FileList

This assignment (and the next) will also emulate UNIX tools. The `FileList` class will, in part, emulate the `ls` command. You should already be pretty familiar with the `ls` command. In case you are not, it lists the files in a particular directory – so will your `FileList` class. However, your `FileList` class will also maintain a list of files as a member variable so that the class can be used for more than just a command line tool. To get an idea for how it should work, play with the system `ls` command or with the `FileList.class` file I have provided.

Commmand Line Behavior

Your `FileList` class should implement a `main()` method, so that it can be run from the command line. You should account for the following command line options:

```
-A   :   show almost-all files (incl. those starting with a dot)
-l   :   use a long listing format
-c   :   display canonicalized path from the root
```

Note: the `-A` option does not need to show `.` and `..`, but all other files starting with a dot should be shown. The `-l` option displays a list of file names and series of attributes about the files – i.e. owner, group, size, etc. The `-c` option displays the full file name – i.e. the canonical path from the root to the file. Of course, be sure all of the options work in conjunction, as well. Running your `FileList` from the command line should look roughly like so (it will be packaged, by the way):

```
[user@notnotbc]$ java cli.FileList
a_directory/
another/
another_file
file.txt
[user@notnotbc]$
```

The `-A` option:

```
[user@notnotbc]$ java cli.FileList -A
.hidden_file
a_directory/
another/
another_file
file.txt
[user@notnotbc]$
```

The `-c` option:

```
[user@notnotbc]$ java cli.FileList -c
/CS3120/MW2/michael.g/a_directory/
/CS3120/MW2/michael.g/another/
/CS3120/MW2/michael.g/another_file
/CS3120/MW2/michael.g/file.txt
[user@notnotbc]$
```

The `-l` option:

```
[user@notnotbc]$ java cli.FileList -l
drwxr-xr-x michael.g michael.g 4096 2016-03-02T08:05:14Z a_directory/
drwxr-xr-x michael.g michael.g 4096 2016-03-02T08:05:14Z another/
-rwxr-xr-x michael.g michael.g 4096 2016-03-07T06:54:43Z another_file
-rwx------ michael.g michael.g 4096 2016-02-13T02:27:40Z file.txt
[user@notnotbc]$
```

All the options in conjunction:

```
[user@notnotbc]$ java cli.FileList -lcA
-rwxr-xr-x michael.g michael.g 4096 2016-02-29T19:39:55Z /CS3120/MW2/michael.g/.hidden
drwxr-xr-x michael.g michael.g 4096 2016-03-02T08:05:14Z /CS3120/MW2/michael.g/a_directory/
drwxr-xr-x michael.g michael.g 4096 2016-03-02T08:05:14Z /CS3120/MW2/michael.g/another/
-rwxr-xr-x michael.g michael.g 4096 2016-03-07T06:54:43Z /CS3120/MW2/michael.g/another_file
-rwx------ michael.g michael.g 4096 2016-02-13T02:27:40Z /CS3120/MW2/michael.g/file.txt
[user@notnotbc]$
```

With a directory name given as an argument after the options:

```
[user@notnotbc]$ java cli.FileList -Ac a_directory
/CS3120/MW2/michael.g/a_directory/.hidden_sub_dir/
/CS3120/MW2/michael.g/a_directory/file_in_dir.txt
/CS3120/MW2/michael.g/a_directory/sub_directory/
[user@notnotbc]$
```

With a file name given as an argument after the options:

```
[user@notnotbc]$ java cli.FileList -c file.txt
/CS3120/MW2/michael.g/file.txt
[user@notnotbc]$
```

There are a few differences to note from the system `ls`:

- The default behavior of `ls` outputs the files in columns as opposed to one per line. `FileList` always outputs one file per line.
- `ls` has a `-c` option, though it has a different function altogether. `FileList`'s `-c` option prints the canonical file name, as mentioned above.
- `ls` does not append a `/` to the names of directories. `FileList` will append a trailing `/`.
- The output for the `-l` option is somewhat different. For the `-l` option, the fields that will need to be displayed are: permissions, owner, group, file size, last modified date, and file name. The formatting does not have to be perfect, you may use tabs to separate the fields.

Minimum Requirements

The class should contain: a single `private` member variable, and single `private` constructor, and a set of `public` methods and variables. It must also be packaged in: `cli`. This means you are required to create a directory in your `submissions` directory called `cli`. Also, there are a number of packages you will need to import in order to complete the assignment. You will likely need all of the following:

```
java.util.*                java.nio.file.*            java.text.*
java.io.*                  java.nio.file.attribute.*
java.nio.*                 java.lang.*
```

Below is the skeleton of the `FileList` class to which your code must conform. You are free to add any helper functions as you wish, *but they must be* `private`. Remember that any `public` methods are part of the API in virtue of being `public`.

```
 package cli;

 public class FileList {
     // variables
     private List<File>;
     public static final int
         ALL, EXTENDED, CANONICAL;

     // constructor
     private FileList(List<File>);

     // methods
     public static FileList of(String) throws FileNotFoundException, SecurityException;
     public static FileList empty();
     public List<File> files();
     public boolean contains(File);
     public void add(File) throws FileNotFoundException, SecurityException;
     public static List<String> format(FileList, int ... options) throws IOException;
     public static void main(String[]);
}
```

FileList API

```
public static FileList of(String path)
   throws FileNotFoundException, SecurityException
```

A static factory method. It is given a path and it returns a `FileList` object loaded with all the files from last element on the path. If the last element on the path is a directory, it constructs the `FileList` object with a `List` of all the files in that directory. If the last element on the path is a regular file, it constructs the `FileList` object with a `List` of only that file.

It should check that the JVM has read and execute priveleges in all the parent directories along the path. And, of course, that each of the elements on the path exist. If the file or directory at any element along the path does not exist or if priveleges are lacking, throw the following `Exceptions` (respectively):

```
FileNotFoundException("cannot access <filename>:  No such file or directory")
SecurityException("cannot open directory <filename>:  Permission denied")
```

You will find that the following methods will all be of use to you:

```
Paths.get(String)          File.exists()             File.isDirectory()
Path.isAbsolute()          File.canRead()            File.listFiles()
Path.getNameCount()        File.canExecute()
Path.subpath(int,int)      File.isHidden()
```

Note: `listFiles()` returns an array of `File` objects *including the hidden files!*.

`public static FileList empty()`

A second factory method which simply constructs a `FileList` object with an empty `List`.

`public static List<File> files()`

Returns a **copy** of the `List` object containing all the `File`s. Be sure that you make a copy and that you don't simply return a reference to the internal `List` object.

`public static boolean contains(File f)`

Checks whether the `List` contains the specified `File f`. Returns `true` if `f` is in the `List`, `false` otherwise. `File.equals(File)` will be useful here.

```
public void add(File f)
   throws FileNotFoundException, SecurityException
```

Adds `File f` to the `List<File>`. If the file or directory at any element along the path does not exist or if priveleges are lacking, throw the following `Exception`s (respectively):

```
FileNotFoundException("cannot access <filename>:  No such file or directory")
SecurityException("cannot open directory <filename>:  Permission denied")
```

`public static List<String> format(FileList fl, int ... options)`

Takes a `FileList` object as an argument and any number of `int`s.

Note: an ellipsis signifies any number of parameters of a particular type. This can also be any linear collection of data – i.e. an array, a `Vector`, a `List`, etc. For more information on this: read about `varargs`.

Returns a sorted `List` of `String`s each of which represents a single file from within the `FileList` given. Each `String` should be formatted according to the options provided. The `static final` members should function as constants given as parameters to help determine how to format the `String`s. `String`s should be formatted according to the following guidelines:

`FileList.ALL` : Shows hidden files – i.e. those beginning with a dot.

`FileList.EXTENDED` : Shows extended file information.

```
(f/d)  (permissions)  (owner)    (group)  (size)  (last modified)       (name)
d      rwxr-xr-x      michael.g  MW2      4096    2016-02-13T06:12:47Z  some_dir/
-      rw-r-x---      michael.g  MW2      4096    2016-01-29T07:52:17Z  some_file
```

`FileList.CANONICAL` : Shows the canonical name of the file – i.e. path from root to file.

```
/CS3120/MW2/michael.g/some_file
/CS3120/MW2/michael.g/some_dir/
```

Note: a trailing slash is appended to directory names. Also, regarding the format of the extended file information: the alignment does not have to be perfect. You may simply use tabs, as mentioned earlier.

The following methods, along with the ones mentioned above, will be useful here:

```
Collections.sort(List<?>)
File.getCanonicalPath()
Files.readAttributes(Path, Class<A>
PosixFileAttributes.permissions()
PosixFilePermissions.toString(Set<PosixFilePermission>)
PosixFileAttributes.owner()
PosixFileAttributes.group()
PosixFileAttributes.size()
PosixFileAttributes.lastModifiedTime()
```

`public static void main(String [] args)`

Parses command line arguments according to the guidelines given in the previous sections. Usage is as follows:

```
java FileList [options] [argument]
```

Note: the [argument] field in this case is a file or a directory. This means `FileList` may be initiatied with any of the following conditions:

(1) no options, no arguments (simple display of contents in current directory)
(2) options, no arguments (display of contents in current directory, format accordingly)
(3) no options, arguments (simple display of contents of the argument given)
(4) options, arguments (display contents of argument given, format accordingly)

Your code may ignore any text after the argument.

After parsing the command line arguments, `main()` should create a `FileList` object using the `of()` static factory method. It should then pass this `FileList` object to the `format()` method which should return a `List<String>`. This `List<String>` may be printed one at a time to the console, or joined with `String.join()` and then printed all at once.

Extra Credit

You may earn extra credit for any of the following improvements on your `FileList` class:

(1) Format the extended output so that the columns line up properly (as they do in the system `ls` command. You will find the `Formatter` class to be of use if you make an attempt at this.

(2) Format the "last modified" field so that they appear as they do in the system `ls`. For example: `Feb 22 09:17`

(3) Include the number of symbolic links in your extended output. This is the field between the permissions and the owner fields. Also, include the total disk space allocated for all files in the directory. This refers to the line at the top of the output of `ls -l`.

(4) Implement a fuzzy finding feature. For instance:

```
java FileList a*
java FileList a??
```

The first lists only filenames that start with an 'a' and can be followed by any number of characters. The second lists only filenames that start with an 'a' and can be followed by only two characters. The `File.listFiles()` method is overloaded to accept a `FileFilter` object which will be useful here.

Remember that the name of the file must be the same as the name of the class with `.java` appended to it. In other words, your file should be named `FileList.java`.

Since it is packaged with the `cli` package, it must be in a directory named: `cli`. Submit your assignment by copying a directory named: `cli` along with the `FileList.java` file into your personal submissions directory. There is no need to place the `FileList.class` file in the directory.