Homework 1: Warm-Up

If you have already created a `hw1` branch and want to try again to do better, you can create a new branch named `hw1a`, or `hw1b`, or `hw1c`, and so on. I will grade the alphabetically-latest such branch in your repository. You must submit a Regrade request on Gradescope for your late submission to be graded.

*Note: The rules on collaboration are slightly relaxed for this assignment. You may get help by allowing other people to look at your work. You still **may not** share written code (including literal commands to run), and you **may not** allow others to use your mouse or keyboard.*

1. Log into gitlab.com.
   Perform the following tasks using the gitlab.com website:
   a. Go to the "course repo" at https://gitlab.com/rmculpepper/cs410-f23.
      You should have received an email inviting you to the repository, with one of the following subjects:
      - "Access to the Ryan Culpepper / cs410-f23 project was granted" —Your GitLab account was associated with your `@umb.edu` email, and you were automatically granted access.
      - "Ryan Culpepper invited you to join GitLab" —The email has a "Join Now" button/link. Click it for more instructions.
   b. Fork the course repo. The rest of these instructions will refer to that fork that you created as "your repo" or "your fork".
      When creating the fork, you must choose a `Project URL`. You must use your use name as the "namespace" part, and you must leave the "Project slug" as `cs410-f23`.
   c. Make sure that your repo is *private* by going to `Settings > General > Visibility, project features, permissions` and checking the `Project visibility` setting.
   d. Add your instructor (user `rmculpepper`) to your repo with the role `Developer`. Do this by going to `Manage > Members`, then click `Invite Members`. Do not add anyone else to your repo.
2. Make a local working copy (aka "clone") of your repo. (This is sometimes called "checking out" the repository, but Git mainly uses "checkout" to refer to switching the branch you are actively working on.)
   Perform the following tasks on your local copy:
   a. Create a branch called `hw1` based on the `start-hw1` branch. Check out `hw1`.
   b. Edit the file `hw1/info.json` according to the directions below. Commit your changes.
   c. Merge the course repo's `fix-hw1` branch (probably visible to you as `origin/fix-hw1`) into your `hw1` branch. Fix the merge conflicts. (Do not rebase, if you know what that is. You must create a merge commit.)
   d. Use IntelliJ to build and run the Scrambler project according to the directions below.

e. Push your hw1 branch to your repository on GitLab.

3. Check your work.

1.1 Edit hw1/info.json

Edit the JSON object in `hw1/info.json`.

First, correct the values of the `"course ID"` and `"course name"` keys. Do not change the names of the keys.

Then add the following information to the JSON object using the given keys exactly. Add your new entries after the existing entries.

- `"name"` — (String) Your name.
- `"major"` — (String or null) Your declared major.
- `"PLs"` — (Array of String) Programming languages that you have some experience with.
- `"400-level prerequisite"` – (String or null) The 400-level course that you have taken to satisfy the CS410 prerequisites.
- `"favorite algorithm"` — (String) The name of your favorite algorithm.
- `"favorite data structure"` — (String) The name of your favorite data structure
- `"CS interests"` — (Array of String) A list of areas within Computer Science that interest you.
- `"non-CS interests"` — (Array of String) A list of things outside of Computer Science that interest you.

Make sure that `hw1/info.json` is valid JSON and that each key has the requested type of value.

1.2 Build and run the Scrambler

1. Use IntelliJ to create a new project called Scrambler, and copy the Java file at `hw1/Scrambler.java` into the project. *Warning:* Make sure that the new `Scrambler.java` is in the right directory, and make sure that IntelliJ does not "helpfully" delete the package declaration!
   For the build system, select either IntelliJ or Maven. Do not select Gradle.

2. Build the project.

3. Run the Scrambler on your `@umb.edu` email address. Commit the result as a new file called `hw1/scrambled-email.txt`.

4. Use the JShell Console to evaluate `scramble("Software Engineering", 410)` *without editing the Java source file!* Commit the result to a new file called `hw1/scrambled-cs410.txt`.
   Warning: Configuring JShell Console in IntelliJ is somewhat finicky and unintuitive. Follow the JShell Console tool guide carefully. In particular:

a. In the Libraries page, you must add the path that immediately contains the `cs410` directory of compiled `.class` files. If you selected the IntelliJ build system when creating the project, the path should look like `.../hw1/Scrambler/out/production/Scrambler`. If you select the Maven build system, the path should look like `.../hw1/Scrambler/target/classes`.

b. In the Modules page, make sure the Scrambler library is listed in the Dependencies tab. (It does not appear to matter whether the checkbox is checked.)

5. Check in your project directory.

1.3 Check your work

Visit your fork on gitlab.com.

Go to `Manage > Members`. You should see two users listed: your username listed as "Owner" and mine (`rmculpepper`) listed as "Developer".

Go back to your fork's main page. The page should have a drop-down box listing your repo's branches; it probably has `main` selected by default. (The drop-down box should be right above the box saying "Forked from ...".)

Click the box. There should be a branch named `hw1`. Select it, then click on the `hw1` directory and verify that it contains all of the files you were expected to check in.

Your fork may have other branches, too, but `hw1` is the one that matters.

Note: You should *not* create a "merge request" (also known as a "pull request") with your changes. If you have created a merge request, you should close it.

Homework 2: Single-Class Design

In this homework, you will design (that is, plan, implement, and test three classes according to the rules for Single-Class Design (as discussed in Lectures 02 through 04).

Treat the three classes as three unrelated Single-Class Design exercises. That is, do not treat them as a single Multi-Class Design problem. They are in the same package, in the same project, purely for convenience.

This homework does not fully specify the signature of every method, the types you should use in the class representations, and so on. You are expected to make reasonable inferences about what to implement and how to implement it.

The homework will be graded based on the principles of class design, the rules for Class Design Documentation, the correctness of your implementation, the quality and comprehensiveness of your testing, and the reasonableness of your inferences about issues that the homework does not explicitly specify.

Additional rules:

- Each class must be in the `cs410` package.

- Classes and methods must be named exactly as written.

- All methods should be instance (non-`static`) methods except those explicitly described as static in the homework specification.

- Your work must be on a branch named `hw2`. See Making the `hw2` branch.

- On the `hw2` branch, there must be in a directory named `Homework2`, and the contents of that directory must follow the Maven Directory Layout. It must be possible to build and test your project using Maven (`mvn compile` and `mvn test`, respectively). IntelliJ users should see IntelliJ and Maven below.

- If your work is not in the correct branch, or not in the correct directory, or cannot be built with Maven, then it will not be graded and you will not receive credit for the assignment.

2.1 `Duration`

Design a class named **`Duration`** representing a duration of time, with a precision of seconds. That is, do not support fractional seconds. Duration objects should be immutable.

Define two static factory methods (both named **`of`**). One should take a single argument representing the total number of seconds in the duration. The other should take three arguments containing the numbers of hours, minutes, and seconds in the duration.

Define an **`add`** method that adds two **`Duration`** objects together.

Define a **`seconds`** method that returns the Duration's total number of seconds.

Override **`toString`** to display the duration in `H:MM:SS` format. For example, the nominal duration of a CS410 lecture is `"1:15:00"`. One million seconds corresponds to `"277:46:40"`. (Hint: See the **`String.format`** method.)

2.2 `Rectangle`

Design a class named **`Rectangle`** for representing a rectangle on a 2D grid with integer coordinates.

Define a static factory method named **`of`** that takes four integer arguments: the "x" and "y" coordinates of one corner of the rectangle, and the "x" and "y" coordinates of the opposite corner.

Define a method to calculate the **area** of the rectangle.

Define a method that determines whether a rectangle **contains** a point, represented by integer "x" and "y" arguments.

Define a method that determines whether a rectangle **overlaps** with another rectangle.

2.3 TwoLaneQueue

Design a class named **TwoLaneQueue**. It is like a standard mutable queue, except that it has two "lanes": a slow lane and a fast lane. (Think of airport boarding lines with separate lanes for General vs Priority boarding, for example.)

Define a public constructor that takes zero arguments. *(Clarified 10/1 2pm for compatibility with the autograder tests. Since this change was posted late, reasonable alternatives will not be penalized during manual grading, but automated feedback through the autograder is only possible if the public constructor is defined.)*

The items contained by the lanes are simple **String** objects.

The **enqueueFast** operation adds an item to the end of the fast lane.

The **enqueueSlow** operation adds an item to the end of the slow lane.

The **dequeue** operation removes and returns an item from the front of one of the lanes. Specifically:

- If there is an item waiting in the fast lane, it is dequeued and returned.
- If there is an item waiting in the slow lane, it is dequeued and returned.
- If both lanes are empty, a **java.util.NoSuchElementException** is thrown.
- Except, as a kind of limited fairness, if the fast lane has produced three or more items in a row, and if the slow lane is not empty, then the slow lane gets to take a turn instead.

You must not place a bound on the number of items in the fast and slow lanes.

You may use any class present in **java.util.*** (as of JDK 11) in your implementation.

2.4 Making the hw2 branch

You can use the following command to make a new branch called hw2 that does not share any history with any of your other branches:

```
git switch --orphan hw2
```

2.5 IntelliJ and Maven

IntelliJ makes it fairly easy to satisfy the homework's requirements for directory structure and Maven-compatibility.

- Before you create the new project, make sure your git working copy is in the `hw2` branch.
- Create a new IntelliJ project named `Homework2`. The Location should be your git working copy's directory. Select **Maven** for the Build System option. IntelliJ will automatically create a `pom.xml` file, which contains information about your project for Maven.

  Add the following text to the bottom of your `pom.xml` file, just before the closing `</project>` tag:

  ```xml
  <build>
      <plugins>
          <plugin>
              <artifactId>maven-surefire-plugin</artifactId>
              <version>3.1.2</version>
          </plugin>
      </plugins>
  </build>
  ```

  IntelliJ should automatically download the necessary plugins.
- Create your testing classes by opening the source file of the class to test (eg, `Duration.java`), selecting the name of the class, and hitting Alt-Enter to open a special context menu, then selecting "Create Test". That will automatically create the `src/test/java` directory and mark it as a test source root. It will also name the testing file according to the convention expected by the Maven Surefire plugin (eg, `DurationTest`).

2.6 Gradescope Autograder

There is now a Homework 2 assignment available on Gradescope with a limited autograder.

The autograder expects a single file named `hw2.json`, which should contain a JSON object with a single key named `"gitlab_user"`, whose value must be your Gitlab username. It must be the same Gitlab username you used for Homework 1. For example, my `hw2.json` file would have the following contents:

```
{ "gitlab_user": "rmculpepper" }
```

The autograder *currently* checks that your repo on Gitlab is accessible, that repo has the correct branch (`hw2`), that branch contains the correct directory (`Homework2`), and the project contains the expected directory structure and files, and the project can be built using Maven (`mvn compile`).

You can resubmit exactly the same file to make the autograder pull the most recent version of your code from Gitlab.

~~The autograder does not currently perform any tests about the correctness of your code. Correctness tests may be added before the homework deadline.~~ The autograder now runs correctness tests. *(Updated 10/1 2pm, tests added by 9am.)*

Passing the autograder checks means that your code can be graded, but the autograder does not assign the final grade for the homework. The grade depends on many other factors; grading criteria are listed at beginning of the homework assignment.

Homework 3: Multi-Class Design

In this homework, you will design, implement, and test extensions to the existing WebFilmz code base.

The homework will be graded based on the principles of class design, the rules for Class Design Documentation, the correctness of your implementation, the quality and comprehensiveness of your testing, and the reasonableness of your inferences about issues that the homework does not explicitly specify.

Additional rules:

- Each class must be in the `cs410.webfilmz` package.
- Classes and methods must be named exactly as written.
- You cannot use this document as an excuse to avoid writing important information in the Class Design Documentation.
- You must not change the signature of any existing `public` method, unless the homework explicitly says to.
- You must not add any non-`private` methods, fields, or constructors to `Film`, `User`, `Catalog`, or `ILikeFilm` unless the homework explicitly says to. You may override the `toString` method on any of those classes. You may add `private` helper methods to any class.
- You must not modify the `BaseFilmTest`, `BaseUserTest`, and `BaseCatalogTest` classes. You should add your test cases to `UserTest` and `CatalogTest` instead.
- Your work must be on a branch named hw3, which should be based upon the course repository's hw3 branch. See Making the hw3 branch.
- On the hw3 branch, there must be a directory named WebFilmz, and the contents of that directory must follow the Maven Directory Layout. It must be possible to build and test your project using Maven (mvn compile and mvn test, respectively).
- If your work is not in the correct branch, or not in the correct directory, or cannot be built with Maven, then it will not be graded and you will not receive credit for the assignment.

3.1 Fix personalized recommendations

The **BaseUserTest** has two failing test cases: **getRecommendationsNewReleases** and **getRecommendationsByDirector**. The tests cases reflect the correct intended behavior.

Both test cases fail because the methods in **User** do not remove films that the user has already watched from the set of recommendations. In general, the recommendation methods in **User** should not include films the user has already watched.

Fix the code in the **User** class so that the test cases pass.

Note: the **getAllRecommendations** method now takes an integer argument called **initialGenericRecsCount** that represents the number of recommendations to be generated from the catalog *before* any watched films are removed. The rationale is that if the user has already watched all recent films, the "New Releases" recommendations should be empty, rather than containing movies from the 1990s.

3.2 Complete by-genre recommendations

Design, implement, and test the **public getRecommendationsByGenre** method in **Catalog**. Remove the direct dependence on the **User** class in the same way as the **getRecommendationsByDirector** method.

Design, implement, and test the **public getRecommendationsByGenre** method in the **User** class.

Update the **getAllRecommendations** method to also include entries for "Favorite Genres", "Most Watched", and "Most Liked" in its result.

3.3 Cache liked directors and genres in `User`

Change **User** to store the set of the user's liked directors and genres. Update **isLikedDirector** and **isLikedGenre** to use this data instead of the existing loop over all liked films. Update the class's design documentation as appropriate.

3.4 Making the `hw3` branch

Here is how you should create your `hw3` branch.

Go to your local working repo. Run `git remote show`. It will probably print one line, `origin`. Git uses the name `origin` by default for the remote repository you cloned your working copy from. Run `git remote show origin`. It will print a bunch of information, but at the top should be two URLs pointing to your GitLab repo.

Run the following commands in order:

- either `git remote add course-repo https://gitlab.com/rmculpepper/cs410-f23` or `git remote add course-repo git@gitlab.com:rmculpepper/cs410-f23` —depending on whether you are using an SSH key to access GitLab. This adds the course repo as a separate "remote" (that is, a reference to a remote repository) named `course-repo`.

- `git fetch course-repo` — Gets any new content from the `course-repo` remote.

- `git branch hw3 course-repo/hw3`

- `git switch hw3`

- `git push --set-upstream origin hw3` or `git push -u origin hw3`

Go to your GitLab page and verify that it now contains a `hw3` branch.

3.5 Gradescope Autograder

There is now a Homework 3 assignment available on Gradescope with a limited autograder. It works in about the same way as for Homework 2.

The autograder expects a single file named `hw3.json`, which should contain a JSON object with a single key named `"gitlab_user"`, whose value must be your Gitlab user name. It must be the same GitLab user name you used for Homework 1. For example, my `hw3.json` file would have the following contents:

```
{ "gitlab_user": "rmculpepper" }
```

You can click "Rerun Autograder" to make the autograder pull the most recent version of your code from GitLab.