Project 1: Hosn's Housing Prototype

DUE: Friday, Feb 21th at 11:59pm

Draft submission due Feb 6

Extra Credit Available for Early Submissions

Basic Procedures

You must:

- Fill out a `readme.txt` file with your information (goes in your user folder, an example is provided)
- Have a style (indentation, good variable names, etc.)
- Comment your code well in JavaDoc style (no need to overdo it, just do it well)
- Have code that compiles with the command: `javac *.java` in your user directory
- Have code that runs with the command: `java Setup [inputFileName]`

You may:

- Add additional methods and variables, however these **must be private**.

You may NOT:

- Make your program part of a package.
- Add additional public methods or variables
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no `ArrayList`, `LinkedList`, `HashSet`, etc.).
- Use any arrays anywhere in your program (except the data field provided in the `MyArray` class)
- Alter any method signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Alter provided classes/methods that are complete (`Setup`, `MyArray.toString()`, etc.).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Add any additional libraries/packages which require downloading from the internet.

First Steps

- Download the `project1.zip` and unzip it. This will create a folder `section-yourGMUUserName-p1`
- Rename the folder replacing `section` with the `006`
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address
- After renaming, your folder should be named something like: `006-jsmith-p1`
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

Draft Submission:

- On or before Feb. 6 submit `MyArray.java` on blackboard

Final Submission:

- Make backup copies of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip `section-username-p1.zip` (no other type of archive) following the same rules for `section` and `username` as described above.
  - The submitted file should look something like this:

```
006-jsmith-p1.zip --> 006-jsmith-p1 --> JavaFile1.java
                                        JavaFile2.java
                                        ...
```

- Submit to blackboard.
- It is each student's responsibility to make sure that they submit the right files. After final submissions, students should download the submitted files from blackboard, unzip them, and double check.

<u>Grading Rubric</u>

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Overview

You have been hired by a company named Hosn's Housing Analytics to create a prototype that will display information about housing data[1]. In this prototype you will use dynamic array data structures to store the data and compute metrics. For the requirements of the prototype, your client has given you a list of metrics and operations. The user interface will be a menu that displays this list and prompts the user to choose which option to execute.

You will create two classes. First you will create your own dynamic array class called `MyArray`. Your `MyArray` class will contain a simple Java array called `data`. You will write methods such that `MyArray` has the functionality of a dynamic array. Next you will create a class called `Menu`. `Menu` will use your `MyArray` class. The user interface is a menu that displays a list and prompts the user to choose an option. You will write methods to execute each of the menu options. The methods in the `Menu` class will call the methods in your `MyArray` class to create dynamic arrays and compute metrics. The template for the `Menu` class has a stub method for each menu option. Your work is to implement these stub methods. The program template compiles with the stub methods. As you work through the project, compile often and make sure that your code doesn't get too many bugs at once. A third class called `SetUp` is given to you, you do not need to change anything in `SetUp`.

Requirements

An overview of the requirements are listed below, please see the grading rubric for more details.

- **<u>Implementing the classes (75 pts)</u>** - You will need to implement required classes and provided template files.
- **<u>Big-O (10 pts)</u>** - The template files provided to you contain instructions on the REQUIRED Big-O runtime for many methods. Your implementation of those methods should NOT have a higher Big-O.
- **<u>Style (5 pts)</u>** – You must follow the coding and conventions specified by the style checker.
- **<u>JavaDocs (10 pts)</u>** - You are required to write Java Doc comments for all the required classes and methods.

<u>JavaDocs (10 pts) and Style Checker (5 pts)</u>

You need to document your code correctly and conform to a given code style (many companies have their own style requirements, but there are also some standard ones like Sun and Google). We'll be using `checkstyle` (https://checkstyle.org) which is tool to automate style checking. It has a command line interface (CLI) which has been provided to you. See the "Testing" and "Command Reference" section for more details. It also has plugins for Eclipse, NetBeans, jGRASP, and many others (see their website).

[1] https://www.census.gov/programs-surveys/ahs/data/2017.html
The data is a subset of the American Housing Survey conducted by the US Census Bureau and is available for public use

Implementation/Classes

This project will be built using a number of classes representing the component pieces of the project. Here we provide a description of these classes. Template files are provided for each class in the project folder and these contain further comments and additional details.

MyArray<T> (MyArray.java): This class represents a generic collection of objects in an array whose capacity can grow and shrink as needed. It supports the usual operations of adding and removing objects including **`add()`** and **`delete()`**. Check the included template file for all required methods and their corresponding big-O requirements. The capacity of a `MyArray` may change as items are added or removed and the following rules must be followed in your implementation:

- The default initial capacity is fixed to be 2 if not specified.
- When you need to add an item and there is not enough space, grow the array to double its capacity.
- When you delete an item and the size falls below 1/4 of the capacity, shrink the array to half its capacity.

Menu (Menu.java): This class implements the menu. It must use `MyArray` objects for storage and you are not permitted to use any arrays anywhere in this file (the main method which interacts with args[] has already been implemented). You will be implementing a typical "menu loop" structure:

- `init()` – this starts the program, reads in the raw data from the input text file, and creates a simple array of survey record objects
- `step()` – this contains a loop that displays the menu and executes user options
- `finish()` – this ends the program

Check the included template file for all required methods. Below is the menu followed by additional details where needed. Note that some of the menu options depend on earlier options above. When this is the case, your code needs to display an error message instructing the user to first execute the correct option above.

Here is the menu:

```
 1. Load objects into an array called myArray1
 2. Print the first 5 records
 3. Create an array called myArray2 with 5 specified records
 4. In myArray2, insert record 11008724 at index 3
 5. In myArray2, append record 11063694
 6. In myArray2, delete record 11011113
 7. In myArray2, swap record 11069423 with record 11067039
 8. Compute how many units there are for each value of totRooms
 9. Compute a count of bike records
10. Create an array containing arrays by decade
11. Create an array of bike record counts by decade
12. Create an array of condo units
13. Compute the average number of people per condo
14. Sort myArray1 by control number
15. Exit
```

Here are additional details where needed:

1. `Load objects into an array called myArray1`
   - Load a plain Java array of survey record objects, *plainArray*, into a dynamic array called *myArray1*
   - Test: the size of *myArray1* = 16688

2. `Print the first 5 records`

3. `Create an array called myArray2 with 5 specified records`
   - Create a dynamic array called *myArray2* with the following control numbers:
     `11029175, 11069423, 11011113, 11013828, 11067039`
   - Print the final array to the screen

4. `In myArray2, insert record 11008724 at index 3`
   - Print the final array to the screen

5. `In myArray2, append record 11063694`
   - Print the final array to the screen

6. `In myArray2, delete record 11011113`
   - Print the final array to the screen

7. `In myArray2, swap record 11069423 with record 11067039`
   - Print the final array to the screen

8. `Compute how many units there are for each value of totRooms`
   - Create a dynamic array called *arrayTotRooms* where size = 28, index values will be 0 - 27
   - Each element will store the count of records where *totRooms* equals the index value
   - For example, if the number of records where *totRooms* = 8 is six, then *arrayTotRooms[8]* = 6
   - For categories 1 – 26, compute the average total rooms across all units and store the result in *arrayTotRooms[0]*
   - Test: *arrayTotRooms[0* should round to 5.57

9. `Compute a count of bike records`
   - Create an int called *bikeCount* that stores the number of records where bike = '1'
   - Test for *myArray1*, *bikeCount* = 75

10. `Create an array containing arrays by decade`
    - Create an array of arrays called *arraysByDecade* using the variable *yrBuilt*
    - There will be eleven sub-arrays, one for each value in *yrBuilt*
    - Test: the size and capacity of the sub-array for 1970 is 2553 and 4096 respectively
    - Test: the sum of the sizes of the sub-arrays should match the size of *myArray1*

11. `Create an array of bike record counts by decade`
    - Compute *bikeCount* for each sub-array in *arraysByDecade*
    - Store the counts in an array called *arrayBikeCounts*
    - Test: make sure the total across all the sub-arrays matches the result for option 8 above

12. `Create an array of condo units`
    - Create a dynamic array called *arrayCondoYes* that stores the number of records where condo = ‘1’
    - Clone myArray1 and shrink it
    - Test: the size of *arrayCondoYes* = 1385, the capacity = 4172

13. `Compute the average number of people per condo`
    - Use *arrayCondoYes* and the *numPeople* to compute a double called *avgPersonsPerCondo*
    - Test: *avgPersonsPerCondo* should round to 1.6

14. `Sort myArray1 by control number`
    - Sort in ascending order, from smallest to largest

15. `Exit`

SetUp (SetUp.java): This class calls `Menu` and is fully provided (do not edit).

Input Data Format(s)

The input file is a CSV formatted text file. An input file has been provided (`inputFile.txt`) The table below provides a description of the variables you will use. (There are more variables in the data set, but you do not need to use those.)

Variable Name	Description	Values
control	A unique 8 digit number for each record	10000001 - 21099999
yrBuilt	The year in which the unit was built, each year represents a decade or the range shown	2010 = 2010 to 2017
		2000 = 2000 to 2009
		1990 = 1990 to 1999
		1980 = 1980 to 1989
		1970 = 1970 to 1979
		1960 = 1960 to 1969
		1950 = 1950 to 1959
		1940 = 1940 to 1949
		1930 = 1930 to 1939
		1920 = 1920 to 1929
		1919 = 1919 or earlier
numPeople	Number of persons living in this unit	1 - 29 = The number of people
		30 = 30 or more people
		All other values = Not applicable
totRooms	Number of rooms in the unit	1 - 26 = The number of rooms
		27 = 27 or more
bike	In a typical week do you bike from home all the way to work?	1 = Yes
		2 = No
		-9 = Not reported
		-6 = Not applicable
condo	Is this unit part of a condominium?	1 = Yes
		2 = No
		All other values = Not applicable

How To Handle a Multi-Week Project

There are three weeks to work on this project. Students are not likely to complete it in one week. If students procrastinate, they are not likely to be able to get their questions answered before the due date. Here is an example schedule:

- Step 1 (Understand the concepts, code `MyArray`): First week
    - Read the project specification, inspect the given template files, and get familiar with the menu options, start thinking of which methods the menu options could use
    - Review the `Scanner` class along with creating and using generic classes
    - Make sure you remember how to compile and run `javadocs`, `checkstyle`, and `junit` tests
    - Start coding `MyArray`
    - If you're struggling... ask questions at the end of class or go to office hours as soon as you can
    - Submit the progress you've made on `MyArray.java` by Feb. 6, 2020 (2 pts)

- Step 2 (Finish `MyArray`, code `Menu`): Second week
    - At this point you've learned about array lists in class, so complete `MyArray` and move on to the `Menu` class.
    - For `Menu`, complete the methods for the menu options in the menu order. The coding complexity increases down the menu options.
    - Plan out which object members you need to use to partition the data and make subset arrays, and how you want to store them

- Step 3 (Finish `Menu`): Third week
    - Finish up the menu options and test to verify your results
    - Extra credit: earn 1% for each 24 hours you submit early, up to 5%

Read the grading rubric for complete details. If you begin work promptly during the 3 week period, you'll have plenty of time to test and debug your implementation before the due date.

Testing

MyArray and Menu

The main methods provided in the template classes can be used to contain useful code to test your project as you work. You can use a command like "`java MyArray`" or "`java Menu`" to run the code in `main()`. You can edit these `main()` functions to perform additional testing. JUnit test cases will not be provided for these classes, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.

Menu and SetUp

The best way to start testing the menu is to run it with the following command:

```
java SetUp [inputFileName]
```

- `inputFileName` – a csv file containing the data

For example, from your user directory you can run:

```
        java SetUp inputFile.txt
```

Style Checking

The provided `cs310code.xml` checks for common coding convention mistakes (such as indentation and naming issues). The provided `cs310comments.xml` checks for JavaDoc issues and in-line comment indentation issues. You can use the following to check your style:

```
java -jar checkstyle.jar -c [style.xml file] [userDirectory]\*.java
```

For example, for a user directory 001-krusselc-p1 checking for JavaDoc mistakes I would use:

```
java -jar checkstyle.jar -c cs310comments.xml 001-krusselc-p1\*.java
```

Note: if you have a lot of messages from `checkstyle`, you can add the following to the end of the command to output it to a file called out.txt: `> out.txt`

Command Reference

All commands are from inside your user folder and assume you left the style items and dictionaries in an outer folder (as in you unzipped project1.zip into a single place as suggested).

From in your **user** directory:

Compile:	`javac *.java`
Run:	`java SetUp inFile.txt`
Compile JavaDocs:	`javadoc -private -d ../docs  *.java`

From ***above*** your user directory:

Style Checker:	`java -jar checkstyle.jar -c cs310code.xml [userDirectory]/*.java`
Comments Checker:	`java -jar checkstyle.jar -c cs310comments.xml [userDirectory]/*.java`
Compile Unit Tests:	`javac -cp .;junit-4.11.jar;[userDirectory] SetupTest.java`
Run Unit Tests:	`java -cp .;junit-4.11.jar;[userDirectory] SetupTest`

Make sure your files are located in the correct directory

Make sure you are executing the commands from the correct directory

Project 1: Grading Rubric

No Credit

- Non submitted assignments
- Assignments late by more than 24 hours
- Non compiling assignments
- Non-independent work
- "Hard coded" solutions
- Code that would win an obfuscated code competition with the rest of CS310 students

How will my assignment be graded?

- Grading will be divided into two portions:
  - Manual/Automatic Testing (75%): To assess the correctness of programs.
  - Manual Inspection (25%): A checklist of features your programs should exhibit. These comprise things that cannot be easily checked via unit tests such as good variable name selection, proper decomposition of a problem into multiple functions or cooperating objects, overall design elegance, and proper asymptotic run-time complexity. These features will be checked by graders and assigned credit based on level of compliance. See the remainder of this document for more information.
- You CANNOT get points (even style/manual-inspection points) for code that doesn't compile or for submitting just the files given to you. You CAN get manual inspection points for code that (a) compiles and (b) is an "honest attempt" at the assignment, but does not pass any unit tests.
- Extra credit for early submissions:
  - 1% extra credit rewarded for every 24 hours your submission made before the due time.
  - Up to 5% extra credit will be rewarded (no more).
  - Your latest submission before the due time will be used for grading and extra credit checking. You CANNOT choose which one counts.

Manual/Automated Testing Rubric

For this assignment a portion of the automated testing will be based on JUnit tests and a manual run of your program. The JUnit tests used for grading will NOT be provided for you (you need to test your own programs!), but the tests will be based on what has been specified in the project description and the comments in the code templates. A breakdown of the point allocations is given below:

2 pts	Submit *MyArray.java* on Feb. 6
38 pts	MyArray
35 pts	Menu

<u>Manual Code Inspection Rubric</u>

Up to 25 points can be earned for these categories:

Inspection Point	Points	High (all points)	Med (1/2 points)	Low (no points)
Big-O Requirements	10pts	The Big-O requirements for listed methods are met and/or exceeded. Code is very efficient.	The Big-O requirements are sometimes met and sometimes not. Code could be more efficient.	The Big-O requirements are never (or almost never) met. Code is extremely inefficient.
Code Reuse	5pts	Code is reused rather than copied.	There is some code reuse, but other code is copied and pasted which could have been reused.	There is no code reuse apparent.
JavaDocs	10pts	Code passes all checks for cs310comments.xml [and] The entire code base is well documented with meaningful comments in JavaDoc format. Each class, method, and field has a comment describing its purpose. Occasional in-method comments used for clarity.	Code passes all checks for cs310comments.xml [and] The code base has some comments, but is lacking comments on some classes/methods/fields or the comments given are mostly "translating" the code.	Code does not passes all checks for cs310comments.xml [and/or] The only documentation is what was in the template and/or documentation is missing from the code (e.g. taken out).

The points below are deducted from the total score

Inspection Point	Points	High (all points)	Med (1/2 points)	Low (no points)
Submission Format (Folder Structure)	5pts ("off the top")	Code is in a folder which in turn is in a zip file. Folder is correctly named.	Code is not directly in user folder, but in a sub-folder. Folder name is correct or close to correct.	Code is directly in the zip file (no folder) and/or folder name is incorrect.
Code Style Basics	5pts ("off the top")	Code passes all checks for cs310code.xml [and] Code has a set indentation and formatting style which is kept consistent throughout and code looks "well laid out".) [and] Code has good, meaningful variable, method, and class names.	Code passes all checks for cs310code.xml [but] Code looks "messy" and/or names often have single letter identifiers and/or incorrect/meaningless identifiers. (Note: i/j/k acceptable for indexes.)	Code does not pass cs310code.xml checks