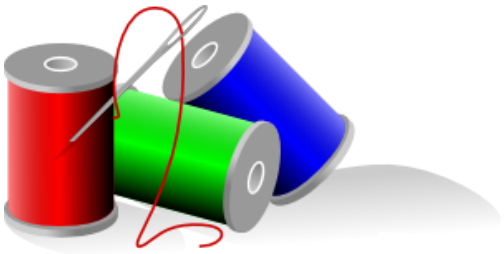
COMP 346 Operating Systems



Assignment 1

Due Date: 11:55 PM, Wednesday, July 15

Note: All submissions are made via Moodle (see "deliverables" section). Late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied automatically.

After you upload the assignment, please verify that your submission is recorded as submitted.

Description: This first assignment will allow you to become more comfortable with threads and the basic issues associated with thread programming. In particular, you will be creating a simple multi-threaded application called **Robin Hoods**. As you may recall, Robin Hood was celebrated for "robbing the rich and giving to the poor." Your application will do something similar.

In short, you will create a multi-threaded competition in which opposing Robin Hoods will attack one another and try to take each other's gold coins. Occasionally, they will donate some of their winnings to the community. To simulate the attack, each Robin Hood will be assigned a Band of Merry Men (or Women) with which he (or she) can do battle. When given the chance, Robin will challenge other competitors. If he is victorious, he will then take some share of the gold coins of the loser. Robin will enter combat by: (1) randomly selecting one of his enemies as a target, (2) selecting a small random percentage of his gold coins as a wager, and (3) selecting a random number of his merry folk to do battle. Victory always goes to the competitor who brings the most soldiers into battle.

So how does this work? As you may have guessed, each of our Robin Hoods is represented by a thread. Each thread operates independently from the others. (i.e., it will be scheduled independently by the operating system). That being said, the threads will have to share data in some fashion. While there are a number of ways that this process could be represented, you will do so by providing a "challenge" list for each competitor. As challenges are issued, they will be placed into a list assigned to a specific Robin/thread. A challenge would consist of (i) the challenger's ID (ii) the number of soldiers sent by the challenger (iii) the amount of the wager. When the target thread receives CPU control, it will be able to read from that list and respond to the challenge. It does so by first checking to see if it has enough money to match the wager. If not, this Robin/thread automatically loses and all his money goes to the challenger. At that point, he is officially "bankrupt" and can no longer compete in the competition. If he does have enough money, he will select a random number of his own men. This number is then compared to the number provided by the challenger. The winner gets the appropriate number of gold coins from the loser (in a tie, nobody wins).

Of course, Robin wouldn't be a hero if he didn't donate some of his winnings to the poor. Therefore, after every ten victories, Robin *may* (as per the command line arguments) select 10% of his current wealth and donate it to a community fund (the amount should be rounded to the nearest dollar so that

we only work with integers). He may make many such contributions as the competition is running. It must be possible to keep track of everyone's contributions since the "winner" will ultimately be the most generous Robin.

There are a few other details that need clarification. First, if the counts of coins and soldiers are purely random, then this would produce a sort of stalemate. In other words, every Robin would win about as much as he loses. Real life isn't like this so we will provide an option that allows some Robins to do a little better than others. To do this, we will vary the size of the band of Merry Men. Specifically, each Robin/thread gets one more soldier than the preceding Robin. So Thread 1 has one soldier, Thread 2 has two soldiers, …Thread *n* has *n* soldiers. What this means is that, over time, a winner would actually emerge. That being said, the randomness in the challenge process allows some upsets. For example, Thread 100 could send just 2 soldiers to do battle with Thread 4, who sends 3 soldiers. Thread 100 would lose this particular battle.

Second, each competitor is given one million gold coins at the start of the competition. During each challenge, the challenging Robin will randomly select a number between 1 and 10 to represent the current wager. Using a small percentage of the total coins ensure that the competition will not end immediately since it takes a while to lose this many coins (or donate all of them if you win).

Parameters: We also need a mechanism to provide basic parameters to the game. We will do this through some simple command line parameters. Assuming that the application is called RobinHoods, an invocation might look like this:

```
RobinHoods thread_count iteration_count skew_odds donations
```

All parameters are integers. `thread_count` should be a number between 2 and 20. `iteration_count` can be a number between 1 and 1 million. Skew_odds is either 0 or 1 and indicates whether threads with higher thread IDs can send more soldiers into battle (1 = yes, 0 = no). If skew is turned off, then Robins simply pick a random number between 1 and the thread_count to represent their soldier count. In this case, all competitors have the same chance to win. Conversely, if skew is turned on, the number of solders is defined as a random number between 1 and the ID of the current thread. So, on overage, threads with higher ID values send more soldiers to battle and are therefore more likely to win.

Donations is also a 0/1 value. If set to 1, donations are made. Otherwise, Robin keeps all of his money. Your code should provide proper checks to guarantee that parameters are in the proper ranges.

Summary: So, to recap, the idea is as follows. You will accept the arguments passed to the application at run time. A summary of these parms will be printed immediately to the console, along with the total number of coins available (i.e., `thread_count` * `coin_count`). You will then generate `thread_count` threads, and each will compete when it gets the CPU. It will do so by generating

`iteration_count` challenges, using a simple looping mechanism (i.e., it will loop `iteration_count` times, creating a new challenge object on each iteration. However, before each individual challenge is generated, the thread must check its own challenge queue. If it is not empty, it must process all challenges that have been placed there by other Robins/threads. In other words, it must adjust the two coins counts as necessary (i.e., winner and loser). Only then does it continue with the loop. Each time ten victories have been recorded, Robin *may* donate 10% of his current total to charity, as per the command line options. If Robin runs out of money, however, a notice must be printed immediately to the console and the thread should no longer participate in the competition. You may use additional Objects to manage the competition if you like.

Once the iterations are completed, and ALL challenge lists are empty, the main thread will summarize the results and print the results to the console. In short, it should order the Robins from most to least successful, in terms of the number of coins donated to charity. You should also list the coins still owned by each Robin (i.e., not donated). If no donations are made, then the results should be sorted by coins remaining. You should then provide the total of all donated coins, plus the total of all coins still possessed by the remaining Robins. If your threads cooperated properly, this total should be the same as the number of coins available at the beginning of the process. Finally, you should list the total elapsed time for the competition (Java's System.currentTimeMillis() method can be used to record the start and end times).

Below, a sample output is listed for a 6 thread competition with 1 million iterations. The actual command line is: **`RobinHoods 6 1000000 1 0`**

Here, the odds of winning are skewed towards higher thread IDs, but no donations are made. In this case, the low threads IDs go bankrupt and the high threadIDs win more of the money.

```
** The Robin Fest **
--------------------

Total competitors: 6
Total challenge iterations: 1000000
Skew the odds: Yes
Donations expected: No
Total coins available: 6 * 1000000 = 6000000


Start the contest!

Robin-0 is now bankrupt!
Robin-1 is now bankrupt!
Robin-2 is now bankrupt!


Competition Summary
-------------------

Coins donated:
Robin-5: 0 (3354611 remaining)
```

```
Robin-4: 0 (2069657 remaining)
Robin-3: 0 (575732 remaining)
Robin-0: 0 (0 remaining)
Robin-1: 0 (0 remaining)
Robin-2: 0 (0 remaining)

Total coins in circulation: 0 donated + 6000000 remaining = 6000000

Total elapsed time for competition: 5.363 seconds
```

It can also be informative to display output for a very small number of iterations. This can demonstrate that the individual competitions are being properly handled. In the following example, we run the contest for just ten iterations (with donations). Here, we can see that no competitor has gone bankrupt yet. We can also see that a couple of the most successful Robins have donate money, while the remaining competitors have either lost a little or won a little.

```
** The Robin Fest **
--------------------

Total competitors: 10
Total challenge iterations: 10
Skew the odds: No
Donations expected: Yes
Total coins available: 10 * 1000000 = 10000000


Start the contest!


Competition Summary
-------------------

Coins donated:
Robin-1: 100003 (900045 remaining)
Robin-0: 100001 (899985 remaining)
Robin-4: 0 (1000037 remaining)
Robin-9: 0 (1000019 remaining)
Robin-2: 0 (1000010 remaining)
Robin-7: 0 (1000009 remaining)
Robin-3: 0 (999979 remaining)
Robin-8: 0 (999979 remaining)
Robin-6: 0 (999974 remaining)
Robin-5: 0 (999959 remaining)

Total coins in circulation: 200004 donated + 9799996 remaining = 10000000

Total elapsed time for competition: 0.102 seconds
```

Supporting Material: Most of you will not have much, if any, experience will Java Threads. A general overview can be found at the link below. You will want to pay particular attention to the discussion of *synchronization.*

http://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html

Note that you cannot use the classes defined in java.util.concurrent, since your job in this assignment is to learn how to provide this kind of functionality yourself.

Deliverables: All code is to be written in Java. Multiple source files should be combined into a single zip file and submitted to Moodle (not the Faculty's EAS system). The zip file should be named

LastName_FirstName_A1.zip (e.g., Smith_John_A1.zip)

No other files should be included (no class files or extra files generated by your development environment). The source code will be extracted and compiled by the marker, who will then execute a series of test cases against the code to ensure correctness.

Finally, you should include a README text file to indicate what parts of the assignment work or don't work. This may make it easier for the grader to give you points if there are problems/limitations with your solution.

Good luck...