University of Washington Bothell
Computing and Software Systems
Graduate Certificate Program
Programming Competency Self-Assessment: Second Assessment (of two)

There are two program specifications in this self-assessment. Candidates to the CSS Graduate Certificate Program should find the completion of these programs to be challenging but *achievable* as assignments. Candidates who find completing the following programs to be beyond their means should consider taking CSS162.

Candidates who have not taken formal undergraduate programming classes are encouraged to include in their application package the listings of their solutions to these program specifications.

Program Specification One (of two):

Welcome to Wumpus Mountain

The goal of Wumpus Mountain is for your Wumpus Hunter to search the mountain pathways for the golden scales left behind by all baby wumpus hatchlings. In the original game of ``Hunt the Wumpus," the hero explored caverns searching for gold, avoiding bottomless pits, and using a bow and arrows to kill the wumpus. Fortunately, baby wumpi aren't dangerous. While there is no risk of death, and no shooting arrows in this version of the Wumpus World, you *will* be exploring new territory in search of gold.

The paths on Wumpus Mountain are arranged like what a computer scientist would call a ``tree". This means that paths branch repeatedly, but they never come back together. So, once you reach the end of a path, you always need to backtrack to a preceding branch to continue your search. Also, there is only one way to get to any point on the mountain, and the only way to get off the mountain is the way you went in. So, you will need to navigate the tree structure in some logical order using a stack. Note that this assignment could be done recursively, and Java's runtime would then manage the stack, but you should explicitly use a stack structure (and an iterative algorithm) for this assignment.

Code Details

To start Wumpus Mountain:

1. Download `all the classes`. (see http://depts.washington.edu/cssinfo/gcsdd-self-assessment).
2. `class MyWumpusHunter extends WumpusHunter`
3. Override the `getName()` method to return your hunter's name
4. Override the `startAt()` method, which is where all of your mountain traversal code (and stack) will go

Sample output from the program might look like:

```
Hunter's name: StackBasedV1Hunter
Hunter's report:
Entering the Mountain Top
Entering the Crooked Way
Entering the Snake Room
Entering the Hidden Nook
We've neared the scales!
Entering the Treasure Room
We've found the scales!
...The path is...
```

```
Start at the Mountain Top
and then visit the Crooked Way
and then visit the Hidden Nook
and then visit the Treasure Room
```

You can use the included `BinaryMountainFactory` and `TrinaryMountainFactory` classes to build mountains for your hunter to explore. Note that you can and should implement different versions of `BinaryMountainFactory` to test your program: producing 0—2 children per cave, but with different arrangements.

Your `startAt()` method should use a stack to keep track of caves on the path between the Mountain Top and the current cave. It can use the public accessors of the `MountainCave` class to get information about the current cave (such as its children, whether the scales are there, etc.). It should add to `actionLog` each time it enters a cave. When it finds the scales, it should trace the path from the Mountain Top to the cave with the scales, appending the description to `actionLog`.

Statement of Work

Build a `WumpusHunter` subclass that searches a tree-structured maze looking for a goal. Your class should use a stack and an iterative algorithm, rather than recursion. Your class should record its moves and report the final path from the start to the goal.

You should also develop your own `MyMountainFactory` class that creates random cave tree structures to search. These random structures should be larger than the small ones created by the provided factories. Demonstrate that your hunter works in these.

Program Specification Two (of two):

In this program, you will use a queue to implement an algorithm for finding all of the prime numbers up to a specified value n. The technique, called the Sieve of Eratosthenes, is ancient (third century BCE) and was developed by a Greek named Eratosthenes. The algorithm is given in the following pseudocode:

> Create a queue and fill it with the consecutive integers 2 through n inclusive.
> Create an empty queue to store primes.
> Do
> Obtain the next prime p by removing the first value in the queue of numbers.
> Put p into the queue of primes.
> Go through the queue of numbers, eliminating numbers divisible by p.
> while (p <= sqrt(n)).
> All remaining values in numbers queue are prime, so transfer them to primes queue.

You must use a Queue interface. When you construct a Queue object, allocate a new LinkedList, but store it in a Queue variable. The LinkedList class implements the Queue interface. Both should be imported from java.util. For example:

```
Queue<Integer> myQueue = new LinkedList<Integer>();
```

Define a class called Sieve with the following public methods:

- Sieve() – Constructs a sieve object.

- void computeTo(int n) - This is the method that should implement the sieve algorithm. All prime computations must be implemented using this algorithm. The method should compute all primes up to and including n. It should throw an IllegalArgumentException if n is less than 2.

- void reportResults() - This method should report the primes to System.out. It should throw an IllegalStateException if no legal call has been made yet to the computeTo method. It is okay for it to have an extra space at the end of each line.

- int getMax() - This is a convenience method that will let the client find out the value of n that was used the last time computeTo was called. It should throw an IllegalStateException if no legal call has been made yet to the computeTo method.

- int getCount() - This method should return the number of primes that were found on the last call to computeTo. It should throw an IllegalStateException if no legal call has been made yet to the computeTo method.

Your reportResults method should show a list of the primes, 12 per line with a space after each prime. There is no guarantee that the number of primes will be a multiple of 12. The calls to reportResults must exactly reproduce the format of the sample log below. The final line of output that appears in the log reporting the percentage of primes is generated by the main program, not by the call to reportResults.

You must guarantee that your object is never in a corrupt state. For example, your sieve object might be asked to compute up to one value of n and then asked to compute up to a different value of n without a call to reportResults ever being made. Similarly, your object might be asked to compute up to some value of n and then be asked to reportResults more than once. Each call to reportResults, getMax, and getCount should behave appropriately given the previous call to computeTo, no matter how often they are called or in what order. Finally, notice that if reportResults, getMax, or getCount are called before a legal call to computeTo, they throw an exception to indicate that the operation is not legal given the object's state.

A sample driver HW8.java is provided. You can use this program to test your class, but keep in mind that it does not test the internal consistency of your object.

Sample log of execution (user responses are underlined):

```
Maximum n to compute (0 to quit)? 20

Primes up to 20 are as follows:
2 3 5 7 11 13 17 19
% of primes = 40

Maximum n to compute (0 to quit)? 100

Primes up to 100 are as follows:
2 3 5 7 11 13 17 19 23 29 31 37
41 43 47 53 59 61 67 71 73 79 83 89
97
% of primes = 25

Maximum n to compute (0 to quit)? 500

Primes up to 500 are as follows:
2 3 5 7 11 13 17 19 23 29 31 37
41 43 47 53 59 61 67 71 73 79 83 89
97 101 103 107 109 113 127 131 137 139 149 151
157 163 167 173 179 181 191 193 197 199 211 223
```

```
227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359
367 373 379 383 389 397 401 409 419 421 431 433
439 443 449 457 461 463 467 479 487 491 499
% of primes = 19

Maximum n to compute (0 to quit)? 0
```

Resources:

- Queue interface: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Queue.html
- LinkedList class: http://java.sun.com/j2se/1.5.0/docs/api/java/util/LinkedList.html
  (Use this only for creating an object that implements the Queue interface. Use the Queue interface for interacting with the object.)