# COS 126 General Computer Science Spring 2022 Programming Exam 2

**Before the exam.** Read this page of instructions before the exam begins. However, do not start the exam (not even reading the next page!) until instructed to begin.

**Exam duration.** Once the exam begins, you have eighty (80) minutes to complete it.

**Submission & Grading.** As with the COS 126 assignments, you can and should submit your code multiple times, but only the last version that compiles will be graded. You are responsible for uploading the correct version of your solution. Your final submitted program *must* compile. If not, we will attempt to grade the latest submission that does actually compile. Your program will be graded on whether each method has the specified behavior (i.e., correct output).

**TigerFile plugin for IntelliJ.** For those of you experimenting with this plugin, please **do not** use it during the exam! Instead, access TigerFile through its regular web interface.

**Resources.** This exam is "open-book" but not "open-internet" – for example, do not use Google to find the answer to questions like: "How to do X in Java." During the exam, you may only use: the textbook, the booksite, your notes, your code from programming assignments, the code on the COS 126 course website, materials from lectures, precepts, and existing Ed posts.

**No communication is permitted** (e.g., talking, texting, Ed, etc.) during the exam, except with course staff. We will be sitting outside the exam room if you have a question.

**Do not discuss it later.** Due to multiple exam times, and various conflicts, some of your peers will take the exam at different times. Do not discuss the exam contents with anyone (not even other students that you know already took the exam!) until after the graded exams are returned.

**Honor Code pledge.** You must electronically sign the honor code by retyping the pledge below and then /s/ in the file `honor-code.txt`, and upload it to TigerFile as part of your solution. Please do this now. (The /s/ is your "electronic signature.")

*"I pledge my honor that I have not violated the Honor Code during this examination. /s/"*

**Do not examine or start the exam until instructed to do so.**

**Problem.** Develop a Java program that predicts people's movie preferences based on the ratings from other people with similar preferences. Much of the exam will be spent working on this Java class that keeps track of a person's preferences:

```
public class Preferences {

    // Constructor. Preferences for a person with this name.
    public Preferences(String name)

    // Return the person's name.
    public String getName()

    // Insert this person's rating for a particular movie.
    public void addRating(String movie, int rating)

    // Has this person rated this movie?
    public boolean hasRating(String movie)

    // If this person has rated this movie, return the rating.
    // Otherwise, throw a custom runtime exception.
    public int getRating(String movie)

    // Return a character symbol indicating a given movie rating:
    // plus sign ('+') for +1 or lowercase 'x' for -1.
    public static char ratingSymbol(int rating)

    // Return a string containing all movie ratings from this person.
    // See exam for details.
    public String toString()

    // Return similarity between this person's ratings and those of another person.
    // Defined as: only considering movies that both people rated,
    // the number of movies where they agree minus the number where they disagree.
    // Assume ratings are either +1 or -1.
    public int similarity(Preferences other)

    // You can use the main method to test your functions above.
    public static void main(String[] args)
}
```

To get started, you are given a file **`Preferences.java`** that contains this public API for this class – do not change this API. You may add your own **`private`** helper functions, but it is not needed in order to complete the exam.

First, implement the **constructor**. It should initialize two private instance variables – one that stores the person's name, and a second that keeps track of the person's preferences. The second variable should hold a symbol table (`ST`) that maps from a string (a movie title) to an integer (this person's rating for the movie). Throughout the exam, ratings are represented by integers – assume they are either `+1` (the person liked the movie) or `–1` ( they did not like it). Also, you may assume that movie titles all have spaces replaced by the underscore character, like this: `"The_Graduate"`.

Implement the remaining methods in order – **`getName`**, **`addRating`**, **`hasRating`**, **`getRating`**, etc. – testing them as you go using the **`main`** method. (*Hint: the methods that deal with ratings should be able to utilize the symbol table to do most of the work*.) Each method has a comment describing what it should do (shown above, and also in the source file) and there are some additional details for a few of these methods in the following paragraphs – so scan over the next page before starting. The provided **`main`** method already has some useful tests for all the methods, and you are welcome to modify it as you like. For this exam, unless otherwise noted, you may also assume that all inputs are valid, including function arguments and standard input, and raters only rate a movie once.

## More Notes on Preferences.java

The **`getRating`** method should return a movie rating if the person has rated that title. Otherwise, it should throw a custom runtime exception, formatted just like this for `Bobby` and `The_Graduate`: `"Bobby did not rate The_Graduate"`

The **`toString`** method should return a string containing the person's name, followed by a colon on one line. *(Hint: to add a newline at the end of the line add `"\n"`.)* Next, it should add the list of movies rated by the person (one per line, with newlines at the end, even on the last line.) They should be in alphabetical order – note that the ST iterator automatically loops over keys in alphabetical order. Each title should be preceded by the symbol indicating that person's rating of that movie, like below, output by the method **`ratingSymbol`**. You may use `String` or `StringBuilder` for this task.

```
Alice:
+ Casablanca
x Star_Wars
+ The_Graduate
```

The **`similarity`** method returns an integer indicating how similar another person's preferences are to this person's. If the number is positive it means that they are similar, and if it is a high positive number it means *very* similar. Likewise, if it is negative then they have dissimilar preferences. It is computed as the number of the movies where their ratings are the same minus the number of the movies where their ratings differ (only considering movies that they have both rated).

Once you have all the functions implemented, you can test them all together **`main`** method provided in the supplied file **`Preferences.java`**, by compiling and running it as follows:

```
% javac-introcs Preferences.java
% java-introcs Preferences
Alice, before adding ratings:
Alice:

Alice, after adding ratings:
Alice:
+ Casablanca
x Star_Wars
+ The_Graduate

Bobby, after adding ratings:
Bobby:
+ Casablanca
x Star_Wars

Alice-Bob similarity: 2
This next function should throw an exception:
Exception in thread "main" java.lang.RuntimeException: Bobby did not rate The_Graduate
	at Preferences.getRating(Preferences.java:34)
	at Preferences.main(Preferences.java:99)
```

The commands and output that are shown in the gray box above, as well as those in all the other gray boxes in this exam, are also available in the file **`example-command-output.txt`** in your project folder (for easy examination and cut-and-paste).

## MovieMatcher.java

The next phase is to complete some missing methods in **`MovieMatcher.java`**. The primary purpose of this class is to read the preferences of a group of people from `StdIn` and make movie recommendations to people, based on other people's preferences. The program reads the first command-line argument (`SIMILAR`, `PREDICT`, or `RECOMMEND`) and calls an appropriate test function based on that string.

First, implement **`getPersonPreferences`**, which gets as an argument the name of a person and returns their preferences (assume they exist). It does so by looking up their name in the symbol table (ST) `people`, which has already been populated by the provided method `addRatingsFromStdIn` (by reading people's preferences from `StdIn`).

Next implement the method **`mostSimilar`**, which returns the name of the person whose preferences are most similar to a given person. You may assume that there are at least two people. For example, if there are only two people's preferences read from `StdIn` – named Alice and Bob – and the argument is `"Alice"`, this method would return `"Bob"`. On the other hand, if there are more than two people, this method would return the person whose preferences are most similar to those of Alice. Note that this method should call the **`similarity`** method you already implemented to determine how similar two people are. (*Hint: you may wish to use the constant `Integer.MIN_VALUE`, which is the lowest possible integer.)* In case of a tie (two equally similar other people) choose the earlier one in alphabetical order. Here are example outputs:

```
% javac-introcs MovieMatcher.java
% java-introcs MovieMatcher SIMILAR Angel < ratings.txt
Most similar to Angel is: Genie
% java-introcs MovieMatcher SIMILAR Briar < ratings.txt
Most similar to Briar is: Genie
% java-introcs MovieMatcher SIMILAR Casey < ratings.txt
Most similar to Casey is: India
```

*Congratulations if you made it this far! These next two functions are a bit more difficult, to provide an extra challenge for those with sufficient time. These functions account for a small portion (<10%) of your exam, so only attempt them if you are confident in your previous steps.*

Next implement the method **`predict`**, which returns a numerical prediction of how a person would rate a movie. Return a positive integer if they are likely to like the movie (or negative if they are expected to dislike it), whose magnitude reflects confidence. This prediction $p_{im}$ for a person $i$ liking movie $m$ is computed as a sum over the ratings ($r_{jm}$) of all *other* people $j$ who rated this movie, weighted by the similarity of $i$ and $j$ ($s_{ij}$): $p_{im} = \sum_{j \neq i} r_{jm} s_{ij}$

When the other person is similar (positive $s_{ij}$), their rating increases the sum, weighted by *how* similar they are. Likewise, if they are dissimilar (negative $s_{ij}$) their rating will reduce the sum. For example, suppose that there are only three people, Angel, Briar, and Casey. And suppose the similarity between Angel and Briar is `+3` and the similarity between Angel and Casey is `-2`. And suppose Briar liked the movie Casablanca (rating `+1`), and Casey did not like it (rating `-1`), and we want to predict if Angel would like the movie. We would use the formula: $p$ = ( `+1` × `+3` ) + ( `-1` × `-2` ) = `+5`. Since $p > 0$ we think Angel would like the movie Casablanca. Here are some example outputs:

```
% javac-introcs MovieMatcher.java
% java-introcs MovieMatcher PREDICT Angel Amadeus < ratings.txt
Prediction for Angel and Amadeus: 54
% java-introcs MovieMatcher PREDICT Angel Ben-Hur < ratings.txt
Prediction for Angel and Ben-Hur: -28
% java-introcs MovieMatcher PREDICT Casey Ben-Hur < ratings.txt
Prediction for Casey and Ben-Hur: -19
% java-introcs MovieMatcher PREDICT Casey Casablanca < ratings.txt
Prediction for Casey and Casablanca: 135
```

Finally, implement the method **`recommend`**, where the goal is to recommend a movie to a person who has not seen it. The function argument is the name of a person. Loop over all movies and check to see if that person has seen them. If not, use the predict function above to predict how much the person would like it. Remember which movie had the highest prediction score, and output that movie. Here are some example outputs:

```
% javac-introcs MovieMatcher.java
% java-introcs MovieMatcher RECOMMEND Angel < ratings.txt
Recommendation for Angel is: Unforgiven
% java-introcs MovieMatcher RECOMMEND Briar < ratings.txt
Recommendation for Briar is: Amadeus
% java-introcs MovieMatcher RECOMMEND Casey < ratings.txt
Recommendation for Casey is: Annie_Hall
```

## *This is the end of the exam!*

Here is some food for thought, for your consideration after the exam:

- If you wanted to make music recommendations instead of movie recommendations – what would be different?
- The one hundred movies in **`ratings.txt`** are the 100 Greatest American Movies of All Time selected by the American Film Institute.
- The 1299 ratings in **`ratings.txt`** came from real ratings of those movies, made by twenty-six real people. The data is from a research group called MovieLens.org. The data is anonymized so the names here are fake. Also, the original ratings were on a 1-5 star scale; we simply replaced any rating with +1 or -1 depending on whether it was higher or lower than the average rating made by that person over the 100 movies in our set.
- How would you extend this algorithm from this exam to handle ratings on a 5-star scale, as in the original MovieLens data?
- If you were to use the algorithm implemented in this exam to predict each of the 1299 ratings in the file **`ratings.txt`** and then compare each prediction against the *actual* rating, the algorithm would get it right about 75% of the time. That is pretty good, considering that random guessing would only get it right 50% of the time; and people are not all that consistent with each other (otherwise 100% would be easy).
- If you had even more data (say thousands of people instead of just twenty-six, and/or thousands of movies instead of just 100), would you expect this to work even better?
- Can you think of an algorithm that would work even better for predicting people's movie preferences based on those of other people?
- Netflix ran a competition for the best algorithm for this kind of prediction and awarded $1M to the winner in 2009.