From: **Alan Liang** alan.5100@yahoo.com
Subject: Damn Lab
Date: October 13, 2016 at 4:32 PM
To: Winnie Liang winnieliang27@yahoo.com

AL

lab07 a53

Due tonight at 11:59 pm

All packages in lab07

In this lab you will implement some basic file editor functionality. It will allow you to open, type, draw, and save files. We will use inheritance and subclassing to aid us. We won't be doing any graphical stuff but will be coding a basic backend.

First you will start with a basic class to represent buffers, the place to edit text.

Make a class `Buffer` with:

- a protected `List<String> lines` field. Do not instantiate it to anything
- a public `int getNumlines()` which returns the length of the `lines` field
- a public `String getText()` field which returns `String.join("\n", this.lines)`
- a public `void draw()` method which prints out `this.getText()`
- a public `void save() throws Exception` method which throws an `UnsupportedOperationException`
- a public `void type(String toType)` method which will add the characters in `toType` to `lines`
    - first check if `lines` is empty, if it is add the empty string to `lines`
    - loop through each character in `toType`
        - if the character is a newline, **add** an empty string to `lines`
        - otherwise **append** the character to the last String in `lines`

Now you will write a specialized version which is a buffer backed by a file.

Make a class `FileBuffer` which extends `Buffer` with:

- a private `String` field named `filename`
- a constructor that accepts a String to set the `filename` with
    - It also needs to read any lines in the given files and set it to `lines`.
    - the easiest way to do this is to use

the easiest way to do this is to use the `java.nio.file.Files.readAllLines` function. Look up the documentation (here and here).
    - this method could throw an IOException so the call to `readAllLines` needs to be in the try block of a try/catch. In the catch block, set lines to a new ArrayList.
- a `save` method which overrides `Buffer`'s save method. Remember when overriding, the function signatures must match. It is good practice to annotate any overridden methods with`@Override` to ensure you are overriding. The method writes all the Strings in `lines` to the `filename` file. The easiest way to do this is to use the `java.nio.file.Files.write`method. documentation. The second argument to the write method should be `lines`.

Next we will create another type of buffer which isn't backed by a file, but still accepts text. While it may not be backed by a file, it still shares operations with the `FileBuffer` class.

Make a class `ScratchBuffer` which extends (subclasses) `FileBuffer`.

- provide a constructor with no arguments that calls its parent (super) constructor with the argument `"scratch"`.

All of the other methods are inherited from our parent class, `FileBuffer`, and need no other functionality.

We will now create a third type of buffer which will mimic a terminal shell (command-line). We will again reuse existing functionality through inheritance.

Make a class `ShellBuffer` which extends `Buffer`

- add a `private final` String called PROMPT, initialized to `"> "`. Note the space after the angle-bracket. This is what will be printed out to prompt the user for a command. Feel free to customize it.
- provide an no-argument constructor that initializes lines to a new ArrayList. When we refer to the variable `lines`, we are refering to `lines` in our parent class, `Buffer`.
- Override the save method to do nothing. This is because our editor will not allow a command-line session to be saved to file, it only exists as long as our editor is open.
- Override the type method

`ShellBuffer`'s `type` method will type the PROMPT string, then the command the user enters, then the output of that command. First we need to type the prompt. We cannot say`type(PROMPT)` because that would call the method we are currently in. Instead we want to call the `type` method in `Buffer`. Thus, we write `super.type(PROMPT)` to

want to call the type method in Buffer. Thus, we write super.type(PROMPT); to add PROMPT to lines. Next call Buffer's type method twice, once with the String passed into this type, and once with a newline. Lastly call the method execute, provided below, with the String argument that our type method took in.

```
/**
 * Has the OS execute the shell command `command`, appending its
output to `lines`.
 * If the command fails in any way, a failure message is instead
appended to `lines`.
 * @param command the command to execute as one space-separated
string
 */
private void execute(String command) {
    // split `command` into list of space-separated words
    List<String> commandAsWordList =
Arrays.asList(command.split(" +"));
    ProcessBuilder pb = new ProcessBuilder(commandAsWordList);
    File pipe = null;
    try {
        pipe = new File("pipe");
        // capture the output of our `command` into the `pipe`
file
        pb.redirectOutput(pipe);
        Process proc = pb.start();
        proc.waitFor(); // wait for it to finish before reading
from it
        List<String> output =
Files.readAllLines(Paths.get("pipe"));
        super.lines.addAll(output);
        super.type("\n");
    } catch (Exception e) {
        super.type("command '" + command + "' failed\n");
    } finally {
        if (pipe != null) {
            pipe.delete(); // delete the pipe file
        }
    }
}
```

Now we'll write a class that will give status information such as number of lines in the buffer.

Make a class `StatusBar` which extends `Buffer` with:

- a private `Buffer attachedTo` field

- a constructor which takes a `Buffer` and sets the `attachedTo` field to it
- Override `getText` to return the number of lines in the `attachedTo` buffer. You can stylize the output. For example it could return `------ lines: 10 ------` assuming there were 10 lines in the attached buffer.

Look back to the `draw` method inside `Buffer` where `this.getText()` is called. Does this always call the `getText` inside `Buffer`? Hint: polymorphism (dynamic-dispatch) is at play.

Putting all theses classes together, we write the main text editor class.

Make a class `Editor` with:

- a private `Buffer` field and a private `StatusBar` field
- a constructor which takes one `Buffer` argument. It sets the `Buffer` field to the argument and uses the argument to initialize the private `StatusBar` field to a new `StatusBar`
- a `draw` method which calls `draw` on the `Buffer` field and then the `StatusBar` field

This `draw` method is a delegating method, meaning its implementation is delegated to a corresponding method of one of its fields, `buffer` and `sb` in this case.

Eclipse can generate delegation methods for you. We'd like to generate delegate methods `save` and `type` that delegate to those of `buffer`. Click `Source->Generate Delegate Methods`and check the boxes `save` and `type` from `buffer` and you should see two methods have been added. Add `throws Exception` to the save method declaration.

Now you will test your editor.

Create a `Driver` class with a main method. Add `throws Exception` to the main method signature. Remember when Eclipse asks you for the name of the class to create, check the checkbox for the main method and Eclipse will generate it for you.

Create an Editor(s) and experiment with supplying it each of the three different Buffer subclasses we created. Play around with `draw`, `type`, and `save` in different orders to see what happens. Instead of being able to actually type in text or click a Save button, we have to call the corresponding methods in our main method to simulate those actions. Make sure to test drawing a buffer that hasn't had anything typed into it. If save a FileBuffer and run your program, pressing F5 in Eclipse will refresh the file list and you should be able to open the file that has the contents you typed. Try typing a command such as `ls` or `pwd` in an Editor with an underlying ShellBuffer and then calling draw. Observe the command and its output are automatically put into the buffer, just as if you entered it into a terminal!