CSCE 314 Programming Languages – Fall 2015
Hyunyoung Lee
Assignment 8
Assigned on Monday, November 23, 2015

Electronic turn-in on eCampus due at 9:00 a.m., Wednesday, Dec. 9, 2015
Signed *coversheet* due at the beginning of class on Wednesday, Dec. 9, 2015
If you do not turn in a signed coversheet your work will not be graded.

"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."

Typed or printed name of student	Section (501 or 502)
Signature of student	UIN

Note 1: This homework set is *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-a8.zip` or `.tar` file that contains all your Java programs and a readme.txt that you deem necessary for the grader to compile and run your programs.

Note 3: All Java code that you submit must compile without errors using the `javac` command of Java 8 (no IDE please). If your code does not compile, you will likely receive zero points for this assignment.

Note 4: Remember to put the head comments in your file, including your name and acknowledgements of any help received in doing this assignment. Remember the honor code.

Note 5: **Due to the limited time for grading, submissions over two days late will not be accepted** (even with the usual late penalty).

You will earn total 140 points.

Problem 1. (40 points) Synchronization. Below is a sketch of a messaging client-server implementation:

```
import java.util.*;

class PostBox implements Runnable {
  static final int MAX_SIZE = 10;

  class Message {
    String sender;
    String recipient;
    String msg;
    Message(String sender, String recipient, String msg) {
      this.sender = sender;
      this.recipient = recipient;
```

```
      this.msg = msg;
    }
  }

  private final LinkedList<Message> messages;
  private LinkedList<Message> myMessages;
  private String myId;
  private boolean stop = false;

  public PostBox(String myId) {
    messages = new LinkedList<Message>();
    this.myId = myId;
    this.myMessages = new LinkedList<Message>();
    new Thread(this).start();
  }

  public PostBox(String myId, PostBox p) {
    this.myId = myId;
    this.messages = p.messages;
    this.myMessages = new LinkedList<Message>();
    new Thread(this).start();
  }

  public String getId() { return myId; }
  public void stop() {
    // make it so that this Runnable will stop
  }

  public void send(String recipient, String msg) {
    // add a message to the shared message queue
  }

  public List<String> retrieve() {
    // return the contents of myMessages
    // and empty myMessages
  }

  public void run() {
    // loop forever
    //   1. approximately once every second move all messages
    //      addressed to this post box from the shared message
    //      queue to the private myMessages queue
    //   2. also approximately once every second, if the message
    //      queue has more than MAX_SIZE messages, delete oldest messages
    //      so that size is at most MAX_SIZE
  }
}
```

A `PostBox` object is both a server and a client to the messaging system. In its role as a client, it maintains the client's id (`myId`). This id is attached to every message sent to the post box. It also maintains a personal message queue (`myMessages`). In its role as a server, a post box has access to a message queue (`messages`) shared by all clients.

`PostBox` has two constructors. The first one creates a new message queue. The second one takes an existing post box as a parameter and uses the message queue of that post box. Both constructors start running the server on a new thread. The messaging system stays alive as long as there is at least one participant; after construction, there is no difference in how the post box that was created first or how the other post boxes operate.

The `send()` method adds a message to the shared message queue. The `retrieve()` method returns the contents of the `myMessages` queue as a list of strings (it has to compose a nicely formatted string from the `Message` structure), and then clears the `myMessages` queue. The `run()` method of `PostBox` periodically wakes up to iterate through the message queue and look for messages whose recipient is the current post box. All such messages are removed from the shared message queue, and moved to the private (non-shared) `myMessages` queue.

Task. Finish the implementation of the `PostBox` class. Make sure it is properly synchronized. Please explain your use of synchronization primitives in comments – Why did you make which parts of code protected by what lock? Think about when if ever does one need to lock the shared message queue, the private message queue, and the entire post box object. Be careful not to create the possibility of deadlock. The test program below shows how the `PostBox` class can be used.

```
public class Main1 {
  static void pause(long n) {
    try { Thread.sleep(n); } catch (InterruptedException e) {}
  }
  public static void main (String[] args) {
    final String bond    = "Bond";
    final String blofeld = "Blofeld";
    final String osato   = "Mr. Osato";

    final PostBox pBond    = new PostBox(bond);
    final PostBox pBlofeld = new PostBox(blofeld, pBond);
    final PostBox pOsato   = new PostBox(osato, pBond);

    // send out some messages on another thread
    new Thread( new Runnable() {
      public void run() {
        pBond.send(blofeld, "Yes, this is my second life"); pause(1000);
        pBlofeld.send(bond, "You only live twice, Mr. Bond."); pause(500);
        String msg = "I gave Number 11 the strictest orders to eliminate him.";
        pOsato.send(blofeld, msg); pause(2000);
        pOsato.send(bond, msg);
        for (int i=0; i<20; ++i) pOsato.send(bond, "flooding the message queue...");
      }
     }).start();
```

```
    PostBox[] boxes = { pBond, pBlofeld, pOsato };
    long startTime = System.currentTimeMillis();

    // poll for messages in a loop for 5 secs
    while (true) {
      for (PostBox box : boxes) {
        for (String m : box.retrieve()) System.out.println(m);
      }
      if (System.currentTimeMillis() - startTime > 5000) break;
    }

    // stop each mailbox
    for (PostBox box : boxes) {
      box.stop();
    }
  } // end of main()
} // end of Main1
```

The output of the above program varies from one run to another. The following is one possible output:

```
From Bond to Blofeld: Yes, this is my second life
From Blofeld to Bond: You only live twice, Mr. Bond.
From Mr. Osato to Blofeld: I gave Number 11 the strictest orders to eliminate him.
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
From Mr. Osato to Bond: flooding the message queue...
```

Problem 2. (40 points) Threads, threads, more threads. (Modified from 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 fifteen 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 seven 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.

Write `Main2` class that contains the `main` method that tests your implementation. The output should look as follows:

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

Problem 3. (30 points) Reflection.

Task 1. Write a method `displayMethodInfo` (as a method of a class of your choosing), with the following signature:

```
static void displayMethodInfo(Object obj);
```

The method writes, to the standard output, the *type* of the methods of `obj`. Please format the method type according to the following example:
Let `obj` be an instance of the following class:

```
class A {
  void foo(T1, T2) { ... }
  int bar(T1, T2, T3) { ... }
  static double doo() { ... }
}
```

The output of `displayMethodInfo(obj)` should be as follows

```
foo (A, T1, T2) -> void
bar (A, T1, T2, T3) -> int
doo () -> double
```

As you can see, the *receiver* type should be the first argument type. Methods declared `static` do not have a receiver, and should thus not display the class type as the first argument type.

Task 2. Write a class named `Main3`, which, in its `main` method, tests your `displayMethodInfo` implementation.

Problem 4. (30 points) More reflection, command line arguments and automated testing.

Task 1. Write a program (your main method should be in class `Main4`) that (i) takes a class name as a command line argument; (ii) in that class locates all `public static` methods

1. that take no parameters,
2. return a `bool`, and
3. whose name starts with `test`;

and (iii) invokes those methods.

If such a method returns `true`, your program should write

`OK: testX succeeded`

where `testX` is the name of the tried test method. In the case of `false` the output should be

`FAILED: testX failed`

Task 2. Write a class `MyClass` that has some test methods, and try your program with that class.