Lab Sheet 01 - Multi-threaded Java Application

**Part 1: Introduction to Threads in Java**

1. Create a Simple Thread Class

**Steps:**

1. create a new project named MultiThreadApp.
2. Inside the project, create a new class called SimpleThread.java.

**Code:**

public class SimpleThread extends Thread {

@Override

public void run () {

System.out.println(Thread.currentThread().getId() + " is executing the thread.");

}

public static void main(String[] args) {

SimpleThread thread1 = new SimpleThread();

SimpleThread thread2 = new SimpleThread();

thread1.start(); // Starts thread1

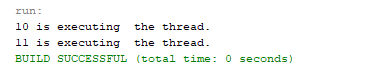
thread2.start(); // Starts thread2

}

}

**Explanation:**

1. SimpleThread extends Thread
   * Creates a custom thread class.
2. run() method
   * Contains the code that runs when the thread starts.
3. Thread.currentThread().getId()
   * Gets the ID of the running thread.
4. main() method
   * Creates two threads and starts them with start(), so they run at the same time.

**Output:**

**Part 2: Using Runnable Interface**

1. Create a Runnable Class

**Steps:**

1. Create a new class called RunnableTask.java.

**Code:**

public class RunnableTask implements Runnable {

@Override

public void run() {

System.out.println(Thread.currentThread().getId() + " is executing the runnable task.");

}

public static void main(String[] args) {

RunnableTask task1 = new RunnableTask();

RunnableTask task2 = new RunnableTask();

Thread thread1 = new Thread(task1);

Thread thread2 = new Thread(task2);

thread1.start(); // Starts thread1

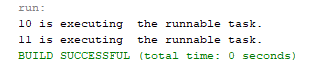
thread2.start(); // Starts thread2

}

}

**Explanation:**

1. implements Runnable
   * Defines a class that can be executed by a thread.
2. run() method
   * Contains the task to run in a thread.
3. Thread.currentThread().getId()
   * Prints the ID of the thread running the task.
4. main() method
   * Creates two RunnableTask objects.
   * Wraps them in Thread objects.
   * Starts both threads to run concurrently.

** Output:**

**Part 3: Synchronizing Threads**

In a multi-threaded environment, synchronization is used to control access to shared resources like variables or files. It prevents data inconsistency caused by multiple threads accessing or modifying the resource at the same time.

1. Synchronizing Shared Resources

**Steps:**

1. Create a new class called Counter.javato demonstrate synchronization with shared  resources.

**Code:**

// Enter class Counter.java

class Counter {

 private int count = 0;

 // Synchronized method to ensure thread-safe access to the counter

public synchronized void increment() {

 count++;

  }

  public int getCount() {

  return count;

 }

}

//Enter class SynchronizedExample.java

public class SynchronizedExample extends Thread {

private Counter counter;

public SynchronizedExample(Counter counter) {

this.counter = counter;

}

@Override

public void run() {

for (int i = 0; i < 1000; i++) {

counter.increment();

}

}

public static void main(String[] args) throws InterruptedException {

Counter counter = new Counter();

// Create and start multiple threads

Thread thread1 = new SynchronizedExample(counter);

Thread thread2 = new SynchronizedExample(counter);

thread1.start();

thread2.start();

// Wait for threads to finish

thread1.join();

thread2.join();

System.out.println("Final counter value: " + counter.getCount());

}

}

**Explanation:**

1. Shared Object:

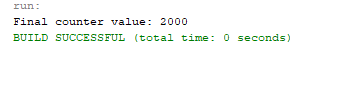
* Both threads use the same Counter object.

1. Thread Safety:

* increment() is synchronized, so only one thread can modify count at a time.

1. join():

* Waits for both threads to finish before printing the final result.

** Output:**

**Part 4: Thread Pooling**

A thread pool is a collection of pre-created, reusable threads. Instead of creating a new thread for every task, tasks are submitted to the pool, and available threads pick them up.

1. Using ExecutorService for Thread Pooling

**Steps:**

* 1. Create a new class called ThreadPoolExample.java.

**Code:**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

class Task implements Runnable {

private int taskId;

public Task(int taskId) {

this.taskId = taskId;

}

@Override

public void run() {

System.out.println("Task " + taskId + " is being processed by " + Thread.currentThread().getName());

}

}

public class ThreadPoolExample {

public static void main(String[] args) {

// Create a thread pool with 3 threads

ExecutorService executorService = Executors.newFixedThreadPool(3);

// Submit tasks to the pool

for (int i = 1; i <= 5; i++) {

executorService.submit(new Task(i));

}

// Shutdown the thread pool

executorService.shutdown();

}

}

**Explanation:**

1. **Task class**:

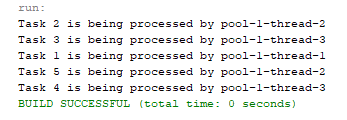
* A simple task that prints its ID and the thread name.

1. **ExecutorService**:

* Manages a fixed pool of 3 threads.

1. **submit()**:
   * Sends 5 tasks to the thread pool.
2. **shutdown()**:

* Closes the pool after tasks are done.

****

**Output:**

**Part 5: Thread Lifecycle and States**

In Java, threads can exist in several states during their lifecycle. These states include:

1. New: A thread is created but not yet started.

2. Runnable: A thread is ready to run, waiting for CPU time.

3. Blocked: A thread is blocked waiting for a resource.

4. Waiting: A thread is waiting indefinitely for another thread to perform particular  action.

5. Terminated: A thread has finished execution.

5.Thread Lifecycle Example

**Steps:**

1. Create a new class called ThreadLifecycleExample**.**java

**Code:**

public class ThreadLifecycleExample extends Thread {

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + " - State: " + Thread.currentThread().getState());

try {

Thread.sleep(2000); // Simulate waiting state

}

catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " - State after sleep: " + Thread.currentThread().getState());

}

public static void main(String[] args) {

ThreadLifecycleExample thread = new ThreadLifecycleExample(); System.out.println(thread.getName() + " - State before start: " + thread.getState());

thread.start(); // Start the thread

System.out.println(thread.getName() + " - State after start: " + thread.getState());

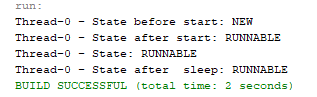
}

}

**Explanation:**

1. Before start()
   * Thread is in NEW state.
2. After start()
   * Thread moves to RUNNABLE, waiting to run.
3. Inside run()
   * When executing, it’s in RUNNING state.
4. Thread.sleep(2000)
   * Thread enters TIMED\_WAITING state for 2 seconds.
5. After sleep
   * Thread resumes and completes, ending in TERMINATED state.

**Output:**

****