Lab 2: Priority, Scheduling, and Basic Synchronization

In this lab, we will try to schedule the execution of multiple threads by using setPriority(), yield(), and join(). In addition to this, we will also look into what would happen when multiple threads are trying to update a shared variable at the same time, and how we can ensure that we will always get the correct answer by using synchronization.

A. Thread Priority

As shown below is an example of how we can (i) set the priority of a thread by using the setPriority() method, and (ii) check the priority of a thread by using the getPriority() method.

```
//Example 1
class PrintNameAndValue implements Runnable {
        public void run() {
                for(int x = 1; x <= 10; x++) {
                        System.out.println("By " + Thread.currentThread().getName() + ", x is " + x);
                }
        }
}

public class TestClass1 {
        public static void main ( String[] args ) {
                PrintNameAndValue pnav = new PrintNameAndValue();
                Thread t0 = new Thread(pnav);
                Thread t1 = new Thread(pnav);
                Thread t2 = new Thread(pnav);

                // Set the priority of thread t0 and t2.
                t0.setPriority(1);
                t2.setPriority(10);

                // Check the priority of thread t2.
                System.out.println("Priority of thread t2 is " + t2.getPriority());

                t0.start();
                t1.start();
                t2.start();
                System.out.println("This is the end of main() method.");
        }
}
```

In Java, the priority is represented as a number, whereby 1 represents the lowest priority and 10 represents the highest priority. But instead of trying to set the priority by using a number, we can also choose to replace the number with the following static variables (defined in the Thread class):

- MIN_PRIORITY, which is equivalent to priority number of 1.
- NORM_PRIOTIRY, which is equivalent to priority number of 5.
- MAX_PRIORITY, which is equivalent to priority number of 10.

They can be better candidates than using the numbers because they always map to the lowest priority level, normal priority level, and highest priority level when run on different OS.

Exercise 1

a) Try to find out what is the default priority of the main thread and thread t1. What conclusion can you draw from the looking at the priority numbers?
b) Replace the priority number in example 1 with the static variables and run the code for several times. Describe what you can observe from the output.

B. Yield Control to Another Thread

Similar to the sleep() method, yield() is also a static method (but sleep() and yield() work differently). As shown below is an example of how we can use the yield() method to hint the scheduler that the thread is ready to pause. In the following example, thread t0 can stop after every single print out, but the effect is not guaranteed. Overall, this method is not frequently used. This is only useful when small tuning of execution order is really necessary.

```
// Example 2
class GoodThread implements Runnable {
        public void run() {
                for(int x = 1; x <= 10; x++) {
                        System.out.println("By " + Thread.currentThread().getName() + ", x is " + x);
                        Thread.yield();
                }
        }
}

class NormalThread implements Runnable {
        public void run() {
                for(int x = 1; x <= 10; x++) {
                        System.out.println("By " + Thread.currentThread().getName() + ", x is " + x);
                }
        }
}

public class TestClass2 {
          public static void main ( String[] args ) {
                  Thread t0 = new Thread(new GoodThread());
                  Thread t1 = new Thread(new NormalThread());
                  t0.start();
                  t1.start();
          }
}
```

C. Join the "End" of Another Thread

When compared with setPriority() and yield(), the join() method has a more deterministic effect when comes to setting the execution order. For example, when a thread called t0.join(), the thread that executed this must wait for thread t0 to finish before it can continue to run. This allow us to schedule

the execution of certain threads, so that they can only start after certain conditions are met. As shown below is an example of how you can schedule thread t1 and thread t2 to start only after thread t0 has finished.

```
// Example 3
class MyThread implements Runnable {
        public void run() {
                for(int x = 1; x <= 10; x++) {
                        System.out.println("By " + Thread.currentThread().getName() + ", x is " + x);
                }
        }
}

public class TestClass3 {
        public static void main ( String[] args ) {
                Thread t0 = new Thread(new MyThread());
                Thread t1 = new Thread(new MyThread());
                Thread t2 = new Thread(new MyThread());

                t0.start();
                // Remember to include in a try-catch block.
                try {
                        t0.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                t1.start();
                t2.start();

                System.out.println("This is the end of main() method.");
        }
}
```

Exercise 2

a) Based on example 3, create another thread t3, and try to schedule the execution of these four threads based on the following order: t3 -> t2 -> t0 -> t1. In order words, thread t2 should only start after thread t3 has finished, and so on.
b) Compare sleep(), setPriority(), yield(), and join() in terms of how they could be used to affect the execution order of multiple threads. No coding is required for this.

D. Synchronization

When you have several threads that will attempt to modify a shared variable, we need to "synchronize" the threads to ensure that we can always get the correct answer. There are several ways to achieve this and we are going to cover the most basic one first, which is by using the synchronized keyword. In this case, all you need to do is to add the synchronized keyword to methods that will be accessing the shared variable. As shown below is an example of how we can add the synchronized keyword to the two methods in the BankAccount class.

```
// Example 4
class BankAccount {
	private int balance = 0;

	public synchronized void changeBalance(int amount) {
		balance = balance + amount;
	}

	public synchronized int checkBalance() {
		return balance;
	}
}

class GoodPerson implements Runnable {
	BankAccount bankAccount;
	private int amount = 0;

	GoodPerson(BankAccount bankAccount, int amount) {
		this.bankAccount = bankAccount;
		this.amount = amount;
	}

	public void run() {
		bankAccount.changeBalance(amount);
	}
}

public class TestClass {
	public static void main(String[] args) {
		BankAccount myBankAccount = new BankAccount();
		Thread goodPerson1 = new Thread(new GoodPerson(myBankAccount, 1000));
		Thread goodPerson2 = new Thread(new GoodPerson(myBankAccount, -500));

		goodPerson1.start();
		goodPerson2.start();

		while (goodPerson1.isAlive() || goodPerson2.isAlive()) {
			//This is meant to be empty.
			//Wait until both have finished only proceed to check the balance.
		}

		System.out.println("Balance: " + myBankAccount.checkBalance());
	}
}
```

After the synchronized keyword is added, a thread will need to first acquire the “lock” associated with the object, before it can proceed to run the method “guarded” by the synchronized keyword. In this example, the lock that belongs to the bankAccount object is used (each object in Java comes with a lock). Hence, when one thread is executing the synchronized method, no other thread can run the other methods that also have the synchronized keyword.

You may try to run the example, but I think most of the time you will be able to get the correct answer because we are not doing any excessive number of modification to the shared variable (balance in bankAccount), and we only have two threads here.

Exercise 3

a) Create a class that can be used as a counter. Three methods should be included in the class:

Method Name	Description
setCounterValue(int x)	To set the counter value.
increaseCounterValueByOne()	Increase the counter value by 1.
getCounterValue()	Return the counter value.

b) Initialize the counter value to 0.
c) Create 6 threads that each will call the increaseCounterValauByOne() method for 10,000 times.
d) Compare the results with and without using the synchronized keyword.