Lab 3: Volatile Variable, Synchronized Block, and Wait and Notify

As suggested by the title, we will be looking into the use of (i) volatile variable, (ii) synchronized block, and (iii) the wait() and notify() methods in this lab.

A. Volatile Variable

Atomicity and visibility are two important properties of a concurrent program (a program with threads that can run concurrently). While synchronization can help us to achieve both atomicity and visibility, volatile variable can guarantee only visibility.

When a variable is declared as a volatile variable, all the other threads will be able to read the most updated value. The most updated value must be written back to the main memory before it is accessed by another thread. Volatile variable can be used when multiple threads could be reading the variable, but only one of them is allowed to make changes. As shown below is an example of how we can declare a volatile variable by using the volatile keyword.

```
//Example 1
class Counter {
        volatile private int value = 0;

        public void increaseCounterValueByOne() {
                value = value + 1;
        }

        public int getCounterValue() {
                return value;
        }
}
```

Exercise 1

a) Based on the Counter class shown in example 1, create 6 threads that each will call the increaseCounterValueByOne() method for 10,000 times.
b) Run the code several times and compare the results with those that you obtained using the synchronized methods (from Lab 2 – Exercise 3). Describe your observation.

B. Synchronized Block

Synchronization is often considered as a "stronger" approach when compared with volatile variable, because it has greater impacts on the performance (e.g. execution speed). When a method is "guarded" by synchronization, the threads will need to first compete for the lock associated with an object. When one of the threads is executing the method, the other threads will need to wait. This hurts concurrency since the threads are unable to run concurrently. Therefore, the general idea is to ensure that we only apply synchronization when it is really necessary to do so. For example, we should try to minimize the scope (or section of code) that should be synchronized whenever it is possible.

Instead of synchronizing the entire method, which might not be necessary, we can also choose to only synchronize a smaller section of code (critical section) by using the synchronized block. It works in the same way when compared with synchronized method. The threads are still required to compete for the lock associated with an object. But in the case of synchronized block, the lock associated with an object other than "this" can be used. As shown below is an example of how you can make use of the synchronized block to synchronize only the critical section in a method.

```
//Example 2
class ModifiedCounter {
        private int valueOutsideBlock = 0;
        private int valueInsideBlock = 0;

        public void increaseCounterValueByOne() {
                valueOutsideBlock = valueOutsideBlock + 1;
                synchronized(this) {
                        valueInsideBlock = valueInsideBlock + 1;
                }
        }

        public int getCounterValueOutsideBlock() {
                        return valueOutsideBlock;
        }

        public int getCounterValueInsideBlock() {
                synchronized(this) {
                        return valueInsideBlock;
                }
        }
}
```

Exercise 2

a) Similarly, based on the ModifiedCounter class shown in example 2, create 6 threads that each will call the increaseCounterValueByOne() method for 10,000 times.
b) Run the code several times and compare the results with those that you obtained using the synchronized methods (from Lab 2 – Exercise 3). Describe your observation.
c) Check and compare the values stored in valueInsideBlock and valueOutsideBlock. Describe your observation.

C. Wait and Notify

In addition to the join() method, we can also use the wait() and notify() methods to coordinate the execution order of multiple threads. Different from the join() method, both wait() and notify() methods must be executed within a synchronized method or synchronized block, and you are also required to enclose the wait() method in a try-catch block.

On one hand, a thread can be put onto the "waiting list" associated with an object when the wait() method is called. In this case, the thread is merely using the "waiting list" provided by an object (similar to utilizing the lock associated with an object), it does not mean that the thread is waiting for the

object. On the other hand, threads in the “waiting list” can be “woke up” when the notify() or notifyAll() method is called. The former will randomly “wake up” one of the threads in the list, whereas the latter will “wake up” all the threads. However, it is also possible for a thread to “wake up” by itself without using the notify() method. This can be done by setting the time out period when calling the wait() method. For example, calling the wait(1000) method will cause a thread to “wake up” automatically after 1000ms.

As shown below is an example that demonstrates the behaviour of a chef and a customer in a restaurant that offers lobster buffet. To start, the chef will first check the lobster availability on the buffet line. If the quantity is more than 0, then the chef will do nothing and wait (call the wait() method) inside the restaurant (waiting list). The customer is allowed to get as many lobsters as he/she wants from the buffet line, but not more than what is available. If all the lobsters are taken, the customer will have to call the chef to refill the buffet line with new lobsters (execute the notify() method), and wait (call the wait() method) inside the restaurant (waiting list) for the chef to cook. The chef will also inform the customer after the buffet line is refilled with new lobsters (execute the notify() method), so that he/she knows it is time to eat for another round.

```
//Example 3
import java.util.Random;

class BuffetLine {
	private int lobsterQuantity = 0;
	Random rand = new Random();

	public synchronized void addLobster() {
		if (lobsterQuantity > 0) {
			try {
				wait();
			} catch(InterruptedException e) {
				e.printStackTrace();
			}
		}
		lobsterQuantity = lobsterQuantity + rand.nextInt(10);
		System.out.println("Chef Refill " + lobsterQuantity + " Lobster(s)");
		notify();
	}

	public synchronized void getLobster() {
		if (lobsterQuantity == 0) {
			try {
				wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		int lobsterToGet = rand.nextInt(lobsterQuantity) + 1;
		lobsterQuantity = lobsterQuantity - lobsterToGet;
		System.out.println("Customer Took " + lobsterToGet + " Lobster(s)");
		notify();
	}
}
```

```
class Chef implements Runnable {
	BuffetLine buffetLine;

	Chef(BuffetLine buffetLine) {
		this.buffetLine = buffetLine;
	}
	public void run() {
		for(int i=0; i<10; i++) {
			buffetLine.addLobster();
		}
	}
}

class Customer implements Runnable {
	BuffetLine buffetLine;

	Customer(BuffetLine buffetLine) {
		this.buffetLine = buffetLine;
	}
	public void run() {
		for(int i=0; i<10; i++) {
			buffetLine.getLobster();
		}
	}
}

public class Ex3 {
	public static void main(String[] args) {
		BuffetLine buffetLine = new BuffetLine();
		Thread chef = new Thread(new Chef(buffetLine));
		Thread customer = new Thread(new Customer(buffetLine));

		chef.start();
		customer.start();
	}
}
```

Exercise 3

a) Modify the code given in example 3, so that the chef will "wake up" regularly and check if he/she would need to refill lobsters. You might also need to adjust how frequent the customer would take the lobsters from the buffet line. To do this, you can use the sleep() method.
b) Sometimes the application might not be able to stop properly. Identify why this could happen and modify the code given in example 3 to fix the issue.

Exercise 4

a) Create a code that could simulate the following scenario of a restaurant by using wait and notify.
   - A chef will do nothing until he/she is notified by a waitress/waiter that there is an order he/she would need to process.
   - The chef will notify a waitress/waiter when the order is ready to serve.
   - There are two chefs and two waitresses/waiters in the restaurant.
   - An order can be processed by either one of the chefs, and similarly the order can be served to the customers by either one of the waitresses/waiters.

Hints:

- You should have the following four classes in your code, (i) chef, (ii) waitress/waiter, (iii) order, (iv) main. You are **not** required to create a customer class for this exercise.
- You can adapt some of the code (or concepts) from example 3.