CMSC131, Fall 2020, Final Exam
Deadline: Thursday, Dec 17, 7:15 pm (No late deadline)

INSTRUCTIONS:

- Code distribution available in the StudentsDistribution folder, available in the same folder this pdf file is available.
- Grading Distribution:
  - 113 pts → release tests
  - 87 pts → secret tests, style and requirements. The secret tests include all the multiple choice questions (50 pts) and secret tests for the code (37 pts). There are 25 multiple choice questions each worth 2 pts.
  - There are no public tests.
- Student tests that you provide will not be graded.
- **You must verify that your code compiles and works in the submit server; if your code does not compile in the submit server, you will not get credit for this exam.**
- We expect you to submit often during the exam period and to check your code works in the submit server. Network problems, Eclipse submission problems, or submit server overloading are not valid excuses for not submitting your work. If you have a compilation problem, post a note in Piazza.
- You have 8 release tokens and you can release-test your work anytime.
- You can only post clarification questions in Piazza. Use the Piazza folder **final** to post your questions.
- We will use the Piazza folder **finalclarifications** to post any clarifications. Please don't use this folder :). Check this folder often to see clarifications we may have made. After we post an answer to a question, we may remove the question and the answer after 15 minutes. Any clarification that applies to the whole class, will be added to the clarifications 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 get assistance from a TA during office hours.
- You may not address questions other students post in Piazza (only TAs and instructors can address questions).
- **You must work by yourself.**
- You may use class resources (lecture notes, lecture/lab examples, videos, your projects, etc.), but no other resources (e.g., code from the web).
- Submissions will be checked with cheating detection software.
- **If you are an ADS student:** The provided timeframe 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 is.

Specifications

General Information

- You don't need to add comments to your code, but you must have good variable names, indentation and you should avoid code duplication.
- Don't cut and paste text from javadoc text or this 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.

- The methods you need to implement for each class have the following statement that you need to remove and replace with the expected code: **throw new UnsupportedOperationException("You must implement");**

Part #1 (Multiple choice)

Provide answers to the multiple choice questions you will find in the code distribution **FinalPartOneMultipleChoice.zip.** To answer a question, remove the throw statement and add a return statement with the character representing your answer. After submitting this part, verify the results of the release test named **MultipleChoiceAnsweredNotGraded**. If you pass the test, you have provided an answer for every multiple choice question. The correctness of your answers will be determined by secret tests. Each question is worth 2 pts for a total of 50 pts. **It is your responsibility to verify you pass the release test.** To perform release testing for the multiple choice question, click on the "view" option of the submit server entry in order to see the release test option.

Part #2 (Coding)

For this part you need the code distribution **FinalPartTwoCoding.zip**. In the distribution you will find the classes (with empty methods) that you need to implement. The methods you need to implement for each class are described below.

1. **Utilities Class**

a. **public static int getLengthLargestRow(int[][] data)** - This method returns the length of the largest row in the two dimensional array **data** parameter. Keep in mind that a row can be null (e.g., data[0] = = null) in which case you can assume the length of that row is 0. The method throws an IllegalArgumentException (with any message) if the **data** array parameter is null. The following is an example of the expected processing:

```
Array[ [30]  [10,6,9,2,89]  [77,20,2] ]
MaxLength: 5
```

b. **public static int[] duplicateAndFill(int[] data, int newLength, int filler)** - This method returns a new array with a length that corresponds to the **newLength** parameter. All the elements of the **data** array will be copied to the new array. After copying the array, if there are entries available in the new array, they will be initialized with the value specified by the **filler** parameter. The method throws an IllegalArgumentException (with any message) if the array parameter is null or if the **newLength** parameter is less than the length of the **data** array. The following is an example of the expected processing, where **filler** and **newLength** have a value of 9 (those values don't need to be the same when calling the method):

```
Original Array: [30, 4, 6, 8]
Result Array: [30, 4, 6, 8, 9, 9, 9, 9, 9]
```

c. **public static int[][] getNonRaggedArray(int[][] data, int filler)** - This method will create a new two-dimensional array, that has the same number of rows as the **data** array. Each row of this new array will be initialized using the **duplicateAndFill** method you implemented above, using the corresponding row in the **data** array and the **filler** parameter. The **newLength** value to use will correspond to the largest row in the two-dimensional array **data** parameter, which you can compute using the **getLengthLargestRow** method you defined above. **Important:** if a row in the **data** array is null (e.g., data[0] == null), you need to call the **duplicateAndFill** method with an array of size 0 (new int[0]). The method throws an IllegalArgumentException (with any message) if the array parameter is null. The following are examples of the expected processing, using a value of 4 for **filler**.

```
Original Array
[ [30]  [10,6,9,2,89]  [77,20,2] ]

Result Array
[ [30,4,4,4,4]  [10,6,9,2,89]  [77,20,2,4,4] ]
```

```
Original Array
[ null  [10,6,9,2,89]  [77,20,2] ]

Result Array
[ [4,4,4,4,4]  [10,6,9,2,89]  [77,20,2,4,4] ]
```

In the code distribution you will find a sample driver (**SampleDriverUtilities.java**) and expected output (**SampleDriverUtilitiesOutput.txt**) that illustrates some of the functionality expected from the methods. Feel free to use the driver to write your own tests.

2. **RecursionProblem Class**

The **RecursionProblem** class defines two methods that you must implement using recursion. You may not use the for, do, or while constructs during the implementation of the methods. Although you will not be penalized for using static variables or instance variables, there is no need for them (and you should implement your code without them). Do not use the words **for, while**, or **do** in your **RecursionProblem.java** file 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 private static auxiliary methods.

a. **public static String replaceVowelsWith(String str, String replacement)** - This method returns a string where vowels (in uppercase or lowercase) have been replaced with the **replacement** string parameter. The characters that were not replaced, will appear in lowercase in the string returned by the method. If the replacement string has uppercase characters, those will not be turned to lowercase. You can use the methods Character.toLowerCase(), Character.toUpperCase(), and Character.toString (returns a string given a character). Also, from the String class you can use length(), substring(), and charAt(). The method throws an IllegalArgumentException (with any message) if either parameter (or both) are null. The following is an example of the expected processing:

```
Replacement: *-
Original: thEskyisblue
  Result: th*-sky*-sbl*-*-
```

b. **public static int setToPositive(int[] array)** - This method sets to positive any negative values in the parameter **array**. The method returns the number of values that were set to positive. The method throws an IllegalArgumentException (with any message) if the parameter is null. The following is an example of the expected processing:

```
Original Array: [-89, 4, 700, -6, -9]
After Processing: [89, 4, 700, 6, 9]
Number of values changed: 3
```

In the code distribution you will find a sample driver (**SampleDriverRecursionProblem.java**) and expected output (**SampleDriverRecursionProblemOutput.txt**) that illustrates some of the functionality expected from the methods. Feel free to use the driver to write your own tests.

3. TrainCar Class

The **TrainCar** class (see TrainCar.java) represents a car in a train. The instance variables associated with this class are:

```
private int tons;  // Represents the number of tons (weight) associated with the car
private ArrayList<String> trainCarItems; // Represents the items the car is carrying
```

The methods you need to implement are:

a. **Constructor** - This method initializes **tons** to 0 and **trainCarItems** to an empty ArrayList of string references.
b. **Copy Constructor** - This method initializes the new object in such a way that changes to the new object will not affect the original.
c. **public TrainCar addItem(String itemsName, int tons)** - This method adds to the ArrayList, the string resulting from appending the **itemsName** string and the number of **tons** parameter. For example, if we call the method with the values "desks" and 4, the string to add to the ArrayList will be "desks4". The method will increase the number of tons of the car by the parameter value. The method throws an IllegalArgumentException (with any message) if **itemsName** parameter is null, or if the number of tons is less than or equal to 0. The method will always return a reference to the current object.
d. **public String getItems()** - Returns a string with the ArrayList items, by just calling the ArrayList toString() method.
e. **public int getTons()** - Returns the number of tons.
f. **String toString()** - We have written this method for you; don't modify it.
g. **public boolean equals(Object obj)** - Determines whether two **TrainCar** objects are equal or not. Two **TrainCar** objects are equal if the toString() method for each object returns the same string.

In the code distribution you will find a sample driver (**SampleDriverTrainCar.java**) and expected output (**SampleDriverTrainCarOutput.txt**) that illustrates some of the functionality expected from the methods. Feel free to use the driver to write your own tests. You can add private non-static methods and instance variables during the implementation of the **TrainCar** methods.

4. Train Class

The **Train** class (see Train.java) represents a train. The instance variables associated with this class are:

```
private String name;  // Represents the name of the train
private int numberOfTrainCars; // Represents the number of TrainCar references that have been added to the train
private TrainCar[] allTrainCars; // Array of TrainCar references
```

The methods you need to implement are:

a. **Constructor (public Train(String name, int maxTrainCars))** - This method initializes the **name** instance variable using the corresponding parameter, sets the **numberOfTrainCars** to 0, and creates an array of **TrainCar** references with a size that corresponds to **maxTrainCars**.
b. **public boolean addTrainCar()** - This method creates a new **TrainCar** object (using the **TrainCar** default constructor) and adds the reference to the object to the next available entry in the **allTrainCars** array. The first reference added will be assigned to the array entry with index position 0, the next reference will be added to the array entry with index position 1, etc. The method will return false (and do no processing) if there is no space in the array to add the reference; otherwise the method will return true. Remember to increase the **numberOfTrainCars** instanceVariable if a train car is added.
c. **public boolean addItemToCar(String itemsName, int tons, int trainCarIndex)** - This method adds an item to a train car of the train. The train car to add the item to corresponds to the train car of the **allTrainCars** array associated with the **trainCarIndex**. Remember, **trainCarIndex** is an index. For example, if we want to add to the first car of the train, **trainCarIndex** will be 0. The method will not do any processing and return false, if the value of **trainCarIndex** is negative or larger than or equal to **numberOfTrainCars**. When the method calls the **TrainCar addItem** method, an exception may take place if invalid parameters were identified by the **addItem** method; your code should handle this exception and return false, instead of letting the code crash. The method will return true if the item is added. You don't need to generate any error message when the exception takes place.
d. **String toString()** - We have written this method for you; don't modify it.
e. **public String getName()** - We have written this method for you; don't modify it.

f. **public int getNumberOfTrainCars()** - We have written this method for you; don't modify it.
g. **public int getTons()** - This method returns the total number of tons associated with the train cars that have been added to the train. Remember that not all the **allTrainCars** array entries may be associated with a train car.
h. **public StringBuffer getItems()** - This method returns a StringBuffer with the items of the train cars that have been added to the train. Just append the items to the StringBuffer (no separator is needed). Remember that not all the **allTrainCars** array entries may be associated with a train car.
i. **public int compareTo(Train train)** - This method compares the number of tons of two trains. The method will return a negative value if the number of tons of the current object is larger than the parameter, 0 if the number of tons is the same, and a positive value otherwise.
j. **public Train getTrainWithCars(String name, int[] trainCarIndices)** - This method returns a new train with copies of **TrainCar** objects present in the current object. The train car objects to copy to the new train are represented by the **trainCarIndices** array. The **trainCarIndices** array has the indices of the train cars we need to copy from the current object to the new train. To make a copy of a train car, use the **TrainCar** copy constructor. The size of the **allTrainCars** array of the new train will correspond to the length of the **trainCarIndices** array. The train car copies will be placed in the new train one after another. Remember to initialize the **numberOfTrainCars** instance variable of the new train to the size of the **trainCarIndices** array. The name of the new train will correspond to the **name** parameter and you can assume it will not be null. You can also assume the **trainCarIndices** parameter will not be null and will have values that corresponds to train cars that exist in the current object.
k. **public static int getNumberOfTrainCars(ArrayList<CargoVehicle> vehicles)** - The **Train** class implements the **CargoVehicle** interface that defines the methods getTons() and getItems() (described earlier). There are other classes that implement the **CargoVehicle** interface. An example is the **Truck** class that is part of the code distribution. The **getNumberOfTrainCars** method has as parameter an ArrayList of **CargoVehicle** references; some could be trains, some trucks or anything that implements the **CargoVehicle** interface. The **getNumberOfTrainCars** method will return the total number of train cars (**not trains**) of trains that could exist in the **vehicles** parameter. You can assume the **vehicles** parameter will be different than null. There is nothing you need for the **Truck** class (we have implemented it for you); it is used in the driver we have provided.

In the code distribution you will find a sample driver (**SampleDriverTrain.java**) and expected output (**SampleDriverTrainOutput.txt**) that illustrates some of the functionality expected from the methods. Feel free to use the driver to write your own tests. You can add private non-static methods and instance variables during the implementation of the **Train** methods.