CS3114 (Fall 2013)
PROGRAMMING ASSIGNMENT #1

Due Wednesday, September 18 @ 11:00 PM for 100 points

Due Tuesday, September 17 @ 11:00 PM for 10 point bonus

Updated: 9/11/2013

Assignment:

This is the first phase of a project to create an earthquake notification system. In Project 2, you will add a spatial data structure to speed up some of the key operations.

Your project will receive input from two data streams. The first stream comes from an earthquake tracking service provider (USGS), which provides information on the time, location, and magnitude of earthquakes around the world. The second data stream will be a series of requests related to users of the notification system. These requests will be things like: add a watcher to the database; remove a watcher; and query for recent, large earthquakes. The project can be viewed as a discrete event simulation, where events (either earthquakes or user requests) take place at discrete times, and are processed in the order that they occur.

Java Code:

For this project, you may only use standard Java classes, Java code that you have written yourself, and Java code supplied by the CS3114 instructor (see the class website for the distribution). You may not use other third-party Java code. You may **not** use any built-in Java list classes for this assignment.

Data Structures:

For this project you will implement three data structures. The first is a simple linked list of watchers who have signed up for earthquake notifications. You will store watchers on the list in the order that they arrive, and all processing to delete watchers or generate notifications will be done by sequentially working through the list. (Note that in the second project, you will replace this simple list with a more sophisticated spatial data structure.)

The second data structure is a linked queue to store the list of recent earthquake records, in order of arrival. You will only be maintaining earthquake records for the most recent 6 hours time period. As simulation time moves on, you will delete records with old timestamps from the queue. Earthquake records are stored in order of arrival time.

The third data structure also stores these same earthquake records (again, removing earthquakes records that are more than 6 hours old). This data structure is a max-heap, ordered by earthquake magnitude. One of the queries in the user data stream is for the biggest earthquake seen in the past 6 hours. This query is answered by looking in the heap. As earthquake records arrive or expire, they are added to or removed from both the queue and the heap. Since the queue is ordered by time, it is easy to find which records are old. But you will also need to have an efficient way to find those records in the heap. It is not sufficient to do a sequential search through the heap's array to find the records. Since an array for the heap needs to be allocated at the start of the program, you will use a size of 1000. We will never test with more than 1000 unique earthquakes.

Event Processing:

Your two data steams (the earthquake records and the user requests) each have timestamped records. As the records come in, you will process the various events. When you process a new

watcher add request, the watcher is added to the watch list (and you print a suitable message to standard output). When you process a new watcher delete request, the watcher is removed from the watch list (and you print a suitable message to standard output). When you process a new "largest recent earthquake" query, you will output a copy of the information for the largest earthquake currently in the heap (it will be at the root since you are storing the records in a max-heap by magnitude).

The earthquake data API might not work as you expect. It is not event driven. Instead, you must poll (every 5 minutes in simulation time), at which point you will receive a single record that contains all earthquakes from the past hour. So you will have to look at this list of earthquakes, and deduce which (if any) are actually new earthquakes. Any new earthquakes are then added to the rear of the earthquake queue. Also, you must add the new earthquake record to the heap, with its position determined by the magnitude of the earthquake. Finally, you need to check the timestamp for the oldest earthquake(s), and remove any that are more than six hours old. To do this, you will look at the front of the queue. For any earthquakes that must be removed, you will need to find their position in the heap. You may not do this using sequential search of the heap. Instead, you must store and maintain in the earthquake record the heap position. All heap update methods must update this heap position appropriately when records are move around within the heap's array.

In addition to updating the queue and the heap, you will also check against the watcher list when you see a new earthquake. You will output notification messages for each watcher that is "close enough" to the earthquake. Each watcher record contains an (X, Y) position, as does the earthquake record. A watcher should receive a notification message if it is within a certain distance of the earthquake, as defined by the following formula: Distance $< 2*$ Magnitude^3. A notification is a suitable message sent to standard output.

Program Invocation:

The program will be invoked from the command line as:

`java EqSimple [--all] <watcher-file> <earthquake-stream>`

where `--all` is an optional parameter. If the `--all` parameter is given, then a suitable message is output for every earthquake processed. Otherwise, when the `--all` option is not given, you will only be outputting notifications messages and messages related to the user request stream. (That is, you would only output messages about earthquakes that are close enough to a watcher to trigger a notification.)

The `<earthquake-stream>` parameter should either be the word `live` or a filename. If it is `live`, then your program should use the real-time data stream. Otherwise, it should use the given local file to simulate the real-time data stream.

Data Streams:

In order to access the earthquake data, you will be using a real-time Java library. This library (optionally) connects to a free, online Earthquake Service that provides structured data about earthquakes around the world (including their position and magnitude). The library and documentation on how to use it is provided at `http://mickey.cs.vt.edu/cs3114-earthquake/` This page also provides an API for processing the watcher data file, and discusses the polling process.

In an ideal world, your program would receive events from the Earthquake service when a new earthquake takes place. However, in practice, this is not how most web services are built. Instead, your program (the client) must **poll** the service (the server). This means that once every five

minutes, you will query the server for new information. This architecture is common, and used by most of the web applications that you see on a daily basis (including Facebook and most online email services).

If you make queries—also called "requests"—too fast, the web service might decide you are spamming them and will ban you from their service. Additionally, network connections can be unstable, or you might have limited access to the web. Since this is inconvenient for development, the library provides the ability to connect to a local data file instead of the online real-time service. That way, when your program is finished, you can switch to live data, but use the local file for development. The course website will provide access to various test files that you can use, both for the earthquake stream and the watcher stream. You will want to make sure you test your program with different data sets. When developing, you can safely "poll" the data files as fast as you can process records, rather than putting in a delay on the polling rate that you will need for the real-time service.

Output Format

Since your programs will be automatically graded for correctness (with feedback returned to you by Web-CAT), it is necessary that you follow this output format.

1. When a watcher is added to the watch list, you should print out:

   `<name> is added to the watchers list`

   For example:

   Jace is added to the watchers list

2. When you process a new watcher delete request, you should print out:

   `<name> is removed from the watchers list`

   For example:

   Jace is removed from the watchers list

3. When you process a "largest recent earthquake" query, you should print out:

   ```
   Largest earthquake in the past 6 hours:
   Magnitude <Earthquake.magnitude> at <Earthquake.getLocationDescription()>
   ```

   For example:

   Largest earthquake in past 6 hours:
   Magnitude 3.999 at 60km E of Cape Yakataga, Alaska

   Note: if the MaxHeap is empty, you should print out:

   `No record on MaxHeap`

4. If the `--all` option has been given, then you should print the following message each time that an earthquake is added to the queue/heap:

   `Earthquake <Earthquake.getLocationDescription()> is inserted into the Heap`

   For example:

   Earthquake 60km E of Cape Yakataga, Alaska is inserted into the Heap

5. When a new earthquake comes in and is added to the MaxHeap, you should print a single line for each watcher that is within the appropriate distance. Each such line should appear as:

   `Earthquake <Earthquake.getLocationDescription()> is close to <name>`

   For example:

   Earthquake 60km E of Cape Yakataga, Alaska is close to Jace

   Note that these notification messages should come after the message printed for the `--all` command.

Programming Standards:

You must conform to good programming/documentation standards. Note that Web-CAT will provide feedback on its evaluation of your coding style. While Web-CAT will not be used to define your coding style grade, the grader will take note of Web-CAT's style grade when evaluating your style. Some specific advice on a good standard to use:

- You should include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment should describe what your program does; don't just plagiarize language from this spec.
- You should include a comment explaining the purpose of every variable or named constant you use in your program.
- You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. Use a consistent convention for how identifier names appear, such as "camel casing".
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or the codebase provided by the instructor. Note that the textbook code is not designed

for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point.

Deliverables:

You will submit your project through the automated Web-CAT server. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases with your program, and part of your grade will be determined by how well your test cases test your program, as defined by Web-CAT's evaluation of code coverage. Of course, your program must pass your own test cases. Part of your grade will also be determined by test cases that are provided by the graders. Web-CAT will report to you which test files have passed correctly, and which have not. Note that you will **not** be given a copy of grader's test files, only a brief description of what each accomplished in order to guide your own testing process in case you did not pass one of our tests.

When structuring the source files of your project (be it in Eclipse as a "Managed Java Project," or in another environment), use a flat directory structure; that is, your source files will all be contained in the project root. Any subdirectories in the project will be ignored. If you used a makefile to compile your code, or otherwise did something that won't automatically compile in Eclipse, be sure to include any necessary files or instructions so that the TAs can compile it.

If submitting through Eclipse, the format of the submitted archive will be managed for you. If you choose not to develop in Eclipse, you will submit either a ZIP-compressed archive (compatible with Windows ZIP tools or the Unix `zip` command) or else a tar'ed and gzip'ed archive. Either way, your archive should contain all of the source code for the project, along with any files or instructions necessary to compile the code. If you need to explain any pertinent information to aid the TA in the grading of your project, you may include an optional "readme" file in your submitted archive.

You are permitted (and ecouraged) to work with a partner on this project. When you work with a partner, then **only one member of the pair** will make a submission. Be sure both names are included in the documentation. Whatever is the final submission from either of the pair members is what we will grade unless you arrange otherwise with the GTA.

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement near the beginning of the file containing the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor:
//
// - I have not used source code obtained from another student,
//   or any other unauthorized source, either modified or
//   unmodified.
//
// - All source code and documentation used in my program is
//   either my original work, or was derived by me from the
//   source code published in the textbook for this course.
```

```
//
// - I have not discussed coding details about this project with
//   anyone other than my partner (in the case of a joint
//   submission), instructor, ACM/UPE tutors or the TAs assigned
//   to this course. I understand that I may discuss the concepts
//   of this program with other students, and that another student
//   may help me debug my program so long as neither of us writes
//   anything during the discussion or modifies any computer file
//   during the discussion. I have violated neither the spirit nor
//   letter of this restriction.
```

Programs that do not contain this pledge will not be graded.