Make, Checkstyle, Junit Lab

Appalachian State University

Mx. Willow Sapphire

CS 3667

This project will help you get familiar with using makefiles, checkstyle (in the command line), and Junit (also in the command line). As always, you should clone the repo on the student machine and then go into the root directory. All the commands in these instructions assume that you are running them from the root of the repo.

For every section you should do the following steps (some sections may remind you of some, but will not reiterate all).

1. Make and checkout a new branch
2. When you make edits, add them to the staging area
3. Commit your changes to the new branch at identifiable points with clear messages

4. Push your new branch to GitHub (this can be done at any point between making the branch and finishing the section)
5. When you are done, make a pull request on GitHub and merge it.
6. Finally, in your local repo, run these commands to update your local repo with the changes made on GitHub:

```
git checkout main
git fetch
git merge origin/main
```

Table of Contents

Makefiles

In this section we are going to examine the makefile in the repository and learn how it works. We will be editing and using the makefile throughout the other sections of this assignment. Create a new branch named make and check it out.

1. Run the command `make`
2. This will run the target "default" that you can read in the makefile. Can you tell what @echo is doing after running the command?
3. Run the `make compile` command. As expected, it compiles your code (in this case, the default code that was already given to you).
4. `ls` the bin directory and any subdirectories to see what was generated. Do you understand why this is what was generated?
5. Run the `make clean` command and `ls` the bin directory and subdirectories again. What happened to it? Do you understand why? Look at the compile and clean commands in the makefile to help make sense of it all.
6. Create a macro at the top of the makefile named JFLAGS and move the flags from the javac

command to the macro. The flags to move are "-d bin -sourcepath src" then add the macro to the javac command using the $(MACRO_NAME) syntax.

7. Run `make compile` again to ensure it still works.
8. If you have it all working, then you can finish this section using our standard git wrap-up commands (see top of instructions).

Checkstyle

In this section we are going to learn how to run checkstyle in the command line and how to write and edit our own checkstyle xml files. This allows us to tailor checkstyle to our coding style preferences. We will also be using a makefile to make the process easier. Start by making and checking out a branch named checkstyle.

Part 1: Checkstyle Basics

1. Look at the styles.xml file. This is the file that the checkstyle program will read to determine what checks to make.
2. ls the lib folder and you should see a file named checkstyle-10.5.0-all.jar. Checkstyle is written in Java and we will use this jar file to run it.
3. Run the following command: `java11 -jar lib/checkstyle-10.5.0-all.jar -c ./styles.xml src/**/**/*.java`
4. Let's break down this command
    a. `java11`: a symlink on the student machine to run java 11. The java command on the student machine runs java 8 by default which will not

work with this version of checkstyle. You can try running this command with java instead of java11 to see the error.

b. `-jar` This is a flag for java to tell it to use code in a jar file
c. `lib/checkstyle-10.5.0-all.jar` This is an argument applied to the `-jar` flag to tell it what jar file to use. In our case, we want to use the checkstyle jar which we have stored in our lib directory.
d. `-c` This is a flag that tells checkstyle what xml file to use to determine what checks to make.
e. `./styles.xml` This is an argument to the `-c` flag that tells checkstyle that we want to use our styles.xml file
f. `src/**/**/*.java` This is the argument to the checkstyle program to tell it what files to check. We are telling it to check all of the java files in directories in the src directory (which will be our program files and test files).

5. You should see two checkstyle errors. Find them in the Calculator.java file and fix them. Then run the checkstyle command again to make sure that they are fixed. (This would probably be a good point to commit changes at, perhaps with the message "fixed checkstyle errors").
6. As you can see, this is a very long command to type out in order to run checkstyle. This is where our makefiles will start coming in especially handy.
7. Create a new target in the makefile named check. Put it above the clean target since we like to keep that target as the last one in the makefile.
8. Add styles.xml and lib/checkstyle-10.5.0-all.jar since we need those files to exist for the command to work.
9. Add the command we just used to the check target
10. Exit the make file and run `make check` to ensure it works
11. Now make a new macro named CHECKSTYLE and set it equal to lib/checkstyle-10.5.0-all.jar
12. Then replace the two occurrences of that path with the macro.

13. Run `make check` again to ensure it is still working (another good time to commit).

Part 2: Customizing Checkstyle

14. Now it is time to edit the styles.xml file. Go into the file and find the LeftCurly module. In the option property, change the value from nl to eol. nl stands for newline and eol stands for end of line. This change means that left curly braces should appear at the end of the previous line rather than being placed on a new line.
15. Run `make check` and you should see a lot of errors. Go into Calculator.java and fix them.
16. Now look at the other modules in the file as well as the possible modules which you can find in the Checkstyle Checks link on AsuLearn.
17. Add or remove a module from styles.xml that caters to your styling preference. The change you make must require you to update Calculator.java. For example, I might remove the NeedBraces module because I like to sometimes omit braces with ifs or loops with only one statement inside them. I

strongly recommend that you pick something that checks for styles that you care about.

18. Update Calculator.java to pass the new checks if needed. If not needed (such as my example of not needing braces) update the file to include something that would have previously failed (such as removing braces from a one-statement loop).
19. When commiting this change to GitHub, make sure you state what change you made to the checkstyle file. If I cannot tell what you did, you will not receive credit for this portion.
20. Finally, add all of your changes to GitHub.

Junit

In this section we are going to learn how to write Junit tests for our code and how to run them using the command line and makefile. Make a new branch named junit and check it out.

Part 1: Junit Setup & Basic Test

1. Under src/tests/calculator make a file named CaclulatorTest.java and set up a basic, empty class. The package should be called calculator. Both the source code and tests are in the calculator package, but they are under separate directories so that we can keep them apart for easy organization. We want them in the same package so that the tests can access package protected content. Note: if you create the file in vscode it may try to make the package tests.calculator. Change it to calculator and reload vscode to make it stop complaining.
2. Create a public void method in Calculator.java named testConstructor. Inside of the method, create a calculator variable and store a new calculator in it.

3. Now we want to add some Junit. Import the Test decorator from org.junit.jupiter.api.Test and add the decorator above the method you created. If you are using vscode, you may need to add the junit jar to your referenced libraries to stop it from complaining.
4. Import the assertNotNull assertion from org.junit.jupiter.api.Assertions.assertNotNull. Note that all of the assert methods (but not the decorators) are static, so your import statements should be "import static ... "
5. In the testConstructor method, use assertNotNull to check that the calculator object was successfully created. This is not a check we would normally do, since all other tests will require the object to be created anyway, but it is an extremely simple test that we can do as an introduction to Junit.
6. Now we need to run our tests. The first thing we need is to compile our tests. Try running `make compile` and you will see that it cannot find junit. This is because our make compile method does not have the junit jar on its class path.

7. Create a macro named JUNIT and set it to lib/junit-platform-console-standalone-1.8.2.jar.
8. In the compile command, after $(JFLAGS), add -cp $(JUNIT). This puts the junit jar file on the class path.
9. Run `make compile` again to verify that it works. Check the bin folder to see if you have the test class file. Note that it is in the same directory as the calculator class file since they are in the same package.
10. To run the tests after you have compiled with, run the command: `java -cp bin:lib/junit-platform-console-standalone-1.8.2.jar org.junit.platform.console.ConsoleLauncher --scan-class-path`
11. As you can see, Junit is also a Java program that we are running using Java. It is also a very long command so let's break it down, then add it to the makefile so we never have to type it out again.
    a. `java` This runs Java (you should know that already)
    b. `-cp` This is a flag for the java command that stands for class path. It is used to tell Java where our files are.

c. `bin:lib/junit-platform-console-standalone-1.8.2.jar` This is our class path. Different parts of the class path are separated by colons. We have two parts: our bin folder where the compiled code goes, and the Junit jar file.
d. `org.junit.platform.console.ConsoleLauncher` This is the actual program that we are running. It is the Junit Runner.
e. `--scan-class-path` Finally, this is a flag for Junit to tell it to scan through the class path and look for all the tests to run.

12. Now let's add this command to the makefile. Make a new target named test.
13. Create a new macro named JUNIT_RUNNER and set it equal to org.junit.platform.console.ConsoleLauncher
14. Create a new target named test.
15. Add the JUNIT macro as a dependency.
16. Add the command we used to run the tests as the command to the target.
17. Use the JUNIT and JUNIT_RUNNER macros to simplify the command.
18. Run `make test` to ensure this is working.

19. Run `make test` once more to check that everything is still working. (This is a great time to commit).

Part 2: Writing Tests

1. Go back to CalculatorTest.java and refactor the code. Create a field to store the calculator, add a beforeEach method using the @BeforeEach decorator (remember to import it), and initialize the calculator field in the beforeEach method. Use this field in the existing test instead of creating a new calculator.
2. Now create a test to test the calculate method. This method accepts an expression as a string and returns the result as a double. Test the expression "5 + 4 * (10 / 2)" (answer is 25) using the assertEquals method (imported from org.junit.jupiter.api.Assertions.assertEquals. The expected value should be the first parameter and the actual value should be the second. Then run `make compile` and `make test` to see if it works. The test should pass.
3. Testing one expression is not enough to make sure this calculator works. If we really wanted to test it thoroughly, we would test many, many different expressions. For this assignment though, let's just test one more: "20 / 5 - 1" (answer is 3). Add this

to the testCalculate method in another assertEquals statement.

4. Run the tests again (remember to compile again first) and you should see that the test fails! Look at the test output to see what the calculator thought the answer is. Try to consider what could cause the calculator to give this answer instead of the correct one. Find the bug in Calculator.java and fix it (hint: the bug is in a helper method).
5. Compile and run the tests again and you should see that everything passes.
6. You are done! add, commit, push, and merge your code.