CSE 8B: Introduction to Programming and Computational Problem Solving - 2

Assignment 5

Objects and Classes in Java

Due Date: Wednesday, May 17, 11:59 PM

Welcome back! Be sure to start this assignment as EARLY as possible! You got this!

Learning goals:

- Implement classes with getters and setters.
- Write unit tests using objects and instance methods.

Your grade will be determined by your most recent submission. If you submit to Gradescope after the deadline, it will be marked late and the late penalty will apply regardless of whether you had past submissions before the deadline.

If your code does not compile on Gradescope, you will receive an automatic zero on the assignment.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 8B Coding Style Guidelines. These guidelines can also be found on Canvas. Please ensure to have COMPLETE file headers, class headers, and method headers, to use descriptive variable names and proper indentation, and to avoid using magic numbers.

Part 0: Getting started with the starter code (0 points)

1. If using a personal computer, then ensure your Java software development environment does not have any issues. If there are any issues, then review Assignment 1, or come to the office/lab hours before you start Assignment 5.
2. First, navigate to the `cse8b` folder you created in Assignment 1 and create a new folder named `assignment5`
3. Download the starter code. You can download the starter code from Piazza → Resources → Homework → `assignment5.zip`. The starter code contains four files: `Assignment5.java`, `MyArrayList.java`, `FifoList.java`, and `Student.java`. Place the starter code within the `assignment5` folder you just created

4. Compile the starter code within the `assignment5` folder. You can compile all files using the single command `javac *.java` and you should get a compiler error saying that methods in either `MyArrayList.java`, `FifoList.java` or Student.java are missing return statements and have not been implemented yet. For example:

```
FifoList.java:32: error: missing return statement
    }
    ^
FifoList.java:36: error: missing return statement
    }
    ^
FifoList.java:40: error: missing return statement
    }
    ^
```

5. The objective of this assignment is to get the `MyArrayList`, `FifoList` and `Student` classes working by implementing the class methods in these classes and testing them.

Part 1: Implement Student Class (30 points)

Overview

In this part of the assignment, you will implement a very minimalistic `Student` class that represents properties of a typical college student.

WARNING: you must NOT change any data fields or method signatures in the starter code. As such, please observe the starter code and read the instructions below to make sure you understand what each field means before you start to implement.

The Student Class

In `Student.java`, you will be implementing **2 constructors, 5 instance getter methods, 5 instance setter methods, and 2 instance methods.** This class is a POJO (Plain Old Java Object) class without any great functionality of its own but acts as a template to model a 'Student.'

Each `Student` object, which is an instance of the `Student` class, contains the following fields (all are provided in the starter code ; do not change any of these):

1. `private String name`: the name of the student
2. `private String college`: the name of the college the student attends
3. `private double currentGpa`: the student's current gpa
4. `private int[] weeklyHoursOfSleep`: an `int` array of length **7** that stores the number of hours the student slept on each day last week. Index 0 being Sunday, index 1 being Monday, and so forth.
5. `private boolean isTired`: a boolean that denotes whether the student is currently tired or not

6. `private ArrayList<Student> friends`: an `ArrayList` of other `Students` that are this student's friends
7. `private ArrayList<Double> testScores` an `ArrayList` of `Doubles` that represent this student's test scores.

Notice how each member field is declared `private`. This means that the member is only visible *within* the class, not from any other class. In other words, you will need to use accessors (i.e., getter methods) and mutators (i.e., setter methods) to access and modify, respectively, these `private` members. **You must also use the `this` keyword to access member variables hidden by local variables.**
The `Student` class contains the following member methods:

1. **`public Student()`**
   This is the no-arg constructor of the `Student` class. This constructor needs to set the `name`, `college`, `currentGpa`, `weeklyHoursOfSleep`, and `isTired` member variables according to what is shown below:
   - `name` must be set to `null`
   - `college` must be set to `null`
   - `currentGpa` must be set to `0.0`
   - `weeklyHoursOfSleep` must be an array of length **7** that contains 7 zeros, that is `{0,0,0,0,0,0,0}`
   - `isTired` must be set to `false`
   - `friends` must be set an empty `ArrayList<Student>` that stores `Students`
   - `testScores` must be set to an empty `ArrayList<Double>` that stores `Doubles`

2. **`public Student(String name, String college, double currentGpa, int[] weeklyHoursOfSleep, boolean isTired, ArrayList<Student> friends, ArrayList<Double> testScores)`**
   This is another constructor of the `Student` class. This constructor needs to set the `name`, `college`, `currentGpa`, `weeklyHoursOfSleep`, and `isTired` member variables using the constructor parameters. For `weeklyHoursOfSleep`, you must perform a copy of the array referred to by the constructor parameter array reference variable to the array referred to by the instance array reference variable. This means that you cannot simply set the `private weeklyHoursOfSleep` data field to refer to the same array being referred to by the constructor parameter. Instead, you must copy the elements of the array it refers to into the array referred to by the instance variable. The same is true for `friends` and `testScores`. You must iterate over the `friends` and `testScores` that were passed in, and add every element to this student's `friend` and `testScores`.

   You can assume that `weeklyHoursOfSleep`, `friends`, and `testScores` will always be non null.

Remember, you must use the `this` keyword to access member variables hidden by local variables.

3. Six getters/accessors -
    a. **`public String getName()`**
    b. **`public String getCollege()`**
    c. **`public double getCurrentGpa()`**
    d. **`public int[] getWeeklyHoursOfSleep()`**
        i. This getter method is special in that it must return a new integer array that stores the same integers referenced to by the instance variable `weeklyHoursOfSleep`. In other words, do not simply return the same instance reference variable.
    e. **`public boolean getIsTired()`**
    f. **`public ArrayList<Student> getFriends()`**
        i. This getter method is special in that it must return a new `ArrayList<Student>` that stores the exact same `Student` references that are currently stored in `friends`. In other words, do not simply return the `friends` reference. Make a new `ArrayList`, copy all `Student` references over to the new `ArrayList` and return that.

Each getter method must simply return the corresponding `private` data field of the `Student` object, except for `getWeeklyHoursOfSleep` and `getFriends`, as described above.

4. Five setters/mutators -
    a. **`public void setName(String name)`**
    b. **`public void setCollege(String college)`**
    c. **`public void setCurrentGpa(double currentGpa)`**
    d. **`public void setWeeklyHoursOfSleep(int[] weeklyHoursOfSleep)`**
    e. **`public void setIsTired(boolean isTired)`**

Each setter method must simply set the corresponding instance variable to the one provided as an argument to the method. Except, `public void setWeeklyHoursOfSleep()` must perform a **copy** of the array referred to by the method parameter array reference variable to the array referred to by the instance array reference variable. This means that you cannot simply set the `private weeklyHoursOfSleep` data field to refer to the same array being referred to by the method parameter. Instead, you must copy the elements of the array it refers to into the array referred to by the instance variable. Again, remember you must use the `this` keyword to access member variables hidden by local variables.

5. **`public String dayOfLeastSleep()`**

This method must return a string which is either `"Sunday"`, `"Monday"`, `"Tuesday"`, `"Wednesday"`, `"Thursday"`, `"Friday"`, or `"Saturday"`. It returns whichever day the student

has had the least sleep on. Note the starter code provides constants for the days of the week. You are encouraged to use these to remain consistent.

For example, if the `weeklyHoursOfSleep` data field is `{7, 9, 3, 10, 12, 3, 13}`, then you would return `"Tuesday".` This is because 3 hours is the lowest amount the student slept in the week and remember that index 0 corresponds to Sunday, index 1 corresponds to Monday, and so forth. **Important:** If there are **ties**, return the first day of the week that the least hours of sleep occurred (minimum index). In this case, because 3 is the least and occurred twice (Tuesday and Friday), you would return the earliest day of the week which is Tuesday.

6. **`public int totalHoursSlept()`**

This method must return the total hours the student slept for the week.

7. **`public void makeFriend(Student friend)`**

This method adds `friend` to the end of this student's `friends` `ArrayList`. You do not need to make a deep copy of the friend object that was passed in. Just add the reference that was passed in to the `ArrayList`.

8. **`public void takeTest()`**

This method must add a random `Double` from [0, 100] to `testScores` of this student. A random `Double` means that the `Double` chosen must be completely random in the range [0, 100] (every single number must have an equal chance being chosen). If you choose numbers in a non-random way, you might fail the Gradescope autograder. Note that we are using `Doubles` so the new test score could be a decimal. `Math.random()` may be useful here.

9. **`public double averageTestScore()`**

This method must return the average of all test scores currently in the student's `testScores` `ArrayList`. You must use floating point division. If the student currently has zero test scores (have not taken any tests), then you must return `0.0`.

The methods that are provided in the starter code **MUST** return the correct output (if any) as that is how the Gradescope autograder will test your code.

Part 2: Implement MyArrayList Class (30 points)

In this part of the assignment, you will implement `MyArrayList` which is a dynamic array that can expand in length when the capacity is being reached. We will dive into more details later on.

At a high level, you worked with arrays in Java in the previous weeks of this class which could not change lengths once you created it. As you may know, you had to specify the length of the array upon declaration and the length remains fixed. If you create an array of length `10` and fill the array with `10` elements, there is no way to add an additional element to the array without creating a new one from scratch. `MyArrayList` allows the user to store items just like an array but also includes many operations that are not included with a simple array including expanding its capacity as needed.

WARNING: you must NOT change any data fields or method signatures in the starter code. As such, please observe the starter code and read the instructions below to make sure you understand what each field means before you start to implement.

In `MyArrayList.java`, you will be implementing **1 constructor and 7 instance methods.**

A `MyArrayList` object contains the following fields:

1. `private int[] data`: the "backing array" that is used to store the data of `MyArrayList`. It is called a "backing array" because under the hood (not exposed to the user of your class) `MyArrayList` is itself actually implemented with an array.
2. `private int size`: the number of integers currently in the `MyArrayList`

One `static` class variable are also provided and are used when printing out an error when it occurs:

1. `private static final String INDEX_OUT_OF_BOUNDS`: printed when the user tries to `get`, `insert`, `remove`, or `popBack` in invalid ways (more on that below).

Notice how each member field is declared `private`. This means the member is only visible *within* the class, not from any other class. **You must also use the `this` keyword to access member variables hidden by local variables.**

You are free to declare any additional instance or `static` member variables as per your convenience. However, if you do so, you must declare them as `private`.

The `MyArrayList` class must contain the following member methods. Please read through **every single** method description before implementing the individual methods as some may depend on others:

1. **`public MyArrayList()`**
   The no-arg constructor for the `MyArrayList` class, which will initialize `data` to an integer array of length `10` and also `size` to `0`.

2. **`public void add(int element)`**
   A method to add an element to the end of the `MyArrayList`. This method must never fail so it always increases the `size` by `1`. If adding the element would make the `size` of the `MyArrayList` to be greater than or equal to half the capacity of `data` (`data.length /`

`2`), then you must create a new array that is double the length of `data`, copy all the elements over and set the new array created as `data`. Examples will make this clear.

Examples:
Initially, `data` starts off as `{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}` with `size` = `0`

`myArrayList.add(10)` →`{10, 0, 0, 0, 0, 0, 0, 0, 0, 0}` with `size` = `1`
`myArrayList.add(20)` →`{10, 20, 0, 0, 0, 0, 0, 0, 0, 0}` with `size` = `2`
`myArrayList.add(30)` →`{10, 20, 30, 0, 0, 0, 0, 0, 0, 0}` with `size` = `3`
`myArrayList.add(40)` →`{10, 20, 30, 40, 0, 0, 0, 0, 0, 0}` with `size` = `4`
`myArrayList.add(50)` →`{10, 20, 30, 40, 50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}` with `size` = `5`

3. **`public int get(int index)`**
   A method to get the element at the specified `index`. If the user tries to get an element with an `index` less than `0` or tries to get an element that does not exist in the `MyArrayList`, you must print out the error message and return `-1`. Otherwise, return the element at the specified index of the `MyArrayList`. Examples will make this clear.

   Examples:

   Imagine the user has done some adding and the state of the backing array is `{10, 20, 30, 40, 50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}` with `size` = `6` (which means they added a zero in the end).

   `myArrayList.get(-1)` → print out `INDEX_OUT_OF_BOUNDS` on an new line and return `-1`
   `myArrayList.get(1)` → `20`
   `myArrayList.get(3)` → `40`
   `myArrayList.get(5)` → `0`
   `myArrayList.get(6)` → print out `INDEX_OUT_OF_BOUNDS` on an new line and return `-1`

4. **`public void insert(int element, int index)`**
   A method that inserts the specified element at the corresponding index with the elements to the right of the `index` shifted to the right by 1. If the `index` is less than 0 or if the `index` is greater than the current `size` (number of elements) of the `MyArrayList`, then you must print out the error message and return. Otherwise, insert the element at the specific `index` and increment `size` by 1. If inserting this element would cause the number of elements to be greater than or equal to half the capacity of `data` (`data.length / 2`), then you must create a new array that is double the length of `data`, copy all the elements over and set the new array created as `data`. Examples will make this clear.

Examples:
Imagine the user has done some adding and the state of the backing array is `{10, 20, 30, 0, 0, 0, 0, 0, 0, 0}` with `size` = `3`

`myArrayList.insert(100, 1)` →`{10, 100, 20, 30, 0, 0, 0, 0, 0, 0}`
`myArrayList.insert(1, 0)` → `{1, 10, 100, 20, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}`

5. **`public int remove(int index)`**
   A method that removes the element specified at the particular `index` and returns the removed element. If the user tries to remove with an `index` less than `0` or an element that does not exist in the `MyArrayList`, then you must print out `INDEX_OUT_OF_BOUNDS` on a new line and return `-1`. Otherwise, remove elements specified at `index` and shift all elements to the right of index to the left by 1 and decrement `size` by 1. Examples will make this clear.

   Imagine the user has done some adding and the state of the backing array is `{10, 20, 30, 0, 0, 0, 0, 0, 0, 0}` with `size` = `3`

   `myArrayList.remove(1)` →`{10, 30, 0, 0, 0, 0, 0, 0, 0, 0}`
   `myArrayList.remove(2)` →print out `INDEX_OUT_OF_BOUNDS` on a new line and return `-1`. This is because there are only two elements left in `MyArrayList` (10 and 30). There is no element at `index` `2` to be removed.

6. **`public int popBack()`**
   A method that removes the element at the end of the `MyArrayList`. If there is no element to remove, then print out `INDEX_OUT_OF_BOUNDS` on a new line and return `-1`. Otherwise, you must remove the last element from `MyArrayList`, decrement the `size` by 1, and return the removed element. Examples will make this clear.

   Imagine the user has done some adding and the state of the backing array is `{10, 20, 30, 40, 50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}` with `size` = `6`

   `myArrayList.popBack()` →`{10, 20, 30, 40, 50, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}`
   `myArrayList.popBack()` →`{10, 20, 30, 40, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}`
   `myArrayList.popBack()` →`{10, 20, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}`

7. **`public int getCapacity()`**
   This must simply return the capacity of the MyArrayList, (the length of the backing array that is used to implement your dynamic array list).

   Imagine the user has done some adding and the state of the backing array is `{10, 20, 30, 40, 0, 0, 0, 0, 0, 0}` with `size` = `4`

   `myArrayList.getCapacity()` → `10`
   `myArrayList.add(10)` → `{10, 20, 30, 40, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}`
   `myArrayList.getCapacity()` → `20`

8. **`public int getSize()`**
   This must simply return the number of elements in the `MyArrayList`. NOTE that this is different from the capacity, which is defined above. The capacity of `MyArrayList` is how much the backing array is able to hold while the `size` of the MyArrayList is the number of elements that currently exists which was added by the user.

   Imagine the user has done some adding and the state of the backing array is `{10, 20, 30, 40, 0, 0, 0, 0, 0, 0}` with `size` = `4`

   `myArrayList.getSize()` → `4`
   `myArrayList.add(10)` → `{10, 20, 30, 40, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}`
   `myArrayList.getSize()` → `5`
   `myArrayList.getCapacity()` → `20`

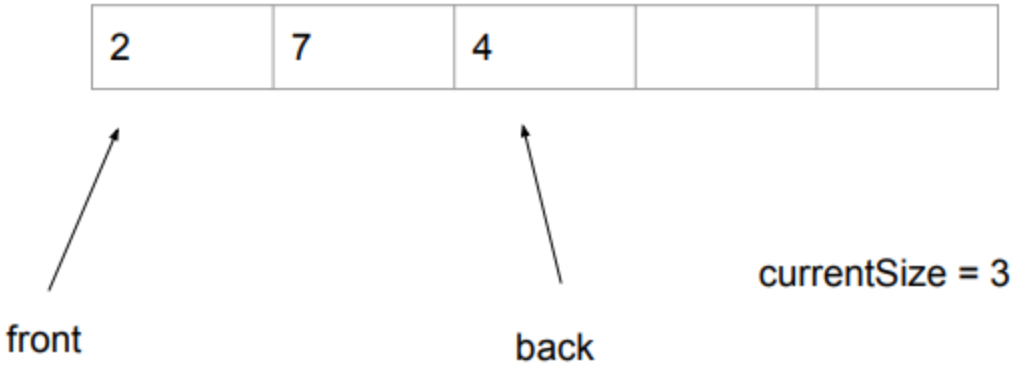
The methods that are provided in the starter code **MUST** return the correct output (if any) as that is how the Gradescope autograder will test your code.

Part 3: Implement FifoList Class (20 points)

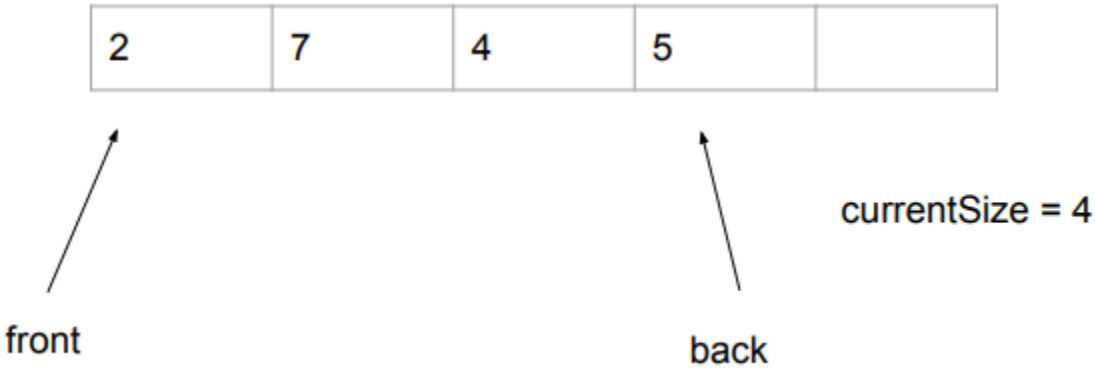
The FifoList Class

In this part of the assignment, you will implement a “first in, first out” list. A `FifoList` is a data structure that maintains its elements in First-In-First-Out (FIFO) order. This means an element is earlier in the data structure's order if it was added before other elements were added. This is described below:

Creating `FifoList` of `maxSize` 5 and adding 2, then adding 7, and then adding 4.

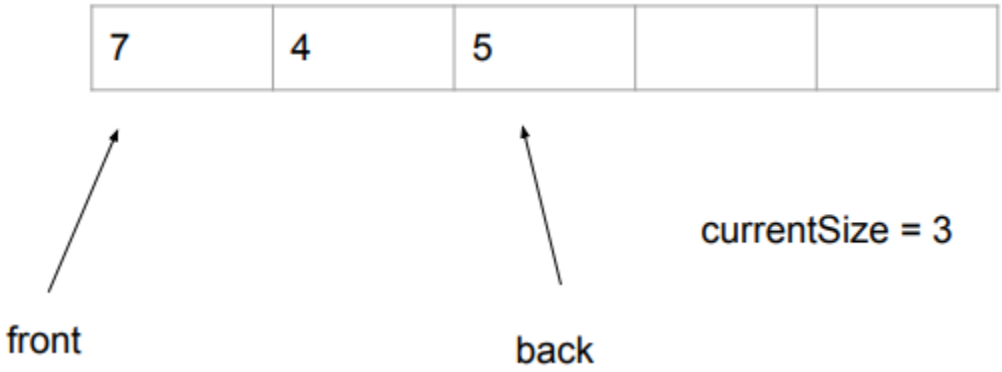


Notice the **add** operation adds items to the back of the FIFO. Adding `5` to the `FifoList` yields



The **peek** operation returns the front element from the FIFO *without* deleting the element from the FIFO. Since `FifoList` maintains elements in FIFO order, the front element of this FIFO is 2. Hence, the peek operation returns 2.

The **delete** operation deletes the front element from the FIFO. Calling the delete operation on this `FifoList` yields



A simple way to think about how a FIFO list works is that it represents the concept of standing in line. Whoever gets to the line first will be served first. Just like whatever gets added to the FIFO list first will be deleted or looked at first.

Please note that the figures above are only one representation of the array that is used to construct the first-in-first-out list. Your implementation may differ and as long as your methods (as described below) return the correct values; it does not matter how the internals of your `FifoList` works.

WARNING: you must NOT change any data fields or method signatures in the starter code. As such, please observe the starter code and read the instructions below to make sure you understand what each field means before you start to implement.

In `FifoList.java`, you will be implementing **2 constructors and 4 instance methods.**

A `FifoList` object contains the following fields:

3. `private int maxSize`: the maximum number of integers the `FifoList` can store
4. `private int currentSize`: the number of integers currently in the `FifoList`
5. `private int[] array`: an array of length `maxSize` that stores the integers in the `FifoList`

Two `static` class variables are also provided and are used when printing out an error when it occurs:

2. `private static final String EMPTY_ERROR`: printed when the user tries to `peek()` or `delete()` an empty `FifoList`
3. `private static final String MAX_SIZE_ERROR`: printed when the user tries to `add()` an element to a `FifoList` that already has a `maxSize` amount of elements.

Notice how each member field is declared `private`. This means the member is only visible *within* the class, not from any other class. **You must also use the `this` keyword to access member variables hidden by local variables.**

You are free to declare any additional instance or `static` member variables as per your convenience. However, if you do so, you must declare them as `private`.

The `FifoList` class must contain the following member methods:

9. **`public FifoList()`**
   The no-arg constructor for the `FifoList` class, which will initialize `maxSize` to `0`, `currentSize` to `0`, and `array` to `null`.

10. **`public FifoList(int maxSize)`**
    This constructor for the `FifoList` class takes the input parameter `maxSize` to initialize the maximum size of the array. For example, creating an object as `new FifoList(5)` must create a `FifoList` whose maximum size is `5`. If the parameter passed is invalid

(i.e., a negative number) then initialize your class variables `maxSize` to `0`, `currentSize` to `0`, and `array` to `null`.

11. **`public void add(int element)`**
    A method to add an element to the back of the `FifoList`. If the `FifoList` is already full or `FifoList` is `null`, then this method must print the contents of the `MAX_SIZE_ERROR` string and return.

12. **`public int delete()`**
    A method to delete the front element. Since elements in `FifoList` are maintained in FIFO order, this method must delete the front element within `FifoList` and return the element just deleted. If `FifoList` is empty or `FifoList` is `null`, then this method must print the contents of `EMPTY_ERROR` and return `-1`.

13. **`public int peek()`**
    A method that returns the front element without deleting it. Since elements in `FifoList` are maintained in FIFO order, this method must return the front element of the `FifoList`. If `FifoList` is empty or `FifoList` is `null`, then this method must print the contents of `EMPTY_ERROR` and return `-1`.

14. **`public int size()`**
    A method that returns the current size of the `FifoList` based on how many elements are currently in the `FifoList`. For example, If the user creates a new `FifoList` with a `maxSize` of `5`, performs three `add()` operations followed by one `delete()` operation, then this method must return `2`.

The methods that are provided in the starter code **MUST** return the correct output (if any) as that is how the Gradescope autograder will test your code.

Part 3: Compile, Run, and UnitTest Your Code (10 points)

Just like in previous assignments, **in this part of the assignment, you need to implement your own test cases in the method called `unitTests` in the `Assignment5.java` class.**

In the starter code, two test cases for `Student`, one simple test case for `MyArrayList`, and one simple test case for `FifoList` have already been implemented for you. You can regard each of these as an example to implement other cases though you will need to test more to get full credit. To get full credit on this section, you **must** follow the instructions below.

For `Student`, you must create **three** additional test cases that test either the getter/setter methods, constructors, `getDayOfLeastSleep`, `makeFriend`, `takeTest`, `averageTestScore`, or a combination of these.

For `MyArrayList`, you must create **three** additional test cases. Each test case must have all the following:

1. Invoke the constructor
2. invoke **each** of the 7 instance methods at least once

For `FifoList`, you must create **three** additional test cases. Each test case must have all the following:

1. Invoke either one of both of the constructors
2. have at least 3 add operations
3. have at least 3 delete operations
4. test either the `peek()` or `currentSize()` method

You cannot make all your test cases follow the same structure. For example, you cannot make all your test cases add three elements then delete three elements (for `MyArrayList`/`FifoList`) Your best option is to test a **random combination** of the methods to see if it gives you the right output as that is what the autograder will do.

Please note that points for unit tests are "all or nothing." For example, if you only create two additional tests for `Student` rather than three, you will get zero points on the `Student` unit test portion of the assignment. You will **not** receive partial credit for the two unit tests you created.

Recall, the general approach is to come up with different inputs and manually give the expected output, then call the method with that input and compare the result with expected output.

You are encouraged to create as many test cases as you think to be necessary to cover all the edge cases. The `unitTests` method must return `true` only when all the test cases are passed; otherwise, it must return `false`.

To compare strings by the equality of their characters, you must use the `String` instance method `equals()`. See the given unit tests for examples.

If a test is not passing, try temporarily printing the result of your method(s) and comparing them to the expected output.

You can compile all the files present in the starter code and run your unit tests from `main()` using the following commands: (Make sure you are in the correct directory, else navigate to the starter code using `cd`)

```
> javac *.java
> java Assignment5
```

The first command `javac *.java` compiles all the files in the folder with a `.java` extension, which is what is required. After the `.java` files compile successfully and `.class` files are generated, run the `main()` in `Assignment5` using `java Assignment5`.

Submission

You're almost there! Please follow the instructions below carefully and use the **exact submission format**. Because we will use scripts to grade, **you may receive a zero** if you do not follow the same submission format.

1. Open Gradescope and login. Then, select this course → Assignment 5.
2. Click the DRAG & DROP section and directly select the required files `Student.java`, `MyArrayList.java`, `FifoList.java`, and `Assignment5.java`. Drag & drop is fine. Please make sure you do not submit a zip, just the three files in one Gradescope submission. Make sure the names of the files are correct.
3. You can resubmit unlimited times before the due date. Your score will depend on your final (most recent) submission, even if your former submissions have higher scores.
4. Your submission should look like the below screenshot.