CSE 373, Winter 2013
Homework Assignment #1: Stable Marriage (25 points)

Due Friday, January 18, 2013, 11:30 PM

This program focuses on using the Java collections framework. Turn in a file named `MatchMaker.java` from the Homework section of the course web site. You will also need to download the support file `Main.java` and various input text files from the Homework section of the course web site; place it in your Java project and add to it. If you are using Eclipse, place .java files in your project's `src/` folder and .txt files in the project root folder (the parent of `src/`). We also provide a 'stub' version of `MatchMaker.java` on the web site as a starting point if you like.

Program Description:

The "stable marriage" problem is a classic computer science problem of matching men and women into married couples. The problem examines a set of input data for an equal number of men and women, where each person has an ordered queue of preferences for whom that person would like to marry. Your task is to pair up the men with the women in such a way that no one is unsatisfied with their partner. You will use a particular algorithm to pair up ("engage") men to women and repeatedly improve the pairings as necessary until every person is satisfied.

Consider an engagement (M, W) between a man M and woman W. We say that (M, W) is *unstable* if either M or W has another person they would rather marry (and who would rather marry them as well). In other words, an unstable engagement meets *one or both* of the following conditions:

- There is another woman W_2 such that M and W_2 both prefer each other to their current partners; that is, M prefers the pair (M, W_2) to (M, W) and W_2 is either single or is engaged to a man she likes less than M; **or,**
- There is another man M_2 such that M_2 and W both prefer each other to their current partners; that is, W prefers the pair (M_2, W) to (M, W) and M_2 is either single or is engaged to a woman he likes less than W.

Gale-Shapley Algorithm for Stable Marriages:

In 1962 David Gale and Lloyd Shapley (mathematics/economics professors from Berkeley and UCLA) created an algorithm for computing a set of stable marriages for any *N* men's and women's preferences. This Gale-Shapley algorithm guarantees that within *N* passes over the data you will find a way to marry everyone with no unstable pairings.

The input into the algorithm is a collection of men and a collection of women, where each person carries an associated collection of preferences of people to marry, ordered in descending order from most to least desirable.

```
    MEN   = collection of all men.   (initially all are single)
    WOMEN = collection of all women. (initially all are single)
```

The algorithm consists of a series of match-making "rounds" in which pairs become engaged. A round is considered "stable" if there are no single men left and if no one changes their partner during the entire round. Generally match-making rounds are performed until a round with no changes, at which point the solution is stable and the algorithm stops.

```
Algorithm for One Match-Making Round:
for each man M in MEN:
    if M is single and still has at least one woman left to consider:
        let W = remove M's most-preferred remaining woman from consideration.
        if W is single:
            (M, W) become engaged.
        else, W must be currently engaged to some other man M2.
            if W prefers M over M2:
                (M, W) become engaged.
                M2 becomes single.
```

Consider the input data on the next page. The men are listed first, then the women. Each line has a person's name and gender, then a list of whom that person would like to marry, from most to least. For example, Jerry would most like to marry Miranda and would least like to marry Charlotte. Miranda desires Newman and would be least happy with George.

```
George:M:Charlotte,Carrie,Miranda,Samantha
Jerry:M:Miranda,Samantha,Carrie,Charlotte
Kramer:M:Samantha,Charlotte,Carrie,Miranda
Newman:M:Samantha,Charlotte,Miranda,Carrie

Carrie:F:Newman,Jerry,Kramer,George
Charlotte:F:Jerry,George,Kramer,Newman
Miranda:F:Newman,Jerry,Kramer,George
Samantha:F:Jerry,Kramer,George,Newman
```

If you run the Gale-Shapley algorithm on the above data, the algorithm produces the following results:

Round 1:
- George proposes to Charlotte, who was single, so she accepts.
- Jerry proposes to Miranda, who was single, so she accepts.
- Kramer proposes to Samantha, who was single, so she accepts.
- Newman tries to pursue Samantha, but she's engaged to Kramer and prefers Kramer, so nothing changes.

  Couples so far: (George, Charlotte), (Jerry, Miranda), (Kramer, Samantha).

Round 2:
- George is already engaged, so he does nothing.
- Jerry is already engaged, so he does nothing.
- Kramer is already engaged, so he does nothing.
- Newman tries to pursue Charlotte, but she's engaged to George and prefers George, so nothing changes.

  Couples so far: (George, Charlotte), (Jerry, Miranda), (Kramer, Samantha).

(Though no couples change here, the algorithm is not stable yet because Newman and Carrie are single.)

Round 3:
- George is already engaged, so he does nothing.
- Jerry is already engaged, so he does nothing.
- Kramer is already engaged, so he does nothing.
- Newman tries to pursue Miranda; she's engaged to Jerry but prefers Newman. Jerry gets dumped.

  Couples so far: (George, Charlotte), (Kramer, Samantha), (Newman, Miranda)

Round 4:
- George is already engaged, so he does nothing.
- Jerry tries to pursue Samantha; she's engaged to Kramer but prefers Jerry. Kramer gets dumped.
- Kramer tries to pursue Charlotte, but she's engaged to George and prefers him, so nothing changes.
- Newman is already engaged, so he does nothing.

  Couples so far: (George, Charlotte), (Jerry, Samantha), (Newman, Miranda)

Round 5:
- George is already engaged, so he does nothing.
- Jerry is already engaged, so he does nothing.
- Kramer proposes to Carrie, who was single, so she accepts.
- Newman is already engaged, so he does nothing.

  Couples so far: (George, Charlotte), (Jerry, Samantha), (Kramer, Carrie), (Newman, Miranda)

Round 6:
- Nothing changes; the algorithm has stabilized.

Provided Files:

You are given one file to use to help you write this assignment:

- `Main.java`: The client program to run the stable marriage simulation. Performs all file I/O. Do not modify.
- `MatchMaker.java`: An incomplete "stub" version that declares all the method headers. Use as a starting point.

A set of provided data files to use as input for the algorithm is also posted on the Homework section of the web site. We will not provide testing code for you other than the `Main.java` program, but we do not guarantee that the main program is an exhaustive test. You should perform additional testing of your own before you submit your program.

MatchMaker Implementation Details:

For this assignment you are to write a class called `MatchMaker` that performs the Gale-Shapley algorithm on a given collection of men and collection of women. You must write the following methods with exactly the headers shown, so they can be called by the provided `Main` class. You **may define additional methods** in your class if you like.

`public MatchMaker()`
In this method you should initialize a new match maker object. Initially there are no men or women.

`public void addPerson(String name, String gender, String[] partners)`
In this method you will add a single person to your match maker. This method is called by the `Main` class, once for each line in the input file, before the actual rounds of match-making begin. The gender will be `"M"` or `"F"`. The `partners` array lists the names of all possible partners for this person, in decreasing order of desirability. For example, given the Seinfeld input data on the previous page, the `Main` class might make the following call on your match maker:

```
    maker.addPerson("George", "M", {"Charlotte", "Carrie", "Miranda", "Samantha"});
```

`public int getRank(String name, String partner)`
In this method you should return the desirability of `partner` in the eyes of `name`. The most desirable person should be rank 1, the second-most should be rank 2, and so on. These rankings come from the orderings present in the arrays that were passed when each person was added. For example, the call of `maker.getRank("George", "Charlotte")` would return 1. The order of the parameters does matter; the call of `maker.getRank("Charlotte", "George")` would return 2 because George is second on Charlotte's preferences list.

`public boolean isStable()`
In this method you should return true if the `MatchMaker` is in a stable state; that is, if the Gale-Shapley algorithm has finished running to produce a stable set of engagements between the men and women. When the `MatchMaker` is initially created, it is not stable, so this method should return `false`. If `nextRound` is called and the round is a "stable" round as defined previously, you should remember this and return `true` for subsequent calls to `isStable`.

`public void nextRound()`
In this method you should perform a single round of the Gale-Shapley algorithm as previously described to match men and women. You should examine the men *in alphabetical order*, regardless of the order in which they were added previously. As you perform each round you should also make note of whether the round was "stable" as previously defined, because you will need to report this information from your `isStable` method. As the round is running, each time any man proposes to a woman and she accepts, print a line of output to `System.out` in the following format:

```
George proposes to Charlotte
```

You may assume that `addPerson` will not be called again after `nextRound` is called.

`public void printMatches()`
In this method you should print output to `System.out` representing the current state of the men and women and their engagements, with each man *in alphabetical order* on one line followed by each woman *in alphabetical order* on one line. For each person, print their name, followed by a colon (`:`), followed by either "single" if they are not engaged, or "engaged to *name* (rank *rank*)" if they are engaged. The rank to show is the person's rank toward their partner; for example, "Charlotte: engaged to George (rank 2)" indicates that George is Charlotte's second choice.

For example, after one round using the Seinfeld example data on the previous page, the output would be:

```
George: engaged to Charlotte (rank 1)
Jerry: engaged to Miranda (rank 1)
Kramer: engaged to Samantha (rank 1)
Newman: single
Carrie: single
Charlotte: engaged to George (rank 2)
Miranda: engaged to Jerry (rank 2)
Samantha: engaged to Kramer (rank 2)
```

Calling `printMatches` does not perform a match round or otherwise change the state of the `MatchMaker` object. Do not assume that `printMatches` is called exactly once per call of `nextRound`, nor always called after `nextRound`.

For all methods, **you may assume that the parameters passed are valid** in all aspects. No parameter will be `null` or empty; duplicates will not be passed to `addPerson`; the gender will be `"M"` or `"F"`; the partners array will contain every name of the opposite sex; the names passed to `getRank` will be ones of opposite genders added through `addPerson`.

Designing Your MatchMaker Class; Choosing Collections to Use:

At various points during the program, you will find yourself wanting information such as:

- What are the names of all people in the simulation? Of just the men? Of the women?
- Is George single or engaged? If engaged, to whom?
- What is the next person Jerry would like to pursue? Does Kramer have any women left that he wants to pursue?
- What is Newman's preference rank for Miranda?

A big part of this assignment is using **Java's collection classes** to solve a tricky algorithmic problem, as well as being able to make intelligent choices about which collections are best for each part of the task. The methods you are required to write for class `MatchMaker` do not outwardly accept any collections as parameters (except for an array in `addPerson`) nor return them, but you will want to use several collections as **private fields** inside your `MatchMaker` to solve the problem. Think about what kinds of collections would allow you to easily answer questions such as the ones in the list above. Here are some general guidelines you should follow in choosing your collections if you want full credit:

- Favor Java collections over arrays throughout your code. `addPerson` passes an array, but don't use any others.
- In the absence of other constraints, favor the collection that implements the needed behavior most efficiently.
- When a collection cannot contain duplicates, favor a set over a list.
- If elements are to be removed from the collection in order of insertion until the collection is empty, favor a queue. If they are to be removed in the opposite order or reversed, favor a stack.
- When pairs of pieces of data are related to each other, use a map.
- When items of a map or set should be processed in a sorted order, favor a tree-based collection implementation. When order is irrelevant, favor a hash-based implementation.
- You should not need to loop over a collection to search it for a value (or call `indexOf` on a list). If you find yourself doing so, you should probably have chosen a different collection, such as a map/set rather than a list.
- You should not need to explicitly sort any collection after it has been filled with data. If you find yourself doing so, you probably should have chosen a different collection that already stores its elements in sorted order.
- You should not need to re-compute a complex value that you have previously computed. If you find yourself doing so, you probably should have saved/remembered the value in a variable or as an element of a collection.

You will most likely need to use one or more **compound collections**; that is, a collection of collections, such as a list of lists, or a map of sets, a map of maps, etc. *Hint:* Our own sample solution uses five total collections, two of which are compound collections. In total, our code makes use of the `List`, `Queue`, `Set`, and `Map` collection types.

Optional Person Class:

Your program keeps track of various data about each person: their name, gender, whether they are engaged (to whom), their preferences and rankings of partners, etc. If you would like to create a Java class to represent each person, you may optionally write a **`Person` class** stored in a file `Person.java` and turn it in with your `MatchMaker.java`. You should not add any other classes besides `Person` to this program. The exact contents of the `Person` class are up to you, but your overall program is still subject to the style guidelines specified in this document, either way.

Style Guidelines and Grading:

We will grade your program's behavior and output to give you an **External Correctness** score. Part of this comes from correctly following the Gale-Shapley algorithm as described previously. Your output must match ours exactly for the same input data, including console output formatting. Use the Output Comparison Tool on the class web page to be sure.

We will grade your program's code quality (**Internal Correctness**). Part comes from choosing appropriate collections to store your data. There are also many general standards Java coding style that you should follow, such as naming, indentation, comments, avoiding redundancy, etc. For a list of these standards, please see the **Style Guide document** posted on the class web site. You should follow all of its guidelines as appropriate on this and future assignments.