Lab #11
CSCI 201

Title

Message Queue

Lecture Topics Emphasized

Threads

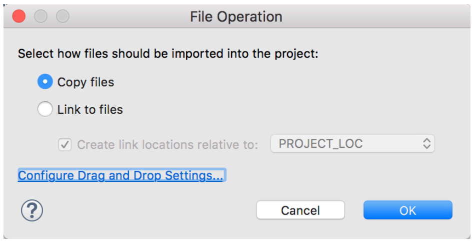
Introduction

Multi-threading is a very important task that is involved in nearly every program you run. The ability to have multiple sections of code appear to be executing simultaneously has enabled applications such as auto-save, gaming, message notification, and AJAX. You will get some experience using multiple threads to implement a message queue.

Description

You will create a message queue class that allows messages to be stored from multiple threads simultaneously. Messages queues are often used for notification-based applications. For example, one thread can put a message into the queue while another thread can "subscribe" to receive notifications. Each thread will operate independently of the other, sharing the message queue.

Start with creating a new Java project (NOT a dynamic web project). Download the two provided Java files (*MessageQueue.java* and *Util.java)* from the assignment page, and drag them into your newly created Eclipse project. If prompted, select *Copy files* instead of *Link to files*.



Select how files should be imported into the project:

● Copy files

○ Link to files

☑ Create link locations relative to: PROJECT_LOC

MessageQueue.java contains an ArrayList that allows inserting and removing. It has two methods, namely 1) *void addMessage(String s)*, which adds a new message string to the end of the queue, and 2) *String getMessage()*, which removes and returns the first message in the queue (in the event that the queue is empty, it returns an empty string).

Util.java has a static method called *getCurrentTime()*, which returns the current time in a string.

Now, create a *Messenger* class that extends from *Thread*. The goal of the *Messenger* class is to add messages to the specified *MessageQueue*. It takes a *MessageQueue* as a parameter in its constructor, and then later adds to that *MessageQueue* when it is its turn to run. Because it is a *Thread*, we will need to put our implementation in the *run()* method. In the *run()* method, write a loop that iterates 20 times and inserts a different message into the *MessageQueue* each time. The message can be whatever you want, but make sure you include a **unique identifier** with each message (such as the message number being inserted). After inserting each message, the *Messenger* should sleep for a random amount of time between 0-1 seconds by calling *Thread.sleep()*. Upon inserting a new message into the *MessageQueue*, print the inserted message, along with a timestamp, to the console. Make sure the outputs are distinct from other print statements that we will write later.

Create a *Subscriber* class that also extends from *Thread*. The goal of the *Subscriber* class is to query the messages in the specified *MessageQueue*. Again, make sure the *Subscriber* class takes a *MessageQueue* in its contractor. In the *run()* method, it should query the *MessageQueue* by calling the *getMessage()* method in the *MessageQueue*. It should continue to query in a loop until it has read 20 messages. The *Subscriber* thread should sleep for a random amount of time between 0-1 seconds after attempting to read a message by calling *Thread.sleep()*. If there is no message, do NOT increment the number of read messages. That will ensure that 20 messages will eventually be read before terminating. Output each message to the console after it has been read, along with a timestamp. If there is no message to be read, print a message to indicate that as well. Again, please make sure the outputs are unique from the other classes (for example, the *Messenger* class).

Now, it is the time for us to test the threads that we just wrote. Create a class called *MessageTest.* This is where we will write our main. In the main, create an instance for each of the classes above (*MessageQueue*, *Messenger*, and *Subscriber*). Make sure of the *ExecutorService* to help you manage the threads. Create a new *executor* by calling its constructor: *Executors.newFixedThreadPool(size)*. Add the *Messenger* and *Subscriber* to the newly created *executor* by using the *executor.execute(thread)* method. There is no need to add *MessageQueue* because it is not a thread and only serves as our data structure. Do NOT explicitly call the *start()* method on the threads as they are now managed and executed by the *executor*. Make sure you call *executor.shutdown()* after adding the two threads to let the executor know that "no more new tasks will be accepted" (from Oracle documentation). The *executor.isTerminated()* method can help you determine if the two tasks are finished. As long as the two threads are not done, use the *Thread.yield()* method to allow them to finish in a timely manner. Hint: An if statement only executes once, whereas a while loop keeps looping until the specified condition is met.

Have an outer for loop in your main to run the program twice, since multi-threading does not always generate the same output. In each iteration, you will create a new instance for each class. Make sure all threads are finished before moving to the next iteration by using the *Thread.yield()* method (see the hint above).

Please note, your program should not have any exceptions thrown.

Below is one possible output of your program, though there are many variations that would still be correct.

```
2017-07-17 7:13:23.03 Messenger - insert “message #1”
2017-07-17 7:13:23.09 Messenger - insert “message #2”
2017-07-17 7:13:23.40 Subscriber - read “message #1”
2017-07-17 7:13:24.00 Subscriber - read “message #2”
2017-07-17 7:13:24.30 Subscriber - tried to read but no message...
2017-07-17 7:13:24.51 Messenger - insert “message #3”
2017-07-17 7:13:24.58 Subscriber - read “message #3”
2017-07-17 7:13:25.03 Messenger - insert “message #4”
2017-07-17 7:13:25.10 Messenger - insert “message #5”
2017-07-17 7:13:25.15 Subscriber - read “message #4”
2017-07-17 7:13:25.29 Messenger - insert “message #6”
2017-07-17 7:13:25.50 Subscriber - read “message #5”
2017-07-17 7:13:26.11 Subscriber - read “message #6”
2017-07-17 7:13:26.30 Subscriber - tried to read but no message...
<program continues>
```

Grading Criteria

Labs are graded based on your understanding of the course material. To receive full credit, you will need to 1) complete the lab following the instructions above **AND** 2) show your understanding of the lab material by answering questions upon check-off.

If there is a discrepancy between your understanding of the material and your implementation (i.e. if your code is someone else's work), you will receive a grade of **0** for the lab. Please note, it is the professor's discretion to report the incident to SJACS.

Instructors, to ensure consistency across all lab sections, please strictly stick to the following criteria:

1) *Messenger*
   a) 0.2 - the *Messenger* class is complete and working
   b) 0.15 - the *Messenger* class is complete but has bugs
   c) 0.1 - the student is on the right track, but the implementation is incomplete (***has more than 50% done***)
   d) 0 - the student implements less than 50%
2) *Subscriber*
   a) 0.2 - the *Subscriber* class is complete and working
   b) 0.15 - the *Subscriber* class is complete but has bugs
   c) 0.1 - the student is on the right track, but the implementation is incomplete (***has more than 50% done***)
   d) 0 - the student implements less than 50%
3) *MessageTest*
   a) 0.2 - the *MessageTest* class is complete and working
   b) 0.15 - the *MessageTest* class is complete but has bugs
   c) 0.1 - the student is on the right track, but the implementation is incomplete (***has more than 50% done***)
   d) 0 - the student implements less than 50%

Check-off Questions

Please randomly choose one question from each section. Each question is worth 0.07%.

Question 1

a) Is the data structure in the *MessageQueue* class first in first out or first in last out? Explain.
b) What are some other possible data structures for the *MessageQueue* class?

Question 2

a) Is the *Messenger* class an instance of Thread? Explain.

b) Where do you implement your code in the *Messenger* class? Explain.

c) Why do we need to make the print statements unique?

Question 3

a) How does the *Subscriber* class differ from the *Messenger* class in terms of their functionalities?

b) What happens if you increase your count every single time when querying for messages?

c) What parameter does the *Subscriber* class take? Why is it necessary?

Question 4

a) How do we make sure the program finishes as soon as possible?

b) How do we execute the threads in the *MessageTest* class?

b) How do we know if all threads are finished?
c) Do we need to call the *start()* method on threads? Why or why not?