CSCE 314 [Sections 500, 501] Programming Languages – Spring 2020
Hyunyoung Lee
Assignment 7
Assigned on Wednesday, April 15, 2020

Electronic submission to eCampus due **at 10:00 p.m., Monday, 4/27/2020.**

By electronically submitting this assignment to eCampus by logging in to your account, you are signing electronically on the following Aggie Honor Code:

"On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment."

In this assignment, you will practice concurrent programming using Java threads. You will earn total 120 + 10 extra points = 130 points.

Note 1: This homework set is an **individual** homework, not a team-based effort. Discussion of the concept is encouraged, but actual write-up of the solutions must be done individually.

Note 2: Turn in on eCampus one `yourLastName-yourFirstName-hw7.zip` file that contains all your java source codes (.java files) and a **README** file that explains how to compile (using the javac command in the terminal), how to run your program (using the java command in the terminal) and what is an expected output of your code when run.
So, once unzipped, it should expand to a subdirectory `yourLastName-yourFirstName-hw7` that contains all of your source files and the README file. No .class files please.

Note 3: All Java code that you submit must compile without any error using `javac` of Java 8 (or higher version). If your code does not compile, you will likely receive zero points for this assignment.

Note 4: Remember to put the head comment in your files, including your name, your UIN, and *acknowledgements of any help received* in doing this assignment. Again, remember the honor code.

Important: Before solving these problems, carefully read Chapter 14 (especially, the first seven sections) of the Java textbook.

Problem 0. (10 Points) README. Explain in a README.txt file how to compile (using the javac command in the terminal) and execute your codes (using the java command in the terminal), and what is an expected output of your codes when run (putting all or some part of the output while you are test running your code into your readme file is also a good idea). It should be detailed enough so that whoever grades your codes can understand what you were doing with your code clearly and should be able to reproduce your tests following what you wrote in it.

Problem 1. (80 points) Printing Server (Modified Exercise 14.2, page 345 of the textbook.) Consider the following print server that outputs the requested messages to the computer screen as follows: Client threads invoke the `printRequest` method to submit the messages (strings) to be output. But all the `printRequest` method actually does is place the message in the print job queue, and a separate (manager) thread then dequeues messages from the job queue and outputs them to the screen. A crude framework of the PrintServer class can be given as below.

```
public class PrintServer implements Runnable {
   private final Queue<String> requests = new LinkedList<String>();
   public PrintServer() {
      new Thread(this).start();
   }
   public void printRequest(String s) {
      requests.add(s);
   }
   public void run() {
      for(;;) realPrint(requests.remove());
   }
   private void realPrint(String s) {
      // do the real work of outputting the string to the screen
   }
}
```

The code above does not work as it is. First, you need to come up with an idea how to make the `requests` job queue shared among the threads. There can be different ways to implement the shared job queue, one simplest way would be to make it a `static` object. Since it is shared among multiple threads, you need to take care of the synchronization issue. You will practice two different programming constructs to solve the synchronization issue (see details below).

Second, as it is, when you run it, you will get "NoSuchElement" exception because every thread tries to remove elements from the request queue (without adding anything!). In comparison to the BankAccount example, this can be a similar situation to when overdraft occurs. Thus, this problem can also be resolved by carefully synchronizing the accesses to the shared request queue and checking whether the queue is not empty before trying to remove a message. If the queue is empty, the manager thread needs to wait until client threads add some messages, and the client thread, after adding a message, will let the manager thread know by signaling.

Only the thread whose title is "manager" can remove an element from the queue, and a thread whose title is not "manager" can only become a client.

Please write two versions of `PrintServer` and one test class as below.

Version 1. (40 points) Use lock and condition objects explicitly (as you have seen in the BankAccount example). For this version, the structure of your code will be similar to the BankAccount example code as follows. In the BankAccount example, the account balance is shared among the withdrawing (who takes the amount from the account) and depositing (who adds the amount to the account) threads, whereas in PrintServer, the request queue is shared among the manager (who takes the messages from the queue) and client (who adds the messages to the queue) threads. Name the class for this version `class PrintServerV1`. The class header should be: `public class PrintServerV1 implements Runnable`. Place the class definition in a file named PrintServerV1.java.

Version 2. (30 points) Use `synchronized` methods (or blocks). The basic idea is the same as in Version 1, but practice different programming constructs for synchronization. Name the class for this version `class PrintServerV2`. The class header should be:

`public class PrintServerV2 implements Runnable`. Place the class definition in a file named PrintServerV2.java.

Meaningful message composition. (10 points) Make the messages added by the clients easy to read/understand by adding the client ID, a serial number of the message by the client, and its class information (PrintServerV1 or PrintServerV2, you can get this information by invoking `getClass()` method on the object) as part of the message.

The `main` method can be as simple as follows:

```
public static void main(String[] args) { // for PrintServerV1
  // The following invocations of the constructor,
  // the first argument is title and the second argument is ID
  PrintServerV1 m = new PrintServerV1("manager", 1);
  PrintServerV1 c1 = new PrintServerV1("client1", 2);
  PrintServerV1 c2 = new PrintServerV1("client2", 3);
  new Thread(m).start();
  new Thread(c1).start();
  new Thread(c2).start();
}
```

Or, if you put the statement `new Thread(this).start();` at the end of the constructor code, then the main can be even simpler without the last three lines of creating Thread and starting it.
Exactly the same `main()` with `V1`s replaced with `V2`s can be used as the `main()` method for `PrintServerV2`. Include the `main` method that tests your implementation in each of your `class PrintServerV1` and `class PrintServerV2`.

Problem 2. (40 points) Time and messages. (Modified Exercise 14.6, page 358 of the textbook.) Consider a shared counter whose values are non-negative integers, initially zero. A *time-printing thread* increments the counter by one and prints its value each second from the start of execution. A *message-printing thread* prints a message every *five* seconds. Have the message-printing thread be notified by the time-printing thread as each second passes by. Add another message-printing thread that prints a different message every *eleven* seconds. Such addition must be done without modifying the time-printing thread implementation.

Have all involved threads share the counter object that is updated by the time-printing thread every second. The time-printing thread will notify other threads to read the counter object each time it updates the counter, then each message-printing thread will read the counter value and see if its assigned time period has elapsed; if so, it will print its message.

The `main` method can be as simple as:

```
public static void main(String[] args) {
  Time counter = new Time();
  // counter is shared among all three runnables below
  TimePrinting tp = new TimePrinting(counter);
  MessagePrinting mp5 = new MessagePrinting(counter, 5);
  MessagePrinting mp11 = new MessagePrinting(counter, 11);
}
```

Again, instead of creating and starting threads in `main()`, you can include the task in the constructor of each runnable class by placing `new Thread(this).start();` as the last step of the constructor.
Include the `main()` in the `Time` class, and the class header of `Time` should be `public class Time`. In addition, you will have two classes, `TimePrinting` and `MessagePrinting`, both of which implement `Runnable`. Put them in the same file as class `Time` in the Time.java file.
The output should look something like this:

```
1 2 3 4 5
-- 5 second message
6 7 8 9 10
-- 5 second message
11
***** 11 second message
12 13 14 15
-- 5 second message
16 17 18 19 20
-- 5 second message
21 22
***** 11 second message
23 24 25
-- 5 second message
 . . .
```

As summary, your .zip directory should contain the following files, nothing else.

1. README.txt
2. PrintServerV1.java (contains public class `PrintServerV1`)
3. PrintServerV2.java (contains public class `PrintServerV2`)
4. Time.java (contains public class `Time`, class `TimePrinting` and class `MessagePrinting`)

Have fun!