PS1: Model-View-Controller

This assignment explores the following topics:

- the model-view-controller paradigm as it is implemented in a particular Swing widget, JTree;
- sending events between model and view/controller;
- data-bound widgets.

In this problem set, you will implement a user interface that displays a subtree of the filesystem, and allows the user to rename files.

Provided Resources

We provide you with the following:

- FileSystem.java: a skeleton for your solution.

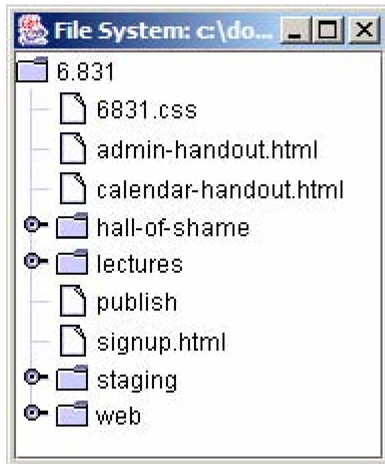
Feel free to change this class as you see fit, or even replace it entirely.

Problem 1: Static Tree

When you compile and run the FileSystem class we provide you, you'll see that it prints the names of all files in the current directory and its subdirectories, recursively. If you specify a different directory on the command line, it displays that directory instead.

Modify this program so that instead of printing filenames to the console, it displays them in a JTree widget. For now, just use the JTree's built-in model to do this: DefaultTreeModel and DefaultMutableTreeNode.

The resulting window should look something like this:



Only the top-level directory should be expanded initially (JTree's default behavior). Obviously, make sure the window can scroll if the tree is too large for it.

Problem 2: Data-Bound Tree

The approach you took in problem 1 doesn't scale well. Try pointing it at a deep subtree (`C:\Windows` is a good choice on Windows platforms; `/usr` works on Unix platforms). What happens?

The solution to this problem, sometimes called *data binding*, is to connect the tree widget directly to a live filesystem model, instead of a copy of the model.

Modify FileSystem so that it can handle deep subtrees without noticeable delay. It should scan the filesystem *lazily*, fetching a directory's contents only when those contents are actually needed. There are two approaches to doing this with JTree: (1) implement your own TreeModel; or (2) use DefaultTreeModel, but implement your own TreeNodes.

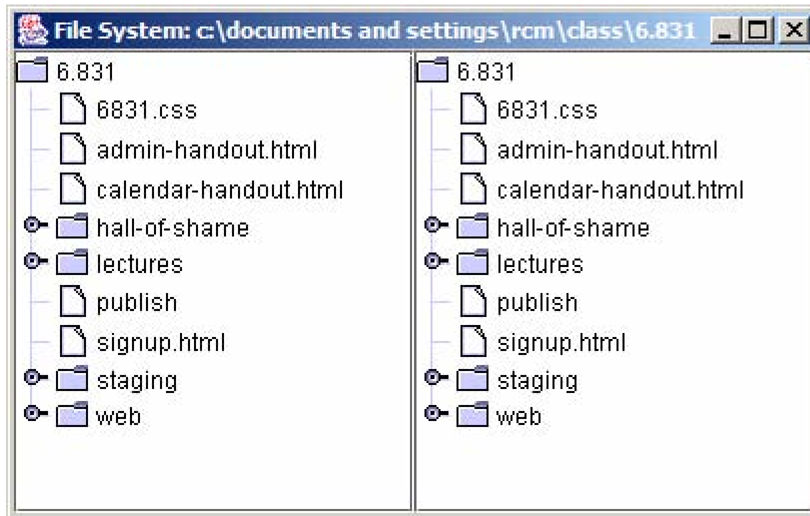
Note that lazy loading doesn't mean that the model is *changing*, only that it doesn't bother creating parts of itself until a client actually looks at those parts. You don't need a background thread for lazy loading, and the model shouldn't be sending events to its listeners as it builds itself lazily.

Your interface should still look like it did in problem 1.

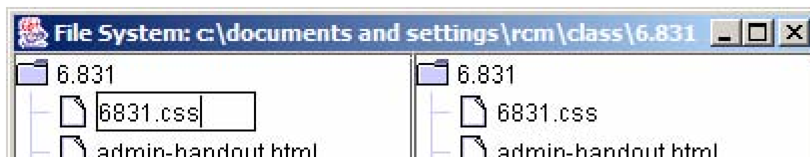
Problem 3: Mutable Model

Until now, your filesystem model has been *immutable*: it never changed while the JTree was displaying it. Now we'll make it mutable.

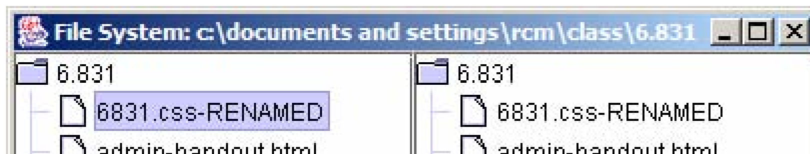
Modify FileSystem so that it displays two JTree widgets side-by-side, both pointing to the same model:



Make both JTree widgets editable (hint: JTree.setEditable()). In an editable JTree, you can edit the label of any file or directory by triple-clicking (yikes!) or pressing F2:



When the user edits a label in a JTree and presses Enter, your program should respond by renaming the corresponding file in the filesystem. Since both JTrees have the same model, both JTrees should then reflect the renamed file:



Making this work depends on which approach you chose in Problem 2:

- If you implemented your own TreeModel, then you will need to implement TreeModel.valueForPathChanged() to handle the user's edits, and you will need to broadcast a treeNodesChanged event to your listeners. It's a little tricky to set up the TreeModelEvent object that you pass along to treeNodesChanged. The TreeModelEvent should describe the node that changed by giving the path to the changed node's *parent* (not to the changed node itself!), plus the index of the changed node among its siblings.
- If you implemented your own TreeNode, you need to implement MutableTreeNode instead, and implement setUserObject to handle the user's edits. If you did it this way, then the DefaultTreeModel takes care of managing listeners and broadcasting events to them.

Questions

Answer the following questions in readme.txt.

1. Point FileSystem at a directory that's too long to fit in the window, so that scrollbars appear. Which parts of the model does JTree have to request in order to display the window initially? (Don't just guess: find out.) Why does it do that?
2. Does your filesystem model cache the directories it looks at, or does it access the filesystem whenever the model is accessed? What are the advantages and disadvantages of caching?
3. Your filesystem model is not fully data-bound. What filesystem changes does it fail to notice? How might you fix that?

What to Hand In

Package your completed assignment as a jar file. Your jar file must contain:

- **Class files**. Compiled class files for your program.
- **Source files**. All Java source files you wrote.
- **Resources.** All the resources your code needs to run. For example, the `words` file must be included in your jar file; so should any images or other data files your code requires. All resources of this kind should accessed with Class.getResource(). Your code shouldn't have any absolute filenames in it, since they're not likely to work on our machines.
- **Readme.** A plain text file called `readme.txt` that answers the questions we asked in this assignment, if any.
- **Manifest.** Your jar file's manifest file must include a `Main-Class` parameter that specifies the main class, so that we can run your jar using `java -jar` *`yourfile.jar`*

The Java tutorial has a section about jar files that explains how to create a jar file and define its manifest file. If you're using Eclipse, you can create a jar file from your project using File/Export. The Export wizard offers options for including source files, readme and other resources, and specifying the Main-Class for your jar's manifest.

If your program depends on any third-party libraries, you have two choices:

- Unpack the other jars into class files, and include those class files in your jar. This is the least-error-prone approach, and is strongly recommended.
- Put a `Class-Path` parameter in your jar's manifest file specifying the jar files it depends on. For example:

  ```
  Class-Path: TableLayout.jar xerces.jar
  ```

  Then hand in TableLayout.jar and xerces.jar alongside your own jar file. This is error-prone because you might forget to send us a required jar file.

Before you submit your solution, put all the jar files you plan to submit in an empty directory and make sure you can run it:

```
java -jar yourfile.jar
```