# CSE 8B Fall 2022
# Assignment 8

## Abstract Classes & Interfaces

Due: Wednesday, November 23, 11:59 PM

Hi again! Be sure to start this assignment as EARLY as possible! You got this!

**Learning goals:**

- Apply knowledge of Abstract and Concrete Classes, Interfaces, and Inheritance by building a Virtual File System.

**NOTE: This programming assignment must be done individually. Paired programming is NOT allowed for this assignment.**

**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.**

**Because this assignment contains a lot of files, we will only be grading a random subset of your files for Coding Style. However, it is still your responsibility to follow proper CSE 8B Coding Style Guidelines for EVERY single file.**

---

## 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.

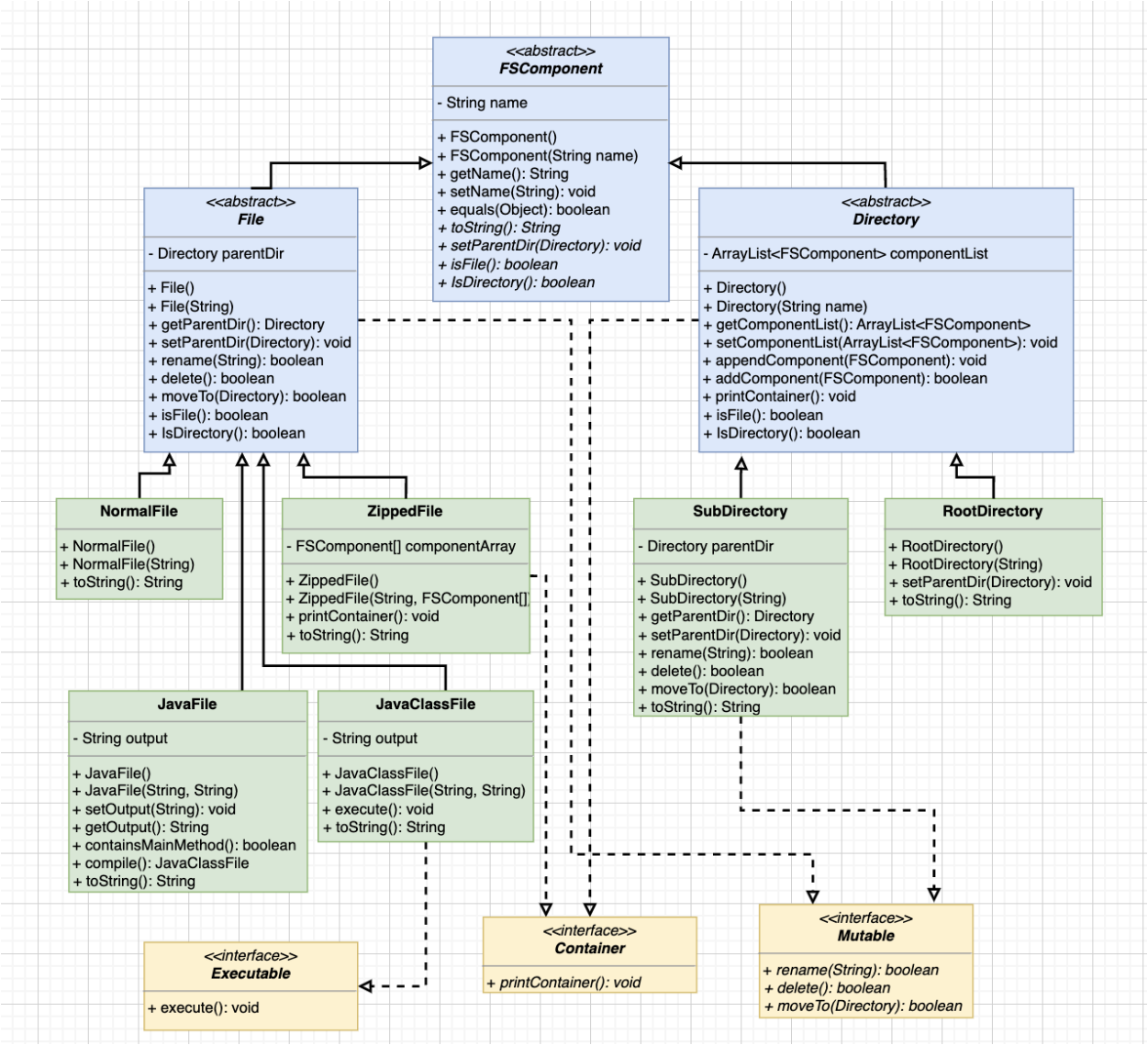
---

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

1. Make sure there is no problem with your Java software development environment. If there is any, then review Assignment 1, or come to the office/lab hours before you start Assignment 8.
2. First, navigate to the `cse8b` folder that you have created in Assignment 1 and create a new folder titled `assignment8`
3. Download the starter code. You can download the starter code from Piazza → Resources → Homework → `assignment8.zip`. The starter code should contain seven files: `Assignment8.java`, `Directory.java`, `File.java`, `FSComponent.java`, `Mutable.java`, `RootDirectory.java`, and `PA8_UML.pdf`. Place the starter code within the `assignment8` folder that you have just created
4. Compile the starter code within the `assignment8` folder. You can compile all files using the single command `javac *.java` and you should get a series of compiler errors since you have not implemented the classes yet. The objective of this assignment is to get the classes working by implementing the class methods and testing them.
5. You will be turning in all of the original files included in `assignment8.zip` and more.

# Part 1: Overview

For this assignment, 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 `assignment8` directory.




In the UML diagram above, there are 3 abstract classes: `FSComponent`, `Directory`, and `File`. Likewise, we have 6 concrete classes: `NormalFile`, `ZippedFile`, `JavaFile`, `JavaClassFile`, `SubDirectory`, and `RootDirectory`. We also have 3 interfaces called `Mutable`, `Container`, and `Executable`. 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)**
| **+-- ZippedFile.java** **Create and edit (WILL BE GRADED)**

| | | |
|---|---|---|
| | **+-- Container.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** |
| | **+-- JavaFile.java** | **Create and edit (WILL BE GRADED)** |
| | **+-- Executable.java** | **Create and edit (WILL BE GRADED)** |
| | **+-- JavaClassFile.java** | **Create and edit (WILL BE GRADED)** |
| | **+-- 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. When you first download it, 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 `assignment8` directory after you finish implementing.

**NOTE:** do NOT change any of the methods that are implemented already. You must implement and comment on everything with a "TODO". Do NOT forget to adhere to the CSE 8B style guidelines.

**NOTE:** We will not be giving partial credit for incorrect output. Please make sure that the format of your output matches **EXACTLY** with what's expected.

If you have any questions regarding implementing/testing, please first check the Q&A at the bottom of this document!

**Be sure to compile your code often**, so that you can catch compile errors early on! Recall, to compile multiple Java files, use:

```
> javac *.java
```

---

## Part 2: `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 four abstract methods (`toString()`, `setParentDir(Directory dir)`, `public abstract boolean isFile()`, `public abstract boolean isDirectory()`) 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 names 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 3: `File.java`

The `File` abstract class inherits directly from the `FSComponent` abstract class. `File` has one instance variable:

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

`File` has pairs of setter and getter methods for each of the instance variables. `File` also contains two public constructors. Later in the assignment, you will be writing more methods in `File`. For now, **just implement the following constructor and two methods:**

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

3. **`public boolean isFile()`**
   Returns `true` only if this `FSComponent` is a `File` (which in this case, for us writing code in the `File` class, it always **is**!)

4. **`public boolean isDirectory()`**
   Returns `true` only if this `FSComponent` is a `Directory` (which in this case, for us writing code in the `File` class, it always is **not**!)

---

## Part 4: `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.

**For now, you need to complete the following methods:**

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

2. **`public boolean isFile()`**
   Returns `true` only if this `FSComponent` is a `File` (which in this case, for us writing code in the `Directory` class, it always is **not**!!)

3. **`public boolean isDirectory()`**
   Returns `true` only if this `FSComponent` is a `Directory` (which in this case, for us writing code in the `Directory` class, it always **is**!)

4. **`public boolean addComponent(FSComponent newComp)`**
   This method adds an `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 `isFile` to check if `newComp` is a `File`.
     - Note: name is a `private String` member declared in the `FSComponent` class. How can you access this private member from inside the `Directory` class?
   - 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 `isDirectory()` 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). Once appended safely, **return** `true`. **HINT:** Look at `appendComponent()`.

---

# Part 5: Implementing the `Mutable` interface

The `Mutable` interface will be implemented by all classes that can be “mutated” (or “changed”) at any point in time.

As shown in the above UML diagram, there are 2 classes in this assignment that **implement the interface Mutable**. They are `File` and `SubDirectory`.

This means that any instance of classes that inherit the abstract class File and instances of the `SubDirectory` class can call methods `moveTo()`, `rename()`, and `delete()`.

**For this part of the assignment, you will have to write brand new methods in `File.java` from scratch as well as create and edit `SubDirectory.java` from scratch.**

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

## 1. `File.java`

First, make sure that the `File` abstract class implements the `Mutable` interface (use the `implements` keyword). Now we need to implement the following methods that were declared in the `Mutable` interface, giving functionality for us to “mutate” our file. Remember that you will need to write **brand new methods in `File.java` from scratch.**

1. **`public boolean rename(String name)`**
   This method takes in a `String` representing the new name to change to. There are a few different cases:
   - If this file **has no** `parentDir`, then simply set the member variable name of the current file to parameter `name`, and **return** `true`.
   - 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` (checking only components in the `parentDir`'s `componentList`). If we do find a file with the same name as the parameter `name`, then **return** `false` since renaming isn’t safe.
     **HINT**: Use `isFile()` 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 the parameter name, and **return** `true`.

   Don’t forget the `@Override` annotation!!

2. **`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.**
Don't forget the `@Override` annotation!!

3. **`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 `isFile()` 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` at the end.

   Don't forget the `@Override` annotation!!

## 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 the `Mutable` interface (use the `implements` keyword). As seen in the UML, `SubDirectory` should have a private member variable `parentDir`. 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 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 the `parentDir` member variable to the `parentDir` parameter.

4. **`public Directory getParentDir()`**

   This is a getter method. Simply return the `parentDir` member variable.

5. **`public boolean rename(String name)`**

Similar to `rename()` in `File`, this method takes in a `String` representing the new name to change to.

- If this directory has no `parentDir`, then simply set the member variable name of the current directory to parameter `name`. Then, **return** `true`.
- 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 `isDirectory()` 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 directory to parameter name, and **return** `true`.

Don't forget the `@Override` annotation!!

### 6. `public boolean delete()`

Similar to `delete()` in `File`, 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 directory to `null`. **NOTE: you can always assume that the parent exists.**

Don't forget the `@Override` annotation!!

### 7. `public boolean moveTo(Directory dir)`

Similar to `moveTo()` in `File`, 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 `isDirectory()` 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.

Don't forget the `@Override` annotation!!

### 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();
```

Don't forget the `@Override` annotation!!

---

## Part 6: `NormalFile.java`

You will have to create this (relatively short) 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).

Since `NormalFile` is a concrete class, it must override and implement all unimplemented abstract methods. Recall the abstract methods it gets from the `Mutable` interface (`moveTo()`, `rename()`, and `delete()`) were already overridden in its parent class, `File` in Part 5. The starter code has already overridden the abstract method `setParentDir()` in `File`. Therefore, `NormalFile` must only override the `toString()` method! Here is what you need to do:

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

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

3. **`public String toString()`**
   This method returns 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();
   ```
   Don't forget the `@Override` annotation!!

---

## Part 7: `JavaFile.java`

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

The `JavaFile` class extends from the `File` abstract class (use the `extends` keyword). `JavaFile` is a class that represents a typical Java file that you would write in CSE 8B. However, for simplicity, we are only dealing with two types of Java files for the purpose of this assignment.

1. Has a main method that can be compiled to an executable file (`JavaClassFile`) which will print an output. In this case, its output field will be non-null

2. Does not have a main method and **cannot** be compiled into a `JavaClassFile`. In this case, the output field is `null`.

Although in reality, you can definitely compile Java files that do not have a main method into their respective `.class` files, we are assuming for the purpose of this assignment that we can't.

Since `JavaFile` is a concrete class, it must override and implement all unimplemented abstract methods. Recall that the abstract methods it gets from the `Mutable` interface (`moveTo()`, `rename()`, and `delete()`) were already overridden in its parent class, File in Part 5. The starter code has already overridden the abstract method `setParentDir()` in `File`. Here is what you need to do:

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

2. **`public JavaFile(String name, String output)`**
   Implement this constructor by initializing the `name` instance variable in its parent class. Then set the `output` instance variable to the `output` passed in as an argument to the constructor. You can assume that the name that is passed into the constructor will always end with `.java`. It is possible that `output` is null. In this case, still set the instance variable to `null`.

3. **`public void setOutput(String output)`**
   Setter that should set the `output` field to the argument passed into the method

4. **`public String getOutput()`**
   Getter that should return the `output` field

5. **`public boolean containsMainMethod()`**
   Returns `true` if the output field is not `null`, otherwise `false`

6. **`public JavaClassFile compile()`**
   Method that "compiles" a `JavaFile` into a `JavaClassFile` and returns it. You should first check if the output field is `null`. If that is the case, then return `null`. Otherwise, create an instance of the `JavaClassFile` class and pass the name of current `JavaFile` and output to the `JavaClassFile` constructor. However, instead of passing the name that ends with `.java` you must modify it so that it ends with `.class` before you pass it into the `JavaClassFile` constructor.

7. **`public String toString()`**

This method should return the string representation of the `JavaFile` object. **To ensure full compatibility with the Gradescope Autograder, you should return the following EXACTLY:**

```
return "JavaFile: " + this.getName();
```

Don't forget the `@Override` annotation!!

---

# Part 8: Implementing the `Executable` interface

First, you will need to **create and edit** a (really, really short) file called `Exectuable.java` from scratch. This file will contain the `Executable` interface, which should have the following abstract method:

```
void execute()
```

Look at `Mutable.java` (which contains the `Mutable` interface) for help!

The above `Executable` interface will be used for all of our classes that are able to be "executed" like the `JavaClassFiles`. As shown in the UML, the `JavaClassFile` class will implement the `Executable` interface.

## 1. `JavaClassFile.java`

You will need to **create and edit** a file called `JavaClassFile.java` from scratch. Ensure that the full file name (including the file extension) is `JavaClassFile.java`**.**

The `JavaClassFile` concrete class extends from the `File` abstract class (use the `extends` keyword) and also implements the `Executable` interface.

LIke `NormalFile`, `JavaClassFile` will have to override the `toString()` method. Here is what you need to do:

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

2. **`public JavaClassFile(String name, String output)`**
   Implement this constructor by initializing the `name` instance variable in its parent class and setting its `output` field to the `output` argument passed into the constructor

3. **`public void execute()`**
   This method prints out the value of the `output` field on a new line using `System.out.println()`
   Don't forget the `@Override` annotation!!

4. **`public String toString()`**
   This method returns the string representation of the `JavaClassFile` object. **To ensure full compatibility with the Gradescope Autograder, you should return the following EXACTLY:**
   ```
   return "Java class file: " + this.getName();
   ```
   Don't forget the `@Override` annotation!!

---

# Part 9: Implementing the `Container` interface

First, you will need to **create and edit** a (really, really short) file called `Container.java` from scratch. This file will contain the `Container` interface, which should have the following abstract method:

```
void printContainer()
```

Look at `Mutable.java` (which contains the `Mutable` interface) for help!

The above `Container` interface will be used for all of our classes that "contain" `FSComponents` in them. As shown in the UML, the `Directory` and `ZippedFile` classes will implement the `Container` interface.

## 1. `Directory.java`

First, make the `Directory` class implement the `Container` interface. Then, implement the `printContainer()` method in the `Directory` class!

1. **`public void printContainer()`**

   This method will print out all files and directories **directly** under the current directory. This method should **ONLY** include elements in the current `componentList`. First, you should print out the String from 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. Please make sure that the format of your output matches exactly with what's expected.

Don't forget the `@Override` annotation!!

## 2. `ZippedFile.java`

You will need to **create and edit** a file called `ZippedFile.java` from scratch. Ensure that the full file name (including the file extension) is `ZippedFile.java`**.**

The `ZippedFile` concrete class extends from the `File` abstract class (use the `extends` keyword) and also implements the `Container` interface.

Because `ZippedFile` is essentially a directory that has been compressed, and since its contents shouldn't be changed at any point in time, `ZippedFile` somewhat resembles the `Directory` class but has an **array** of `FSComponent` objects rather than an `ArrayList` like `Directory` does (its `componentList`). You will need to declare the following member variable inside of the `ZippedFile` class:

```
private FSComponent[] componentArray;
```

LIke `NormalFile`, `ZippedFile` will have to override the `toString()` method, but it also must override the `printContainer()` method it inherits from implementing the `Container` interface. Here is what you need to do:

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

2. **`public ZippedFile(String name, FSComponent[] componentArray)`**

Implement this constructor by initializing the `name` instance variable in its parent class. For this constructor, there's one special thing about initializing names: zipped file names must end in **`.zip`**!! Check if the parameter name ends with `.zip` - if not, append `.zip` to the end of name before setting it. Otherwise, initialize the name variable with the parameter as is. (Hint: use the `endsWith` method. You may also want to use `setName()`).

Then, set the `componentArray` member variable to the `componentArray` parameter.

> (**A note for testing later:** Recall that all unwanted changes made to the `componentArray` outside of the `ZippedFile` class will still be reflected by the `componentArray` member variable. This is just because setting the `componentArray` member variable to the `componentArray` parameter only makes both references refer to the same, singular array object, not two different arrays. Keep this in mind so you don't accidentally change the array referenced by `componentArray` and create confusing situations!)

3. **`public void printContainer()`**
   This method prints out the result of the `toString()` method of the `ZippedFile` object, immediately followed by this exact String:
   `" " + componentArray.length + " FSComponents"`
   All of the printing should be done in **one line** that ends with a **newline character**.

For example, if we have a `ZippedFile` object with the name `NewZip.zip` and if the object has one `FSComponent` in its `componentArray`, the following output would be printed:

```
Zipped file: NewZip.zip 1 FSComponents
```

Please make sure your output matches **exactly** for full points.
Don't forget the `@Override` annotation!!

4. **`public String toString()`**
   This method returns the string representation of the `ZippedFile` object. **To ensure full compatibility with the Gradescope Autograder, return the following EXACTLY:**

   ```
   return "Zipped file: " + this.getName();
   ```

   Don't forget the `@Override` annotation!!

---

## Part 10: `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 11: Compile, Run and UnitTest Your Code (10 points)

First, read the Q&A for other specifications on what are some test cases that we will **not** be testing.

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 `Assignment8` class.**

We already provide one testing method called `testOne()`. We have written code in `unitTests()` that 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 more 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. Each of your tests must be similar in scope and scale to the example test case that we have provided in order to get full credit. 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`.

**IMPORTANT NOTE:** The Gradescope Autograder will be reading the output from `printContainer()` to ensure correctness, so make sure that you **do NOT** leave any other, unnecessary print statements inside the method `printContainer()`. Otherwise, it is OK if your unit tests print to standard output.

Remember that it is OK to have magic numbers in your unit tests.

You can compile 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 Assignment8
```

---

## 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 8.
2. Click the DRAG & DROP section and directly select the **THIRTEEN** required files: `FSComponent.java`, `File.java`, `NormalFile.java`, `JavaFile.java`, `JavaClassFile.java`, `ZippedFile.java`, `Container.java`, `Directory.java`, `SubDirectory.java`, `RootDirectory.java`, `Mutable.java`, `Executable.java`, and `Assignment8.java`. Drag & drop is fine. Please make sure you don't submit a zip, just the separate 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. If you have any questions, feel free to post on Piazza!

# Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

Upload  GitHub  Bitbucket

Add files via Drag & Drop or Browse Files.

| NAME | SIZE | PROGRESS |
|---|---|---|
| Assignment8.java | **2.9** KB | |
| Container.java | **0.3** KB | |
| Directory.java | **2.7** KB | |
| Executable.java | **0.2** KB | |
| File.java | **2.6** KB | |
| FSComponent.java | **1.5** KB | |
| JavaClassFile.java | **1** KB | |
| JavaFile.java | **1.2** KB | |
| Mutable.java | **0.3** KB | |
| NormalFile.java | **0.6** KB | |
| RootDirectory.java | **1.7** KB | |

STUDENT NAME (OPTIONAL)

Enter student name

Upload  Cancel

SubDirectory.java **2.9** KB

ZippedFile.java **1.2** KB

STUDENT NAME (OPTIONAL)

Enter student name

Upload Cancel

# Q&A

*Is it possible that an object instantiated somewhere in the program is-a FSComponent but is none of the concrete class objects?*
This is not possible because only concrete classes can be instantiated. Any object that is-a FSComponent must have an actual type of one of the 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.