CSCE 314 [Sections 595, 596, 597] Programming Languages – Spring 2024
Hyunyoung Lee
Homework Assignment 6
Assigned on Monday, April 8, 2024

Electronic submission on Canvas due **at 11:59 p.m., Friday, April 19, 2024**
By electronically submitting this assignment to Canvas 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 Java generics (wildcards) and Java concurrency. You will earn total 120 points. Here are some general instructions.

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 by yourself. Your final product – the code as well as comments, explanations, the README file – should never be shared.
2. Read the problem descriptions and requirements carefully! There may be significant penalties for not fulfilling the requirements.
3. **Explain each line (or a small block) of your code in your own words.** In addition, some problems ask you to explain the working of your code with the given test scenario in the main method. Your explanation must be consistent with your implementation. Your work will be graded not only on the correctness of your answer, but also on the consistency and clarity with which you express it.
4. Turn in one *yourFirstName-yourLastName-hw6.zip* file on Canvas, nothing else. Your zip directory must include the four files, nothing else: One README.txt plain text file (see Problem 1 statements below; it has three parts) and the three .java files – Market.java (for Problem 2) and SimBox.java and SimMain.java (for Problem 3).
5. All Java code that you submit must compile without errors using `javac` (at the terminal) of Java version 11 or higher. If your code does not compile using javac, you risk receiving zero points for this assignment.
6. Remember to put the head comment in *all* of your files *including the README.txt file*, including your name, your UIN, and *acknowledgements of any help received* in doing this assignment/problem. Again, **remember the honor code!**

Problem 1. (30 Points) This problem has three parts. All three parts are answered in the README.txt file, where you put Part 1, Part 2, and Part 3 in this order. *Clearly mark where each part begins.* (See the skeleton file.)

[Part 1.] (10 Points) Explain in the README.txt file (a plain text file) how to compile (using the javac command) and execute your codes (using the java command), and what the expected output of your codes is when tested (desirably for each problem/task). It should be detailed enough so that whoever grades your code can understand what you were doing with your code clearly and should be able to reproduce your tests following what you wrote in your README.txt. *For Problem 3, show at least two executions and explain why the output is not always the same.*

[Part 2.] (10 Points) Using the provided `main()` and your implementation of class Market in Problem 2, explain in your own words how the wildcard works (be specific and to the point!), and (in a separate paragraph) compare and explain what would happen without wildcards in Problem 2, that is, just using type variables. Be specific and to the point!

[Part 3.] (10 Points) For Problem 3, explain (in one or two paragraphs) why you need to synchronize those message queues using the example messaging scenarios provided in `SimMain`. Also, explain (in a separate paragraph) why/how your implementation does not create the possibility of deadlock. Be specific and to the point!

Problem 2. (30 points) Practice wildcards at the market.

Below is a `Market<T>` class that maintains a stock of objects of type `T`. There are two `sell` methods (a customer sells what they have to the market, thus the items are added to the stock of the market): one for selling a single item of type `T` and another for selling all items in a `List<T>`. There are two `buy` methods (a customer buys from the market, thus reducing the stock of the market): one to buy a single item of type `T` and the other to buy n items at once to be placed into the customer's basket passed as an argument to the parameter (`List<T> items`) of the `buy` function. Here, if the customer wants to buy more than what are in stock, the customer should be able to buy whatever in stock and should be informed that now the market is sold out.

```
import java.util.LinkedList;
import java.util.List;

public class Market<T> {
  List<T> stock; // stock of the market

  public Market() { stock = new java.util.LinkedList<T>(); }

  void sell(T item) {
    // implement this method
  }
  public T buy() {
    // implement this method
  }
  void sell(List<T> items) { // modify the parameter type
    // implement this method
  }
  void buy(int n, List<T> items) { // modify the parameter type
    // implement this method
  }
```

Modify the class so that you can buy items into and sell items from any `Collection` type, not just `List`. Also, the element types of the collection should not have to match exactly when buying and selling: allow for maximally flexible (but still static) typing using wildcards. The test class `Main` with a `main()` method is provided.
An example output is as below.

```
Here's what I bought
Apple
Gala
Apple
Apple
Gala
Fruit
Enjoy!
```

Have fun at the farmer's market!

Problem 3. (60 points) The Simpsons messaging system. Below is a sketch of a messaging system `SimBox` for The Simpsons.

```
class SimBox 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 SimBox(String myId) {
    messages = new LinkedList<Message>();
    this.myId = myId;
    this.myMessages = new LinkedList<Message>();
    new Thread(this).start();
  }

  public SimBox(String myId, SimBox s) {
    this.messages = s.messages;
    this.myId = myId;
    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 (messages)
    // you will have to synchronize the message queue
  }

  public List<String> retrieve() {
    // return the contents of myMessages
    // and empty myMessages
    // you will have to synchronize myMessages
    // each message should be in the following format:
    //   From (the sender) to (the recipient) (actual message)
  }

  public void run() {
  // loop forever
  // 1. Approximately once every second move all messages
  //    addressed to this mailbox from the shared message queue
  //    to the private myMessages queue
  //    To do so, you need to synchronize messages and myMessages.
  //    Furthermore, you need to explicitly use the iterator() of messages
  //    with a while loop.  A for-each loop will not work here.
  // 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. This part of code is provided
  //    below (see the skeleton code for more details)

    for(;;) {
       . . .
    } // endfor
  } // end run()
}
```

A `SimBox` 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 mailbox. It also maintains a personal message queue (`myMessages`). In its role as a server, a mailbox has access to a message queue (`messages`) shared by all clients.

`SimBox` has two constructors. The first one creates a new message queue. The second one takes an existing SimBox as a parameter and uses the message queue of that SimBox as the shared message queue. Both constructors start running the SimBox object 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 mailbox that was created first or the other mailboxes 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 `SimBox` periodically wakes up to iterate through the message queue and look for messages whose recipient is the current mailbox; if so, the messages are removed from the shared message queue, and moved to the private `myMessages` queue.

Task. Complete the implementation of the `SimBox` class. Make sure it is properly synchronized.

Explain in the README.txt file (in one or two paragraphs) why you need to synchronize those message queues using the example messaging scenarios provided in `SimMain`. Also, explain (in a separate paragraph) why/how your implementation does not create the possibility of deadlock. Be specific and to the point! This is the third part (ten points) of Problem 1.

The test program in class SimMain shows how the `SimBox` class can be used/tested.

```
class SimMain {
  static void pause(long n) {
    try { Thread.sleep(n); } catch (InterruptedException e) {}
  }

  public static void main (String[] args) {
    final String homer = "Homer"; // "My doctor said don't walk."
    final String marge = "Marge"; // "That was a traffic signal!"
    final String bart  = "Bart";  // "There's a 4:30 in the morning now?"

    final SimBox sHomer = new SimBox(homer);
    final SimBox sMarge = new SimBox(marge, sHomer); // shares sHomer.messages
    final SimBox sBart  = new SimBox(bart, sHomer);  // shares sHomer.messages

    // send out some messages on another thread
    new Thread( new Runnable() {
      public void run() {
        sHomer.send(marge, "My doctor said don't walk."); pause(1000);
        sMarge.send(homer, "That was a traffic signal!"); pause(500);
        String msg = "There's a 4:30 in the morning now?";
        sBart.send(homer, msg); pause(500);
        sHomer.send(bart, "D'oh!"); pause(500);
        //sBart.send(homer, msg);
        for (int i=0; i<20; ++i) {
            sBart.send(homer, "flooding the message queue...");
        }
      } // end run()
    } ).start();

    SimBox[] simpsons = { sMarge, sHomer, sBart };
    long startTime = System.currentTimeMillis();

    // poll for messages in a tight loop for 5 secs
    while (true) {
      for (SimBox aSimpson : simpsons)
        for (String m : aSimpson.retrieve()) System.out.println(m);
      if (System.currentTimeMillis() - startTime > 5000) break;
    } // endwhile

    // stop each mailbox
    for (SimBox aSimpson : simpsons) { aSimpson.stop(); }
  } // end main()
} // end class SimMain
```

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

```
From Homer to Marge: My doctor said don't walk.
From Marge to Homer: That was a traffic signal!
From Bart to Homer: There's a 4:30 in the morning now?
From Homer to Bart: D'oh!
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
From Bart to Homer: flooding the message queue...
```

Have fun!