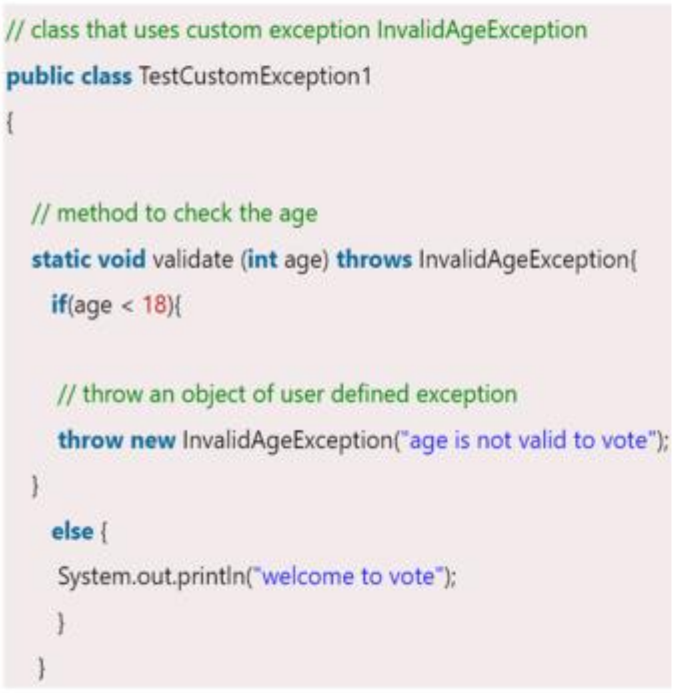
In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as a custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

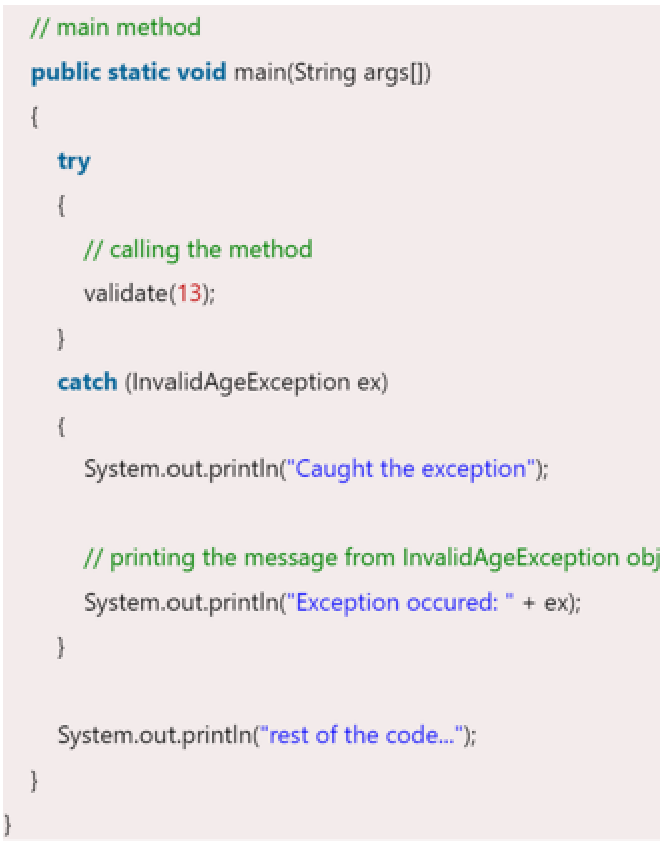
Using the custom exception, we can have our own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getM essage() method on the object we have created.



```
// class representing custom exception
class InvalidAgeException  extends Exception
{
  public InvalidAgeException (String str)
  {
    // calling the constructor of parent Exception
    super(str);
  }
}
```



```
// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

  // method to check the age
  static void validate (int age) throws InvalidAgeException{
    if(age < 18){

      // throw an object of user defined exception
      throw new InvalidAgeException("age is not valid to vote");
  }
    else {
      System.out.println("welcome to vote");
    }
  }
```



```
    // main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");

            // printing the message from InvalidAgeException obj
            System.out.println("Exception occured: " + ex);
        }

        System.out.println("rest of the code...");
    }
}
```

In the above code, constructor of InvalidAgeException takes a string as an argument. This string is passed to the constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

Algorithm / Implementation Steps:

1. Create License Class:

- Define attributes: name, age, gender, validLLR, noOfAccidents.
- Add a constructor to initialize these fields.

2. Create Driver Class with main Method:

- Read user input for name, age, gender, validLLR, and noOfAccidents.
- Create an instance of the License class.
- Call validateApplicant(applicant) method.

3. Define validateApplicant(applicant) Method:

- Check conditions:
- If age < 18, throw UnderAgeException.
- If validLLR is false, throw ValidLLRException.
- If noOfAccidents > 0, throw NumAccidentsException.
- If all conditions are met, print "License can be issued."

4. Create Custom Exception Classes:

- UnderAgeException: Thrown if age < 18.
- ValidLLRException: Thrown if validLLR is false.
- NumAccidentsException: Thrown if noOfAccidents > 0.
- Each exception should pass a custom error message.

5. Handle Exceptions in main:

- Use a try-catch block to catch and display exception messages.

6. Test the Program:

- Validate with various inputs to check correctness and handle errors.

Code:

```
import java.util.Scanner;
class UnderAgeException extends Exception{
   UnderAgeException(String s){
      super(s);
   }
   @Override
   public String toString(){
      return "Sorry. You are too young for the license";
   }
}
class ValidLLR extends Exception{
   ValidLLR(String s){
      super(s);
   }
   @Override
   public String toString(){
      return "Sorry. You do not hold a valid LLR";
   }
}
class NumAccidents extends Exception{
   NumAccidents(String s){
      super(s);
   }
   @Override
   public String toString(){
      return "Sorry. There are accidents in last one year";
   }
```

```
}
class License{
   String name;
   int age, no_of_cases;
   char gender;
   char validLLR;
   void readData(){
      Scanner in=new Scanner(System.in);
      System.out.println("Enter the name: ");
      name = in.nextLine();
      System.out.println("Enter the age: ");
      age = in.nextInt();
      System.out.println("Enter the gender: ");
      gender = in.next().charAt(0);
      System.out.println("Do you have Valid LLR (Y/N)? ");
      validLLR = in.next().charAt(0);
      System.out.println("How many number of cases in past one year? ");
      no_of_cases = in.nextInt();
   }
}
public class TW8 {
   public static void main(String[] args) {
      License applicant =new License();
      applicant.readData();
      validateApplicant(applicant);
   }
   static void validateApplicant(License a){
     try{
        if(a.age<18)
           throw new UnderAgeException("Underageexception:");
        if(a.validLLR!='Y')
           throw new ValidLLR("ValidLLRexception:");
        if(a.no_of_cases>0)
           throw new NumAccidents("Numberofaccidentsexception:");
        System.out.println("Congrats!! Your license is being posted");
     }
     catch(UnderAgeException e){
        System.out.println(e.getMessage()+e);
     }
     catch(ValidLLR e){
        System.out.println(e.getMessage()+e);
     }
     catch(NumAccidents e){
            System.out.println(e.getMessage()+e);
        }
        catch(Exception e){
            System.out.println(e.getMessage()+e);
        }
    }
}
```

Output

```
Enter the name:
abe
Enter the age:
15
Enter the gender:
M
Do you have Valid LLR (Y/N)?
N
How many number of cases in past one year?
0
Underageexception:Sorry. You are too young for the license

Enter the name:
pqr
Enter the age:
19
Enter the gender:
M
Do you have Valid LLR (Y/N)?
Y
How many number of cases in past one year?
0
Congrats!! Your license is being posted
```

TERMWORK 9

Title: Write a program to demonstrate the implementation of Collection framework and interfaces

Problem Definition: You are developing a feature for a student management system that uses an ArrayList to keep track of various student names. The system needs to perform multiple operations on the list, such as displaying names, modifying the list, and sorting it. Additionally, you need to implement search functionality to check for the presence of specific names.

In your student management system, you need to:

1. Maintain and Update the Student List:
   - Start with an initial set of student names and be able to modify it as needed.
2. Display and Modify List:
   - Allow for displaying the list, modifying it (adding/removing names), and viewing the updated list. (use Iterator/ListIterator)
3. Sort and Search:
   - Ensure that the list can be sorted for display purposes and that you can quickly find whether a student name is present.

1. **Collection Interface**

- The root interface of the Collection Framework.
- It represents a group of objects, known as elements.
- Subinterfaces include List, Set, and Queue.

---

2. **List Interface**

- An ordered collection (sequence) that allows duplicate elements.
- Provides positional access and insertion of elements.
- Examples:
  - **Implementations**: ArrayList, LinkedList, Vector
  - **Common Methods**:
    - add(int index, E element)
    - get(int index)
    - remove(int index)

3. **Set Interface**

- A collection that does not allow duplicate elements.
- Implements the mathematical concept of a set.
- Examples:
  - **Implementations**: HashSet, LinkedHashSet, TreeSet
  - **Special Features**:

        - TreeSet is sorted.
        - HashSet uses a hash table for storage.

4. **Queue Interface**

- Represents a collection designed for holding elements prior to processing.
- Typically follows **FIFO** (First In, First Out).
- Examples:
    - **Implementations**: PriorityQueue, ArrayDeque
    - **Variants**: Deque (Double-Ended Queue)

5. **Map Interface**

- Not a subinterface of Collection, but part of the framework.
- Represents a collection of key-value pairs.
- Examples:
    - **Implementations**: HashMap, LinkedHashMap, TreeMap
    - **Key Methods**:
        - put(K key, V value)
        - get(Object key)
        - remove(Object key)

Key Characteristics of Interfaces

Interface	**Ordered?**	**Allows Duplicates?**	**Key Features**
List	Yes	Yes	Positional access, index-based
Set	No	No	Unique elements, no duplicates
Queue	Yes	Depends	FIFO or priority-based processing
Map	No	Keys - No, Values - Yes	Key-value pairs, unique keys

Advantages of the Collection Framework

1. **Standardization**: Provides a common way to manipulate data structures.
2. **Efficiency**: Optimized implementations for common tasks.
3. **Flexibility**: Interchangeable implementations via interfaces.
4. **Ease of Use**: Readily available utility methods like sorting, searching, etc.

Code:

```
import java.util.*;
public class IteratorDemo {
   public static void main(String[] args) {
      ArrayList<String> al = new ArrayList<>();
      al.add("Alpha");
      al.add("Beta");
      al.add("Gamma");
      al.add("Delta");
      al.add("Epsilon");
      al.add("Zeta");
      al.add("Eta");

      System.out.println("Original Contents");
      Iterator<String> i = al.iterator();
      while(i.hasNext())
         System.out.print(i.next() + " ");
      System.out.println("\n");
      //to remove Gamma from the list
      i = al.iterator();
      while(i.hasNext()){
         if(i.next().equals("Gamma"))
            i.remove();
      }
      System.out.println("Contents after deletion");
      i=al.iterator();
      while(i.hasNext())
         System.out.print(i.next() + " ");
      System.out.println("\n");
      //to add Gamma back to the list
      ListIterator<String> li = al.listIterator();
      while(li.hasNext()){
         if(li.next().equals("Beta"))
         li.add("Gamma");
      }
      System.out.println("Contents after addition ");
      li = al.listIterator();
      while(li.hasNext())
         System.out.print(li.next() + " ");
      System.out.println("\n");
      //to modify the objects
      String str;
      li = al.listIterator();
```

```
while(li.hasNext()){
   str = li.next();
   switch (str) {
      case "Eta":
         li.set("Omega");
         break;
      case "Zeta":
         li.set("Psi");
         break;
      case "Epsilon":
         li.set("Chi");
         break;
      case "Delta":
         li.set("Dlt");
         break;
      default:
         break;
   }
}
System.out.println("Contents after changes ");
li= al.listIterator();
while(li.hasNext())
   System.out.print(li.next() + " ");
System.out.println("\n");

//ListIterator to display the backwards
System.out.println("Modified list backwards ");
while(li.hasPrevious()){
   System.out.print(li.previous() + " ");
}
System.out.println("\n");
System.out.println("Array elements before sorting ");
for(String s:al)
   System.out.print(s+" ");
System.out.println("\n");
System.out.println("Array elements after sorting in ascending order ");
Collections.sort(al);
for(String s:al)
   System.out.print(s+" ");
System.out.println("\n");
System.out.println("Array elements after sorting in descending order ");
Collections.sort(al,Collections.reverseOrder());
for(String s:al)
```

```
            System.out.print(s+" ");
        System.out.println("\n");

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the element to be searched in the list: ");
        String key = sc.nextLine();
        for(String s:al){
            if(s.equals(key)){
                System.out.println("Element found");
                return;
            }
        }
        System.out.println("Element not found");
    }

}
```

Output

```
Original Contents
Alpha Beta Gamma Delta Epsilon Zeta Eta

Contents after deletion
Alpha Beta Delta Epsilon Zeta Eta

Contents after addition
Alpha Beta Gamma Delta Epsilon Zeta Eta

Contents after changes
Alpha Beta Gamma Dlt Chi Psi Omega

Modified list backwards
Omega Psi Chi Dlt Gamma Beta Alpha

Array elements before sorting
Alpha Beta Gamma Dlt Chi Psi Omega

Array elements after sorting in ascending order
Alpha Beta Chi Dlt Gamma Omega Psi

Array elements after sorting in descending order
Psi Omega Gamma Dlt Chi Beta Alpha

Enter the element to be searched in the list:
```

```
Alpha
Element found

Enter the element to be searched in the list:
xyz
Element not found
```

TERMWORK 10

Title: Write a program to demonstrate the implementation of Lambda Expression

Problem Definition: You are developing a financial software system for a company that needs to calculate taxes for employees based on their income. The system must apply different tax rates depending on the income brackets and determine if the calculated tax is below a certain threshold.

Requirements:

1. Tax Calculation:
   - Implement a TaxCalculator interface that includes a method calculateTax(double
   - income) which returns the tax amount based on the following criteria:
   - ▪ For income ≤ 50,000: Tax is income * 0.1
   - ▪ For 50,000 < income ≤ 100,000: Tax is income * 0.2
   - ▪ For income > 100,000: Tax is income * 0.3
2. Tax Threshold Check:
   - Use the Predicate interface to create a predicate isTaxLessThan1000 that takes a tax amount and returns true if the tax is less than 1000, and false otherwise.
3. Process Employee Data:
   - Define an array with the incomes of 5 employees.
   - For each employee's income, use the TaxCalculator to compute the tax and then
   - use the Predicate to check if the computed tax is less than 1000.
   - Print the tax amount and the result of the predicate check for each employee.

Use Case: In the financial software system, the goal is to:
1. Calculate Taxes: Apply specific tax rules based on income brackets to compute the tax for each employee.
2. Check Tax Amount: Determine if the computed tax is below a certain threshold to make further financial decisions.
3. Process and Display Results: Efficiently handle and display tax calculations and checks for multiple employees.

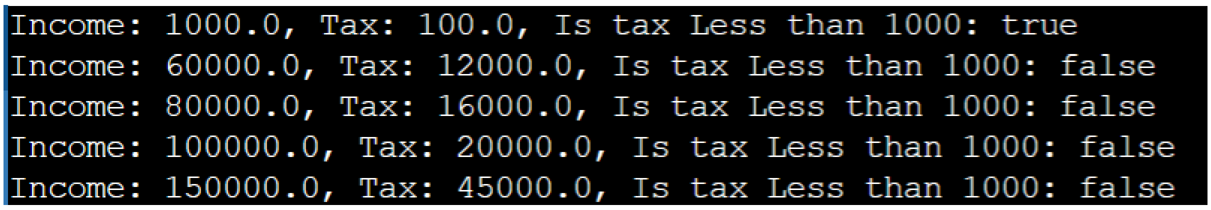
Algorithm / Implementation Steps:

1. Define the TaxCalculator Interface:
    - Create an interface TaxCalculator with a method double calculateTax(double income) that will return the tax amount based on the income.
2. Implement the TaxCalculator Interface:
    - Create a class TaxCalculatorImpl that implements the TaxCalculator interface.
    - Inside the calculateTax method, apply the tax rates based on the provided income:
    - □ If income ≤ 50,000, tax = income * 0.1.
    - □ If 50,000 < income ≤ 100,000, tax = income * 0.2.
    - □ If income > 100,000, tax = income * 0.3.
3. Define the Predicate Interface for Threshold Check:
    - Create a Predicate<Double> interface called isTaxLessThan1000 to check if the calculated tax is below 1000.
    - Implement the test method of Predicate to return true if the tax amount is less than 1000, and false otherwise.
4. Define Employee Data and Calculate Taxes:
    - Create an array (or list) to store the incomes of 5 employees.
    - For each employee, use the TaxCalculator to calculate the tax based on their income.
    - Use the Predicate to check if the computed tax is less than 1000.
5. Print the Results:
    - For each employee, print the tax amount and the result of whether the tax is below the threshold of 1000.

Code

```
import java.util.function.Predicate;  //This line of code is importing the Predicate interface
                                                              from the Java.util.function package.
  interface TaxCalculator {
  double calculateTax(double income);
}
public class Main {
 public static void main(String[] args) {
 TaxCalculator taxCal = (double income) -> {
     if (income <= 50000) {
      return income * 0.1;
    }
else if (income > 50000 && income <= 100000) {
          return income * 0.2;    }
else{
   return income * 0.3;
   }
}  ;
//  The Predicate interface represents a single argument function that returns a boolean value.
Predicate<Double> isTaxLessThan1000 = (tax) -> tax < 1000;
double[] incomes = {1000, 60000, 80000, 100000, 150000};
 for (double x : incomes) {
double tax = taxCal.calculateTax(x);
      System.out.println("Income: " + x + ", Tax: " + tax + ", Is Tax Less than 1000: " +
isTaxLessThan1000.test(tax));
     }
 }
}
```

Output



```
Income: 1000.0, Tax: 100.0, Is tax Less than 1000: true
Income: 60000.0, Tax: 12000.0, Is tax Less than 1000: false
Income: 80000.0, Tax: 16000.0, Is tax Less than 1000: false
Income: 100000.0, Tax: 20000.0, Is tax Less than 1000: false
Income: 150000.0, Tax: 45000.0, Is tax Less than 1000: false
```