|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Object Oriented Programming** | **Semester:** | **III** |
| **Date of Performance:** | **13 / 10 / 2023** | **Batch No:** | **A - 3** |
| **Faculty Name:** | **Pragya Gupta** | **Roll No:** | **16014022050** |
| **Faculty Sign & Date:** |  | **Grade/Marks:** | **\_\_\_ / 25** |

|  |  |  |
| --- | --- | --- |
| **Writing Program (07)** | **Performance in lab**  **and Viva (05 + 03)** | **Post lab questions, conclusion and**  **Completion (03 + 02 + 05)** |
|  |  |  |

**Experiment No: 8**

**Title: Multithreading in Java**

|  |
| --- |
| **Aim and Objective of the Experiment:** |
| Learn the how to do multithreading in Java. |

|  |
| --- |
| **COs to be achieved:** |
| **CO4:** Develop a Java application with threads and generics classes. |

|  |
| --- |
| **Tools used:** |
|  |

|  |
| --- |
| **Theory:** |
| 1. (Syntax of implementing thread in Java, thread group, state diagram of thread)   To create a thread in Java, you have two main ways to do it: by implementing the Runnable interface or by extending the Thread class. Here's an example of both approaches:   1. **Implementing Runnable interface:**   public class MyRunnable implements Runnable {  public void run() {  // Code that will run in the thread  System.out.println("MyRunnable running");  }  public static void main(String[] args) {  Thread thread = new Thread(new MyRunnable());  thread.start();  }  }   1. **Extending Thread class:**   public class MyThread extends Thread {  public void run() {  // Code that will run in the thread  System.out.println("MyThread running");  }  public static void main(String[] args) {  MyThread thread = new MyThread();  thread.start();  }  }  Java thread groups are a way of categorizing threads. They can be used to perform operations on multiple threads simultaneously. However, the usage of thread groups has become less common due to the introduction of the java.util.concurrent package.  The state diagram of a thread in Java represents the different states a thread can be in during its lifetime. A thread can be in one of the following states:   1. **New:** When a thread is created but not yet started. 2. **Runnable:** When a thread is ready to run and waiting for the CPU. 3. **Running:** When the thread is executing its task. 4. **Blocked:** When a thread is waiting for a monitor lock to enter a synchronized block/method or waiting for I/O operations to complete. 5. **Waiting:** When a thread is in a waiting state, often due to a call to wait() method. 6. **Timed Waiting:** When a thread is in a waiting state with a time limit, often due to a call to sleep(long millis) or join(long millis) methods. 7. **Terminated:** When a thread has completed its task and has been terminated either by finishing its execution normally or by throwing an uncaught exception.   These states are illustrated in a state diagram as transitions from one state to another based on certain events or actions performed on the thread. |

|  |
| --- |
| **Code:** |
| 1. **Write a Java program that implements a multi-threaded program has three threads. First thread generates a 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. (use thread class)**   import java.util.Random;  class RandomNumberGenerator extends Thread {      public void run() {          Random random = new Random();          while (true) {              int randomNumber = random.nextInt(100);              System.out.println("\nGenerated number: " + randomNumber);              if (randomNumber % 2 == 0) {                  new EvenThread(randomNumber).start();              } else {                  new OddThread(randomNumber).start();              }              try {                  Thread.sleep(1000);              } catch (InterruptedException e) {                  e.printStackTrace();              }          }      }  }  class EvenThread extends Thread {      private int number;      EvenThread(int number) {          this.number = number;      }      public void run() {          int square = number \* number;          System.out.println("Square of " + number + " is " + square);      }  }  class OddThread extends Thread {      private int number;      OddThread(int number) {          this.number = number;      }      public void run() {          int cube = number \* number \* number;          System.out.println("Cube of " + number + " is " + cube);      }  }  public class exp8\_q1  {      public static void main(String[] args)      {          RandomNumberGenerator randomNumberGenerator = new RandomNumberGenerator();          randomNumberGenerator.start();      }  }   1. **Write a Java program which first generates a set of random numbers and then determines negative, positive even, positive odd numbers concurrently. (use runnable interface)**   import java.util.Random;  class NumberClassifier implements Runnable {      private int[] numbers;      NumberClassifier(int[] numbers) {          this.numbers = numbers;      }      @Override      public void run() {          Thread currentThread = Thread.currentThread();          if (currentThread.getName().equals("NegativeThread")) {              for (int number : numbers) {                  if (number < 0) {                      System.out.println("Negative number: " + number);                  }              }          } else if (currentThread.getName().equals("EvenThread")) {              for (int number : numbers) {                  if (number > 0 && number % 2 == 0) {                      System.out.println("Positive even number: " + number);                  }              }          } else if (currentThread.getName().equals("OddThread")) {              for (int number : numbers) {                  if (number > 0 && number % 2 != 0) {                      System.out.println("Positive odd number: " + number);                  }              }          }      }  }  public class exp8\_q2  {      public static void main(String[] args)      {            int[] randomNumbers = generateRandomNumbers(10);          Runnable negativeRunnable = new NumberClassifier(randomNumbers);          Runnable evenRunnable = new NumberClassifier(randomNumbers);          Runnable oddRunnable = new NumberClassifier(randomNumbers);          Thread negativeThread = new Thread(negativeRunnable);          negativeThread.setName("NegativeThread");          Thread evenThread = new Thread(evenRunnable);          evenThread.setName("EvenThread");          Thread oddThread = new Thread(oddRunnable);          oddThread.setName("OddThread");          negativeThread.start();          evenThread.start();          oddThread.start();      }      private static int[] generateRandomNumbers(int count) {          int[] numbers = new int[count];          Random random = new Random();          for (int i = 0; i < count; i++) {              numbers[i] = random.nextInt(100) - 50; // Generates random numbers between -50 and 50          }          return numbers;      }  }   1. **Write a java program for to solve producer consumer problem in which a producer produces a value and consumer consume the value before producer generate the next value.**   import java.util.LinkedList;  class Buffer {      private LinkedList<Integer> list;      private int capacity;      Buffer(int capacity) {          this.capacity = capacity;          this.list = new LinkedList<>();      }      public void produce() throws InterruptedException {          int value = 0;          while (true) {              synchronized (this) {                  while (list.size() == capacity) {                      wait();                  }                  System.out.println("\nProducer produced-" + value);                  list.add(value++);                  notify();                  Thread.sleep(1000); // Producer waits for the consumer to consume the value              }          }      }      public void consume() throws InterruptedException {          while (true) {              synchronized (this) {                  while (list.size() == 0) {                      wait();                  }                  int val = list.removeFirst();                  System.out.println("Consumer consumed-" + val);                  notify();                  Thread.sleep(1000); // Consumer waits for the producer to produce the next value              }          }      }  }  public class exp8\_q3  {      public static void main(String[] args)      {          Buffer buffer = new Buffer(2);          Thread producerThread = new Thread(() -> {              try {                  buffer.produce();              } catch (InterruptedException e) {                  e.printStackTrace();              }          });          Thread consumerThread = new Thread(() -> {              try {                  buffer.consume();              } catch (InterruptedException e) {                  e.printStackTrace();              }          });          producerThread.start();          consumerThread.start();      }  } |

|  |
| --- |
| **Output:** |
|  |

|  |
| --- |
| **Post Lab Subjective/Objective type Questions:** |
| 1. **What does join() method?**   The join() method is used to make sure that the current thread will wait for the thread on which join() is called to complete its execution. It's especially useful when a thread needs to complete its task before the main thread can continue.   1. **Is it possible to start a thread twice?**   No, it is not possible to start a thread twice. Once a thread has completed its execution or has been stopped, it cannot be restarted. If you attempt to start a thread that has already been started and has not completed its execution, it will throw an IllegalThreadStateException.   1. **Can we call the run() method instead of start()?**   Yes, you can call the run() method directly, but doing so will not create a new thread. It will simply run the code on the current thread without starting a new thread. The start() method is used to create a new thread and then call its run() method.   1. **What is difference between wait() and sleep()?**  * wait() is a method in the Object class that is used for inter-thread communication. It causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object. * sleep() is a static method in the Thread class that causes the currently executing thread to sleep for a specified number of milliseconds. Unlike wait(), sleep() does not release the lock it holds.  1. **What is the purpose of the Synchronized block?**   The synchronized block is used to ensure that only one thread can access the synchronized code at a time. It is used to prevent thread interference and memory consistency errors. By using the synchronized keyword, you can specify critical sections of code that can be executed by only one thread at a time.   1. **What is the difference between notify() and notifyAll()?**  * notify() wakes up a single thread that is waiting on the object's monitor. If multiple threads are waiting, it is not specified which thread will be awakened. * notifyAll() wakes up all the threads that are waiting on the object's monitor. All these threads are placed in the ready-to-run state, and the thread scheduler will schedule them. |

|  |
| --- |
| **Conclusion:** |
| Through our Java multithreading experiment, we gained valuable insights into efficiently managing concurrent tasks, leveraging threads to optimize resource utilization, and ensuring synchronized access to shared resources. |

|  |
| --- |
| **Signature of faculty in-charge with Date:** |