**PART-4**

|  |  |
| --- | --- |
| **NO.** | **AIM OF THE PRACTICAL** |
| 17. | Create a class with a method that prints "This is parent class" and its subclass with another method that prints "This is child class". Now, create an object for each of the class and call 1 - method of parent class by object of parent  **PROGRAM:**  public class Main\_17 {     public Main\_17() {     }     public static void main(String[] var0) {        Parent var1 = new Parent();        var1.printParent();        Child var2 = new Child();        var2.printChild();     }  }  **OUTPUT:**    **CONCLUSION:**  By creating an object of the Parent class and calling its method, we demonstrate that the method of the parent class can be called directly using the parent object. This showcases how object-oriented programming works, where a class contains methods that can be accessed through its objects. |

|  |  |
| --- | --- |
| **NO.** | **AIM OF THE PRACTICAL** |
| 18. | Create a class named 'Member' having the following members: Data members  1 - Name  2 - Age  3 - Phone number  4 - Address  5 – Salary  It also has a method named 'printSalary' which prints the salary of the members. Two classes 'Employee' and 'Manager' inherits the 'Member' class. The 'Employee' and 'Manager' classes have data members 'specialization' and  'department' respectively. Now, assign name, age, phone number, address and salary to an employee and a manager by making an object of both of these classes and print the same.  **PROGRAM:**  // Member class  class Member  {      String name;      int age;      String phoneNumber;      String address;      double salary;      void printSalary()      {          System.out.println("Salary: " + salary);      }  }  class Employee extends Member {      String specialization;      void displayEmployeeDetails() {          System.out.println("Employee Details:");          System.out.println("Name: " + name);          System.out.println("Age: " + age);          System.out.println("Phone Number: " + phoneNumber);          System.out.println("Address: " + address);          System.out.println("Specialization: " + specialization);          printSalary();      }  }  class Manager extends Member {      String department;      void displayManagerDetails() {          System.out.println("Manager Details:");          System.out.println("Name: " + name);          System.out.println("Age: " + age);          System.out.println("Phone Number: " + phoneNumber);          System.out.println("Address: " + address);          System.out.println("Department: " + department);          printSalary();      }  }  public class prac18 {      public static void main(String[] args) {          Employee employee = new Employee();          employee.name = "JNEIL";          employee.age = 20;          employee.phoneNumber = "9727022105";          employee.address = "ahemdabad";          employee.salary = 50000.0;          employee.specialization = "Software Engineering";          employee.displayEmployeeDetails();          System.out.println();          Manager manager = new Manager();          manager.name = "preet ";          manager.age = 21;          manager.phoneNumber = "8320201710";          manager.address = "rajkot";          manager.salary = 70000.0;          manager.department = "blockchain expert";          manager.displayManagerDetails();      }  }  **OUTPUT:**    **CONCLUSION:**  This design allows efficient management of common attributes while providing flexibility to add specialized characteristics to different types of members. |

|  |  |
| --- | --- |
| **NO.** | **AIM OF THE PRACTICAL** |
| 19. | Create a class named 'Rectangle' with two data members 'length' and 'breadth' and two methods to print the area and perimeter of the rectangle respectively. Its constructor having parameters for length and breadth is used to initialize length and breadth of the rectangle. Let class 'Square' inherit the 'Rectangle' class with its constructor having a parameter for its side (suppose s) calling the constructor of its parent class as 'super(s,s)'. Print the area and perimeter of a rectangle and a square. Also use array of objects.  **PROGRAM:**  class Rectangle {      double length;      double breadth;      Rectangle(double length, double breadth) {          this.length = length;          this.breadth = breadth;      }      void printArea() {          System.out.println("Area : " + length \* breadth);      }      void printPerimeter() {          System.out.println("Perimeter : " + 2 \* (length + breadth));      }  }  class Square extends Rectangle {      Square(double side) {          super(side, side);      }  }  public class prac19 {      public static void main(String[] args) {          Rectangle[] rectangles = new Rectangle[2];          rectangles[0] = new Rectangle(5.0, 3.0);          System.out.println("Rectangle:");          rectangles[0].printArea();          rectangles[0].printPerimeter();          System.out.println();          rectangles[1] = new Square(4.0);          System.out.println("Square:");          rectangles[1].printArea();          rectangles[1].printPerimeter();      }  }  **OUTPUT:**    **CONCLUSION:**  This design highlights the benefits of inheritance, code reuse, and polymorphism in object-oriented programming. |

|  |  |
| --- | --- |
| **NO.** | **AIM OF THE PRACTICAL** |
| 20. | Create a class named 'Shape' with a method to print "This  is This is a shape". Then create two other classes named 'Rectangle', and 'Circle' inheriting the Shape class, both having a method to print "This is rectangular shape" and "This is circular shape" respectively. Create a subclass  'Square' of 'Rectangle' having a method to print "Square is a rectangle". Now call the method of the 'Shape' and 'Rectangle' class by the object of 'Square' class.  **PROGRAM:**  class Shape  {  void display()  {  System.out.println("This is the shape");  }  }  class Rectangle extends Shape  {  void displayRectangle()  {  System.out.println("This is a rectangular shape");  }  }  class Circle extends Shape  {  void displayCircle()  {  System.out.println("This is a circular shape");  }  }  class Square extends Rectangle  {  void displaySquare()  {  System.out.println("Square is a rectangle");  }  }  public class prac20  {  public static void main(String[] args)  {  Square s = new Square();  s.display();  s.displayRectangle();  }  }  **OUTPUT:**    **CONCLUSION:**  The example provided illustrates core object-oriented programming principles, including inheritance, polymorphism, method overriding, and encapsulation. By structuring classes in a hierarchical manner that reflects logical relationships between shapes, the design promotes code reusability, flexibility, and maintainability. The use of polymorphism and method resolution order ensures that the correct behavior is invoked for each specific object type, showcasing the power and versatility of object-oriented design. |

|  |  |
| --- | --- |
| **NO.** | **AIM OF THE PRACTICAL** |
| 21. | Create a class 'Degree' having a method 'getDegree' that prints "I got a degree". It has two subclasses namely 'Undergraduate' and 'Postgraduate' each having a method with the same name that prints "I am an Undergraduate" and "I am a Postgraduate" respectively. Call the method by creating an object of each of the three classes.  **PROGRAM:**  class Degree  {  void getDegree()  {  System.out.println("I got a degree!!");  }  }  class Postgraduate extends Degree  {  void getDegree()  {  System.out.println("I am a postgraduate!!");  }  }  class Undergraduate extends Degree  {  void getDegree()  {  System.out.println("I am an undergraduate!!");  }  }  public class prac21  {  public static void main(String[] args)  {  Degree d = new Degree();  Postgraduate p = new Postgraduate();  Undergraduate u = new Undergraduate();  d.getDegree();  p.getDegree();  u.getDegree();  }  }  **OUTPUT:**    **CONCLUSION:**  This demonstrates **polymorphism** in Python, where the method in the subclass overrides the method from the parent class when called on an instance of the subclass. |
|  |  |

|  |  |
| --- | --- |
| **NO.** | **AIM OF THE PRACTICAL** |
| 22. | Write a java that implements an interface AdvancedArithmetic which contains amethod signature int divisor\_sum(int n). You need to write a class  calledMyCalculator which implements the interface. divisorSum function just takes an integer as input and return the sum of all its divisors.  For example, divisors of 6 are 1, 2, 3 and 6, so divisor\_sum should return 12. The value of n will be at most 1000.  **PROGRAM:**  import java.util.\*;  interface AdvancedArithmetic {      int divisor\_sum(int n);    }  class calledMyCalculator implements AdvancedArithmetic{      public int divisor\_sum(int n){          int sum=0;          int sqrt =(int) Math.sqrt(n);          for(int i=1;i<=sqrt;i++){              if(n%i==0){                  sum+=i;                  if(i!=n/i){                      sum +=n/i;                  }              }          }          return sum;      }  }  class P22{      static Scanner sc=new Scanner(System.in);      public static void main(String args[]){       calledMyCalculator cmc=new calledMyCalculator();       System.out.println("enter the Number:");       int x=sc.nextInt();       System.out.println("sum:"+cmc.divisor\_sum(x));      }  }  **OUTPUT:**    **CONCLUSION:**  This design effectively demonstrates the implementation of interfaces in Java, as well as an efficient approach to divisor calculation. |

|  |  |
| --- | --- |
| **NO.** | **AIM OF THE PRACTICAL** |
| 23. | Assume you want to capture shapes, which can be either circles (with a radius and a color) or rectangles (with a length, width, and color). You also want to be able to create signs (to post in the campus center, for example), each of which has a shape (for the background of the sign) and the text (a String) to put on the sign. Create classes and interfaces for circles, rectangles, shapes, and signs.  Write a program that illustrates the significance of interface default method.  **PROGRAM:**  interface Shape {  String getColor();  double getArea();  default void displayInfo() {  System.out.println("Color: " + getColor());  System.out.println("Area: " + getArea());  }  }  class Circle implements Shape {  private double radius;  private String color;  public Circle(double radius, String color) {  this.radius = radius;  this.color = color;  }  @Override  public String getColor() {  return color;  }  @Override  public double getArea() {  return Math.PI \* radius \* radius;  }  }  class Rectangle implements Shape {  private double length;  private double width;  private String color;  public Rectangle(double length, double width, String color) {  this.length = length;  this.width = width;  this.color = color;  }  @Override  public String getColor() {  return color;  }  @Override  public double getArea() {  return length \* width;  }  }  class Sign {  private Shape shape;  private String text;  public Sign(Shape shape, String text) {  this.shape = shape;  this.text = text;  }  public void displaySignInfo() {  System.out.println("Sign Text: " + text);  shape.displayInfo();  }  }  public class prac23 {  public static void main(String[] args) {  Shape circle = new Circle(5, "Red");  Shape rectangle = new Rectangle(4, 7, "Blue");  Sign sign1 = new Sign(circle, "Welcome to Campus");  Sign sign2 = new Sign(rectangle, "Library Ahead");  System.out.println("Sign 1 Information:");  sign1.displaySignInfo();  System.out.println("\nSign 2 Information:");  sign2.displaySignInfo();  }  }**OUTPUT:**    **CONCLUSION:** |

**PART-V Exception Handling**

|  |  |
| --- | --- |
| **No.** | **Aim of the Practical** |
| 24. | **AIM : Write a java program which takes two integers x & y as input, you have to compute x/y. If x and y are not integers or if y is zero, exception will occur and you have to report it.**  **PROGRAM CODE :**  import java.util.Scanner;  public class Prac\_24 {  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  try {  System.out.print("Enter integer x: ");  int x = Integer.parseInt(scanner.nextLine());  System.out.print("Enter integer y: ");  int y = Integer.parseInt(scanner.nextLine());  int d = x / y;  System.out.println("Division " + d);  } catch (NumberFormatException e) {  System.out.println("Invalid input: Please enter valid integers.");  } catch (ArithmeticException e) {  System.out.println("Arithmetic error: Division by zero is not allowed.");  } finally {  scanner.close();  System.out.print("\nName : Sneh \nID : 23DCS093 ");  }  }  }  **OUTPUT:**    OUTPUT: PRACTICAL-24  **CONCLUSION:**  This code is a simple console application that prompts the user to input two integers and performs division. It includes error handling for invalid input and division by zero. The Scanner object is used for input and is properly closed in the finally block. The program also prints a name and ID at the end. This ensures the program handles common input errors gracefully while demonstrating basic exception handling in Java. |
| 25. | AIM : Write a Java program that throws an exception and catch it using a try-catch block. PROGRAM CODE : public class Prac\_25 {  public static void main(String[] args) {  try {  throw new Exception("This is a custom exception");  } catch (Exception e) {  System.out.println("Caught an exception: " + e.getMessage());  }  System.out.print("\nName : Sneh \nID : 23DCS093 ");  }  }  **OUTPUT:**    OUTPUT: PRACTICAL-25  **CONCLUSION:**  This java code demonstrates basic exception handling. It intentionally throws a custom exception and catches it, printing the exception message. The program then prints a name and ID. This example illustrates how to use try-catch blocks to manage exceptions in Java. It ensures that even when an exception occurs, the program continues to execute subsequent statements. |
| 26. | AIM : Write a java program to generate user defined exception using “throw” and “throws” keyword. Also Write a java that differentiates checked and unchecked exceptions. (Mention at least two checked and two unchecked exceptions in program). PROGRAM CODE : import java.io.IOException;  import java.sql.SQLException;  class CustomCheckedException extends Exception {  public CustomCheckedException(String message) {  super(message);  }  }  class CustomUncheckedException extends RuntimeException {  public CustomUncheckedException(String message) {  super(message);  }  }  public class Prac\_26 {  public static void methodThrowingCheckedException() throws CustomCheckedException {  throw new CustomCheckedException("This is a custom checked exception");  }  public static void methodThrowingUncheckedException() {  throw new CustomUncheckedException("This is a custom unchecked exception");  }  public static void methodThrowingStandardCheckedExceptions() throws IOException, SQLException {  throw new IOException("This is an IOException");  }  public static void methodThrowingStandardUncheckedExceptions() {  throw new NullPointerException("This is a NullPointerException");  }  public static void main(String[] args) {  try {  methodThrowingCheckedException();  } catch (CustomCheckedException e) {  System.out.println("Caught checked exception: " + e.getMessage());  }  try {  methodThrowingStandardCheckedExceptions();  } catch (IOException | SQLException e) {  System.out.println("Caught standard checked exception: " + e.getMessage());  }  try {  methodThrowingUncheckedException();  } catch (CustomUncheckedException e) {  System.out.println("Caught unchecked exception: " + e.getMessage());  }  try {  methodThrowingStandardUncheckedExceptions();  } catch (NullPointerException | ArithmeticException e) {  System.out.println("Caught standard unchecked exception: " + e.getMessage());  }  System.out.print("\nName : Sneh \nID : 23DCS093 ");  }  }  **OUTPUT:**    **CONCLUSION:**  This Java code demonstrates handling both checked and unchecked exceptions. It uses multiple try-catch blocks to catch custom and standard exceptions, printing appropriate messages. This ensures robust error handling and program continuity. The program concludes by printing the author's name and ID. This example effectively illustrates exception management in Java. |

|  |  |
| --- | --- |
|  | **PART-VI File Handling & Streams** |
| 27 | Write a program that will count the number of lines in each file that is specified on the command line. Assume that the files are text files. Note that multiple files can be specified, as in "java Line Counts file1.txt file2.txt file3.txt". Write each file name, along with the number of lines in that file, to standard output. If an error occurs while trying to read from one of the files, you should print an error message for that file, but you should still process all the remaining files.  **PROGRAM CODE:**  import java.io.\*;  public class P27 {  public static void main(String[] args) throws Exception {  if (args.length == 0) {  System.out.println("No file Found!");  } else {  for (int i = 0; i < args.length; i++) {  try {  BufferedReader f = new BufferedReader(new FileReader(args[i]));  String j;  int count = 0;  while ((j = f.readLine()) != null) {  count++;  }  System.out.println("File name is : " + args[i] + " and Number of lines are : " + count);  } catch (Exception e) {  System.out.println(e);  } } }  System.out.println("ID :23DCS093\_SNEH PATEL");  } }  **OUTPUT:**    **CONCLUSION:**  This Java program reads several files named by the command line arguments and counts the number of lines in each. If no files are provided as command-line arguments, it will print out the appropriate message. Exception handling ensures graceful error management during file reading, thus a stable program. |
| 28 | Write an example that counts the number of times a particular character, such as e, appears in a file. The character can be specified at the command line. You can use xanadu.txt as the input file.  **PROGRAM CODE:**  import java.io.BufferedReader;  import java.io.FileReader;  import java.io.IOException;  public class P28{  public static void main(String[] args) {  if (args.length < 2) {  System.out.println("Usage: java P28 <character> <filename>");  return; }  char targetChar = args[0].charAt(0);  String fileName = args[1];  int count = 0;  try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {  int ch;  while ((ch = reader.read()) != -1) {  if (ch == targetChar) {  count++;  } }  System.out.println("The character '" + targetChar + "' appears " + count + " times in " + fileName);  } catch (IOException e) {  System.out.println("Error reading " + fileName + ": " + e.getMessage());  }  System.out.println("ID :23DCS093\_SNEH PATEL");  }}  **OUTPUT:**    **CONCLUSION:**  The Java program successfully counts the occurrences of a specified character in a given file, providing the result in a clear format. It handles file read errors gracefully, ensuring robust performance even if issues arise during file access. |
| 29 | Write a Java Program to Search for a given word in a File. Also show use of Wrapper Class with an example.  **PROGRAM CODE:**  import java.io.BufferedReader;  import java.io.FileReader;  import java.io.IOException;  public class P29 {  public static void main(String[] args) {  if (args.length < 2) {  System.out.println("Usage: java P29 <word> <filename>");  return;  }  String searchWord = args[0];  String fileName = args[1];  Integer count = 0;  try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {  String line;  while ((line = reader.readLine()) != null) {  String[] words = line.split("\\W+");  for (String word : words) {  if (word.equalsIgnoreCase(searchWord)) {  count++;  } } }  System.out.println("The word '" + searchWord + "' appears " + count + " times in " + fileName);  } catch (IOException e) {  System.out.println("Error reading " + fileName + ": " + e.getMessage());  }  System.out.println("ID :23DCS093\_SNEH PATEL");  } }  **OUTPUT:**    **CONCLUSION:**  This Java program effectively searches for a specified word in a given file and counts its occurrences. It demonstrates the use of the Integer wrapper class to manage the count, showcasing how wrapper classes can be used for object manipulation in Java. |
| 30 | Write a program to copy data from one file to another file. If the destination file does not exist, it is created automatically.  **PROGRAM CODE:**  import java.io.FileReader;  import java.io.FileWriter;  import java.io.IOException;  public class P30 {  public static void main(String[] args) {  if (args.length < 2) {  System.out.println("Usage: java P30 <source file> <destination file>");  return;  }  String sourceFile = args[0];  String destinationFile = args[1];  try (FileReader fr = new FileReader(sourceFile);  FileWriter fw = new FileWriter(destinationFile)) {  int ch;  while ((ch = fr.read()) != -1) {  fw.write(ch); }  System.out.println("Data copied from " + sourceFile + " to " + destinationFile);  } catch (IOException e) {  System.out.println("Error: " + e.getMessage());  }  System.out.println("ID :23DCS093\_SNEH PATEL");  } }  **OUTPUT:**      **CONCLUSION:**  This Java program efficiently copies data from a source file to a destination file, automatically creating the destination file if it does not already exist. It handles any potential I/O exceptions during the process, ensuring robust performance. |
| 31 | Write a program to show use of character and byte stream. Also show use of BufferedReader / BufferedWriter to read console input and write them into a file.  **PROGRAM CODE:**  import java.io.\*;  public class P31 {  public static void main(String[] args) {  BufferedReader consoleReader = new BufferedReader(new InputStreamReader (System.in));  String fileName = "output.txt";  try (BufferedWriter fileWriter = new BufferedWriter(new FileWriter(fileName))) {  System.out.println("Enter text (type 'exit' to finish):");  String input;  while (!(input = consoleReader.readLine()).equalsIgnoreCase("exit")) {  fileWriter.write(input);  fileWriter.newLine();  }  System.out.println("Data written to " + fileName);  } catch (IOException e) {  System.out.println("Error: " + e.getMessage());  }  System.out.println("ID :23DCS093\_SNEH PATEL");  } }  **OUTPUT:**    **CONCLUSION:**  This program effectively demonstrates the use of character streams via BufferedReader and BufferedWriter for reading console input and writing it to a file. It showcases how to handle text data efficiently while managing resources properly with try-with-resources. |
|  | **PART-VII Multithreading** |
| 32 | Write a program to create thread which display “Hello World” message. A. by extending Thread class B. by using Runnable interface.  **PROGRAM CODE:**  class Thread1 extends Thread{  // by extending Thread class  public void run(){  System.out.println("Hello world");  }}  class Thread2 implements Runnable{ //by using Runnable interface.  public void run(){  System.out.println("Hello world 1");  } }  public class P32 {  public static void main(String[] args) {  Thread1 thread = new Thread1();  thread.start();  Thread2 obj2 = new Thread2();  Thread t1 = new Thread(obj2);  t1.start();  } }  **OUTPUT:**  **CONCLUSION:**  This program demonstrates two approaches to creating threads in Java: extending the Thread class and implementing the Runnable interface. Both methods effectively print "Hello World," showcasing the flexibility of Java's concurrency model. |
| 33 | Write a program which takes N and number of threads as an argument. Program should distribute the task of summation of N numbers amongst number of threads and final result to be displayed on the console.  **PROGRAM CODE:**  import java.util.Scanner;  class SumTask implements Runnable {  private int start;  private int end;  private static int totalSum = 0;  public SumTask(int start, int end) {  this.start = start;  this.end = end;  }  public void run() {  int partialSum = 0;  for (int i = start; i <= end; i++) {  partialSum += i;  }  synchronized (SumTask.class) {  totalSum += partialSum;  } }  public static int getTotalSum() {  return totalSum;  } }  public class P33 {  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  System.out.print("Enter N: ");  int N = scanner.nextInt();  System.out.print("Enter number of threads: ");  int numThreads = scanner.nextInt();  Thread[] threads = new Thread[numThreads];  int range = N / numThreads;  int remainder = N % numThreads;  int start = 1;  for (int i = 0; i < numThreads; i++) {  int end = start + range - 1;  if (i == numThreads - 1) {  end += remainder;  }  threads[i] = new Thread(new SumTask(start, end));  threads[i].start();  start = end + 1;  }  for (Thread thread : threads) {  try {  thread.join();  } catch (InterruptedException e) {  e.printStackTrace();  } }  System.out.println("Total Sum: " + SumTask.getTotalSum());  } }  **OUTPUT:**    **CONCLUSION:**  This program effectively demonstrates how to utilize multiple threads in Java to perform a summation task concurrently. By distributing the workload among threads, it showcases improved efficiency in computation, making it a practical example of multithreading in action. |
| 34 | Write a java program that implements a multi-thread application that has three threads. First thread generates random integer every 1 second and if the value is even, second thread computes the square of the number and prints. If the value is odd, the third thread will print the value of cube of the number.  **PROGRAM CODE:**  import java.util.Random;  class RandomNumberGenerator extends Thread {  private final Object lock;  public RandomNumberGenerator(Object lock) {  this.lock = lock;  }  public void run() {  Random random = new Random();  while (true) {  int number = random.nextInt(100);  synchronized (lock) {  P34.lastNumber = number;  lock.notifyAll();  System.out.println("Generated: " + number);  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  } } } } }  class EvenNumberProcessor extends Thread {  private final Object lock;  public EvenNumberProcessor(Object lock) {  this.lock = lock;  }  public void run() {  while (true) {  synchronized (lock) {  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  if (P34.lastNumber % 2 == 0) {  int square = P34.lastNumber \* P34.lastNumber;  System.out.println("Square: " + square);  } } } } }  class OddNumberProcessor extends Thread {  private final Object lock;  public OddNumberProcessor(Object lock) {  this.lock = lock;  }  public void run() {  while (true) {  synchronized (lock) {  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  if (P34.lastNumber % 2 != 0) {  int cube = P34.lastNumber \* P34.lastNumber \* P34.lastNumber;  System.out.println("Cube: " + cube);  } } } } }  public class P34 {  public static int lastNumber;  public static void main(String[] args) {  Object lock = new Object();  RandomNumberGenerator generator = new RandomNumberGenerator(lock);  EvenNumberProcessor evenProcessor = new EvenNumberProcessor(lock);  OddNumberProcessor oddProcessor = new OddNumberProcessor(lock);  generator.start();  evenProcessor.start();  oddProcessor.start();  }}  **OUTPUT:**    **CONCLUSION:**  This program effectively demonstrates a multi-threaded application where one thread generates random integers, while two other threads process these integers based on their parity. It highlights the use of synchronization in Java to safely share data among threads, showcasing how concurrency can be leveraged for efficient task distribution. |
| 35 | Write a program to increment the value of one variable by one and display it after one second using thread using sleep() method.  **PROGRAM CODE:**  public class P35 extends Thread {  private int value = 0;  public void run() {  while (true) {  value++;  System.out.println("Value: " + value);  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  } } }  public static void main(String[] args) {  P35 incrementer = new P35();  incrementer.start();  } }  **OUTPUT:**    **CONCLUSION:**  This program effectively demonstrates the use of a thread to increment a variable every second. It utilizes the sleep() method to create a delay between increments, showcasing basic thread functionality in Java. |
| 36 | Write a program to create three threads ‘FIRST’, ‘SECOND’, ‘THIRD’. Set the priority of the ‘FIRST’ thread to 3, the ‘SECOND’ thread to 5(default) and the ‘THIRD’ thread to 7.  **PROGRAM CODE:**  class MyThread extends Thread {  MyThread(String name) {  super(name);  }  public void run() {  System.out.println(getName() + " is running with priority " + getPriority());  } }  public class P36 {  public static void main(String[] args) {  MyThread first = new MyThread("FIRST");  MyThread second = new MyThread("SECOND");  MyThread third = new MyThread("THIRD");  first.setPriority(3);  first.start();  second.setPriority(Thread.NORM\_PRIORITY);  second.start();  third.setPriority(7);  third.start();  } }  **OUTPUT:**    **CONCLUSION:**  This program demonstrates thread creation and priority setting in Java by extending the Thread class. Each thread prints its name and priority when executed. Different priority levels (3, 5, 7) are set using setPriority(), showcasing the influence of priority on execution order. However, actual execution may vary due to the system's thread scheduling. |
| 37 | Write a program to solve producer-consumer problem using thread synchronization.  **PROGRAM CODE:**  class Buffer {  private int data;  private boolean isEmpty = true;  public synchronized void produce(int value) {  while (!isEmpty) {  try {  wait();  } catch (InterruptedException e) {  e.printStackTrace();  } }  data = value;  isEmpty = false;  System.out.println("Produced: " + data);  notify();  }  public synchronized void consume() {  while (isEmpty) {  try {  wait();  } catch (InterruptedException e) {  e.printStackTrace();  } }  System.out.println("Consumed: " + data);  isEmpty = true;  notify();  } }  class Producer extends Thread {  private Buffer buffer;  public Producer(Buffer buffer) {  this.buffer = buffer;  }  public void run() {  for (int i = 1; i <= 5; i++) {  buffer.produce(i);  // Produce values from 1 to 5  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  } } } }  class Consumer extends Thread {  private Buffer buffer;  public Consumer(Buffer buffer) {  this.buffer = buffer;  }  public void run() {  for (int i = 1; i <= 5; i++) {  buffer.consume();  try {  Thread.sleep(1500);  } catch (InterruptedException e) {  e.printStackTrace();  } } } }  public class P37 {  public static void main(String[] args) {  Buffer buffer = new Buffer();  Producer producer = new Producer(buffer);  Consumer consumer = new Consumer(buffer);  producer.start();  consumer.start();  } }  **OUTPUT:**    **CONCLUSION:**  This program demonstrates producer-consumer synchronization in Java using the wait() and notify() methods. The producer thread generates data, while the consumer thread consumes it, both synchronized to avoid race conditions. The use of wait() and notify() ensures proper coordination between the threads, allowing for controlled data production and consumption. |
|  | **PART-VIII Collection Framework and Generic** |
| 38 | Design a Custom Stack using ArrayList class, which implements following functionalities of stack.  My Stack -list ArrayList<Object>: A list to store elements.  isEmpty: boolean: Returns true if this stack is empty.  getSize(): int: Returns number of elements in this stack.  peek(): Object: Returns top element in this stack without removing it.  pop(): Object: Returns and Removes the top elements in this stack.  push(o: object): Adds new element to the top of this stack.  **PROGRAM CODE:**  import java.util.ArrayList;  class MyStack {  private ArrayList<Object> list = new ArrayList<>();  public boolean isEmpty() {  return list.isEmpty();  }  public int getSize() {  return list.size();  }  public Object peek() {  if (isEmpty()) {  return "Stack is empty";  }  return list.get(list.size() - 1);  }  public Object pop() {  if (isEmpty()) {  return "Stack is empty";  }  return list.remove(list.size() - 1);  }  public void push(Object o) {  list.add(o);  } }  public class P38 {  public static void main(String[] args) {  MyStack stack = new MyStack();  stack.push(10);  stack.push(20);  stack.push(30);  System.out.println("Top element is: " + stack.peek());  System.out.println("Popped element: " + stack.pop());  System.out.println("Popped element: " + stack.pop());  System.out.println("Is stack empty ? " + stack.isEmpty());  System.out.println("Current stack size: " + stack.getSize());  System.out.println("Top element now: " + stack.peek());  } }  **OUTPUT:**    **CONCLUSION:**  This program demonstrates the implementation of a custom stack using the ArrayList class in Java. It provides functionalities to push, pop, peek, check if the stack is empty, and get the current size of the stack. The program effectively showcases how to manage a dynamic collection of elements while adhering to stack principles. |
| 39 | Imagine you are developing an e-commerce application. The platform needs to sort lists of products based on different criteria, such as price, rating, or name. Each product object implements the Comparable interface to define the natural ordering. To ensure flexibility and reusability, you need a generic method that can sort any array of Comparable objects. Create a generic method in Java that sorts an array of Comparable objects. This method should be versatile enough to sort arrays of different types of objects (such as products, customers, or orders) as long as they implement the Comparable interface.  **PROGRAM CODE:**  import java.util.Arrays;  public class P39 {  public static <T extends Comparable<T>> void sortArray(T[] array) {  Arrays.sort(array);  }  public static void main(String[] args) {  Integer[] numbers = {5, 3, 9, 1, 7};  System.out.println("Before sorting (Integers): " + Arrays.toString(numbers));  sortArray(numbers);  System.out.println("After sorting (Integers): " + Arrays.toString(numbers));  String[] names = {"John", "Alice", "Bob", "David"};  System.out.println("\nBefore sorting (Strings): " + Arrays.toString(names));  sortArray(names);  System.out.println("After sorting (Strings): " + Arrays.toString(names));  Product[] products = {  new Product("Laptop", 1000),  new Product("Phone", 800),  new Product("Tablet", 600),  new Product("Smartwatch", 200)  };  System.out.println("\nBefore sorting (Products by price): ");  for (Product p : products) {  System.out.println(p);  }  sortArray(products);  System.out.println("\nAfter sorting (Products by price): ");  for (Product p : products) {  System.out.println(p);  } } }  class Product implements Comparable<Product> {  private String name;  private int price;  public Product(String name, int price) {  this.name = name;  this.price = price;  }  @Override  public int compareTo(Product other) {  return this.price - other.price;  }  @Override  public String toString() {  return name + ": $" + price;  } }  **OUTPUT:**    **CONCLUSION:**  This program demonstrates the use of generics in Java to create a versatile sorting method for arrays of different types. By implementing the Comparable interface in the Product class, it enables sorting of custom objects based on specific criteria, such as price. The output shows the effective sorting of integers, strings, and products, highlighting the flexibility and reusability of the generic sorting method. |
| 40 | Write a program that counts the occurrences of words in a text and displays the words and their occurrences in alphabetical order of the words. Using Map and Set Classes.  **PROGRAM CODE:**  import java.util.\*;  public class P40 {  public static void main(String[] args) {  Map<String, Integer> wordMap = new TreeMap<>();  Scanner scanner = new Scanner(System.in);  System.out.println("Enter a text:");  String text = scanner.nextLine();  String[] words = text.toLowerCase().split("\\W+");  for (String word : words) {  if (!word.isEmpty()) {  wordMap.put(word, wordMap.getOrDefault(word, 0) + 1);  } }  System.out.println("\nWord Occurrences (in alphabetical order):");  Set<Map.Entry<String, Integer>> entrySet = wordMap.entrySet();  for (Map.Entry<String, Integer> entry : entrySet) {  System.out.println(entry.getKey() + ": " + entry.getValue());  } } }  **OUTPUT:**    **CONCLUSION:**  This program demonstrates how to count and display the occurrences of words in a given text using Java's Map and Set classes. The words are stored in a TreeMap, ensuring that they are presented in alphabetical order. The use of getOrDefault() simplifies the counting process, showcasing efficient word frequency analysis. |
| 41 | Write a code which counts the number of the keywords in a Java source file. Store all the keywords in a HashSet and use the contains () method to test if a word is in the keyword set.  **PROGRAM CODE:**  import java.io.\*;  import java.util.\*;  public class P41 {  private static final HashSet<String> keywords = new HashSet<>();  static {  String[] keywordArray = {  "abstract", "assert", "boolean", "break", "byte", "case", "catch", "char", "class",  "const", "continue", "default", "do", "double", "else", "enum", "extends", "final",  "finally", "float", "for", "goto", "if", "implements", "import", "instanceof", "int",  "interface", "long", "native", "new", "package", "private", "protected", "public",  "return", "short", "static", "strictfp", "super", "switch", "synchronized", "this",  "throw", "throws", "transient", "try", "void", "volatile", "while"  };  for (String keyword : keywordArray) {  keywords.add(keyword);  } }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  System.out.print("Enter the path of the Java source file: ");  String filePath = scanner.nextLine();  try {  File file = new File(filePath);  Scanner fileScanner = new Scanner(file);  int keywordCount = 0;  while (fileScanner.hasNext()) {  String word = fileScanner.next();  if (keywords.contains(word)) {  keywordCount++;  } }  System.out.println("Number of Java keywords in the file: " + keywordCount);  fileScanner.close();  } catch (FileNotFoundException e) {  System.out.println("File not found: " + filePath);  } } }  **OUTPUT:**    **CONCLUSION:**  This program demonstrates the use of a HashSet to efficiently count Java keywords in a source file. By reading each word from the file and checking for its presence in the set of keywords, it showcases how to utilize collections for rapid lookups. The result is the total number of keywords, providing a simple yet effective tool for analyzing Java code. |