FA21 CSE 8B Homework 8: Virtual File System (Part 1)

Due Date: Wednesday, November 24, 11:59 PM

Learning goals:

- Build a Virtual File System using:
  - Java built-in Collections (ArrayList)
  - Inheritance
  - Interfaces
  - Abstract and Concrete Classes

NOTE: This assignment should be completed INDIVIDUALLY. Pair programming is NOT allowed for this assignment.

IMPORTANT: You should NOT have to import any additional packages to complete this assignment. Any unnecessary imports may result in failure to compile on Gradescope.

Coding Style

As always, 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. For reference, please see this mock PA with complete style. **These points are all-or-nothing, so please do NOT forget any headers or magic numbers/strings.**

Part 0: Getting started with the starter code

1. Make sure there is no problem with your Java coding environment. If there are any problems, then review Assignment 1, or come to the office/lab hours before you start Assignment 8.
2. Download the starter code.
   a. If you are working on your local machine, then you can download the starter code from Piazza → Resources → Homework → `Assignment8.zip`. Download the starter code to a directory of your choice, then extract the `zip` file. The extracted directory should contain 6 Java files: `Assignment8.java`, `Directory.java`, `File.java`, `FSComponent.java`, `Mutable.java`, and `RootDirectory.java`. Afterwards, open your terminal or command prompt, then navigate to the directory that contains those 6 Java files.

b. If you are working via UCSD Linux Cloud through your CSE 8B account, then use the commands below to copy the starter code to a new directory called `PA8,` to change your current directory to `PA8/starter`, and to print the files in the `starter` directory:

```
$ cp -r ~/../public/assignments/PA8 ~
$ cd ~/PA8/starter
$ ls
```

`ls` should print out 6 Java files: `Assignment8.java`, `Directory.java`, `File.java`, `FSComponent.java`, `Mutable.java`, and `RootDirectory.java`.

3. Try to compile the starter code (`javac *.java`), and you should expect some compilation errors.

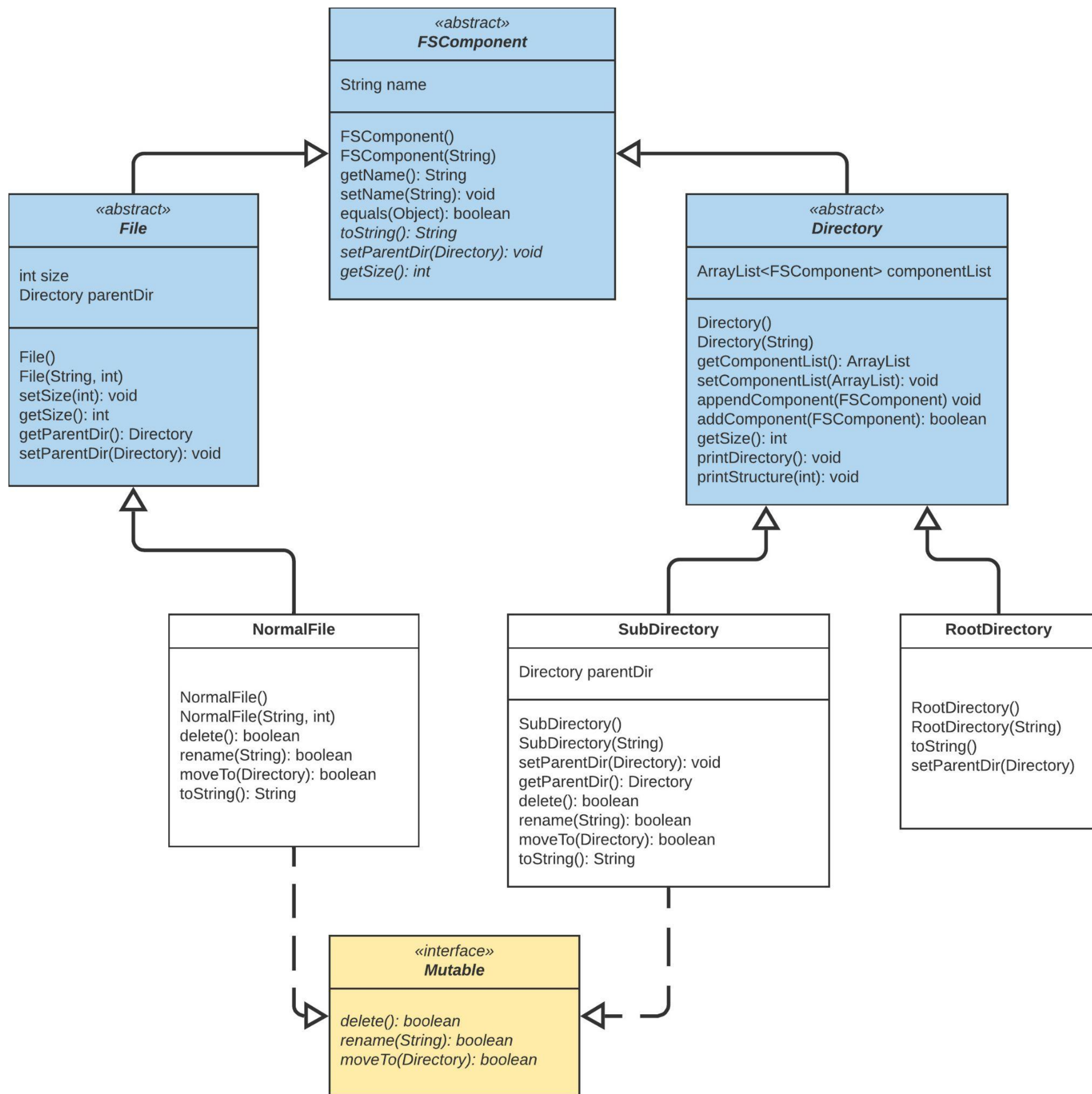
You will have to implement several methods and create new files to finish this programming assignment.

Part 1: Overview

For Assignment 8, you will implement a simplified abstraction of File System (FS). This FS will be able to support creating, deleting, renaming, and moving virtual files and directories. The image below is the UML (Unified Modeling Language) diagram for Assignment 8, showing the relationships between different classes. If the image looks blurry in the write-up, then open `PA8_UML.pdf` in your starter directory.

There are several methods and files that will only be used in Assignment 9, so you should not touch those specific methods nor should you create those files for this assignment.

Everything in Assignment 8 will be done in preparation for Assignment 9 (which will focus on recursion), so be sure to do your best in this assignment!



In the UML above, there are 3 abstract classes: `FSComponent`, `Directory`, and `File`. Likewise, we have 3 concrete classes: `NormalFile`, `SubDirectory`, and `RootDirectory`. We also have 1 interface called `Mutable`. Remember, the solid line with hollow triangle is inheritance (`extends`) and the dashed line with hollow triangle is implementation of interface (`implements`).

After finishing this assignment, this is what your file structure should look like:

```
+-- starter/
|     +-- FSComponent.java          Edit this file (WILL BE GRADED)
|     +-- File.java                 Edit this file (WILL BE GRADED)
```

```
|     +-- NormalFile.java          Create and edit (WILL BE GRADED)
|     +-- Directory.java           Edit this file (WILL BE GRADED)
|     +-- SubDirectory.java        Create and edit (WILL BE GRADED)
|     +-- RootDirectory.java       Do NOT change
|     +-- Mutable.java                  DO NOT change
|     +-- Assignment8.java         Add more tests (WILL BE GRADED)
|     +-- PA8_UML.pdf              UML Diagram
```

It is ***very important*** to organize the files as above to ensure that the provided methods will work correctly. The starter code intentionally contains compiler errors because some of the methods need to be implemented by you. You will run `javac` and `java` from within the `starter` directory after you finish implementing.

NOTE: do **NOT** change any of the methods that are implemented already.

Part 1: `FSComponent.java`

The `FSComponent` abstract class has a single instance variable `name`, getter and setter associated with `name`, and two public constructors. **All of them are implemented.** For this assignment, `FSComponent` defines two abstract methods (`toString()` and `setParentDir(Directory dir)`) that need to be overridden by its subclasses. **Do NOT change these methods**. Here is what you need to do:

1. `public boolean equals(Object obj)`
   Override the public method `equals` inherited from `Object` class. Two `FSComponent` objects are equal *if and only if* they have the same class AND their `name`s are the same. For example, a `NormalFile` instance is not equal to a `SubDirectory` instance even if they have the same `name`. **HINT:** use `.getClass()` to get the class of an object.

Part 2: `File.java`

The `File` abstract class inherits directly from the `FSComponent` abstract class. `File` has two additional instance variables:

1. `private int size`
   This is the size of the file. **NOTE:** `size` will not play any important role in PA8, but `size` will be used in PA9.

2. `private Directory parentDir`
   The parent directory of the file. In other words, `parentDir` is the directory that contains the current file.

`File.java` has pairs of setter and getter methods for each of the instance variables. File also contains two public constructors. **All methods are implemented for you *except* the following constructor that you need to implement:**

1. `public File(String name, int size)`
   Implement this constructor by initializing all instance variables including the `name` instance variable in its parent class.

Part 3: `Directory.java`

The `Directory` abstract class inherits directly from the `FSComponent` abstract class. `Directory` class has a list of `FSComponent` objects stored in `componentList`. You can think of this `componentList` as a data structure that stores all files and directories under the current directory. A no-arg constructor and a pair of setter and getter methods associated with `componentList` are implemented for you. In addition, a method called `appendComponent()` is implemented for your convenience. **HINT:** You should understand how `appendComponent()` works.

You need to complete the following methods:

1. `public Directory(String name)`
   Implement this constructor by initializing `this.componentList` to an empty `ArrayList` of `FSComponent` objects and initializing the `name` instance variable (by using the input parameter) in its parent class.

2. `public boolean addComponent(FSComponent newComp)`
   This method adds a `FSComponent` to its `componentList`. You can think of this method as adding a new file or directory to the current directory. However, there are some rules you need to follow when adding files or directories into the current directory.
   - If `newComp` is a file, then there cannot be another file under the current directory that has the same name as the name of `newComp`. If this is the case, then simply `return false`. **HINT:** Use `instanceof` to check if `newComp` is-a `File`.
   - Similarly, if `newComp` is a directory, then there cannot be another directory under the current directory that has the same name as the name of `newComp`. If this is the case, then simply `return false`. **HINT:** Use `instanceof` to check if `newComp` is-a `Directory`.
   - Otherwise, the `newComp` can be safely added to `componentList`. Simply do so by adding to the end of the `componentList`. Then, set the parentDir of `newComp` to the current directory and return true. (This is commonly referred to as ***two-way binding***, meaning that the parent object and the child object are

aware of each other and can change together). **HINT:** Look at `appendComponent()`.

3. `public void printDirectory()`

   This method will print out all files and directories under the current directory. This method should **ONLY** include elements in the current `componentList`. First, you should print the current directory's `toString()` method. Then, for each `FSComponent` in `componentList`, you should print the `FSComponent`'s `toString()` method, prepended with a tab (`"\t"`). An example is shown below:

```
Root Directory: Home
        Normal file: cat.png
        Normal file: rice.mp3
        Sub Directory: music
```

   In the example above, `"Home"` would be the current directory, and `"cat.png"`, `"rice.mp3"`, and `"music"` are the `FSComponent` objects in `componentList`. **NOTE:** there is a new-line character after every line, even the last line.

Part 4: Implementing the `Mutable` interface

As shown in the above UML diagram, there are 3 concrete classes in this assignment: `NormalFile`, `RootDirectory`, and `SubDirectory`. **Among them, `NormalFile` and `SubDirectory` are mutable (i.e., they implement the interface `Mutable`)**. This means that any instance of these two classes can call methods `moveTo()`, `rename()`, and `delete()`. **For this part of the assignment, you will have to create and edit `NormalFile.java` and `SubDirectory.java` from scratch.**

NOTE: `Mutable.java` is already fully implemented for you, so you do not have to edit `Mutable.java`.

1. `NormalFile.java`

You will have to create this file from scratch. Ensure that the full file name (including the file extension) is `NormalFile.java`.

The `NormalFile` class `extends` from the `File` abstract class (use the `extends` keyword) and `implements` the `Mutable` interface (use the `implements` keyword). Since `NormalFile` is a concrete class, `NormalFile` must override and implement all abstract

methods. **Furthermore, `NormalFile` must override the `toString()` method. You may import the ArrayList class if necessary. Here is what you need to do:**

1. `public NormalFile()`
   This is the default no-arg constructor. You do not need to initialize anything in this constructor.

2. `public NormalFile(String name, int size)`
   Implement this constructor by initializing all instance variables including the `name` and `size` instance variables in its parent class.

3. `public boolean rename(String name)`
   This method takes in a String representing the new `name` to change.
   If this file has no `parentDir`, then simply change the member variable `name` of the current file to parameter `name`, and `return true`. **HINT:** Use `setName()`.
   If this file does have a `parentDir`, then it checks whether the `parentDir` contains any ***file*** that has the same name as the parameter `name` (only those components in the parent's `componentList`). **HINT**: Use `instanceof` to check if a component is-a `File`. If there is a file that has the same name, then simply `return false`. Otherwise, if there is NO file that has the same name, then change the member variable `name` of the current file to parameter `name`, and `return true`.
   Do **NOT** forget the `@Override` keyword.

4. `public boolean delete()`
   This method removes the current object from the `componentList` of its parent and `returns true`. This method should ALWAYS `return true` because one can always delete the current file. Again, implement ***two-way binding*** by removing the file from its parent's `componentList` and setting the parent of the current file to `null`. **NOTE: you can always assume that the parent exists.**
   Do **NOT** forget the `@Override` keyword.

5. `public boolean moveTo(Directory dir)`
   Moves the current file to the designated `Directory` called `dir`. This method checks whether `dir` contains a file that has the same name as our current file (only those components in `dir`'s `componentList`). **HINT**: Use `instanceof` to check if a component is-a `File`.. If there is a file that has the same name, then simply `return false`. Otherwise, if there is NO file that has the same name, then delete the current file from its original `parentDir`, `add` this file to the passed-in `dir` using ***two-way binding***, and `return true` in the end.
   Do **NOT** forget the `@Override` keyword.

6. `public String toString()`

This method should return the string representation of the `NormalFile` object. **To ensure full compatibility with the Gradescope Autograder, you should return the following EXACTLY: `return "Normal file: " + this.getName();`.** You do not need to worry about magic strings for this particular method.
Do **NOT** forget the `@Override` keyword.

2. SubDirectory.java

You will have to create this file from scratch. Ensure that the full file name (including the file extension) is `SubDirectory.java`.

The `SubDirectory` class `extends` from the `Directory` abstract class (use the `extends` keyword) and implements `Mutable` interface (use the implements keyword). As seen in the UML, `SubDirectory` has an additional instance variable `parentDir` compared to the `Directory` class. Since `SubDirectory` is a concrete class, `SubDirectory` must override and implement all abstract methods. **Furthermore, `SubDirectory` must override the `toString()` and `setParentDir(Directory dir)` methods. You may import the ArrayList class if necessary. Here is what you need to do:**

1. `public SubDirectory()`
   This is the default no-arg constructor. You do not need to initialize anything in this constructor.

2. `public SubDirectory(String name)`
   Implement this constructor by initializing the instance variable `name` in its parent class.

3. `public void setParentDir(Directory parentDir)`
   This is a setter method. Simply set `this.parentDir = parentDir;`.

4. `public Directory getParentDir()`
   This is a getter method. Simply `return this.parentDir;`.

5. `public boolean rename(String name)`
   Similar to `rename()` in `NormalFile`, this method takes in a `String` representing the new name to change.
   If this directory has no `parentDir`, then simply the member variable `name` of the current file to parameter `name`. **HINT:** Use `setName()`.
   If this directory has a `parentDir`, then this method checks whether the `parentDir` contains any directory that has the same name as the parameter `name` (only those components in `parentDir`'s `componentList`). **HINT:** Use `instanceof` to check if a component is-a `Directory`. If there is a directory that has the same name, then simply `return false`. Otherwise, if there is NOT a directory that has the same name, then

change the member variable `name` of the current file to parameter `name`, and `return true`.
Do **NOT** forget the `@Override` keyword.

6. `public boolean delete()`
   Similar to `delete()` in `NormalFile`, this method removes the current object from the `componentList` of its parent and `returns true`. This method should ALWAYS `return true` because one can always delete the current directory. Again, implement ***two-way binding*** by removing the directory from its parent's `componentList` and setting the parent of the current file to `null`. **NOTE: you can always assume that the parent exists.**
   Do **NOT** forget the `@Override` keyword.

7. `public boolean moveTo(Directory dir)`
   Similar to `moveTo()` in `NormalFile`, this method checks whether `dir` contains a directory that has the same name as our current directory (only those components in `dir`'s `componentList`). **HINT:** Use `instanceof` to check if a component is-a `Directory`. If there is a directory that has the same name, then simply `return false`. Otherwise, if there is NO directory that has the same name, then delete the current directory from its original `parentDir`, `add` this directory to the passed-in `dir` using ***two-way binding***, and `return true` in the end.
   Do **NOT** forget the `@Override` keyword.

8. `public String toString()`
   This method should return the string representation of the `SubDirectory` object. **To ensure full compatibility with the Gradescope Autograder, you should return the following EXACTLY: `return "Sub Directory: " + this.getName();`.** You do not need to worry about magic strings for this particular method.
   Do **NOT** forget the `@Override` keyword.

Part 5: `RootDirectory.java`

This file is fully implemented for you in the starter code. The object instance created by this class can only be the outmost layer in a file system. Please take a look at this file and understand what `RootDirectory` does.

Part 6: `Assignment8.java`

Inside `Assignment8.java`, we provide one testing method called `testOne()`. We also have `unitTests()` which calls `testOne()`. Because we only provide one testing method, you are encouraged to create as many testing methods as you think to be necessary to cover all the edge cases. **To get full credit, create *at least 5* extra tester *methods* in `Assignment8.java`. In other words, we expect to see a total of at least 6 tester methods being called by `unitTests()`.** There are some comments above `unitTests()` suggesting what to test. We also suggest making some print messages in each of your test cases so that you will know which test case is failing. The `unitTests()` method should `return true` only when all the test cases are passed. Otherwise, you should `return false`. **NOTE: The Gradescope Autograder will be reading the output from `printDirectory()` to ensure correctness, so make sure that you do NOT leave any debug statements inside `printDirectory()`. Otherwise, it is OK if your tests print to standard output.**

Submission

VERY IMPORTANT: Please follow the instructions below carefully and make the exact submission format.

1. Go to Gradescope via Canvas and click on `PA8`.
2. Click the DRAG & DROP section and directly select the 6 required files (`Assignment8.java`, `Directory.java`, `File.java`, `FSComponent.java`, `NormalFile.java`, and `SubDirectory.java`). Drag & drop is fine. **Please make sure you don't submit a zip. Make sure the filenames are correct.**
3. **You can resubmit an unlimited number of times before the due date.** Your score will depend on your final submission, even if your former submissions have a higher score.
4. The autograder is for grading your uploaded files automatically. Make sure your code can compile on Gradescope. You will receive 0 on the Autograder section if your code does not compile correctly on Gradescope.

   NOTE: The Gradescope Autograder you see is a minimal autograder. For this particular assignment, the Gradescope Autograder will only show the compilation results and the results of a few testers. After the assignment deadline, a thorough Autograder will be used to determine the final grade of the assignment. **Thus, to ensure that you would receive full points from the thorough Autograder, it is your job to extensively test your code for correctness via `unitTests`.**

Q&A

Is it possible that an object instantiated somewhere in the program is-a FSComponent but is none of the concrete class objects (NormalFile, RootDirectory, and Subdirectory)?
This is not possible because only concrete classes can be instantiated. For this assignment, since there are only three concrete classes inherited from `FSComponent`, any object that is-a `FSComponent` must have an actual type of one of the three concrete classes.

What if I move the directory to itself by calling something like dir.moveTo(dir)?
You do not need to consider the edge case where a directory is moved to itself. The caller is always different from the parameter.

Can a directory contain a file and a subdirectory with the same name?
Yes, the only conflict is when two files have the same name or two subdirectories have the same name under the same directory.

Can a directory contain a RootDirectory?
No. The `RootDirectory` can only be the outmost directory.

Can SubDirectory be the outmost directory?
Yes. `SubDirectory` can exist on its own and become the outmost directory.

Do we need to consider the case when the root directory or subdirectory does not contain a single file or subdirectory？
Yes, this is certainly possible.

Does calling delete() remove the object from the memory?
Calling `delete()` only removes a file or directory from its parent directory. `delete()` does **NOT** remove the object from the memory. In fact, as a Java programmer, we have no control over memory.

What if I call delete() on the component that does not have a parent (e.g. already been deleted)?
We will not do something like that.

Can the same object instance appear multiple times under the structure of a Directory?
No. All object instances are unique.