**SER421 Fall 2017 Lab1 Javascript Intro and Sockets/Threads, Review**

Assigned 8/30/17, due 9/12/17 at 11:59pm via online submission to Blackboard

**Objectives**:

1. Gain proficiency in Javascript/JSON
2. Review basic programming concepts in socket-based programming.
3. Review basic programming concepts in multithreaded / concurrent programming

**Overview**:

*THIS LAB IS AN INDIVIDUAL ASSIGNMENT, BOTH PARTS!*

**Part 1: Javascript Golf Tournament Leaderboard (45%)**

Golf is an individual competitive sport where a player plays multiple *rounds* (typically 4) in a *tournament*. In each round, a player plays 18 *holes*, where each hole has an expected score, called *par*. Score is tracked during the round with respect to par, so for example a score of -2 after 10 holes means the player has taken 2 less shots than expected (which is good). Likewise, a score of +3 would indicate taking 3 shots more than par (which is not so good). At the conclusion of the round the score is officially recorded in the aggregate sum of the holes. The player with the lowest score after all rounds in the tournament are completed wins the tournament.

For this activity you will implement a collection of Javascript functions and manipulate Javascript objects to implement a scoreboard (or what is called in golf the *leaderboard*).

1. Implement a function to parse the JSON string below into objects of type *Tournament* and *Player*. It should be obvious that a Tournament will have many Players. The JSON string is to be the argument to the Tournament constructor.
2. Implement a function named *leaderboard* on Tournament that returns a JSON string back except with the player entries sorted first by score (lowest score first), then by hole (later holes first). This function should be named *leaderboard*.
3. Implement a function on Tournament named *projectScoreByIndividual* that accepts as parameters the lastname and first initial of a Player and returns a projected score for the player when the round is completed. Project the player’s final score based on his current score and hole (individual rate of progress). For example, if the player has a score of -2 after 12 holes, then project that he has a final score of -3 at the end of the round. For the example JSON below, the projected score would be (-3/17\*(18-17) + -3) = -3.176, rounded to -3.
4. Implement a function named *projectScoreByHole* that accepts as parameters the lastname and first initial of a Player and returns a projected score for the player at the end of the round by projecting the player’s final score based on his current score and hole plus the collective rate of progress (the average score per hole for all Players in the Tournament). For example, if the player’s score is -2 on he has finished the 9th hole, and on average golfers are scoring +0.11 per hole then add -2 plus 9 \* +0.11 to get a projected score of -1. For the example above, the returned score would be -3 + (1 \* 0.11) = -2.89, which rounded is -3.
5. Implement a function named *projectedLeaderboard* that does exactly what *leaderboard* does except it takes another argument representing a function (*projectScoreByXXX*) and uses this function to determine a leaderboard based on each player’s projected finishing score. If you are really clever you can reuse the leaderboard function.
6. Modify the prototype of the *Tournament* object by adding a function *printLeaderboard* that prints the leaderboard (sorted from best score to worst then by hole (later holes first)) to the console using console.log().
7. Add a function to Player named *postScore(s)* where *s* is a score with respect to par (e.g. -1, 0, 1, 2) that represents the Player's score on the next hole. Update the Player's score and hole properties based on *s* and check if the Tournament is completed (requirements h & i). If you are clever you can use event emitters to accomplish (but this is not required).
8. If the tournament is completed (round == 4 and all Players have hole == "finished"), create a new property on Tournament named *winner* that is assigned the player that is the winner of the tournament. Add a *getWinner* function on the object that returns the winner’s last name.
9. Also, when the tournament is over, assign to every player a property named *winnings*. Set the value of winnings to 50% of the tournament award for the winner, 30% for 2nd place, and 20% for 3rd place. If there is a tie for the winner you may arbitrarily select one of the Players as the winner for h and i.

Note: *to pass object functions as parameters, even to other functions on the same object, you need to use a .bind() call*

JSON example:

{ "tournament": {

"name": "British Open",

"year": "1998",

"award": 840000,

"yardage": 6905,

"par": 71,

"round": 1,

"players": [

{"lastname": "Montgomerie", "firstinitial": "C", "score": -3, "hole": 17 },

{"lastname": "Fulke", "firstinitial": "P", "score": -5, "hole": "finished"}

]

} }

**Part 2: Multithreaded and socket-based programming (55%)**

For this lab you will construct a simple web proxy server in Java. In the real world, a web proxy server forwards HTTP requests to a target server based on various rules (filtering, rewriting, etc.). You will construct a special restricted proxy that will exercise your ability to meet the programming competencies outlined in the Objectives 2 and 3.

For this task, you will retrieve files from a destination HTTP server, cache them, and output that HTML back to your client. Your requirements:

1. Your program must be in package edu.asupoly.ser421 with main classname Lab1Part2
2. Your program must take the following command-line arguments:

<local port> - the port your program will listen on

<server> - hostname or IP address of the target HTTP server

<port> - port number the HTTP process is running on the <server> machine

<cache size> - a value in KB representing the max size of the in-memory cache

<delay> - a value in milliseconds representing an artificial delay to introduce on each request

Example: java edu.asupoly.ser421.Lab1Part2 8000 [www.asu.edu](http://www.asu.edu) 80 1024 5000 would create a proxy listening on port 8000 (your program), forwarding requests to [www.asu.edu](http://www.asu.edu) on standard HTTP port 80, with a 1MB cache and 5 second delay on each serving thread.

1. Your program should accept an incoming request to retrieve a file, and if that file content is not already available in the in-memory cache, it should connect to <server>:<port> and retrieve the file, store the file in it's cache, and return the content to the client.
   1. You should not worry about what the content of the file is that you are retrieving, but if it helps you may safely assume it is a text format like HTML, XML, or JSON.
   2. The incoming protocol to your server should be an HTTP GET request. That is "GET <filename>". No use of HTTP request headers is allowed, you are merely accepting a string and constructing the proxy request from it.
      1. Note that use of a 3rd party library for server functionality, or of Oracle's built-in HttpServer library in Java, is not allowed – you must write the server from scratch.
      2. You may however, use Java's built-in HTTP classes to act as a client to the remote HTTP server.
      3. Any request in any format other than what is specified may be rejected with an error message sent back to the client.
2. Handle errors returned from the HTTP server (400 and 500-level errors) by indicating such an error back to the client (as HTML content). You do not have to follow redirects (300-level messages); you may report an error if you get one.
3. Your program must support multiple concurrent requests through multi-threading. You may choose to use a thread pool or spawn threads on demand, but in either case, each incoming request must be served from an individual thread.
   1. The <delay> command-line argument is meant to be an integer in milliseconds to pause that thread while processing the request (Thread.sleep argument). The intent here is to be able to artificially induce a concurrent scenario for testing. You must have a Thread.sleep call as the first line in the run() method of the serving thread that uses this parameter so we are certain you are implementing thread-safety properly.
4. Your in-memory cache should be a "Least Recently Used" (LRU) implementation. That is:
   1. when inserting new file contents into the cache, if the total size of stored content in the cach exceeds <cache size>, then remove the LRU cache item (the one that has been in the cache the longest without being served to the client)
   2. when there is a cache "hit" – that is, when you can serve the HTML content from the cache instead of the remote server, update the LRU cache so it knows this item was recently used.

A significant part of the grading criteria for this problem is connected to the 1) thread-safety of the in-memory cache in the face of various operations that can be performed on it to add new items, update cache entries, or remove cache entries. I expect you will create an efficient implementation based on a java.util.Collection implementation; 2) proper implementation of a socket server and client, and being very robust in the face of errors ("robust" here means not failing catastrophically, but rather trapping errors and taking a recovery action or informing the user in a meaningful way).

**Extra Credit (10 points each)**

There are 2 features this web caching proxy does not do (well lots of things it doesn't do, but here are a couple for you):

1. Keep the cache fresh. Add a command-line parameter that is a timer interval (in seconds) for removing any "stale" cache entries. Right now the entries are only removed when the cache is full; this asks you to augment this with a timer-based expiration of cache entries that have not been accessed within the freshness parameter.
2. Support the ability to persist the in-memory cache if the server exits. Add a command-line parameter to indicate a cache location on disk, and then initialize the in-memory cache based on the on-disk cache (if it exists).