Part 1 (0 points): Warm-up

Do **not** *submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs or instructors during their office hours; they can help you and work with you through the warm-up questions.*

Warm-up Question 1 (0 points)

Write a class `Vector`. A `Vector` should consist of three `private` properties of type double: x,y, and z. You should add to your class a constructor which takes as input 3 doubles. These doubles should be assigned to x,y, and z. You should then write methods `getX()`, `getY()`, `getZ()`, `setX()`, `setY()`, and `setZ()` which allow you to get and set the values of the vector.

Warm-up Question 2 (0 points)

Add to your `Vector` class a method calculateMagnitude() which returns a double representing the magnitude of the vector. The magnitude can be computed by taking

$$\sqrt{x^2 + y^2 + z^2}$$

Warm-up Question 3 (0 points)

Write a method `scalarMultiply` which takes as input a `double[]`, and a `double scale`, and returns `void`. The method should modify the input array by multiplying each value in the array by `scale`. Question to consider: Would this approach work if we had a `double` as input instead of a `double[]`?

Warm-up Question 4 (0 points)

Write a method `deleteElement` which takes as input an `int[]` and an `int target` and deletes all occurrences of `target` from the array. The method should return the new `int[]`. Question to consider: Why is it that we have to return an array and can't simply change the input parameter array?

Warm-up Question 5 (0 points)

Write the same method, except this time it should take as input a `String[]` and a `String`. What is different about this than the previous method? (Hint: Remember that `String` is a reference type.

Part 2

The questions in this part of the assignment will be graded. Note the test program `AssignmentThreeTests.java` *provided with the assignment. There are instructions on how to run the test code after the assignment specification below. You must run your code through these tests. It is OK if your code does not pass every test (although you will lose some marks for this), but your code* ***must*** *compile with the test file. If you cannot get your code to compile with the test file, please see a TA or instructor, who will help you.*

The TAs may use a test program with additional tests to the one we provided, so you should run additional tests. The point of the test file is to help make sure you followed the specifications correctly.

You will not be required to hand in a `main` *method, but you should write one in order to test your code properly. You are also encouraged to add additional auxiliary methods to the required ones below. This will help reduce errors, allow for you to test your code more easily, and make your code more readable and understandable.*

In this assignment, you will first define a Java Object called a `TxtFile` which will be a simple representation of a text file in memory. You will then define a Java Object called `HardDrive` which will be a simplified representation of a computer hard drive and allow you to manipulate the `TxtFile` objects you created in question one. In the third part, you will use the `HardDrive` object in a simulated `Computer` object, for which most of the code has already been provided and you will be required to make a small modification to. Finally, in the 4th part, we will examine a security problem with the computer implementation given. You will see that the computer code given allows for the creating of a "virus" (not a real one, only on the simulated computer), and you will fix the security hole.

Because each section depends on the previous section. It is very important to test your code as you go. For example, if you have a mistake in your `TxtFile` *class, then it will be difficult to find the mistakes in the* `HardDrive` *class that uses it. You can use the test program to help, although to compile it you will have to make sure you create empty* skeleton *methods for all required public methods in both questions 1 and 2 to allow the code to compile.*

Question 1: Representing a text file (20 points)

In this question, you will create your own Object type in Java called a `TxtFile`*. This will allow you to represent the notion of a text file in Java.*

Create a class called `TxtFile`. The class `TxtFile` should have TWO private (non-static) properties, FIVE public (non static) methods, and ONE constructor. In addition to this, you may add a static public method called main and as many PRIVATE helper methods or properties as you like.

The two properties you should create are as follows:

- `private String filename` This String will store the name of the `TxtFile`
- `private String data` This String will store the data of the `TxtFile`

You should write the following five `public` methods:

- `public String getName()` This method should return the `String` representing the name of the file.
- `public void setName(String newName)` This method should set `filename` to have the value of `newName`.
- `public String getData()` This method should return the String representing the data of the file.
- `public void setData(String newData)` This method should set `data` to have the value of `newData`.
- `public void appendData(String newData)` This method should *append* to `data` the value of the `String newData`. For example, if `data` used to have the value `hello` and `newData` has the value `bye` then after calling this method, the value in `data` should be `hellobye`

There is a way to correctly write each of these five methods with just one statement per method.

The final thing you should write in this class is a constructor. Remember that a constructor will be called whenever your Object is created. Keep in mind, that this will be done *always* using the `new` keyword.

The constructor `TxtFile(String name, String text)` should take as input TWO String and assign to the property `filename` the value of `name` and to the property `data` the value of `text`.

Remember that since this constructor takes as input two `String`, it means that to create a `TxtFile`, one will have to call the constructor with TWO arguments. For example, to create a file `FavouriteChessOpenings.txt` with the data `Sicilian, Ruy Lopez, Nimzovitch`, and store a reference to the file into a variable `chessFile`, one would write

```
TxtFile chessFile = new TxtFile("FavouriteChessOpenings.txt", "Sicilian, Ruy Lopez, Nimzovitch");
```

Note that you may assume when writing your methods and constructor that all `String`s passed as input are not null. In other words, if any method/constructor is passed a `null String` then your method can do whatever it likes.

Testing the class `TxtFile`

In order to test what you have written, you should add a `main` method to the class. The main method will not be graded, but will help make sure that your code is correct. To do this, you should create a `TxtFile` using the `new` operator, try calling the methods in it, and see whether the results are what you expect. For example:

```
public class TxtFile {
         private String filename;
         //fill in other properties, methods, and constructors here
         //...
         //
         public static void main(String[] args) {
                   //create a new TxtFile with name foo and data bar
                   TxtFile test = new TxtFile("foo", "bar");
                    //now check whether the name prints correctly.
                    System.out.println(test.getName());
                    //now try to set the name differently
                    test.setName("newname");
                    //look to see if the value is correct or not
                    System.out.println(test.getName());
                    //test getData() and setData() similarly
        }
}
```

Question 2: Representing a hard drive (50 points)

In this question you will write a class called `HardDrive`*. For simplicity, we will assume that a hard drive contains only an array of* `TxtFile`*. (You will not be able to put your MP3s on this hard drive.)*

Create a class `HardDrive`. The class `HardDrive` should have TWO private properties, ONE final private property, EIGHT public methods, and a public constructor. Before coding anything, you should think about the similarities between the various methods, as some of the methods can be very short (even as small as one line), if you realize the similarity between the methods.

You should create the following `private` properties:

- `private TxtFile[] drive` This array will represent the data stored on the drive.

- `private int numUsed` This integer will represent the NUMBER of non-null, non-free spaces on the hard drive/in the array. For example, if the array has non-null values at index 0,2,3,10, then the value for `numUsed` should be 4 since there are four spaces used.

You should create the following `private` constant:

- `private final int DRIVE_SIZE` This should have the value 1000. You should not use the value 1000 anywhere in your code other than to set this constant.

Next, create a `public` constructor: The constructor will take nothing as input and do two things:

- create a new `TxtFile` array of size DRIVE_SIZE and assign it to the variable `drive`
- Set the variable `numUsed` to have the value of 0.

Finally, you should create the following `public` (non static) methods, that modify and access the private properties of the `HardDrive`. Each of these methods may assume that `drive` has already been initialized by the constructor, since it will be impossible for the methods to be called before the constructor is.

- `addFile` : This method should take as input a `String filename` as first argument and a `String data` as second argument and return a `boolean`. It should first check to see if there is space in the array to add a `TxtFile`. (i.e. if `numUsed < DRIVE_SIZE`). If there is no space, the method should return `false`. The method should check to see if there already is a file with the name `filename` on the hard drive. If there is, the method should return `false`.

  If there is space and no file already named `filename`, the method should create a `TxtFile` called `filename` with data `data` and add it to the *first* null space in the array `drive`. It should add one to the variable `numUsed` and then return the value of `true`.

- `indexOfFile` : This method should take as input a `String filename` and return an `int`. The `int` should be the index of the array `drive` that contains the file with name `filename`. If no such file exists, the method should return -1

- `exists` : This method should take as input a `String filename` and return a `boolean` value of `true` if a file with name `filename` exists in `drive` and `false` otherwise.

- `getContent` : This method should take as input a `String filename` and return a `String`. The method should search for a file with name `filename` in the array `drive` and return the *data* that is stored in that `TxtFile`. If no such file exists in the array `drive`, the method should return `null`.

- `deleteFile` : This method should take as input a `String filename` and return a `boolean`. The method should look for the file with name `filename` and set the value of the array `drive` at that index to be null. If no such index exists, the method should return `false`. If the deletion is successful, then it should subtract one from the variable `numUsed` and return `true`.

- `renameFile` This method should take as input a `String from` and a `String to`. It should return a `boolean`. The method should find the index in `drive` of a file with name `from`. If no such index exists, the method should return `false`. It should also check that the file `to` does not exist. If the file `to` does exist, then the method should return `false`.

  If the file `from` exists and `to` does not, then the method should change the name of the file from `from` to be `to`. In this case, the method should return `true`.

- `appendFile` This method should take as input a `String filename` as first argument and a `String extraData` as second argument. It should return a `boolean`. The method should search for the file `filename` in the array `drive`. If no such file exists, it should return `false`. If the file exists, then the method should *append* (i.e. add to) the existing data in that file with the additional data `extraData`.

- `listFiles` This method should take nothing as input and return a `String[]` representing the *names* of all the non-null files stored in `drive`. The `String[]` should be ordered in the same order

that the files appear in `drive`. Hint: Remember that if you have deleted files, there may be some null files stored between non-null files.

You should test the class `HardDrive` in a similar fashion as you did with `TxtFile` before proceeding further.

Question 3: Creating a computer to use this HardDrive (10 points)

In this class you will edit an already existing file called `Computer.java`.

Inside the folder for assignment three, you will find a file called `Computer.java`. Most of the code in this has been written for you. Inside `Computer.java` there will be 3 methods that you should edit. These will be denoted by the marks `//YOUR CODE GOES HERE`

- `turnOn` This method should take nothing as input. It should modify the private property `isOn` and set the value to be `true`.
- `turnOff` This method should take nothing as input. It should modify the private property `isOn` and set the value to be `false`.
- `installOS` This method should take nothing as input. It should create a new `HardDrive` object and assign it to the private property `hardDrive`. It should then create a new file in the `hardDrive` called `.operating_system` (note the dot in front of operating_system). The contents of this file should contain the name of your favourite operating system. So for example, you could create a file with name `.operating_system` and value `windows3.1`

These methods should all be very short.

Running (and then cracking into) the compuer!

Now that you have created these classes, you should be able to run the program `ComputerGUI`. To do so, you should make sure that all .java files are inside the same folder, and then compile them together by typing

```
javac *.java
```

Now, you can launch the program ComputerGUI using the normal command `java ComputerGUI`

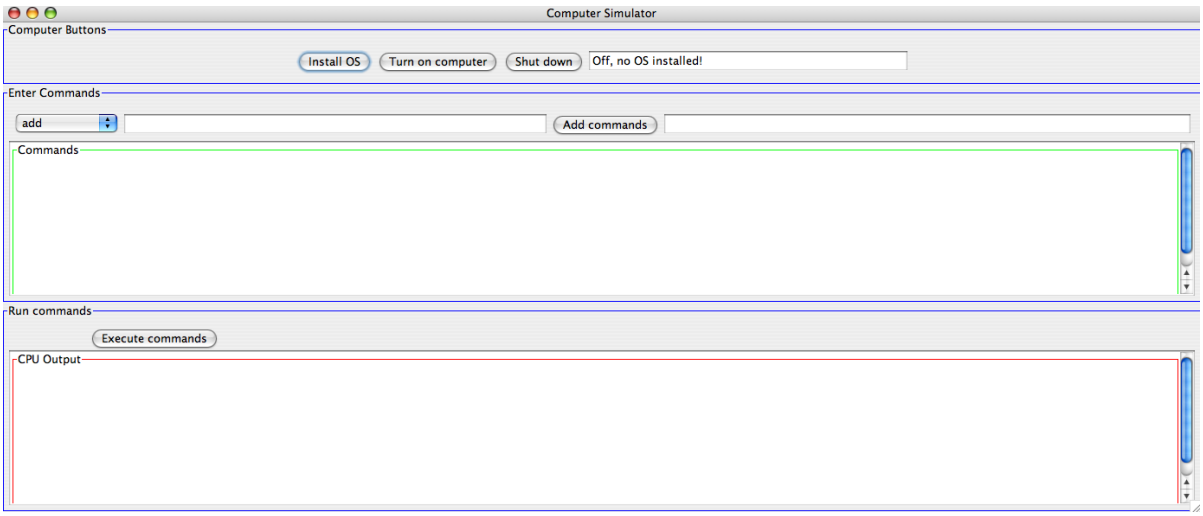

Figure 1: The program ComputerGUI when you first start it.

When you first load the program, you will not be able to do anything, because the simulated computer is off, and no operating system is installed. In order to do anything, you should click the buttons install OS and Turn on computer. These will call the methods that you wrote in question 3. In general, on this

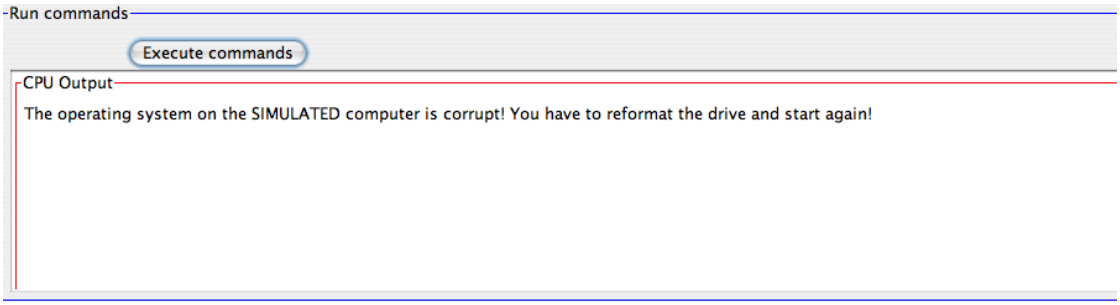

Figure 2: The output of the computer when you try to run it before installing the OS. This message appears whenever the computer can't find the file .operating_system

computer, if you try to execute any commands on the computer without installing an OS you will get a message in the cpu output box.

The reason for this message is that the computer is looking for the file `.operating_system` in order to access "key" components. (It is not really doing anything with the file as the simulated computer has several limitations, but use your imagination to figure out what sorts of things a file like this could do in a real computer.)

Once you have installed the OS and turned on the computer, you can enter commands into the window. To do this, type a command inside the textbox and then click add commands. You can add several commands in a row if you want to do more than one thing at a time.

For any command, you should enter it's *arguments* into the text box. For normal execution, you should not add any parenthesis as these will be added automatically. After you select the kind of instruction to give and enter the commands, you can click the "add command" button. At this point, the command will show up in the command window with parenthesis added.

As you'll see in the next part, adding extra parenthesis or commas can cause major problems, so you should make sure to follow the syntax closely (unless of course you *want* to crash the computer!) In addition, any spaces that you type here will be treated as an argument. So for example, entering a rename command with argument `foo, bar` will rename the file `foo` to be `bar` (with spaces).

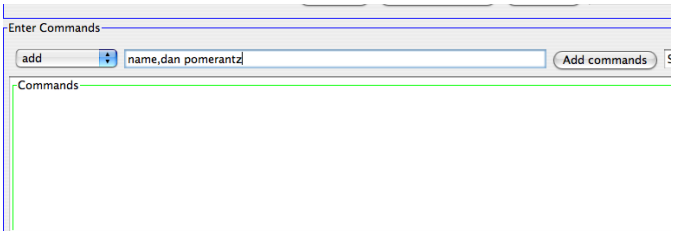

Figure 3: How to enter a command. This would be the command to create a file called `name` with the contents `dan pomerantz` in it. To queue the command up, click the add command button.

Figure 4: After clicking the "Add command" button, the command will show up in the screen.

You can add the following commands:

- add : Takes 2 arguments, separated by a comma. First argument is filename, second is data. Example argument: `name,dan pomerantz`
- rename: Takes 2 arguments, separated by a comma. First argument is old filename, second argument is new filename. Example argument: `name,fullname`
- delete : Takes one argument for which file to delete. Example: `fullname`
- appendFile : Takes 2 arguments, separated by a comma. The first argument is filename, second is data to append to. Example: `fullname, mr` (Note here that we can't add something including a comma since the parser would interpret it differently.)
- list : Takes no arguments. Lists all the files in memory
- showFile : Takes one argument for which file to display the contents of.

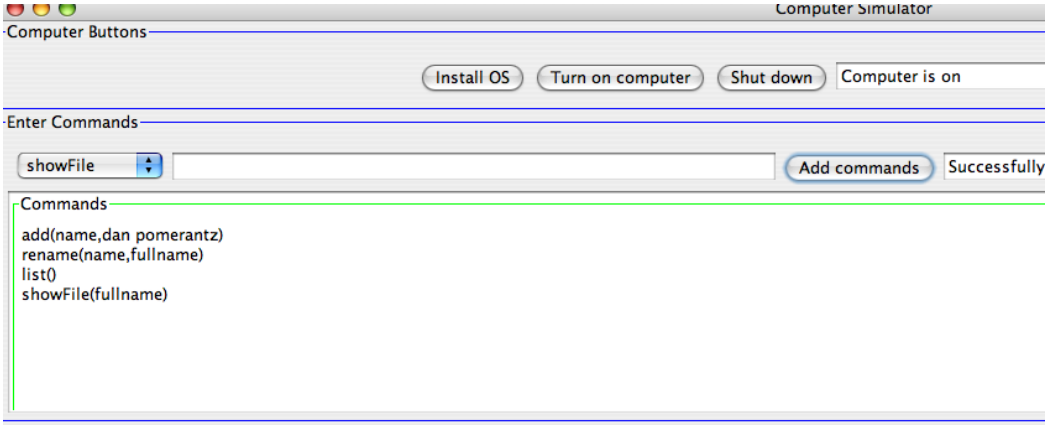

Figure 5: An example of entering many commands at the same time.

Once you have entered the commands you want to run, hit "Execute commands" to run the commands.

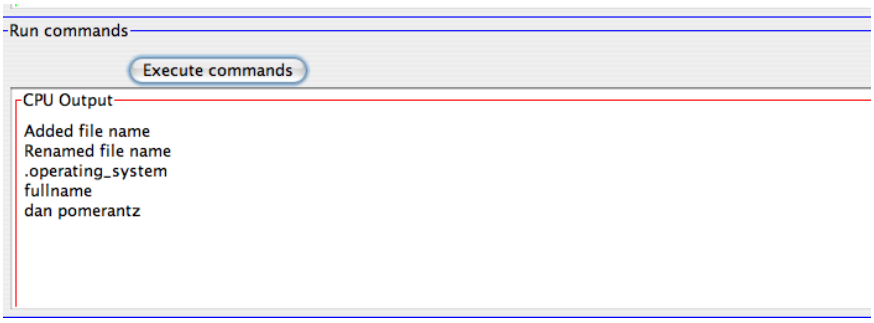

Figure 6: The output of the computer after running several instructions. Notice that it displays a list of the files present after the rename as well as the contents of the file `fullname`

Hacking into the simulated computer by exploiting a security hole!

Now for the fun part!

If you skim through the code inside of `Computer.java` you will find a method `addInstruction`. This method is called whenever you hit the "Add command" button. This method calls the method `addCommand` inside of `InstructionList.java`. Inside the `addCommand` method of `InstructionList.java`, you will find that the instruction type and arguments are added to a list, separated by a semi-colon (sort of like in Java). When the simulated computer processes the instructions, it goes through them one semi-colon at a time.

You'll also see that there is some simple error checking to make sure that the user is not modifying any file that starts with a dot, as this sort of file would be a "critical file" that can't be modified. For example, if you try to enter a command to delete the file ".operating_system" you will see an error that stops you from adding the command to the list.

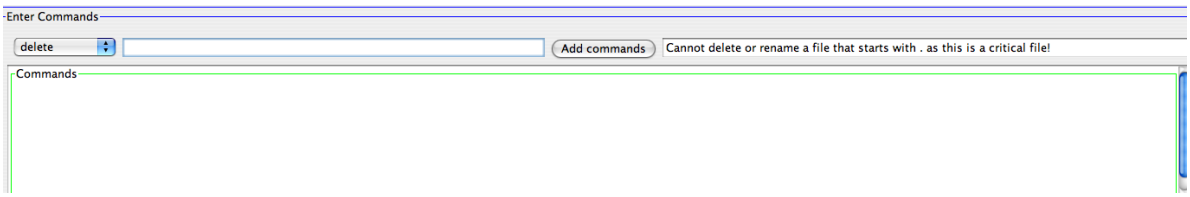

Figure 7: If you click the "Add command" button and your argument was .operating_system you will get an error as you can't delete that file.

This seems to be doing the job, but there is a very clever trick around this. Since we know that the computer likes to separate instructions via a semi-colon, you can enter a semi-colon into the command arguments itself. If you do this, you can then put a delete command in as well! See the screen shot below. The user has added an *add* instruction, and the argument is:

```
pointlessfile,data);delete(.operating_system
```

Since the command appears to be an add command of a file called "pointlessfile," the code to avoid the deletion does not detect this. However, when the computer executes these instructions, it will still find a delete.

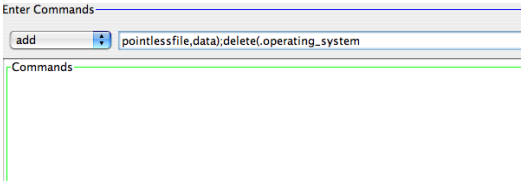

Figure 8: An example of tricking the computer into adding a delete command.

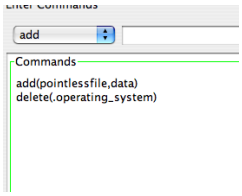

Figure 9: You can see the instruction lined up to be executed now.

Now when the user hits execute instructions, this "key" operating system file will be deleted. The next time a user tries to enter a command, there will be an error saying the operating system is corrupt (since the computer is missing the file `.operating_system`. Since the key file was deleted, you will have to reinstall the operating system, thus being forced to delete all of your data.