29/30

CS361 Project 5

Martin Deutsch, Liwei Jiang, Melody Mao, Tatsuya Yokota

October 12, 2018

The major tasks of this project were to implement buttons for compiling and running code and to add an input/output console for our IDE. Our code is structured to use 7 Java files, a CSS file, and an FXML file. The Java files are a top-level Main file, a StyledCodeArea for syntax highlighting, and controllers for different sections of the IDE. We made the majority of our changes for this project in the ToolbarController class. The ToolbarController is now able to start processes to compile and run Java programs using the built-in javac and java tools. The ToolbarController also handles sending output to and getting input from the console during the execution of the Java program.

When the compile and run button is clicked, the Controller class calls ToolbarController's handleCompileRunButtonAction. That method first checks if the file has been modified since the last save, and, if so, asks the user if they want to save. It then spawns a new concurrent Task that calls the compileJavaFile and runJavaFile methods. To compile, the application starts a new Process that runs *javac* on the given file. To run, the application creates a Process that runs *java* on the file and starts two new threads, inThread and outThread, that handle the program's input and output. The outThread prints each character from the program's output stream to the console. When the output stream is paused, the outThread signals the input thread. The inThread waits for the user to type some input and press enter, then sends the user's input to the program's input stream and signals the output thread. When the process ends, the two threads die. If the user presses the stop button at any time, the two threads are interrupted and the process is destroyed.

One reason our code is elegant is because it uses different threads for the input and output of the Java program. This allows our application to easily alternate between printing output and waiting for input. Another reason it is elegant is because it uses a Semaphore to signal between the input and output threads, which automatically halts execution of the thread that is not needed. Also, in terms of the file structure, the program has one main Controller class distributing the work to more specialized subclasses, which reduces the length of each Controller and makes the code compact and easy to understand. We designed the sub controllers so that for each required field from the main controller, we added a setter method for it. We didn't pass in the shared parameters through constructor because it provides more flexibility if we want to modify the sub controllers and to pass in more parameters in the future. Finally, we set bindings for the Compile, CompileRun, and Stop buttons, and File and Edit menu to enable and disable them accordingly. In order to detect whether the compile or the compile and run threads are active, we wrapped the working threads into a Javafx Service object, so that the threads would have properties indicating whether they are active or not, which can be used in binding setting.

One inelegance in our application is that the output prints character by character, pausing the thread for a short time after each character, instead of line by line like a real IDE. However, this was necessary because reading by line would not include statements like System.out.print, which does not end in a newline. Another possible inelegance is that we have code interacting with both the TabPane and the console in our ToolbarController, instead of having that class deal with only one aspect of the GUI. But, since the buttons in the toolbar cause actions that interact with both the tabs and the console, we could not see a better place to put the code.

That's fine, but pause 1msec between chars, not 5msec.

Tatsuya and Melody implemented the toolbar buttons and the compile and run processes. Martin and Liwei implemented getting output and input to and from the console, set up the bindings, added save-for-compile dialog, and refactored the previous project code. We all looked through the code for bugs and made sure documentation was correct.

- Works great with program input/output and printing runtime and compiler errors. Good job!
- If I edit JavaScannerExample.java so that all print() calls are changed to println() calls and then run the code and enter my name on the same line as the prompt instead of on the next line, the program crashes. If I enter the data on the blank line, it works fine.
- Things to fix for Project 6 (no points taken off this time):
    - Your StyledCodeArea is misnamed. It doesn't style any kind of code; it is geared only for Java. Your StyledCodeArea's constructor should call handleTextChange rather than expecting another class to do it. More generally, all methods in StyledCodeArea except the constructors should be private. No other class should have a reason to call those methods.
    - In StyledCodeArea, the handleTextChange method should have nothing to do with TabPanes. What if you want to use StyledCodeAreas without putting them in TabPanes? Your code won't let you.
    - Your code would be simpler and much cleaner if you got rid of the initial untitledTab. It would save you several methods and two fields and a bunch of code. The tab pane can be initially empty and the user can press Command-N to get a new empty tab.
    - Your handleCompileAction and handleCompileRunAction in ToolbarController take a tab as a parameter. They should instead take the file as the second parameter. These two methods shouldn't need to know about tabs and the tabFileMap. In fact, the whole ToolbarController doesn't need to know about the map because all it ever does with it is find the associated file. So instead, pass the ToolbarController the file instead of the tab.
    - The ToolbarController's checkSaveBeforeCompile should be in FileController, which handles saving. The FileController can find the selected tab and check if it is dirty. The method could get the text for the dialog from a parameter, so that it will be more generally useful than just checking before compilation.
    - In FileMenuController, you have getFileContents and saveFile methods. This is fine, except that, since they are complementary methods, it would be better to change the name of the latter method to setFileContents.
    - The getFileChooser method has an incorrect Javadoc header and it also does nothing other than return a new FileChooser. Wouldn't it always be better to just create a FileChooser directly rather than call this method?