Lab 2

Due 11:59 pm, 21 February

Handout 3
CSCI 136: Spring, 2005
14 February

Housing Draw

1 Warm-up Exercises

There is nothing to hand in for this part, but make sure you understand these examples.

1. What is printed by the following program?

```
class Example {
    public int num;

    public Example(int initial) {
        num = initial;
    }

    public int getNum() {
        num = num + 1;
        return num;
    }

    public static void main(String args[]) {
        Example first = new Example(10);
        Example second = new Example(20);
        System.out.println(first.getNum());
        System.out.println(second.getNum());
    }
}
```

   What would be printed if we changed the declaration of `num` to be

```
    public static int num;
```

   What does this tell you about `static` fields?

2. Browse the javadoc documentation available from the class web page. Look up the `structure` package's `Vector` class. Become familiar with how to look up information on these webpages.

2 Lab Program

This week, you will implement a computerized housing draw system for Williams College. While not as exciting as the big night itself, your program will be able to read in information about dormitories and draw groups and then assign students to dormitories based on their preferences and lottery numbers.

Here is the basic strategy: Each residence on campus has a certain capacity. Students form groups of up to four people. The group decides on a list of preferences, ranked in order from best to worst. The list may only be one or two residences, or a complete list ordering all of the dorms. The group is assigned a lottery number between 1 and 2000. In order of lottery number, groups are assigned to the residence highest in the preference list that has enough space to fit the entire group. If no preferred

residence has enough space, the students in the group are left unassigned. After all the groups have been processed, a second pass is made over them. All unassigned students are then placed in any open dormitory spaces (exactly how this happens is up to you).

Since this is the first lab with a complex data structure, we will give you parts of the class declarations for your data structures. You'll need to fill in the details and add constructors and methods to complete the assignment.

```
class Student {
    protected String name;
    protected int id;
    protected Residence assignedTo;
}

class Residence {
    protected String name;
    protected int capacity;
    protected Vector students;
    protected int lotteryCutoff;
}
```

The `Student` class records information about a Williams student. The `assignedTo` field can simply be `null` to indicate that a student has not been assigned yet. The `Residence` class records information about a dormitory. The `capacity` is the number of students that can fit in that residence. The `students` vector should begin empty and will be filled in as students are assigned to the dormitory. As students are assigned, the `lotteryCutoff` instance variable records the highest (worst) lottery number among the assign students.

Note that when a student is assigned to a residence, a connection is made in both directions. The student has a reference to his or her residence, and the residence keeps a list of all students assigned to it. Both directions are needed to implement the program easily. For example, without the student knowing about the residence, we would have to search through all the residences to find where a specific student lived. While this information is redundant, it improves the efficiency and simplifies the overall implementation.

There is also a class to represent draw groups:

```
class Group {
    protected Vector members;      // members is a list of Student objects
    protected Vector preferences;  // preferences is a list of String objects
    protected int lotteryNumber;
    protected boolean liveAnywhere;
}
```

The `members` vector stores the student records for group members (i.e., members is a vector of `Student` objects). The `preferences` vector stores a list of residence names from most to least preferred. The residence names are stored as `String`s. The `liveAnywhere` flag indicates whether this group will accept any assignment, and not just those on their list. The lottery number should be filled in during the draw process.

Finally, there is the class that manages the housing database and runs the lottery. This class should also contain the `main` method for the program:

```
class HousingDraw {

    protected Residence dorms[];
    protected Group groups[];

    public static void main(String args[]) {
        HousingDraw draw = new HousingDraw(...);
```

```
        draw.makeAssignments(...);
        draw.commandLoop(...);
    }
}
```

The number of dorms and groups will be known when those structures are created, and dorms and groups will not be added or removed, so we shall use arrays rather than vectors.

Each class should be put in a java file with the appropriate name (ie, `HousingDraw.java`). There are many other ways to define these data structures. Some of our Vectors could be arrays, and vice-versa. Our specific choices are partially motivated to give you practice with arrays and vectors in a variety of situations.

2.1 Data Files

The `HousingDraw` class will read the student group information and residence data out of two data files.

Residence file The residence file contains information about residences. The first line contains the number of residences, and each successive line contains the size and name of each dorm (names will not contain any spaces).

```
35
19 Agard
78 Armstrong
17 Brooks
62 Bryant
70 Carter
...
```

You can read this data using the `ReadStream` class, which is described in the appendix of the book. The most relevant pieces for this lab are the following:

```
class ReadStream {
    // create a new ReadStream for the terminal
    public ReadStream()

    // create a ReadStream for a specific file
    public ReadStream(String filename)

    // read an integer
    public int readInt()

    // read a string (deliminated by whitespace)
    public String readString()

    // read all the characters up to the next new line
    public String readLine()

    // return true if there is no more input
    public boolean eof()

    // skip over any white space.
    public void skipWhite()
}
```

Groups file This file stores information about student groups and preferences. The file is organized as follows. You may assume it contains no errors in format:

```
521                                    number of groups
2                                      size of group 1
2736644 Stephen Freund                 ID and name of first student
9294133 Duane Bailey                   ID and name of second student
4 TylerAnnex Spencer Doughty Agard no  number of choices, followed by choices
3                                      size of group 2
6227775 Kristof Redei                  ID and name of first student
2802227 Jared Strait                   ID and name of second student
1242141 Edmund Rucci                   ID and name of third student
2 Rectory Prospect yes                 number of choices, followed by choices
...
```

The first line indicates the number of groups in the file. Each group is stored as:

1. a number indicating how many students are in the group.
2. a line containing the ID and name for each student.
3. a number indicating how many preferences are listed, followed by those preferences (all on the same line). The last entry on that line will always be "yes" or "no" to indicate the groups willingness to be assigned anywhere.

As above, you can use a `ReadStream` to read the data from the file. Use `readInt` and `readString` to read numbers, ID's, and residence names. Since student names will have spaces in them, we need a different tactic. After reading the ID for a student with `readInt`, you can call `skipWhite` to skip the space before the name starts and then `readLine` to read the rest of the text from there to the end of the line. Just store the whole string as the name– do not worry about taking apart first and last names.

We give you two sets of files— large files for most of Williams, and small files for use while developing your programs.

2.2 Generating Random Lottery Numbers

One of your tasks is to assign lottery numbers to draw groups. Each group is assigned a random number between 1 and 2000. Generating these numbers is a fairly straightforward use of a random number generator, but the constraint that each number must only be used once adds a bit of a twist. One way to make sure each number is only used once is to generate an array of all the possible values (kind of like printing a set of numbered tickets to put in a bin). Randomly shuffle the array elements. When you need a number, use the first one in the array, and then on subsequent draws, use the second, then the third, and so on. This strategy will randomly cycle through all the values using each exactly once.

Here is a simple shuffling algorithm given in pseudo-code:

```
for (int i = 0; i < number of array elements; i++)
    swap elements array[i] and array[random index between i and end of array]
```

2.3 Making Assignments

Lower lottery numbers are "better" so groups that have lower numbers are assigned before those with higher numbers. To facilitate processing groups in the proper order, it's easiest to first sort the entire draw group array. Rearrange the elements so that the lowest-numbered group is first, followed by the next lowest, and so on up to the highest numbered group at the end. If your knowledge of sorting from

134 is rusty, the explanation of the simple Selection Sort algorithm on page 110 of the book should serve as a good refresher. Once you have the draw groups in order, you are ready to start making assignments.

The assignment process operates in two passes. In the first pass, you attempt to assign entire groups without splitting to a residence on their preference list. In the second pass, you re-examine groups that failed to be assigned in the first round but that indicated a willingness to be assigned anywhere and put them (splitting the group if needed) in any remaining spaces.

In the first pass of assignments, iterate through the groups in order of lottery number from 1 to 2000. For each group, examine their list of preferences to find the highest ranked choice that has enough space to accommodate all group members without splitting. For example, consider a group of three members with a preference list of West, Prospect, and Fitch. If their first choice, West, only has two spaces remaining, the entire group won't fit, so we move down and try Prospect. If Prospect has twenty open spaces, then three of those spots will be assigned to the group members. If none of the choices has three spaces open, this group would remain unassigned in the first pass.

In the second pass of assignments, go back through the groups (again in lottery number order) and look for students that didn't get assigned in the first round. You may skip any groups that were not willing to live anywhere. Assign the students to residences that still have space (splitting groups as needed). You can search the residences in any order to find remaining spaces. You may assume that there will always be enough space to put all students somewhere.

2.4 The Interactive Query

Once all the assignments have been made, your program should go into an interactive loop that allows the user to query the database to view the results. You should prompt the user with options for printing the list of students assigned to a residence, finding the assignment for a student, and showing the list of cutoffs for all residences. Below is a sample of the interaction:

```
Choices:
  1  Print residence
  2  Print students
  3  Print cutoffs
  4  quit
Enter Command: 1

Enter Residence: West

West has 43/45 spaces allocated.
 7059781  Edmund Rucci
 1901705  Kristof Redei
 3302888  Stephen Freund
      ...

Choices:
  1  Print residence
  2  Print students
  3  Print cutoffs
  4  quit
Enter Command: 2

Enter student name: stephen
 4400830  Stephen Abbot is assigned to Pratt
 2981898  Stephen Freund is assigned to West
```

```
 2981898  Zach Stephens is assigned to Milham

Choices:
  1  Print residence
  2  Print students
  3  Print cutoffs
  4  quit
Enter Command: 3

381 Rectory
450 Milham
583 Sewall
688 Lambert
...
```

When finding a residence by name for the first option, the name given by the user must exactly match (including case) one of the residence names in the database. (This search by name is similar to one you needed earlier- write it once as a helper method to be used in both places!) The list of resident names is required to be printed in alphabetical order. Alphabetizing by first name, as we have shown above, is fine. Probably the simplest way to handle this requirement is to maintain the students in alphabetical order when adding them to the residences.

When finding a student by name for the second option, any student name that contains the string entered by the user should be considered a match. Searching for "St" would print the entries for "Stephen Freund" as well as "Justin Moore". This search should also be case-insensitive. You can print the list of matching students in any order that is convenient.

For the third option, the list of residences should be printed in order of cutoff number from low to high. The easiest way to support this functionality is to sort the residences by cutoff after all of the assignments have been made.

Take care that you gracefully handle malformed input on the part of the user (e.g. choosing an option that doesn't exist, a name that has no matches, and so on).

2.5 Suggested Task Breakdown

Complex programs are always easier to develop if you evolve the program incrementally. Add one feature at a time, test and debug it, and only then move on. We suggest that you write the program in stages, as sketched below:

1. *Understand the data structure.* Carefully examine the data structure definitions. Be sure you understand what each of the fields is for, what methods you use to access them, and how to update them as you read the data files and make assignments. Here are a few examples to think about:

   - How do I find the residence with a certain name?
   - Is there enough space in a specific residence for a group of size 3?
   - How many members does a specific draw group have?
   - What are the names of the students assigned to a residence?
   - Has a student already been assigned to a residence?

   You might want to try writing some expressions to obtain information like that listed above out of the data structure before you start programming. Drawing diagrams of the various objects and relationships may also be helpful.

2. *Read residence data.* First just read a single residence's data and configure its record. Once that's working, extend the program to read the entire file's worth of residences into the database. Add some temporary println statements to print out the data to confirm all is well before going on.

3. *Read group/student data.* Read the first draw group's information from the file and set up the group and student. Again, use print statements to verify that you've got this part under control so you know you are ready to tackle the next step.

4. *Assign lottery numbers and sort draw groups array by number.* Each draw group is given a unique, random number between 1 and 2000. Make sure that the numbers are being chosen randomly and without repetition. Once all groups have a number, re-arrange the array of groups into order by lottery number to facilitate processing. Print the results to make sure all is well before you move on.

5. *Make assignments, first pass.* Starting by assigning just one group. Find the highest-ranked choice in their list that can accommodate all members and make the assignment. Remember that the connection needs to be set up in both directions (the house has a list of residents, the student has a pointer to their house). Once you can assign a single group, assigning all groups is just a loop.

6. *Make assignments, second pass.* On the second pass, only consider those groups that didn't get assigned in the first pass. Those groups can be split up as needed to fit in whatever space is available in any of the residences.

7. *Interactive query.* Once you can have the assignments made, set up the loop that interacts with the user and lets them view the results.

2.6 A Few Random Notes and Suggestions

1. You may find it helpful to do most of your development without randomization, for example, just assign lottery numbers to groups in sequential order as you read them from the file. This will give you reproducible results from run to run which can be quite helpful for debugging. Turn on randomization once you know the basic algorithms are working.

2. Test on the small data files first, since it is easier to hand-examine their limited output and work you way up to the large one.

3. Error-checking is an important part of handling user input. Where you prompt the user for input, such as asking for file to open or a house to show the results for, you should validate that the response is reasonable and if not, ask for another response.

2.7 Getting Started

Download the "lab2.tar.gz" starter file from the course handouts web page. This should decompress automatically and store a "lab2" folder in your directory. If it does not, you can double-click on lab2.tar.gz to decompress the archive, or run `gunzip lab2.tar.gz` and then `tar -xf lab2.tar` from a Unix window when in the directory containing the downloaded file. Move the "lab2" folder into your "cs136" folder.

The starter files contain two sets of data files— large data files for the Williams residences and about 1500 students, and small data files for use while developing your programs.

2.8 Thought Questions

Answer the following in a comment at the top of HousingDraw.java:

1. Bailey, 3.1 on page 65

2. Bailey, 3.6 on page 65 (don't write the class— just answer what the advantage would be)

3. Suppose you wanted to gender balance each residence so that no more than 50% of the capacity could be female or male. Describe how you would implement this feature. What changes to the data files, data structures, and algorithms would be required?

2.9 Deliverables

Turn in each of your Java files with the "`turnin -c 136 File.java`" command as you did last week.

Be sure to clean up and document your source code. Style counts as much (and sometimes more) than correctness.