CSE 8B Spring 2023
Assignment 7

Abstract Classes, Text IO, and Exception Handling

Due: Thursday, June 1 11:59 PM

Learning goals:

- Apply knowledge of Abstract and Concrete Classes, Text IO, and Exception Handling in Java

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.

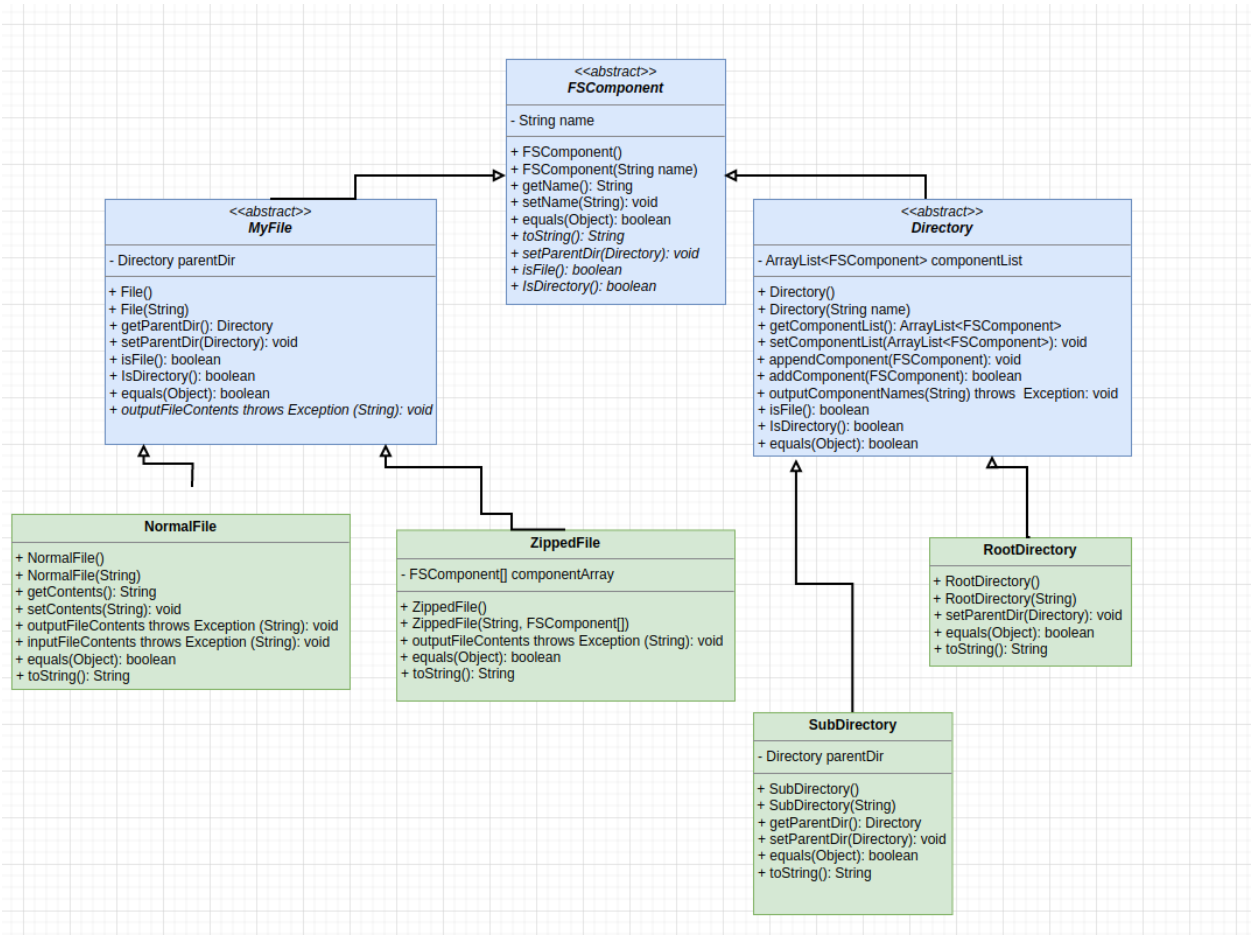
Part 0: Getting started with the starter code (0 points)

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 7.
2. First, navigate to the `cse8b` folder you created in Assignment 1 and create a new folder named `assignment7`
3. Download the starter code. You can download the starter code from Piazza → Resources → Homework → `assignment7.zip`. The starter code contains six files: `Assignment7.java`, `Directory.java`, `MyFile.java`, `FSComponent.java`, `PA7_UML.pdf`, and `RootDirectory.java`. Place the starter code within the `assignment7` folder you just created.
4. Compile the starter code within the `assignment7` 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. <u>You will be turning in all of the original `.java` files included in `assignment7.zip` and more.</u>

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 Unified Modeling Language (UML) diagram for Assignment 7, showing the relationships between different classes. If the image looks blurry in the write-up, then open `PA7_UML.pdf` in your `assignment7` directory.



In the UML diagram above, there are 3 abstract classes: `FSComponent`, `Directory`, and `MyFile`. Likewise, we have 4 concrete classes: `NormalFile`, `ZippedFile`, `SubDirectory`, and `RootDirectory`. Remember, the solid line with hollow triangle represents inheritance (extends).

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

+-- assignment7/

```
|       +-- FSComponent.java        Edit this file (WILL BE GRADED)
|       +-- MyFile.java             Edit this file (WILL BE GRADED)
|       +-- NormalFile.java         Create and edit (WILL BE GRADED)
|       +-- ZippedFile.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
|       +-- Assignment7.java        Add more tests (WILL BE GRADED)
|       +-- PA7_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 `assignment7` directory after you finish implementing.

NOTE: do NOT change any of the methods that are implemented already. 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 protected constructors. **All of them are implemented.** Additionally, `FSComponent` defines four abstract methods (`setParentDir(Directory dir)`, `isFile()`, and `isDirectory()`) that need to be implemented 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. An `FSComponent` object is equal to another object if that object is of type FSComponent and if they have the same names. Otherwise return false.

HINT: use the instanceof operator

Part 3: MyFile.java

The `MyFile` abstract class inherits directly from the `FSComponent` abstract class. `MyFile` 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.

`MyFile` starter code has setter and getter methods for this instance variable already implemented. `MyFile` also contains two protected constructors; the no-arg constructor is implemented for you. **Do NOT change the constructor and other methods implemented in the starter code.** Later in the assignment, you will be writing more methods in `MyFile`. For now, **just implement the following constructor and other methods:**

1. **`protected MyFile(String name)`**
   Implement this constructor by initializing the `name` instance variable in its superclass.

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

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

4. **`public boolean equals(Object obj)`**
   Override the parent's equals method. A `MyFile` object is equal to another object if its parent class's `equals()` method returns `true` AND if this `MyFile` object has the same `parentDir` as `obj`'s `parentDir` by reference. In other words, you must call `super.equals()`, check their type equality, then check if these two objects' `parentDir` are equal by reference.

HINT: use the instanceof operator

5. **`public abstract void outputFileContents(String outputFileName)`**
   Declare this method **without** a body. This method will eventually need to be overridden by the concrete class that extends from `MyFile`. (More on this later)

Part 4: 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 `MyFile` abstract class (use the `extends` keyword) and contains one instance variable:

1. **`private String contents`**
   A `String` variable that represents the contents of the file.

Note the concrete class `NormalFile` extends the abstract class `MyFile` (which contains the abstract method `outputFileContents()`), which extends the abstract class `FSComponent` (which contains the abstract methods `setParentDir()`, `isFile()`, and `isDirectory()`). The starter code already implemented the abstract method `setParentDir()` in `MyFile`, and you implemented `isFile()` and `isDirectory()` in `MyFile`. The only remaining abstract method is `outputFileContents()`, which you will implement in `NormalFile`. Here is all of 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, String contents)`**
   Implement this constructor by initializing the `name` instance variable in its superclass. Then, set the `contents` instance variable.

3. **`public String getContents()`**
   Getter method for `contents`. Simply return `contents`.

4. **`public void setContents(String contents)`**
   Setter method for `contents`. Simply set the instance variable to the (local variable) `contents` that was passed in.

5. **`public void outputFileContents(String outputFileName) throws Exception`**

This method must write the `contents` field to a file in the current directory, where the file has the same name as the string outputFileName. If `contents` is null or empty, then you must `throw` an instance of the `Exception` class with the message `"Empty file contents!"` (hint: create the instance using `new Exception(String message)`); otherwise, write `contents` **and then terminate the line** to the output file using PrintWriter. If a file named outputFileName already exists in the current directory, then overwrite the original file.

If an IOException is thrown during this process, then you must handle it gracefully within this method by simply printing out the IO exception message (hint: use `getMessage()`) **and then terminate the line** to standard output.

6. **`public void inputFileContents(String inputFileName) throws Exception`**
   This method must read in all contents from a file into the instance variable `contents`, where the file has the same name as the string inputFIleName.

   If an IOException is thrown during this process, then you must handle it gracefully within this method by simply printing out the IO exception message (hint: use `getMessage()`) **and then terminate the line** to standard output.

7. **`public boolean equals(Object obj)`**
   Override the `equals()` method. A `NormalFile` object is equal to another object if its parent class's `equals()` method returns `true` AND its instance variable `contents` represent the same sequence of characters `obj`'s contents. In other words, you must call `super.equals()`, check their type equality, then check if these two objects' contents are equal.

   HINT: use the instanceof operator

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

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

Part 5: ZippedFile.java

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

The `ZippedFile` class extends from the `MyFile` abstract class (use the `extends` keyword).

Because `ZippedFile` is essentially a directory that has been compressed, and since its contents must not be changed at any point in time, `ZippedFile` somewhat resembles the `Directory` class (see below) 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 implement `outputFileContents()` and override the `equals()` and `toString()` methods. Here is all of 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 setting the `name` instance variable in its grandparent class. For this constructor, there's one special thing: zipped file names must end in **`.zip`**!! Check if the input variable `name` ends with `.zip` - if not, append `.zip` to the end of `name` before initializing setting the grandparent's instance variable. Otherwise, initialize the grandparent's `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 outputFileContents(String outputFileName) throws Exception`**
   This method must write each of the `name`s of the `FSComponent`s stored in `componentArray` on a new line in the file named `outputFileName` within the current directory. If `componentArray` is `null` or its length is `0`, then you must `throw` an instance of the `Exception` class with the message `"Empty file contents!"`; otherwise, use `PrintWriter` to write the file. If the file named `outputFileName` already exists in the current directory, then overwrite the original file. For example, if `componentArray` has length `2` and has two `FSComponents` whose names are `"Hello.txt"` and

`"Person.java"`, a file named `outputFileName` in the current directory must contain the following contents after the call to this method:

```
Hello.txt
Person.java
```

More specifically, the file must contain the contents: `"Hello.txt\nPerson.java\n"`.

If an `IOException` is thrown during this process, then you must handle it gracefully within this method by simply printing out the IO exception message (hint: use `getMessage()`) **and then terminate the line** to standard output.

4. **`public boolean equals(Object obj)`**
   Override the `equals()` method. A `ZippedFile` object is equal to another object if its parent class's `equals()` method returns `true` AND if both `componentArray` contains the exact same elements (by reference). If one is `null` and the other is not, then return `false`. If both are `null`, then return `true`. Otherwise, compare the elements by reference. Make sure to check for edge cases (e.g. difference lengths). Like before, you should be calling the parent's `equals()` method first, check their type equality, and then compare their `componentArray`.

   HINT: use the instanceof operator

5. **`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();`

Part 6: `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 the methods `getComponentList()`, `setComponentList()`, and `appendComponent()` are implemented for you. **Do NOT change the constructor and other methods implemented in the starter code. HINT:** 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 My`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()`.

5. **`public void outputComponentNames(String outputFileName) throws Exception`**
   This method must write each of the `names` of the `FSComponents` stored in `componentList` on a new line in the file named `outputFileName` within the current directory. If `componentList` is `null` or its length is `0`, then you must `throw` an instance of the `Exception` class with the error message `"Empty directory contents!"`; otherwise, use `PrintWriter` to write the file. If the file named `outputFileName` already

exists in the current directory, then overwrite the original file. For example, if `componentList` has length `2` and has two `FSComponents` whose names are `"Hello.txt"` and `"Person.java"`, a file named `outputFileName` in the current directory must contain the following contents after the call to this method:

```
Hello.txt
Person.java
```

More specifically, the file must contain the contents: `"Hello.txt\nPerson.java\n"`.

If an `IOException` is thrown during this process, then you must handle it gracefully within this method by simply printing out the IO exception message (hint: use `getMessage()`) **and then terminate the line** to standard output.

6. **`public boolean equals(Object obj)`**
   Override the `equals()` method. A `Directory` object is equal to another object if its parent class's `equals()` method returns `true` AND if both `componentList` contains the exact same elements (by reference). If one is `null` and the other is not, then return `false`. If both are `null`, then return `true`. Make sure to check for edge cases (e.g. difference sizes). Otherwise, compare the elements by reference. Like before, you should be calling the parent's `equals()`operator first, perform type checking, and then compare `componentList`.

   HINT: use the instanceof operator

Part 7: `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 concrete class `SubDirectory` extends the abstract class `Directory`, which extends the abstract class `FSComponent` (which contains the abstract methods `setParentDir()`, `isFile()`, and `isDirectory()`). You implemented `isFile()` and `isDirectory()` in `Directory`. The only remaining abstract method is `setParentDir()`, which you will implement in `SubDirectory`.

As seen in the UML, `SubDirectory` must have a private member variable `parentDir`. Here is all of 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 superclass.

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 equals(Object obj)`**

   Override the `equals()` method. A `SubDirectory` object is equal to another object if its parent class's `equals()` method returns `true` AND if this `SubDirectory` object has the same `parentDir` as `obj`'s `parentDir` by reference. Make sure to check edge cases where the `parentDir` can be `null`. Once again, you should be calling the parent's `equals()` method first, check for type equality, and then compare `parentDir`.

   HINT: use the instanceof operator

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

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

   ```
   return "Sub directory: " + this.getName();
   ```

Part 8: `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.

Note: For all methods that override it's parent method, you must use the `@Override` annotation

Part 9: 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 `Assignment7` 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 `Assignment7.java`**. In other words, we expect to see a **total of at least 6 tester methods** that test a ***variety*** of situations being called by `unitTests()`. If we do not see a ***variety*** of tests that are equivalent in scope to the one provided, you *may* lose points. 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 must **return** `true` only when all the test cases are passed; otherwise, you must **return** `false`.

In order to test some methods that throw exceptions, it is necessary to put your test in a try-catch block. Handle the exception gracefully within the unit test method by printing out the exception message if it occurs.

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 Assignment7
```

Submission

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

1. Open Gradescope and login. Then, select this course → Assignment 7.
2. Click the DRAG & DROP section and directly select the **EIGHT** required files: `FSComponent.java`, `File.java`, `NormalFile.java`, `ZippedFile.java`, `Directory.java`, `SubDirectory.java`, `RootDirectory.java`, and `Assignment7.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	✖
RootDirectory.java	**1.8** KB		✖
Assignment7.java	**1.9** KB		✖
Directory.java	**3.9** KB		✖
FSComponent.java	**1.5** KB		✖
MyFile.java	**1.6** KB		✖
ZippedFile.java	**2.3** KB		✖
SubDirectory.java	**1.1** KB		✖
NormalFile.java	**2.7** KB		✖

Submitting For

Darren Yeung

Cancel Upload

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.

Can a directory contain a myfile 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 myfile or subdirectory?
Yes, this is certainly possible.

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