CS 273 Laboratory 13: Recursion and File I/O

This lab gives you experience with reading, writing, and manipulating files as well as recursive methods.

Preliminaries

Create a folder named `lab13` in the `cs273` area of your network drive. Go to the course website and download the `lab13.zip` software. Unzip the software into the folder you just created.

Section 1. Recursion

From your recently unzipped folder, use BlueJ to open the `recurse` BlueJ project.

The `recurse` BlueJ project consists of three classes, `Tree`, `TreeException`, and `BeanFinder`. The file `Tree.java` defines the GUI for the lab. To run the code, right-click on the `Tree` class in the BlueJ window and select `void main(String[] args).`

Executing the program results in the display of a full binary tree, with a random height between 2 and 6. At the top, or the tree's root, is a white circle. This is the finder. Somewhere hidden in the tree is a red jellybean that the finder must find.

The file `Tree.java` defines these methods for moving around in the tree:

- public void moveUp() – Move the white finder up one level in the tree.
- public void moveLeft() – Move the white finder down one level in the tree, to the left.
- public void moveRight() – Move the white finder down one level in the tree, to the right.

It defines these methods for detecting whether its possible to move a given direction in the tree:

- public boolean canGoLeft() – Returns true if it is possible to move down and left in the tree, false otherwise.
- public boolean canGoRight() – Returns true if it is possible to move down and right in the tree, false otherwise.
- public boolean canGoUp() – Returns true if it is possible to move up in the tree, false otherwise.

It defines these methods for detecting the jellybean:

- public boolean foundTarget() – Returns true if the white finder is at the same node as the jellybean.

You will be using these seven methods in the next four checkpoints.

Part 1: Manually Find the Jellybean

At the top of the display are seven buttons. Three of these are up, left, and right. These buttons can be used to manually move the white finder around the nodes of the tree. Experiment with the buttons to manually move the white finder to the jellybean. Note that attempts to move too high or too low in the tree will cause a big 'X' to appear on the window. Once the 'x' appears, the only way to get rid of it is to press 'Reset'.

checkpoint 1 (5 points): Show your lab instructor or assistant that you were able to manually find the jellybean.

Part 2: Traverse to the Bottom of the Tree

Open the `BeanFinder.java` file from your BlueJ project. Notice the top of the BeanFinder class. It has a private instance variable called `tree`. Notice, too, that the file has the following method stubbed out for you with some example calls to `Tree` methods.

```
public void traverse() throws TreeException {

        // this are some EXAMPLE CALLS to Tree methods

        // CS 273 students should REPLACE these with actual calls

        // if I have found the jelly bean, return immediately so that
        // we do not move
        if (tree.foundTarget()) return;

        // move left/down if we can
        if (tree.canGoLeft()) {
            tree.moveLeft();
        }

        // move right/down if we can
        if (tree.canGoRight()) {
            tree.moveRight();
        }

        // move up if we can
        if (tree.canGoUp()) {
            tree.moveUp();
        }
    }
```

This method is called whenever the Go button is pressed from the GUI.

Given what you learned about recursion from CS203 or CS273, use the moveLeft() and canGoLeft() methods to move the white finder all the way to the bottom left of the tree. Consider using this structure for your code:

- *Recursive Case: If the finder can move left, move left. Then traverse again.*
- *Base Case: Otherwise, return.*

<u>checkpoint 2 (10 points):</u> Show your lab instructor or assistant that you are able to click 'Go' and the white finder moves all the way to the bottom left of the tree.

Part 3: Traverse the Entire Tree

Continue working in the `BeanFinder.java`, in the `traverse()` method. Use the moveLeft(), canGoLeft(), moveRight(), canGoRight(), moveUp() and canMoveUp() methods to move the white finder to every single node in the tree. You must use recursion to implement this checkpoint.

Consider this structure for your code:

- *Recursive Case 1: If the finder can move left, move left. Traverse again. Then move up.*
- *Recursive Case 2: If the finder can move right, move right. Traverse again. Then move up.*
- *Base Case: Otherwise, return.*

<u>checkpoint 3 (15 points):</u> Show your lab instructor or assistant that you are able to click 'Go' and the white finder moves to every node in the entire tree. Before getting assistance, make sure your finder can locate the jellybean at different parts of the tree, e.g. at the bottom or in the middle.

Part 4: Stop at the JellyBean

Continue working in the `BeanFinder.java`, in the `traverse()` method. Use the foundTarget() method so that the white finder stops at the node containing the jelly bean. Add a base case to the top of your method such that if the white finder is on the same node as the jellybean, the method returns.

<u>checkpoint 4 (10 points):</u> Show your lab instructor or assistant that you are able to click 'Go' and the white finder traverses the tree until it finds the jellybean.

Section 2. File I/O

From your recently unzipped folder, use BlueJ to open the `FileIO` BlueJ project.

Examine the file `FileHandler.java`, which defines the class `FileHandler`. This class contains one piece of state: a string that gives a message about the previous file operation that it performed. It defines five (presently dummied-up) file-manipulation methods:

- `createEmptyFile` - creates an empty file
- `deleteFile` - deletes an empty file
- `countBytes` - counts the number of bytes in a file
- `listContents` - lists the contents of a file
- `appendString` – appends text to an existing file

Most of the above methods sets the `lastMessage` instance variable to a value that either

- reports the result of the operation (e.g., whether it was successful), or
- contains the data that was requested by the user.

The instance variable `lastMessage` is declared as `private`, but other classes can get its contents by invoking the `getMessage` operation.

The `FileFrame` class provides a graphical user interface (GUI) that allows you to test these operations. The GUI consists of the following parts:

- a text-field where an input file can be specified, via a `Browse...` button.
- a text-field where an output file can be specified, via a `Browse...` button.
- one button for each method you'll be implementing
- a message area that displays the message from the most recent operation (i.e., the result of calling `getMessage` on its `FileHander` object).
- a text-field that is labeled `Call:`, which displays the most recent file-manipulation method call.

In its present state, all the operations result in a "... not implemented" message being displayed in the message area. This is because the methods in `FileHandler.java` are dummied up. You should be able to see calls that are made in the `Call:` text-field, however.

Your job will be to implement methods in `FileHandler.java` so that they behave correctly.

Compile `FileHandler` and run `FileFrame` (using `void main String []`) so that you can see what this looks like.

Part 5: Implement the createEmptyFile method

Edit `FileHandler.java` so that the `createEmptyFile` method creates a file that contains zero bytes, whose path/name is given by its `String` parameter. (This is taken directly from the `Output file:` text-field of the FileFrame interface.) You should be able to do this by:

1. opening the file for writing
2. immediately closing it

The above operations should be "wrapped" in a `try` clause. If an error occurs, the `lastMessage` instance variable should be set to a `String` that reports that the file could not be created; if no error occurs, the `lastMessage` instance variable should be set to a `String` that reports that the file was created successfully. Do not print the error message to the console.

To test that your method works correctly, do the following:

1. From the Windows File Explorer, locate the file `emptyFile.txt` in your ~/lab13/fileIO/ directory. Open it. Notice that it is not actually empty.
2. Return to your Java GUI. Use the Browse button to select the file `emptyFile.txt` in the Output File: text-field. **Note:** You cannot type a filename directly into the output field. It must be specified via the file selection dialog window that appears when you click the Browse button.

checkpoint 5 (10 points): Show your lab instructor or assistant the executing program. You should be able to use the Windows File Explorer to verify that the file is in the directory, and that its length is zero.

Part 6: Implement the deleteFile method

Edit `FileHandler.java` so that the `deleteFile` method deletes the file that is named by its `String` parameter. (This is taken directly from the `Output file:` text-field.) Do this using Java's File class. The `lastMessage` instance variable should be set so that it reports whether the deletion was successful.

To test that your method works correctly, do the following:

1. From the Windows File Explorer, locate the file `deleteThisFileFirst.txt` in your ~/lab13/fileIO/ directory. From your Java GUI, select this as your `Output file`. Click "Delete File". Affirm that the file has disappeared from the Windows File Explorer.
2. From your Java GUI, create an empty file, delete it, then attempt to delete it a second time. The second deletion attempt should result in an error message.

checkpoint 6 (10 points): Show your lab instructor or assistant the executing program. You should be able to use Explorer or Finder to verify that the file is actually deleted.

If you have *fully* completed all the above checkpoints, you have a grade of D- (60) for this lab.

Part 7: Implement the <u>listContents</u> method

Edit `FileHandler.java` so that the `listContents` method sets the `lastMessage` variable to be a `String` whose contents are the contents of the file named by `listContents`' `String` parameter. (This is taken directly from the `Input file:` text-field.)

Important: You must use the `Scanner` class to implement this method. To declare a new `Scanner` that is associated with a given input file,

```
        Scanner sc = new Scanner(source);
```

Where `source` is an object of type `FileInputStream`.

It is recommended that you use the `hasNextLine` and `nextLine` methods from the `Scanner` class. However, `nextLine` will remove the newline character from each line it reads. You'll have to add it back. The `EOL` instance variable at the top of FileHandler.java contains this character.

To test that your method works correctly, do the following:

1. From the Windows File Explorer, locate the file `myFile.txt` in your lab13 directory. From your Java GUI, select this as your `Input file`. The window should show something like this:

```
This is a file that contains ...

    ... a couple lines of text.
```

<u>checkpoint 7 (10 points):</u> Show your lab instructor or assistant the executing program. You should be able to view the contents of text file in the message area. (Non-text files should also be viewable, but they'll look like garbage.)

If you have *fully* completed all the above checkpoints, you have a grade of C- (70) for this lab.

Part 8: Implement the <u>countBytes</u> method

Edit `FileHandler.java` so that the `countBytes` method counts the number of bytes in the file that is named by its `String` parameter. (This is taken directly from the `Input file:` text-field.) You could do this by creating a `File` object and calling its `length()` method, but that's too easy and you need some practice reading a file. Instead, do this by:

- Opening the file for reading, using an instance of the `FileInputStream` class.
  Note: Do not use the `Scanner` class for this method.
- Successively reading one byte at a time from the file until there are no more bytes. Keep count of how many bytes you read as you do this.

- Return your count to the caller.

Use the file `myfile.txt` as a test case.

After you successfully count the bytes, you should be able to delete the file.

checkpoint 8 (20 points): Show your lab instructor or assistant the executing program. You should be able to use Windows to verify that the files byte-lengths of the files are reported correctly.

If you have *fully* completed all the above checkpoints, you have a grade of A- (90) for this lab.

Part 9: Implement the appendString method

Edit `FileHandler.java` so that the `appendString` method appends a `String` to the end of the file. (This should increase the size of the file by a few bytes.) The file to be modified is the one that is named by `appendString`'s first `String` parameter. (This is taken directly from the `Output file:` text-field.) The `String` to be appended is given in the second parameter.

You should be able to do this by opening the file for writing in "append mode" using a `FileOutputStream`. The variable `lastMessage` should be set to a `String` that gives a message that tells whether the operation is successful.

checkpoint 9 (10 points): Show your lab instructor or assistant the executing program. You should be able to create an empty file (`createEmptyFile`), append strings to it, and then display it using `listContents`.

If you have *fully* completed all the above checkpoints, you have a grade of A (100) for this lab.

Part 10: Finish Up

Close all windows and log off. You're done.