CMSC131, Spring 2020, Final Exam
Deadline: Friday, May 15, 5:00 pm (No late deadline)

INSTRUCTIONS:

- Code distribution available in the **StudentsDistribution** folder, available in the same folder where this pdf is available.
- Import the above distribution as you usually import a project.
- Immediately after importing the code, verify you can submit (check the submit server submission). We will not accept projects submitted via e-mail.
- Grading: 132% release tests, 68% secret tests / style and requirements.
- There are no public tests.
- Student tests that you provide will not be graded.
- There is no late submission period; we will grade the highest scoring submission. **There are students that wait until exactly 5:00 pm to submit; please do not wait until 5:00 pm. Your submission might not be accepted as it takes time for the submission to be processed by the submit server.**
- **We expect you to submit often during the exam period.** This will allow you to have a backup in the submit server and to address compilation problems before the deadline. Network problems, Eclipse submission problems, or submit server overloading are not valid excuses for not submitting your work.
- You have 8 release tokens and you can release-test your work anytime.
- You can only post clarification questions in Piazza. Debugging questions, why is code not passing a test, are invalid questions to post in Piazza. Use the Piazza folder **finalquestions** to post your questions; we will use the folder **finalclarifications** to post clarifications (please, don't use this folder). Before posting a question, check the **finalclarifications** folder.
- **Posting of any kind of code in Piazza, during this assignment period, represents an academic integrity violation and will be reported as such.**
- You may not address questions other students post in Piazza (only TAs and instructors can address questions).
- You must work by yourself.
- You can use class resources (lecture notes, lecture/lab examples, videos, etc.), but no other resources (e.g., code from the web).
- Sharing of this assignment solutions represents an academic integrity violation and will be reported as such (even after the semester is over).
- Submissions will be checked with cheating detection software.
- **If you are an ADS student:** The provided time period takes into consideration your time allocation. If you need any other assistance, contact your instructor. Ignore this entry if you don't know what an ADS student is.

Specifications

For this assignment you will implement methods for the classes **MultipleChoice, ArrayUtilities**, **RecursionProblem, Refrigerator** and **Store**. You will find the classes in the **sysImplementation** package. A description of each method is provided below. A driver (that you can ignore if you know what to implement) and associated output is provided at the end. This driver is part of the code distribution. **The driver relies on a class called Toaster that we have implemented for you**. Regarding the code you need to implement:

- You don't need to add comments to your code, but you must have good variable names, indentation and you should avoid code duplication.
- At this point you may want to look at the classes you will find in **sysImplementation**. We have provided a shell for each method you need to implement. You can also see the instance variables associated with each class.
- During the implementation of the above classes, you can add instance variables, constants (static final) and private methods, except for the **RecursionProblem** class (see restrictions below).
- You can assume parameters are valid, unless we indicate otherwise (e.g., if the parameter is null throw …).
- Be careful with cutting and pasting text (e.g., from a Javadoc or the pdf file). We have seen instances where student code does not work in the submit server, but works in Eclipse and the problem has to do with character set problems.
- You must provide an implementation for every method described below, otherwise your code will not work in the submit server. If you don't know what to do for a method, leave the body empty (void method) and for a method returning a value, return any value that makes the code compile.

MultipleChoice Class Specification

This class provides multiple choice questions, where each method represents a question. To answer a question, replace the statement

throw new UnsupportedOperationException("Not Implemented");

with a return statement that returns a character. The following is an example of the kind of question we will provide and the answer you need to provide:

```
/* What we will provide */
public static char question1() {
        /* Question #1

           This course is:

           a. cmsc106
           b. cmsc131
           c. cmsc200
           d. None of the above


        */
        throw new UnsupportedOperationException("Not Implemented");
}
```

```
/* You will replace the throw statement with a return and a character */
public static char question1() {
        /* Question #1

           This course is:

           a. cmsc106
           b. cmsc131
           c. cmsc200
           d. None of the above


        */
        return 'b';
}
```

There is a release test for the multiple choice class. If you pass the release test, it does not mean you got the right answers; it means that you have provided an answer for each question ☹.

ArrayUtilities Class Specification

This class defines the methods:

1. **makeArrayCopy -** The method's prototype is provided below. The method returns a deep copy of the **data** array parameter. You can assume the array parameter will never be null, however, it can be empty. This method will not throw any exceptions.

   `public static int[] makeArrayCopy(int[] data)`

2. **getNumberRowsSetToNullOrEmpty -** The method's prototype is provided below. The method returns the **number** of rows in the **data** array parameter that are set to null or are empty (length of 0). The **data** array is a ragged array and **data** will never be null. This method will not throw any exceptions.

   `public static int getNumberRowsSetToNullOrEmpty(int[][] data)`

3. **getArrayCopyWithoutNullEmptyRows -** The method's prototype is provided below. The method returns an array with **deep** copies of rows from the **data** array parameter that are not null or that are **not** empty. The **data** array is a ragged array and **data** will never be null. This method will not throw any exceptions. Feel free to use the previous methods during the implementation of this **method**.

   `public static int[][] getArrayCopyWithoutNullEmptyRows(int[][] data)`

Recursion Problem Class Specification

The **RecursionProblem** class defines a single method called **removeDigits** whose prototype is provided below. The method is a recursive method that returns a string where all digits present in the **str** parameter have been removed. For this problem you can only use the following String class methods: isEmpty(), length(), charAt(), and substring(). You can use the Character.isDigit() method to determine whether a character is a digit. The parameter will never be null, but it can be the empty string. Your solution must be recursive, you may not use any iteration statement (for, while, do while), and you may not add instance variables nor static variables; local variables are fine. Do not use the words **for, while**, or **do** in your code at all (not even in comments or variable names (e.g., doYourWork)) as our tests might think you are using an iteration statement. You can use one auxiliary method if you understand it is necessary.

```
public static String removeDigits(String str)
```

Refrigerator Class Specification

The **Refrigerator** class represents a home refrigerator. A refrigerator has a brand (**brand** instance variable), a number of bins (**bins** instance variable), and items (**items** instance variable). In addition, a constant representing a default number of bins is defined. The declaration of each variable follows.

```
private String brand;
private int bins;
private StringBuffer items;
private static final int DEFAULT_BINS_NUMBER = 10;
```

The **Refrigerator** class implements two interfaces: **Appliance** and **Comparable**. The method associated with the **Appliance** interface is:

```
public String getBrand();
```

The **Refrigerator** class methods are:

1. **Constructor -** Takes a string (representing the brand) and an integer (representing the number of bins) as parameters, and initializes the corresponding instance variables, accordingly. It creates a StringBuffer object and assigns it to the **items** instance variable. You can assume the parameters are valid (e.g., string parameter is not null).
2. **Constructor -** Takes a string (representing the brand) as parameter. It initializes the brand instance variable using the parameter value, and the number of bins with the DEFAULT_BINS_NUMBER value. You can implement this method by using "this" in order to call another constructor.
3. **Copy Constructor -** Initializes the current object in such a way that changes to the current object will not affect the parameter.
4. **getBrand -** get method for **brand**.
5. **getBins -** get method for **bins**.
6. **addItem -** Appends the provided string parameter to the **items** Stringbuffer if the string parameter is not null. Do not add any delimeter (e.g., a comma) as part of the append process. The method always returns a reference to the current object. If the parameter is null an IllegalArgumentException (with any message) will be thrown.
7. **equals -** Defines an equals method for the class. Two **Refrigerator** objects are considered equal if they have the same brand and the same number of bins.
8. **compareTo -** Returns a negative value if the current object has a number of bins less than the parameter, a positive value if the current object has a number of bins larger than the parameter, and 0 if the number of bins is the same. You can assume the parameter will never be null.
9. **toString() -** We have provided this method; do not modify it, otherwise you might not pass release / secret tests.

Store Class Specification

The **Store** class represents a refrigerator store. A store has a name (**name** instance variable), an array of references to **Refrigerator** objects (**refrigerators** instance variable), and the current number of **Refrigerator** objects (**numberOfRefrigerators** instance variable). Keep in mind that the current number of Refrigerator objects is not necessarily the same as the length of the **refrigerators** array. The declaration of each variable follows.

```
private String name;
private Refrigerator[] refrigerators;
private int numberOfRefrigerators;
```

The **Store** class methods are:

1. **Constructor -** Takes a string (representing the store's name) and an integer (representing the maximum number of refrigerators we can have) as parameters. It creates an array of **Refrigerator** references with a size corresponding to the integer parameter. It initializes the current number of refrigerators (**numberOfRefrigerators** instance variable) to 0.
2. **addRefrigerator -** Takes as parameters a string (representing the refrigerator's brand), and an integer (representing the number of bins for the refrigerator). The method will create a **Refrigerator** object and assign it to the next available array entry (if there is one). Assuming no refrigerators have been added, the first available entry will be the one associated with index 0; the next one the one associated with index 1, etc. If no array entry is available, no addition will take place. The method always returns a reference to the current object. No exception is thrown by this method.
3. **getCountWithBinsInRange -** This method takes two integer parameters representing a lower bound and upper bound for the number of bins in a refrigerator. The method will return the number of refrigerators that have a number of bins that fall in the specified range.
4. **getRefrigerators -** This method returns an array with copies of **Refrigerator** objects that have a number of bins that falls in the range specified by the two integer parameters. The kind of copying you are required to do is one where modifications to the returned array will not affect any of the **Refrigerator** objects that are part of the store. Feel free to use the previous method during the implementation of this method. An empty array will be returned if no refrigerators are found in the specified range.
5. **getRefrigeratorsWithNumberOfBins -** This method takes an ArrayList<Appliance> parameter and an integer parameter representing a number of bins. It will return an ArrayList<Refrigerator> with references to those objects in the ArrayList<Applicance> parameter that have a number of bins corresponding to the integer parameter. Remember that the ArrayList<Appliance> parameter can have objects other than refrigerators (e.g., Toaster objects, where Toaster is class that implements the Appliance interface). Notice that you don't need to implement the Toaster class.
6. **toString() -** We have provided this method; do not modify it, otherwise you might not pass release / secret tests.

Driver / Expected Output (Feel free to ignore)

Driver

```
public static void main(String[] args) {
    String answer = "", brand = "GE";
    int bins = 20;

    Refrigerator r1 = new Refrigerator(brand, bins);
    r1.addItem("milk").addItem("cheese");
    answer += r1 + "\n";

    Refrigerator r2 = new Refrigerator("Cheap", 5);
    answer += r2 + "\n";

    Refrigerator r3 = new Refrigerator("Sony", 20);
    r3.addItem("lettuce").addItem("tomato").addItem("cheese");
    answer += r3 + "\n";

    answer += "Brand: " + r1.getBrand() + "\n";
    answer += "Bins: " + r1.getBins() + "\n";

    answer += "Equality: " + r1.equals(r2) + "\n";
    answer += "Comparison: " + (r1.compareTo(r2) < 0 ? true : false) + "\n";
    answer += "==================================================\n";

    Toaster t1 = new Toaster("Sharp", 35);
    t1.setToastingLevel(50);
    answer += t1 + "\n";
    Toaster t2 = new Toaster("LG", 100);
    t2.setToastingLevel(10);
    answer += t2 + "\n";
    answer += "==================================================\n";

    int maxRefrigerators = 60;
    Store store = new Store("TerpMart", maxRefrigerators);
    store.addRefrigerator("Sony", 20).addRefrigerator("LG", 15);
    store.addRefrigerator("Sharp", 8).addRefrigerator("GE", 4);
    answer += "Store:\n";
    answer += store + "\n";
    answer += "Number of Refrigerators with bins in range: " + store.getCountWithBinsInRange(8, 16);
    answer += "\nRefrigerators in range:\n";
    Refrigerator[] found = store.getRefrigerators(8, 16);
    for (Refrigerator refrigerator : found) {
        answer += refrigerator + "\n";
    }
    answer += "==================================================\n";


    ArrayList<Appliance> appliances = new ArrayList<Appliance>();
```

```java
                appliances.add(r1);
                appliances.add(r2);
                appliances.add(r3);
                appliances.add(t1);
                appliances.add(t2);
                answer += "\nAll Applicances:\n";
                answer += appliances + "\n\n";
                answer += "***Only Refrigerators with number of bins: " + bins + "\n";
                answer += Store.getRefrigeratorsWithNumberOfBins(appliances, bins);
                answer += "\n==================================================\n\n";

                String str = "blue8Hou9se";
                answer += "Before removing digits: " + str + "\n";
                answer += "After removing digits: " + RecursionProblem.removeDigits(str) + "\n";
                answer += "==================================================\n\n";

                int[][] data = { { 10, 30 }, null, { 8 }, null, {} };
                answer += "NumberRowsNullOrEmpty: " + ArrayUtilities.getNumberRowsSetToNullOrEmpty(data);
                answer += "\n==================================================\n\n";

                int[][] result = ArrayUtilities.getArrayCopyWithoutNullEmptyRows(data);
                answer += "ArrayCopyWithoutNullEmptyRows: ";
                for (int i = 0; i < result.length; i++) {
                    answer += Arrays.toString(result[i]);
                }
                answer += "\n==================================================\n\n";

                System.out.println(answer);
    }
```

Output

```
Refrigerator [brand=GE, bins=20, items=milkcheese]
Refrigerator [brand=Cheap, bins=5, items=]
Refrigerator [brand=Sony, bins=20, items=lettucetomatocheese]
Brand: GE
Bins: 20
Equality: false
Comparison: false
==================================================
Toaster [brand=Sharp, cost=35, toastingLevel=50]
Toaster [brand=LG, cost=100, toastingLevel=10]
==================================================
Store:
Name: TerpMart
Refrigerators:
Refrigerator [brand=Sony, bins=20, items=]
Refrigerator [brand=LG, bins=15, items=]
Refrigerator [brand=Sharp, bins=8, items=]
Refrigerator [brand=GE, bins=4, items=]

Number of Refrigerators with bins in range: 2
Refrigerators in range:
Refrigerator [brand=LG, bins=15, items=]
Refrigerator [brand=Sharp, bins=8, items=]
==================================================

All Applicances:
[Refrigerator [brand=GE, bins=20, items=milkcheese], Refrigerator [brand=Cheap, bins=5, items=], Refrigerator
[brand=Sony, bins=20, items=lettucetomatocheese], Toaster [brand=Sharp, cost=35, toastingLevel=50], Toaster
[brand=LG, cost=100, toastingLevel=10]]

***Only Refrigerators with number of bins: 20
[Refrigerator [brand=GE, bins=20, items=milkcheese], Refrigerator [brand=Sony, bins=20,
items=lettucetomatocheese]]
==================================================

Before removing digits: blue8Hou9se
After removing digits: blueHouse
==================================================

NumberRowsNullOrEmpty: 3
==================================================

ArrayCopyWithoutNullEmptyRows: [10, 30][8]
==================================================
```