Lab 6: Executor and Callable

In this lab, we will look into how we can create a thread pool and make use of the threads in the thread pool to execute tasks, and how to use the call() method (to replace the run() method that we have been using) from the Callable interface to return a value.

A. Executor

Before this, we have been creating different Runnable objects and pass them to the Thread class constructor to create threads that help to fetch the tasks for execution. It can be seen that there is a close connection between the task we defined, and the thread we created to run the task. In this case, the thread is created just to run a task, and the thread will turn to the dead state after the task has completed. If we need to create large number of threads to execute different tasks under different conditions, it can be quite tedious to manage all the threads. To overcome the issue, we can use the executor in Java and create a thread pool with certain number of reusable threads.

As shown below is an example that demonstrates how we can define the task we would like to execute in the run() method, and submit the task to the executor. In this case, the task is to calculate and print out the total of each row in a two-dimensional array. A thread pool with only 2 threads is created, but 4 tasks are submitted to the executor. Hence, at one time, only 2 tasks can be executed concurrently (you should be able to observe this from the outputs). The 2 threads will try to fetch new tasks from the queue (an unbounded queue) after they have completed the tasks assigned to them.

```
// Example 1
import java.util.concurrent.*;

class SumRow implements Runnable {
   private int[][] intArray;
   private int rowNumber;
   private int total;

   SumRow(int[][] intArray, int rowNumber) {
      this.intArray = intArray;
      this.rowNumber = rowNumber;
   }

   public void run() {
      total = intArray[rowNumber][0] + intArray[rowNumber][1];
      System.out.println("Row " + rowNumber + " Total: " + total);
      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}

// Continue next page.
```

```
public class ExecutorExample {
   public static void main(String[] args) {
      int[][] intArray = { {1,2},{3,4},{5,6},{7,8} };

      ExecutorService executor = Executors.newFixedThreadPool(2);

      executor.submit(new SumRow(intArray,0));
      executor.submit(new SumRow(intArray,1));
      executor.submit(new SumRow(intArray,2));
      executor.submit(new SumRow(intArray,3));

      executor.shutdown();
   }
}
```

Exercise 1

a) Create a two-dimensional array to store the data given below. You are not required to include those labels (Customer ID, Expenses, Jan – Dec, Total) into the array. In other words, only the cells shaded in green colour are required. Try to use a for loop to fill the array with the desired data.

Customer ID	Expenses												Total
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	
100	1	2	3	4	5	6	7	8	9	10	11	12	
101	13	14	15	16	17	18	19	20	21	22	23	24	
102	25	26	27	28	29	30	31	32	33	34	35	36	
103	37	38	39	40	41	42	43	44	45	46	47	48	
104	49	50	51	52	53	54	55	56	57	58	59	60	
105	61	62	63	64	65	66	67	68	69	70	71	72	
106	73	74	75	76	77	78	79	80	81	82	83	84	
107	85	86	87	88	89	90	91	92	93	94	95	96	
108	97	98	99	100	101	102	103	104	105	106	107	108	
109	109	110	111	112	113	114	115	116	117	118	119	120	

b) Calculate the total expenses (over the year) of each customer concurrently by using multiple threads. Write the total expenses into the last column of the array you created in part (a). Create a thread pool with 5 threads and use these 5 threads to complete all the calculations.

B. Callable

As mentioned above, the run() method in the Runnable interface is unable to return a value. To work around with this, we can write the value into a shared structure (or a file), and then retrieve the value from there later. But if you need a method that can return a value, then you can use the call() method from the Callable interface. Similarly, we just need to define what we need a thread to do in the call() method. Of course, we also need to specify the return type when writing the call() method.

As shown below is an example that demonstrates how we can define the task we would like to execute in the call() method, and submit the task to the executor. In this case, the task is to sum all the values from 0 to 9 (the total is 45). Just like the previous example, a thread pool with only 2 threads is created. Since only two tasks are submitted this round, the two tasks will be executed concurrently.

The return value (the total) of each task will be stored in the respective Future object (futureA and futureB), after the calculation is completed. We can then access the value stored in each Future object by calling the get() method. Please take note that the get() method should be enclosed in a try-catch block.

```
//Example 2
import java.util.concurrent.*;

class SumValue implements Callable<Integer> {
   public Integer call() throws Exception {
      int total = 0;
      for (int i = 0; i < 10; i++) {
         total = total + i;
      }
      return total;
   }
}

public class CallableExample {
   public static void main(String[] args) {
      ExecutorService executor = Executors.newFixedThreadPool(2);

      SumValue sumValueA = new SumValue();
      SumValue sumValueB = new SumValue();

      Future<Integer> futureA = executor.submit(sumValueA);
      Future<Integer> futureB = executor.submit(sumValueB);

      int totalA = 0;
      int totalB = 0;

      try {
         totalA = futureA.get();
         totalB = futureB.get();
      } catch (InterruptedException | ExecutionException e) {
         e.printStackTrace();
      }

      System.out.println("The total sum is: " + (totalA + totalB));

      executor.shutdown();
   }
}
```

<u>Exercise 2</u>

a) Create a two-dimensional array to store the data given below. Similarly, you can try to use a for loop to fill the array with the desired data.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

b) Try to find the largest value in the array by using multiple threads. For example, you can divide the array into two parts (left and right). One thread will try to find the largest value in the left part while another thread will try to find the largest value in the right part. You can then compare the values returned from the two threads to determine the largest value of the entire array. In addition to this, you can also try to divide the array into four quadrants, instead of just two parts.