Banner

Special CPSC 2735 Stuff

Due no later than Thursday, Mar 26th

What must you do?

You need to begin using professional developer tools (IntelliJ IDEA and Git). Let's begin using the major data structures (lists, arrays, stacks, queues, deques, sets, and maps). We'll also talk about using the official Oracle Javadocs. Let's do some problem-solving using data like what's stored in Banner.

1 Git and GitHub

Git is console-based source code management software. GitHub is cloud-based storage that interfaces with git.

(a) If using your personal computer, download git from `https://git-scm.com/downloads`.

(b) Check you have git by typing `git --version` in a Terminal (mac) or PowerShell (windows). You should see 'git version versionNumber' displayed.

(c) Decide on a professional developer's username and then create a GitHub account at `https://github.com`. Do not link this account to your xula.edu account, instead use some personal Gmail, Yahoo, or other user account.

(d) Email me (aedwards@xula.edu) your GitHub account name.

2 Generating Simulated Data

XULA has 3,325 students and all the data about those students is stored in Banner but we can't access that data so we'll generate simulated data. Every row of this data has 3 values: an id [90000..99999], a first name (you choose any 10 names), and a gpa [0.0..4.0].

(a) Create a new project in IntelliJ named "Banner".

(b) Write a function that generates a file that has 3325 rows of student data that adheres to the given business rules.

(c) When your code works, share it with me. How? By creating a GitHub repository. Here's how to create that repo from within IntelliJ: (a) select VCS, Import into Version Control, Share Project on GitHub, and then (b) use a browser to see you've created a Banner repository in your GitHub account.

(d) Add me as a collaborator to your Banner GitHub repo by adding "aedwardsxula" under "Settings" then "Manage Access" in your Banner repo. Once a collaborator, I can push code to your repo and I'll be notified when you make changes to your repo. This is helpful because I'll continually do code reviews. A **code review** is done to make sure code works as expected. You'll push your code to GitHub, I'll pull it to my laptop, review it, add comments to your code with changes you need to make, push this newly commented code to your repo, you'll pull your code, make the changes, push, repeat.

3 Primitive Arrays and ArrayLists

(a) Let's use only the second column of the input file and ignore the other two columns. Store the names in a primitive array in the same order as they are in the input file. Using no data structure other than primitive array(s), how many students have each of the given names?

(b) Every time you complete a task/feature of your project, you should push that code so others can review it while you continue working on other features. You can push your code from within IntelliJ by clicking VCS and then Commit. Push your code to your Banner repo with the commit message "Frequency of student names in primitive array".

(c) Dump the names from the primitive array into an ArrayList. Sort the ArrayList. (javadocs Collections.sort()). Take advantage of the fact that this ArrayList is sorted to count how many students have each of the given names? Push your code to your Banner repo.

(d) Student IDs should be unique. Which rows of the input file have duplicate ids? Push your code.

(e) Every row of the input file is the data for a single student. Create a **Student** class to hold a single student's data. Push your code.

(f) Store all rows of data from the input file in a user-defined **Students** object.

(g) Push to your repo every time you create a new feature.

(h) Student IDs should be unique. Remove duplicate ids from the Students object. If an id is a duplicate, the first student with a particular id is a valid student. For example, if the 5th student has the same id as the 137th student then remove the 137th student from the Students object but keep the 5th student.

(i) It is inefficient to store and then remove some students from the Students object. Starting from an empty Students object, read the rows from the input file and this time ignore duplicate ids as you store students in the Students object. This method should return the number of student objects that were read but not stored in the collection.

(j) Sort the students by id in ascending order.

4 Stacks, Queues, and Deques

(a) Stacks make it very easy to reverse a collection - just push everything then pop it all. Reverse the Students using a Java Stack.

(b) The Java Stack documentation has a preference of the Deque interface over the Stack class. Reverse the collection again but this time use a Deque.

(c) Suppose XULA administrators are trying to decide the best number of entrance lines for the cafeteria. They know that every hour [7am..7pm], 0 to 50 students get in line and 0 to 40 students are admitted to the cafeteria. The cafeteria can hold no more than 200 students at a time. The administrators think a new entrance line should be opened any hour when more than 40 students are in line and a line should be closed any time fewer than 20 students are in line. Of course, at all times at least one line must be open. Design a simulation that shows how many lines are needed every hour. For this simulation, (a) at 7am there is only one line that randomly [0..50] students get in, (b) that same hour randomly [0..40] students are waited on, (c) your program displays reporting data about that hour, (d) pauses for 2 second, and then (e) repeat everything for the next hour. See the Javadocs on **Thread.sleep()** for how to pause your program.

5 Sets and Maps

(a) Use a Java Set to determine if the data has any duplicate ids in the student data.

(b) Use a Java Map to determine how many of each name there is in the student data.