CHAROTAR UNIVERSITY OF SCIENCE & TECHNOLOGY

**DEVANG PATEL INSTITUTE OF ADVANCE TECHNOLOGY & RESEARCH**

Department of Computer Science & Engineering

Subject Name: Java Programming

Semester: III

Subject Code: CSE201

Academic year: 2024-25

Part - VII

|  |  |
| --- | --- |
| **No.** | **Aim of the Practical** |
| 32. | Write a program to create thread which display “Hello World” message. A. by extending Thread class B. by using Runnable interface.  **PROGRAM CODE:**  public class HelloWorld {  static class HelloWorldThread extends Thread {  public void run() {  System.out.println("Hello World");  }  }  static class HelloWorldRunnable implements Runnable {  public void run() {  System.out.println("Hello World");  }  }  public static void main(String[] args) {  HelloWorldThread thread1 = new HelloWorldThread();  thread1.start();    Thread thread2 = new Thread(new HelloWorldRunnable());  thread2.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 ThreadedSummation {  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) {  MultiThreadApplication.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 (MultiThreadApplication.lastNumber % 2 == 0) {  int square = MultiThreadApplication.lastNumber \* MultiThreadApplication.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 (MultiThreadApplication.lastNumber % 2 != 0) {  int cube = MultiThreadApplication.lastNumber \* MultiThreadApplication.lastNumber \* MultiThreadApplication.lastNumber;  System.out.println("Cube: " + cube);  }  }  }  }  }  public class MultiThreadApplication {  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 IncrementVariable 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) {  IncrementVariable incrementer = new IncrementVariable();  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 {  public MyThread(String name) {  super(name);  }  public void run() {  for (int i = 1; i <= 5; i++) {  System.out.println(getName() + ": " + i);  try {  Thread.sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }  public class ThreadPriorityExample {  public static void main(String[] args) {  MyThread firstThread = new MyThread("FIRST");  MyThread secondThread = new MyThread("SECOND");  MyThread thirdThread = new MyThread("THIRD");  firstThread.setPriority(3);  secondThread.setPriority(Thread.NORM\_PRIORITY);  thirdThread.setPriority(7);  firstThread.start();  secondThread.start();  thirdThread.start();  }  }  **OUTPUT:**    **CONCLUSION:**  This program demonstrates thread creation and priority setting in Java. Each thread executes a simple loop, displaying its name and an iteration count, showcasing how thread priority can influence the execution order, although actual execution may vary due to the nature of thread scheduling. |
| 37. | Write a program to solve producer-consumer problem using thread synchronization.  **PROGRAM CODE:**  import java.util.LinkedList;  import java.util.Queue;  class ProducerConsumer {  private final Queue<Integer> queue = new LinkedList<>();  private final int capacity = 5;  public void produce() throws InterruptedException {  int value = 0;  while (true) {  synchronized (this) {  while (queue.size() == capacity) {  wait();  }  queue.add(value);  System.out.println("Produced: " + value);  value++;  notifyAll();  }  Thread.sleep(1000);  }  }  public void consume() throws InterruptedException {  while (true) {  synchronized (this) {  while (queue.isEmpty()) {  wait();  }  int value = queue.poll();  System.out.println("Consumed: " + value);  notifyAll();  }  Thread.sleep(1500);  }  }  }  class Producer extends Thread {  private final ProducerConsumer pc;  public Producer(ProducerConsumer pc) {  this.pc = pc;  }  public void run() {  try {  pc.produce();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  class Consumer extends Thread {  private final ProducerConsumer pc;  public Consumer(ProducerConsumer pc) {  this.pc = pc;  }  public void run() {  try {  pc.consume();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  public class ProducerConsumerExample {  public static void main(String[] args) {  ProducerConsumer pc = new ProducerConsumer();  Producer producer = new Producer(pc);  Consumer consumer = new Consumer(pc);  producer.start();  consumer.start();  }  }  **OUTPUT:**    **CONCLUSION:**  This program effectively demonstrates the Producer-Consumer problem using thread synchronization in Java. The producer generates integers and adds them to a shared queue, while the consumer retrieves and consumes them. Synchronization ensures safe access to the shared resource, preventing data inconsistencies and race conditions. |