GAM/IT/2022/F/0068 – S.A.I.I. Subhasingha

Lab Sheet 01: Multi-threaded Java Application

Part 1: Introduction to Threads in Java

In Java, threads are lightweight processes that allow for concurrent execution of code. We will start by creating a basic multi-threaded application that demonstrates the creation and execution of threads.

1. Create a Simple Thread Class

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package multithreadapp2;

/**
 *
 * @author USER
 */
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
    }
}

```

Part 2: Using Runnable Interface

Another way to create threads is by implementing the Runnable interface. This allows you to separate the task (logic) from the thread management.

2. Create a Runnable Class

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package multithreadapp2;

/**
 *
 * @author USER
 */
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(target:task1);
        Thread thread2 = new Thread(target:task2);
        thread1.start(); // Starts thread1
        thread2.start(); // Starts thread2
    }
}

```

Part 3: Synchronizing Threads

In a multi-threaded environment, synchronization is used to control access to shared resources (e.g., variables, files) to prevent data inconsistency due to concurrent access.

3. Synchronizing Shared Resources

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package multithreadapp2;

/**
 *
 * @author USER
 */
public 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;
    }
}

```

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package multithreadapp2;

/**
 *
 * @author USER
 */
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());
    }
}

```

Part 4: Thread Pooling

Thread pools allow you to reuse a fixed number of threads, which helps in optimizing thread management and system resources.

4. Using ExecutorService for Thread Pooling

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package multithreadapp2;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
/**
 *
 * @author USER
 */
public 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(nThreads: 3);
        // Submit tasks to the pool
        for (int i = 1; i <= 5; i++) {
            executorService.submit(new Task(taskId:i));
        }
        // Shutdown the thread pool
            executorService.shutdown();
        }
    }
}
```

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package multithreadapp2;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 *
 * @author USER
 */
public class ThreadPoolExample {

    public static void main(String[] args) {
        // Create a thread pool with 3 threads
        ExecutorService executorService = Executors.newFixedThreadPool(nThreads: 3);
        // Submit tasks to the pool
        for (int i = 1; i <= 5; i++) {
            executorService.submit(new Task(taskId:i));
        }
        // Shutdown the thread pool
        executorService.shutdown();
    }
}
```

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 a particular action.
5. Terminated: A thread has finished execution.

5. Thread Lifecycle Example

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package multithreadapp2;

/**
 *
 * @author USER
 */
public class ThreadLifecycleExample extends Thread{

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " - State: " +
        Thread.currentThread().getState());
    try {
        Thread.sleep(millis: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());
    }
}

```