CMSC 132 Project #5 Fall 2012

Date due: Tuesday, December 11, 10:00:00 p.m.

1 Introduction and purpose

In this project you will write a utility that will use threads to extract some information from a webpage online (the information will actually be sent by a CGI script on that webpage, if you know what a CGI script is). The purpose of the project is to get some experience using threads and synchronization, and just a little experience with networking concepts.

Note that your program will be reading and processing the text of a webpage, which is written in HTML. You don't have to know anything about HTML other than what's explained below, although of course if you do know a little about HTML it may help slightly.

2 CVS repositories

At some point after the semester ends your CVS repository, and everything in it, will disappear. The repositories are on machines (the Grace machines) run by the campus Office of Information Technology, and the class access expires after some time. If you want to keep accurate copies of your coursework you should follow the steps below. Students are often asked for copies of coursework, for example in the future for internship or job interviews. If you may ever need copies of your coursework for these purposes you have to save them yourself; the instructional staff will not be able to provide copies in the future. Even if you don't want to keep copies of your coursework, if you ever try to run Eclipse after your repository has disappeared you will get dozens of popup error messages when you start it, as it tries to connect to the repository that doesn't exist any more, so everyone should follow these steps after finals are over to remove the old repository connection. This will retain copies of your coursework on your own machine (if you want), but Eclipse will not associate them with the CVS repository any more:

- Run Eclipse and go to the Java perspective.
- In the package explorer, select (highlight) all the projects, homewords, and practice programming exercises that are in your CVS repository (the ones that have "[grace.umd.edu]" next to their names). You can use control–left–click to select multiple items in the package explorer.
- If you **do** want to save any of your coursework from this semester just skip this step and go to the next step. If you **don't** want to save any of your coursework from this semester, just right–click on any of the selected items and choose **Delete**. In this case choose **Delete project contents on disk (cannot be undone)** in the popup window, then go to the last step below.
- Right–click on any of the selected items and choose **Team** → **Update**. (If you get any popup errors, try **Team** → **Commit** and then try updating again). Before going on, you might want to check the "Error Log" to see if there are errors listed for any of the projects, and if so, look at the files in those projects to make sure that they look like your correct final versions before continuing.
- Right–click again on any of the selected items and choose **Team** → **Disconnect...**. In the popup window select **Also delete the CVS meta information from the file system**. This disassociates each Java project from the CVS repository, so they're still stored in your local Eclipse installation, but Eclipse no longer tries to keep them in sync repository.
- Lastly you have to remove the old CVS repository connection entirely. Go to the CVS perspective and right-click on your old repository location and choose **Discard location**, and answer **Yes** when asked. After all the above your repository connection should be gone from the CVS perspective, and if you go back to the Java perspective you should not see any items in the package explorer with "[grace.umd.edu]" next to their names. You should be able to restart Eclipse and it should run without any errors, and should not try to connect to your CVS repository. Optionally (unless you deleted it above) you may want to export some or all of your coursework to other directories or folders so you can refer to it in the future without having to run Eclipse– to do this choose **File** → **Export** → **General** → **File System**

→ **Next** and in the left pane select any of the coursework that you want to export, then click on **Browse** to choose a directory to export the coursework to, choose **OK**, and then click on **Finish**.

Note that at some time in the future the ELMS space for this course will also disappear, so if there's anything that you want to save that you haven't copied already, be sure to save a copy after finals are over.

3 Baby names

There are many sites online that allow people to get ideas for baby names, if they perhaps are, or will be having, a baby, and the baby, as babies tend to do, will require a name at some point. The Social Security Administration has detailed records on names that have been used over time; in particular, since everyone born in the U.S. since 1937 has had to get a Social Security number, they have very detailed records since then on what names have been given to children, how many babies have been given what names every year, and how the popularity of names has changed over time. Besides people looking for ideas for names, these records are useful for various research purposes and sociological investigation. The earliest year that data is available for is 1880, although the data prior to 1937 is not as accurate, since many people born before 1937 never got Social Security numbers. Information is available up through 2011.

If you want to see information about baby names visit their page at `www.ssa.gov/OACT/babynames`. You can enter any year and specify how many names you want to see, or give a name and ask how its popularity has changed over time, and your browser will display the results. In this project your program will send requests for this information to that website, read the results, store them in one or more data structures, and allow information about the names that were downloaded to be looked up. In particular, names for multiple years may be requested, with a different thread in your program processing the data for each year.

The information, incidentally, is very interesting. Who would have guessed that the name Dillard was in the top 1000 most popular boy's names 61 times up through 1947, or that 19 girls were named Sophronia as recently as 1909? Who could have imagined that in 1946 there were 28436 boys born that were named Larry– as well as 83 girls? (It was the 11th most popular boy's name, as well as the 919th most popular girl's name, that year.) Would you have surmised that the same name would have the same popularity for both boys and girls seven times since 1880: with Shirley being the 408th most popular name for both girls and boys in 1894, Laverne the 468th most popular name for both sexes in 1907, Gayle being the 738th most popular name for both genders in 1925, Jessie being 167th in 1942, Jody 162nd in 1971, Kenyatta 799th in 1979, and Jessie– again– being the 227th most popular name for both girls and boys in 1988? Who might have imagined that the name Odin, which had not been among the top 1000 most popular boys names since 1884, would have made a recent resurgence, with 201 people naming their sons Odin in 2008, 202 in 2009, 247 in 2010, and 309 in 2011? And just who with any sense would name their child Odin anyway? (If you are one of the people related to any of the parents of the 959 young Odins, please do not let them read this project assignment.)

4 Methods to be written

In this section the five methods that have to be written for the project are described, while the essential details of how things have to be implemented are discussed afterward. These five methods will be in a class called `Names`, although you probably will have to use at least one other class as well. A `Names` object will contain lists of names downloaded from the Social Security Administration website, and will have methods that allow finding out information about the names that were previously downloaded and are now being stored.

4.1 `void getNames(int startYear, int endYear, int numNames)`

This method should retrieve from the Social Security Administration's web server the list of the top `numNames` names– for both boys and girls– for every year between `startYear` and `endYear` inclusive– and store them in the current `Names` object. For example, if `startYear` is 2003, `endYear` is 2006, and `numNames` is 10, the method should retrieve the list of the top ten girls' names for 2003, 2004, 2005, and 2006, as well as the top ten boys' names for those years, and store them in the current object. **Each year's names must be retrieved using a separate thread.** So for this example the method would create four threads, one reading the top

ten names (both girls' and boys') for 2003, one for 2004, one for 2005, and one for 2006. (The threads are created in this way due to the manner in which the data is downloaded, described in the next section.)

If there are no years between `startYear` and `endYear` then no threads will be created, and consequently no names will be stored. If a valid query is sent to the Social Security Administration's website in which either `startYear` or `endYear` are less than 1880 or greater than 2011, a webpage will be sent back to your thread but instead of a list of baby names it will contain an error message somewhere, which your program doesn't have to handle (it doesn't matter what results are produced in this case). If a query requesting zero names or a negative number of names is sent a webpage will be returned that has no names, so this is not an error, but no names will be stored, and the methods below will not return any useful data subsequently. The Social Security Administration's web server will handle requests for up to and including 1000 names (although see Section 7 below). (Values greater than 1000 can be used, but they have the same effect as 1000.)

Just to be clear: the method **must create one thread for every year between the value of `startYear` and `endYear`**. Each thread will open a connection to the Social Security Administration's website, send a query, and receive a webpage in reply that contains data for the top `numNames` names. Each thread will have to extract the names from the webpage as described below, and the threads must all save– in some common data structure or structures in the `Names` object– the extracted names and data about them. Note this has to be one (or more) **shared** data structures, since multiple threads will be adding data to it (or them) concurrently.

Note that the threads that are created must synchronize their access to the shared data structure they are storing results in, to prevent two or more threads from concurrently modifying it. We showed a simple example in lecture where three threads concurrently modified a simple shared integer variable; with no synchronization between them the results were unpredictable and the variable could have different values afterward. In the same fashion, if two or more threads in your project concurrently tried to modify the shared construct they use to store names, its contents could also be changed unpredictably.

The methods below may be called on the current `Names` object after the name data has been retrieved and stored, but it must be that all of the data has been stored, and you know that the threads have finished, **before** any other methods can be called, otherwise some method might get inaccurate or incomplete data.

4.2 `String getGirlName(int year, int rank)`

This method should return the girl's name that had rank `rank` in year `year` that is stored in the current object (rank means popularity, so the most popular name in a given year had rank 1, the second most popular name had rank 2, etc.). The data would have come from the Social Security Administration's web server, and been downloaded and stored into the current object by an earlier call to `getNames()`. For example, `getGirlName(2005, 3)` should return the third most popular girl's name in 2005 in the current `Names` object. If there are no names for the year `year` in the current object (`year` was not within the range `startYear` to `endYear` when `getNames()` was called on it earlier), or if there is no name with the rank `rank` (the value of `numNames` when `getNames()` was called on it earlier did not include `rank`) the method should return `null`.

4.3 `String getBoyName(int year, int rank)`

This method should have the same effect as `getGirlName()`, except it returns the indicated boy's name.

4.4 `int getGirlRank(int year, String name)`

This method should return the rank of the girl's name `name` in the year `year` that is stored in the current `Names` object. For example, `getGirlRank(1999, "Susan")` would return the position where "Susan" appeared on the list of most popular girl's names in 1999 (it was number 401). If there are no names for the year `year` in the current object (`year` was not within the range `startYear` to `endYear` when `getNames()` was called on it earlier), or if `name` is not one of the girls names stored for the year `year` in the current object (it was not within the `numNames` most popular names when `getNames()` was called on it earlier) -1 should be returned.

4.5 `int getBoyRank(int year, String name)`

This method should have the same effect as `getGirlRank()`, except it returns the rank of the indicated boy's name rather than girl's name.

4.6 Special cases

If `getNames()` is called more than once on a `Names` object, only the names retrieved by the most recent call should be stored; in other words each call should **replace** or overwrite any earlier call's data.

If any I/O error occurs while your program is running (such as if the internet connection is lost, or the Social Security Administration's server goes down) it doesn't matter what results your program produces.

If any object parameter is `null` the method should throw a `NullPointerException`. Note– if you don't add any explicit code to handle this situation, the first time a method tries to use a parameter that is `null` it will result in a `NullPointerException` anyway, so you really do not have to do anything special to cause this behavior to happen as described.

If the other four methods are called before `getNames()` is called on a `Names` object it will not be storing any names, so either `null` will be returned (by `getGirlName()` or `getBoyName()`), or -1 will be returned (by `getGirlRank()` or `getBoyRank()`).

5 Sending a request

- The URL that each thread will have to use to send a request to is:

  http://www.ssa.gov/cgi-bin/popularnames.cgi

  Java's `URL` class may be used to represent a URL, with the name of the URL being given to its constructor.

- The `openConnection()` method of the `URL` class will actually open a connection to the URL that a `URL` object was constructed with; it returns a reference of type `URLConnection`, which can be used for communications with the open connection. Your program (each thread) will have to send a request to the `URLConnection` object (actually a `URLConnection` reference to an object of the actual type `HttpURLConnection`; `URLConnection` is an abstract superclass) telling it what data it wants, and then read or receive a reply from the connection. In other words, your program (each thread) will perform both output to and input from the connection that the `URLConnection` refers to.

- In order to perform output on a `URLConnection` object the method `setDoOutput()` must first be called on it, passing `true`, as in for example `connection.setDoOutput(true)` (assuming `connection` is a reference of type `URLConnection`). As in the simple client/server example either soon to be shown in class or recently shown (depending upon when you are reading this), your program must then open an output stream to the connection; the `getOutputStream()` method of the `URLConnection` class will return an output stream, which may be fed to the constructor of one or more other output wrapper classes as needed (see the two networking examples either already on ELMS, or soon to be there, for illustration).

- Recall again that all of the above will have to be done by each thread, where one thread does this for one year. After the above is done the thread is ready to actually send a request to the CGI script at the Social Security Administration's web server, which is waiting at the other end of the open connection. The request will be a string of the form `year=2011&top=10`. All of the characters must appear **exactly** as shown, except 2011 stands for the year, which will be different for every thread (each thread will substitute its own year), and 10 stands for how many of the top girls and boys names should be downloaded (the value of the `getNames()` method's parameter `numNames` should be substituted in place of 10 here).

- Before explaining the input that each thread will receive from the Social Security Administration's web server, and describing how to extract the necessary data from it, a word about exceptions: various I/O operations described in this section that you will have to use can thrown exceptions. For example, the `URL` constructor can throw a `MalformedURLException` that has to be caught. Although any such exceptions that are checked exceptions must be caught and handled in some way, your code can do anything you like for them. We will assume that your program will always be given valid data, and errors will not occur. For instance, if you just spell the URL correctly in the `URL` constructor, although you must handle the `MalformedURLException` in some way, it will never actually be generated.

Note that you'll have to be online for your project code to work, since the `URLConnection` class allows your program to actually go to the webpage indicated by a URL and open it as an input stream.

6 Handling the input

Your program (each thread) will have to open an input connection using the same `URLConnection` object (there is a `setDoInput()` method in the `URLConnection` class that may optionally be called, passing `true`, but it is not necessary because it is the default). After sending its request each thread must then read the HTML output sent to it by the Social Security Administration's web server and extract the data from it.

Here is a portion of the actual data that would be received for the request `year=2003&top=4` (the top four names for 2003):

```
<caption><h2>Popularity in 2003</h2></caption>
<tr align="center" valign="bottom">
  <th scope="col" width="12%" bgcolor="#efefef">Rank</th>
  <th scope="col" width="41%" bgcolor="#99ccff">Male name</th>
<th scope="col" bgcolor="pink" width="41%">Female name</th></tr>
<tr align="right">
 <td>1</td> <td>Jacob</td> <td>Emily</td>
</tr>
<tr align="right">
 <td>2</td> <td>Michael</td> <td>Emma</td>
</tr>
<tr align="right">
 <td>3</td> <td>Joshua</td> <td>Madison</td>
</tr>
<tr align="right">
 <td>4</td> <td>Matthew</td> <td>Hannah</td>
</tr>
```

There are actually about 50 lines before this (and the first part of the first line of the data was also chopped off above, just for readability, but all lines before a line that contains a string of the form "Popularity in 2003" may be ignored. There are also about 15 lines following the data above that can be ignored as well. (Note that parents are apparently more creative with girls' names than with boys' names, since almost the same number of girls and boys must be born each year, but thousands more boys have the most popular names. Parents apparently go to more effort to choose more unique or less common names for girls.)

What each thread will have to do is to read all of the data received by the input connection created from the same `URLConnection` object, which can just be read as `String`s, and remove everything before a line containing the string "Popularity in 2003" (where in place of 2003 the year that that thread sent in its request will appear). Then it will have to extract the actual data items, each of which is contained inside a `<td>` HTML tag. An HTML tag of this type indicates a table data item, and begins with the string `<td>` and ends with the string `</td>`. Between those two is the actual data. In the above example, data for four names was downloaded, so there are four sets of three `<td>` tags, one for each pair of names. The three `<td>` tags for each pair of names contain the rank for that pair of names, the boy's name with that rank, and the girl's name with that rank. For instance, here is a table showing the data a thread would have to extract from the HTML above:

rank	boy's name	girl's name
1	Jacob	Emily
2	Michael	Emma
3	Joshua	Madison
4	Matthew	Hannah

You may just use the methods of the `String` class to extract the data items. First search in the downloaded text for the position of the substring "Popularity in 2003" (with the actual year the method was called with substituted for 2003), then search starting right after that for the position of the substring `<td>` and the position of the `</td>` immediately following it, and extract the text between them. Start right after the position of the `</td>` and repeat the same thing three times for each pair of names. After the data for **each pair of names** is extracted, the thread must store it **immediately** in a shared data structure or structures.

Note: use the `String` class methods to extract the data from the downloaded HTML; as discussed in Section 9, **do not** use regular expressions (if you don't know what they are, don't worry about them at all).

7 Being nice

The Social Security Administration makes this data available for people to use, but please do not abuse their machines by running your program over and over, making large requests each time. We want to avoid having dozens of students running their programs on thousands of names for dozens of years at the same time, all around the time the project is due, which could negatively impact the Social Security Administration's server. Although up to 1000 names per year can be downloaded, and years from 1880 through 2011 can be used, test your program thoroughly on **small** input data sets– no more than a handful of names and a handful of years, before even thinking about using larger data. If you want, for testing purposes, to **occasionally** run your program on a larger number of names or a larger range of years, **do not** do so over and over– run it on larger data sets **no more than once or twice a day**. Use your web browser, with the Social Security Administration's web interface, to figure out what the results of your own student tests should be.

8 Other notes

The Social Security Administration occasionally changes its baby name data. You would think that the birth name of a baby is pretty much determined when its birth certificate is written, and it doesn't change later, but presumably the data changes when either the Social Security Administration may fix errors that are brought to its attention, or when Social Security numbers are requested for children who did not get one when they were younger. For example, if someone applies for a Social Security Number tomorrow for a little girl named Matilda who was born in 2011 (one of 351 girls named Matilda last year), then eventually the count of Matildas born in 2011 would be incremented by 1 in their data.

Although it seems unlikely that their data would change during the time that this project is being assigned, it's always possible. In that case we may have to update the public tests and provide you with a new copy that you would have to import into Eclipse, replacing the current tests, since the results could change.

By the way, in terms of writing your own tests: it seems that the data from recent years seems to change more often, and for some reason the data for the second 500 names (out of the top 1000) seem to change more often than the first or top 500 names. You will minimize the possibility that any changes to the data could affect your own tests by just writing tests that don't go beyond the top 500 names in any year. As above, only run tests that download large numbers of names **rarely**.

9 Project requirements and related issues

1. Your program **must** create a thread for each year in the range of years indicated by the parameters of the `getNames()` method, in such a way that the process of downloading the names for each year takes place **concurrently**. This is a vital and necessary part of your implementation, and failure to create threads as described will lead to receiving a severe grading penalty that could cause you to lose up to **all points for the entire project assignment**. If you have any questions about whether you're using threads correctly, or aren't sure, ask in office hours.

2. Similarly, to avoid a severe grading penalty each thread **must store each pair of names as they are extracted into a common data structure**. Your threads are **not allowed** to extract all of the names (into a data structure local to that thread) and then store them all into a common data structure. It must be done after **each pair of names** is extracted.

3. To avoid a severe grading penalty you **may not use arrays or `ArrayLists` to store the downloaded names**. To store the downloaded names you may use any data structures you have written (as long as they don't use arrays), or any Java library classes, with the exception that you **may not use any Java collection classes that are internally synchronized**. The purpose of the project is for **you** to figure

out what sections of your code have to be protected or locked, and how to accomplish this, rather than using code that does this because someone else figured it out instead. For instance, there are collection classes in the package `java.util.concurrent` that handle being used my multiple threads concurrently, such as for example a `ConcurrentHashMap` or a `ConcurrentLinkedQueue`– you **may not use these** (do not use **anything at all** in the package `java.util.concurrent`). **You may use any Java collection classes that contain the warning "Note that this implementation is not synchronized" in their description in the Java API**. For example, the classes `LinkedList`, `HashSet`, and `TreeSet` all contain this warning, so you **may** use them (as well as any others with this warning). The class `Vector` does not contain this warning, so you **may not** use it. The class `Stack` extends `Vector`, and does not contain this warning either, so you **may not** use it either. Using disallowed Java library classes (those that already contain synchronization) will lead to receiving a severe grading penalty that could cause you to lose up to **all points for the entire project assignment**.

4. Your program should use the **minimum synchronization necessary** to achieve correct results, but **no more synchronization than that**. At the extreme, if you only allow one thread at a time to do anything your program will work fine– but you will not have used concurrency. This also would result in a severe grading penalty.

5. Even if you know what they are, your threads **may not** use regular expressions, or any type of parser, to extract the data from the downloaded HTML. (Regular expressions will be covered in detail in a later computer science course, but you are not required to know them now.) Although these alternatives would actually be better, you should just use the `String` class methods to extract the necessary values (meaning the `String` class methods that don't use regular expressions). We want to avoid having a number of students coming to office hours needing help getting language features to work that we haven't even had time to cover this semester, which would be a waste of everyone's time.

6. Other than the above you can use any Java library classes you like, or any data structures you wrote for earlier projects. You may use any data structure that you like to store the **threads** that you create (the restrictions above only apply to the data structures storing the downloaded and extracted **names**).

7. You may add any methods to the `Names` class, or any other classes to the `names` package, but you may not add any packages to the project. You may add any classes to the `tests` package, and may add any methods you like to the `StudentTests` class. (Of course keep in mind the stipulation in the project grading policy that fields and methods should be given an appropriate access method.) You must of course create some sort of threads.

8. Grades for this project will be based on:

public tests	50 points
secret tests	30 points
student tests	10 points
source code and programming style	10 points

9. As in the earlier programming coursework your student tests must be in a Java class file named `StudentTests.java`, which must be in the `tests` package. The restrictions above do not apply to your student tests; you may use any Java classes and language features you like in writing them. Do not rely on the public tests to test your methods exhaustively, and don't wait until you've finished coding to write your student tests.

   If you have a bug in your code that you can't figure out, and need to come to office hours, you **must** have written at least one student test that illustrates the problem.

10. After you submit you must **log into the submit server** and verify that your submission compiled and worked correctly there.

10 Academic integrity statement

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.