Assignment 1: Game Score Calculator - Java

Notes

- This assignment is to be done **individually**. Do not share your code or solution, do not copy code found online; ask all questions on Piazza discussion forum (see webpage).
  - You may use code provide by the instructor, or in the guides/videos provided by the instructor (such as instructor's YouTube videos).
  - You may follow any guides you like online as long as they are providing you information on how to solve the problem vs solutions to the assignment.
  - If receiving help from someone, they must be teaching you not writing your code.
  - You may not resubmit code you have previously submitted for any course or offering.
- Cite your sources in your code by putting the URL in a comment.

1. Java Work

In this assignment you will be writing a (plain old) Java application, not an Android app. Much of what you create will be used in Assignment 2 when we implement a similar Android app.

You must submit an Android Studio project which contains your Java code. All work can be done inside Android Studio. See website for video showing how to write "plain old" Java code inside Android Studio.

Install and configure an Android development environment on a computer.

- Download the latest Java SE JDK (it includes the JRE)
  http://www.oracle.com/technetwork/java/javase/downloads/index.html
- Android studios:
  https://developer.android.com/studio/index.html
- You are encouraged to install this software on your own computer; however, if you wish, you can complete this assignment without doing an installation using the Linux CSIL labs.

1.1 Game Background

- In this assignment, you will create a text-based Java application which calculates and tracks player scores in a board game. This fictitious game is based on the scoring of the game Lost Cities, by Reiner Knizia[1].
- The details of playing the game are not important here; all you need to know is:
  - Each game can have between 1 and 4 players (inclusive)
  - Each player plays 0 or more cards
  - Some of the cards are "wager" cards, which multiply the player's score
  - Some of the cards are point cards
- **Scoring**
  - At the end of the game, players will:
    - count the number of cards they have played (including point cards and wager cards)
    - count the number of wager cards they have played
    - sum up the total number of points on the point cards they have played
  - A player's score is calculated using the following rules
    - Add up the number of points on their point cards, then subtract 20.
      This may be negative.
    - Each wager card multiplies this total. For example, if the player has 1, 2, or 3 wager cards, multiply the above total by 2, 3, or 4 respectively (and so on). There is no limit to the number of wager cards a player may have.
    - Finally, if a player has played at least 8 cards (counting both point cards and wager cards), then they earn an overall bonus of 20 points. This bonus is *not* multiplied by wager cards.
  - For example, a player with:
    - 4 cards, a sum of 15 points and 1 wager cards would score -10.
    - 10 cards, a sum of 30 points and 2 wager cards would score 50.
  - In a game, the player with the highest score wins (ties are possible).

1.2 GitLab

You must use SFU's GitLab for this assignment.

- See Appendix A at the end of this document for directions, or a video tutorial linked on the course website.
- You must commit your work to GitLab reasonably frequently. I suggest you do so about every hour or so of work.
- You must have 3 or more commits by the time you are done the assignment. (It might be significantly higher!)

1 Lost Cites is for 2 players (vs the 1 to 4 here) and has multiple different colours to score (vs the one here).

1.3 The Program

Your program must have two packages, as shown in the posted Java videos:

Model Package

- The model is the part of program which manages the data and much of the logic.
  - The UI class(es) will call classes in the model.
  - Classes in the model must *not* call the UI classes[1].

- Have at least three classes in the model package:
  - **Game manager**: Store a collection of games.
    - Class must support adding new games, retrieving a specific game by its index, and removing a game by its index.
    - *Hint: Watch the videos on the course website about learning Java: they show how to create a class very much like this!*
  - **Game**: Represents one game played by 1 to 4 players. Also stores the date/time when the Game was created.
    - You must use the `LocalDateTime` class to store its creation date/time.
    - Must be able to report which player(s) won
  - **Player score**: Represents the score of a single player during a game.
    - See section 1.1 Game Background for the information to store
    - Constraints:
      - Number of cards, sum of points, and number of wager cards must each be non-negative.
      - If there are 0 cards, then the sum of points and number of wager cards must be 0.
      - Beyond this, you don't need to add constraints on the number of cards vs points vs number of wager cards. For example, your program need not detect that with 2 cards one could not have 1000 points or 3 wager cards.
    - The class must enforce constraints on each of its values using exceptions. For example:
      ```
      if (month > 12) {
          throw new IllegalArgumentException("Month must be 12 or less");
      }
      ```
    - Calculate the player's score.

- Classes in the model package must *not* print to the screen or read from the keyboard. Instead they should be called by the classes in the UI which handle the screen and keyboard.

1 A model class can only 'call' a UI class via the observer pattern, which we will learn later in the course.

Text UI Package

- Have a text UI class which interacts with the user by printing to the screen and reading from the keyboard.

- Each time the user enters a number, if the number is out of range for that value then display a message stating what values are allowed and re-ask for the value. Applies to menu choices and entering values.
  - See the sample output on the course website for the required flow when there is an error. Your output need *not* match the sample output exactly, but it should be of equal quality.

- Text UI must repeatedly display a menu with the following choices:
  - List known games
    - If there are no games yet, state “No games”
    - Display all the games, numbering them starting at 1.
    - Display each game in the format:
      `40 vs 40 vs -12, winner player(s): 1, 2 (@2021-08-31 13:37)`
      - The score earned by each player, separated by “ vs ”
      - Comma separated list of winning players’ numbers (1-indexed).
      - Time stamp of when the game was created in format: yyyy-MM-dd HH:mm
  - Add a new game
    - Ask user for how many player, then for each player record the information about their game’s score.
    - If the user enters 0 cards were played for a player, then do not ask about points or wager cards.
  - Remove an existing game
    - Display a list of known games (same format as above)
    - Ask user to enter a game’s number to delete it.
    - Allow user to enter 0 to delete none.
  - Exit
    - Close the program

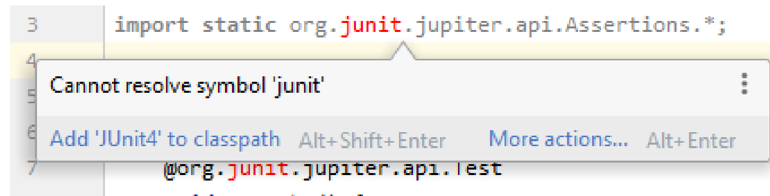
- Text UI Details
  - See the sample text UI code (on website) for how to use the screen and keyboard.
    - Change this provided code as needed.
    - This code tries to make use of the game manager class which you must write.
  - Use a Scanner to read from the keyboard. Assume the user always enters the correct *type* of data. For example, if you are expecting an `int` (using `myScanner.nextInt()`), then assume the user enters an `int` like 5, not a string like “yo”.
  - Avoid duplicate code. If you are writing the same (similar) code twice, make it into a function! Don’t Repeat Yourself (DRY)!
  - Magic Numbers
    - You may hard-code the numbers for your menu.
    - You may *not* use other magic numbers, such as 1 in your code for the minimum number of players; make this a named constant.
    - Heuristic: 0 (and sometimes 1) are often not magical. Most other numbers are.

1.4 Testing

- Write a JUnit 5 test class for your player score class.
- Must achieve 100% code coverage of your player score class.
- Must test each parameter to the player score class out of range at least once.
- Must test score calculation using boundary value analysis.
- Expect to have 5 (or more!) test methods.

Android Studio and JUnit 5 Build Problem

After you create the JUnit 5 test class for your code, you may see an error: `Cannot resolve symbol 'junit'` (or `'jupiter'`)



Solution

- File → Project Structure
- Select dependencies (left side)
- Select your module
- Select "junit-jupiter" in the top area
- In details below, select Requested Version to 5.6 (or newer)

2. Getting to Know Android Studio

Nothing from this Android section is to be submitted or for marks. However, the next assignment will require to to create an Android app. Therefore, it is **strongly** recommended that you start learning Android now.

2.1 Resources

- Read chapters 1 to 5 of Android Programming: The Big Nerd Ranch book.
- You may also want watch some of the provided videos on YouTube. Suggested videos:
  - Creating a button
  - UI Layouts
  - Java Objects in Android Activities
  - Creating a 2nd activity.

3. Deliverables

To CourSys (https://courses.cs.sfu.ca/) you must submit:

1. Screenshot of the commits page of your GitLab repo (on csil-git1.cs.surrey.sfu.ca).
   - In web browser, open your repository and click "Commits".
   - It should now show at least 3 commits.
   (If there were problems with Git while completing the assignment, you are welcome to create three trivial commits, even if you have completed the rest of the assignment).
   - Screenshot this and submit the image.

2. URL and Tag for your Git repository:
   1. **Add the TA for the course as a "Developer" member of your repo:**
      - Goto `csil-git1.cs.surrey.sfu.ca` and select your project
      - On the left hand side, click the cog-wheel drop-down ('Settings')
      - Select "Members"
      - Add just the TA to your repo as a **Developer**. TA SFU ID = **hga50**

   2. **Create a tag for your submission as follows:**
      - In Android Studio, go to VCS --> Git --> Tag...
      - Enter a name for your tag, such as: `final_submission`
      - Leave Commit and Message blank.
      - Click Create Tag
      - Push changes to remote repo. On "Push Commits" dialog, select **"Push Tags: All"**.
      - You can check the tag was pushed correctly in GitLab online.
      - If you resubmit, create a new tag as above and submit the new tag via CourSys.

   3. **Submit the git@... URL and tag name to CourSys**
      - Find the Git URL on csil-git1.cs.surrey.sfu.ca/. Should be similar to:
      `git@csil-git1.cs.surrey.sfu.ca:yourid/myProjName.git`
      - The "tag" is the name you used above, such as "`final_submission`"
      - Git URL may also start with `https://`

3. ZIP file of your project, as per directions on course website.

Please remember that all submissions will automatically be compared for unexplainable similar submissions. Please make sure you do your own original work.

Appendix A: Git Lab

Initial GitLab Checkin

1. Install Git on your computer.

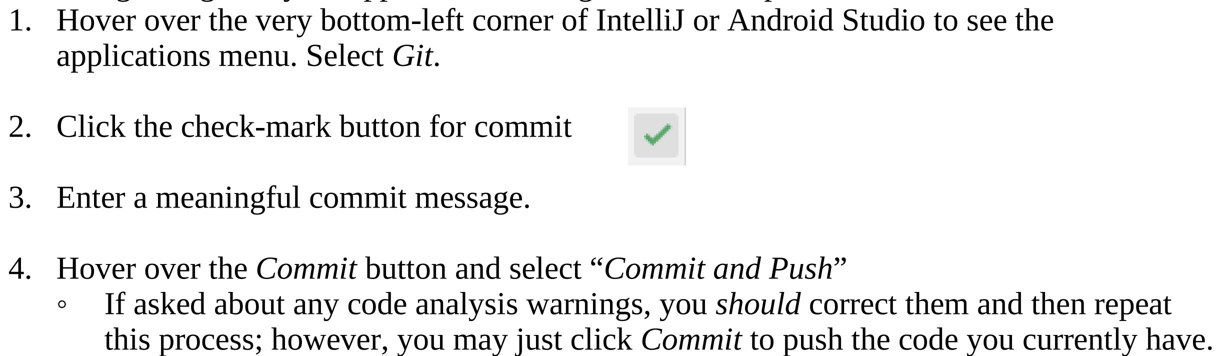
2. Create a new project on SFU Computing Science GitLab server
   a. Via a web browser, log into `https://csil-git1.cs.surrey.sfu.ca/` and click "*New Project*" (top right).
   b. Name it something like `cmpt276As1` and click "*Create project*"
   c. Generate an SSH key on your computer and upload it to GitLab. See video on course website for directions in Windows.
   d. Copy the Git URL for the project. It's near the top middle of the page.
      In the drop-down to its left, select "SSH", then the URL should be something like:
      `git@csil-git1.cs.surrey.sfu.ca:bfraser/cmpt276As1.git`
      Note: You may need to select "Clone with HTTP" instead and type in your SFU username and password to get around the new (2021) firewall.

3. In IntelliJ or Android Studio, enable Git and commit the project:
   a. If you don't have a 'Git' menu, then from the VCS menu select *Enable Version Control Integration* and select *Git* and press *OK*.
   b. In the very bottom left corner of IntelliJ / Android Studio, hover over the button to show the applications menu and select "*Git*"
      - If it's not there, then enabling VCS likely failed because the IDE could not find Git. Ensure you have Git installed and then go to:
        File --> Settings --> Version Control --> Git, and set the path to Git
        (on Windows likely "`C:\Program Files\Git\bin\git.exe`")
   c. Expand the *Unversioned Files* and select them all.
   d. Right click the selected files and say "*Add to VCS*"
   e. From the menu select *Git* --> *Commit Changes*.
      Enter a description like "Initial commit".
      Hover over the *Commit* button, from the drop-down select "*Commit and Push*"
      - If asked about Code Analysis, for the time being you can just *Commit* instead of reviewing issues.
      - In the Push Commits window, click "*Define remote*" in the top left. Enter that `git@csil-git1.cs.surrey.sfu.ca:...` URL you copied from above (or the HTTPS://…) and click *OK*.
      - Click *Push*

4. Ensure the files were pushed by viewing the project in GitLab via the web.

Checking in Changes

After making changes to your app, commit changes to Git and push them to the GitLab server.

1. Hover over the very bottom-left corner of IntelliJ or Android Studio to see the applications menu. Select *Git*.
2. Click the check-mark button for commit 
3. Enter a meaningful commit message.
4. Hover over the *Commit* button and select “*Commit and Push*”
    - If asked about any code analysis warnings, you *should* correct them and then repeat this process; however, you may just click *Commit* to push the code you currently have.