

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

```
1  /**
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5   package multithreadapp2;
6
7   /**
8    *
9    * @author USER
10   */
11  public class SimpleThread extends Thread{
12
13      @Override
14      public void run() {
15          System.out.println(Thread.currentThread().getId() + " is executing the thread.");
16      }
17      public static void main(String[] args) {
18          SimpleThread thread1 = new SimpleThread();
19          SimpleThread thread2 = new SimpleThread();
20          thread1.start(); // Starts thread1
21          thread2.start(); // Starts thread2
22      }
23  }
24
```

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

```
1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5  package multithreadapp2;
6
7  /**
8   *
9   * @author USER
10  */
11 public class RunnableTask implements Runnable{
12
13     @Override
14     public void run() {
15         System.out.println(Thread.currentThread().getId() + " is executing the runnable task.");
16     }
17
18     public static void main(String[] args) {
19         RunnableTask task1 = new RunnableTask();
20         RunnableTask task2 = new RunnableTask();
21         Thread thread1 = new Thread(target: task1);
22         Thread thread2 = new Thread(target: task2);
23         thread1.start(); // Starts thread1
24         thread2.start(); // Starts thread2
25     }
26 }
27
```

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

```
1  /**
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5   package multithreadapp2;
6
7   /**
8    *
9    * @author USER
10  */
11  public class Counter {
12
13      private int count = 0;
14      // Synchronized method to ensure thread-safe access to the counter
15      public synchronized void increment() {
16          count++;
17      }
18      public int getCount() {
19          return count;
20      }
21  }
22
```

```
1  /**
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5   package multithreadapp2;
6
7   /**
8    *
9    * @author USER
10  */
11  public class SynchronizedExample extends Thread{
12
13      private Counter counter;
14
15      public SynchronizedExample(Counter counter) {
16          this.counter = counter;
17      }
18
19      @Override
20      public void run() {
21          for (int i = 0; i < 1000; i++) {
22              counter.increment();
23          }
24      }
25
26      public static void main(String[] args) throws InterruptedException {
27          Counter counter = new Counter();
28          // Create and start multiple threads
29          Thread thread1 = new SynchronizedExample(counter);
30          Thread thread2 = new SynchronizedExample(counter);
31          thread1.start();
32          thread2.start();
33          // Wait for threads to finish
34          thread1.join();
35          thread2.join();
36          System.out.println("Final counter value: " + counter.getCount());
37      }
38  }
39
```

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

```
1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4  */
5  package multithreadapp2;
6
7  import java.util.concurrent.ExecutorService;
8  import java.util.concurrent.Executors;
9
10 /**
11  *
12  * @author USER
13  */
14 public class Task implements Runnable{
15
16     private int taskId;
17
18     public Task(int taskId) {
19         this.taskId = taskId;
20     }
21
22     @Override
23     public void run() {
24         System.out.println("Task " + taskId + " is being processed by " +
25             Thread.currentThread().getName());
26     }
27
28     public class ThreadPoolExample {
29         public static void main(String[] args) {
30             // Create a thread pool with 3 threads
31             ExecutorService executorService = Executors.newFixedThreadPool(3);
32             // Submit tasks to the pool
33             for (int i = 1; i <= 5; i++) {
34                 executorService.submit(new Task(taskId: i));
35             }
36             // Shutdown the thread pool
37             executorService.shutdown();
38         }
39     }
40 }
```

```
1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4  */
5  package multithreadapp2;
6
7  import java.util.concurrent.ExecutorService;
8  import java.util.concurrent.Executors;
9
10 /**
11  *
12  * @author USER
13  */
14 public class ThreadPoolExample {
15
16     public static void main(String[] args) {
17         // Create a thread pool with 3 threads
18         ExecutorService executorService = Executors.newFixedThreadPool(3);
19         // Submit tasks to the pool
20         for (int i = 1; i <= 5; i++) {
21             executorService.submit(new Task(taskId: i));
22         }
23         // Shutdown the thread pool
24         executorService.shutdown();
25     }
26 }
27
```

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

```
1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4  */
5  package multithreadapp2;
6
7  /**
8   *
9   * @author USER
10  */
11  public class ThreadLifecycleExample extends Thread{
12
13      @Override
14      public void run() {
15          System.out.println(Thread.currentThread().getName() + " - State: " +
16              Thread.currentThread().getState());
17          try {
18              Thread.sleep(2000); // Simulate waiting state
19          } catch (InterruptedException e) {
20              e.printStackTrace();
21          }
22
23          System.out.println(Thread.currentThread().getName() + " - State after sleep: " + Thread.currentThread().getState());
24      }
25
26      public static void main(String[] args) {
27          ThreadLifecycleExample thread = new ThreadLifecycleExample();
28          System.out.println(thread.getName() + " - State before start: " +
29              thread.getState());
30          thread.start(); // Start the thread
31          System.out.println(thread.getName() + " - State after start: " +
32              thread.getState());
33      }
34  }
35  }
```