Tutorial & Lab: SonarSource

Using SonarQube and SonarLint to write quality code

Introduction

In the last topic, we looked at the IntelliJ static analyzer and how it could benefit us in writing better, cleaner, and sometimes faster code without changing what your code actually does. It can be beneficial, however, to run through multiple different static analyzers that specialize in different areas to help improve your code.

SonarSource incorporates most known methods to analyze your code, and can run independently of an IDE. We will be exploring SonarQube and SonarLint--two open-source tools from SonarSource that have become widely used in the industry. SonarQube is used as part of an automated code deployment process while SonarLint is an IDE plugin that analyzes your latest edits.

Download the Static Analyzer Tutorial Revised Code Zip File from the last assignment. The instructions below are specific for IntelliJ IDEA version 2020, but you should be able to use other IDEs for this assignment. This is because of SonarSource's robust support for other IDEs.

The code you are given is now part of a Maven project and has written unit tests. It is expected that you have Java 11 or higher installed in your computer. You can look at the comments in the pom.xml file to understand the configuration made to have Maven integrate SonarQube.

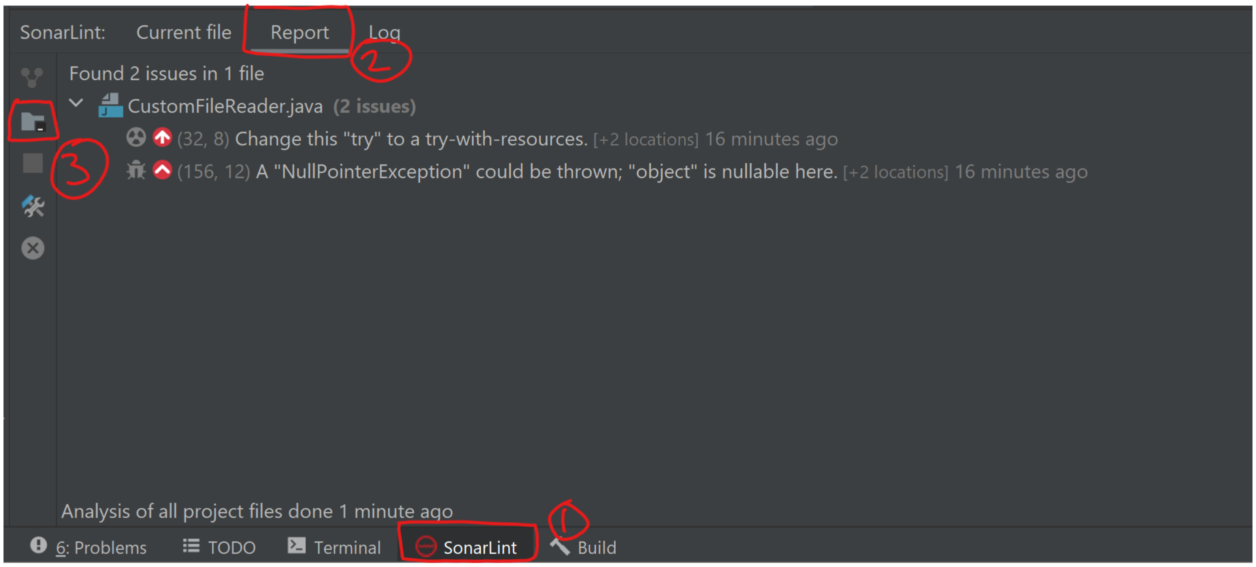
Tutorial

Using SonarLint

1. To import the revised code, unzip the code to a folder which you may hold other IntelliJ projects. After you have done so, open up IntelliJ, then go to File > New > Project from Existing Sources. A window will open in which you can navigate to, and select the pom.xml file that was unzipped. IntelliJ will then import the project. Run the test files in IntelliJ to make sure the project was imported correctly. All the tests should pass.
2. Run the program in Main.java once to create a run configuration and then edit the run configuration. Refer to the instructions from the last tutorial for information on how to set up the run configuration. The readMe files are now in the root directory, so you will not have to change the working directory. This is to simplify the configuration of the unit tests.

3. After you ensure that the program is working properly, run IntelliJ's static analyzer. The green bar should display as it did at the end of the last lab assignment you have done, meaning that it didn't detect any errors. This is to show that even though the IntelliJ static analyzer doesn't show any warnings, there are still other ways to possibly improve our code.

4. Now open the SonarLint tab on the bottom of IntelliJ, then click on the Report tab as shown below. Click on the folder icon to analyze the whole project. You can choose to ignore the warning that shows up since our project is small. After the analysis is done, you should see output similar to the following image in the bottom left corner of your Intellij screen:



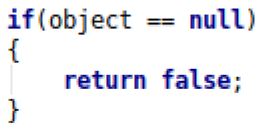
```
Found 2 issues in 1 file
CustomFileReader.java (2 issues)
    (32, 8) Change this "try" to a try-with-resources. [+2 locations] 16 minutes ago
    (156, 12) A "NullPointerException" could be thrown; "object" is nullable here. [+2 locations] 16 minutes ago

Analysis of all project files done 1 minute ago
```

It indicates that there are two potential bugs. Open the one that says “A ‘NullPointerException’ could be thrown…”. It says there is a potential bug in our equals method. The last three methods that you see in the CustomFileReader class have the keyword “@Override” above each declaration. This means it is rewriting a method with the exact same name from the inherited Object class. Overriding these methods for your custom classes can be useful but only if they are implemented correctly.

You may see another issue after running SonarLint. Don't worry about it for now, we will fix it eventually.

5. Something that is easy to forget when writing your own equals method is that a null object can be passed in as a parameter. Since we don't do a null check before we start calling methods on the object, we run the risk of a null pointer exception. To fix this, we need to check if the object is null, and if it is, return false. Add an if statement similar to the one below at the top of your equals method.

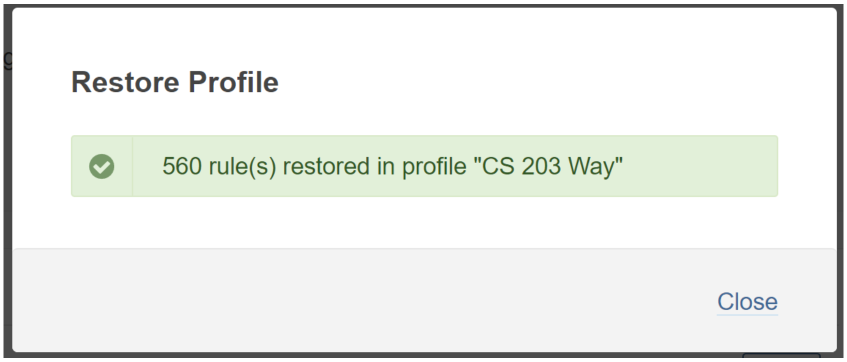


```
if(object == null)
{
    return false;
}
```

6. Run SonarLint again and you should see the “Change this ‘try’ to a try-with-resources” bug. In the bottom right pane, you should see a description of why this is an issue. More effort is actually needed to fix this code. It requires us to not use the Scanner dynamically, which will make our CustomFileReader class more robust. We will fix it later.

Configure SonarQube

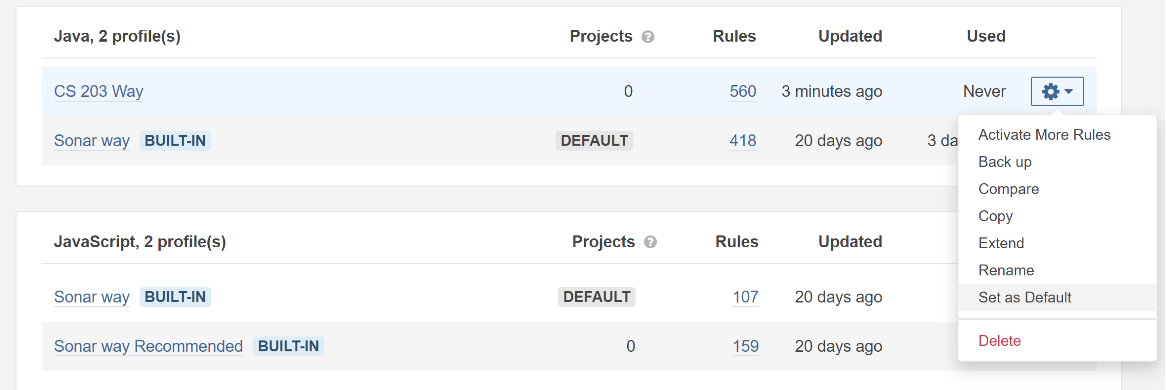
7. Before we connect SonarLint with the SonarQube instance you started in the topic preparation from this module, we will need to do some configuration. Log in to your SonarQube instance at URL: http://localhost:9000. Make sure you are logged in as Administrator by checking the upper right corner. This is needed to make the changes we want to our SonarQube instance.
8. We will first configure a Quality Profile. Quality Profiles contain the rules which we want to enforce on our project. The rule set you would need is this [Quality Profile File](). Download that ruleset and upload it to your SonarQube instance. You can do so by clicking on the “Quality Profiles” tab and then click the “Restore” button close to the upper right corner. You should see this popup when the ruleset was successfully imported:



Restore Profile

560 rule(s) restored in profile "CS 203 Way"

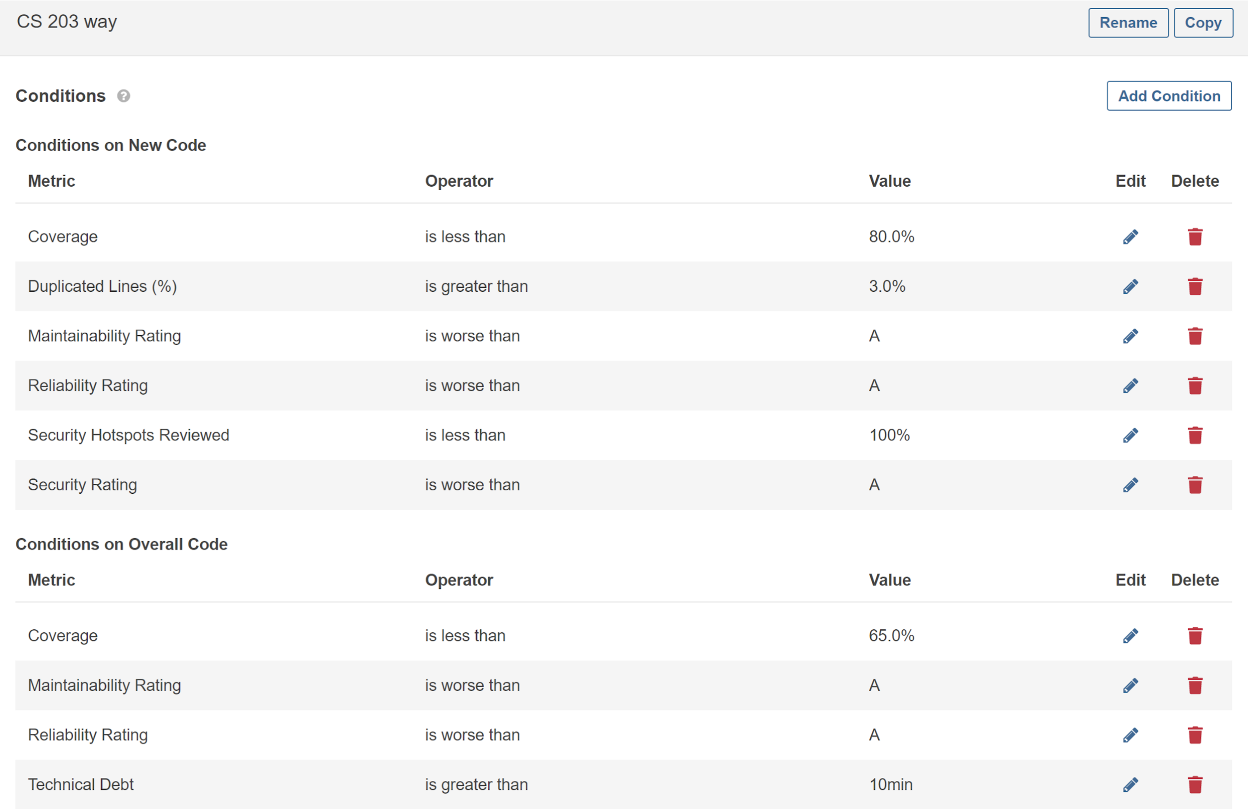
9. Now find the profile in the list of quality profiles under the “Java” subsection. You can see that the currently configured default is the “Sonar way” profile. For this assignment, make CS 203 Way the default configuration by clicking on “Set as Default” as shown below:



Java, 2 profile(s)	Projects	Rules	Updated	Used
CS 203 Way	0	560	3 minutes ago	Never
Sonar way BUILT-IN	DEFAULT	418	20 days ago	3 da

JavaScript, 2 profile(s)	Projects	Rules	Updated
Sonar way BUILT-IN	DEFAULT	107	20 days ago
Sonar way Recommended BUILT-IN	0	159	20 days ago

10. Now we will configure Quality Gates. You have learned about Quality Gates in the reading. The default Quality Gate called “Sonar way” does not check existing code, so we would need to make some modifications to it. Click the “Quality Gates” tab to start configuring a Quality Gate.
11. We still would like the existing configuration in the “Sonar way” quality gate, so click on the “Copy” button on the upper right corner to copy the configuration. Name the configuration “CS 203 way” and then click on “Copy”.
12. Open up the copied configuration, and set it as default by clicking on “Set as Default” on the upper right corner. After you have done that, add conditions for “Conditions on Overall Code” until your quality gate looks like this:



CS 203 way

Conditions

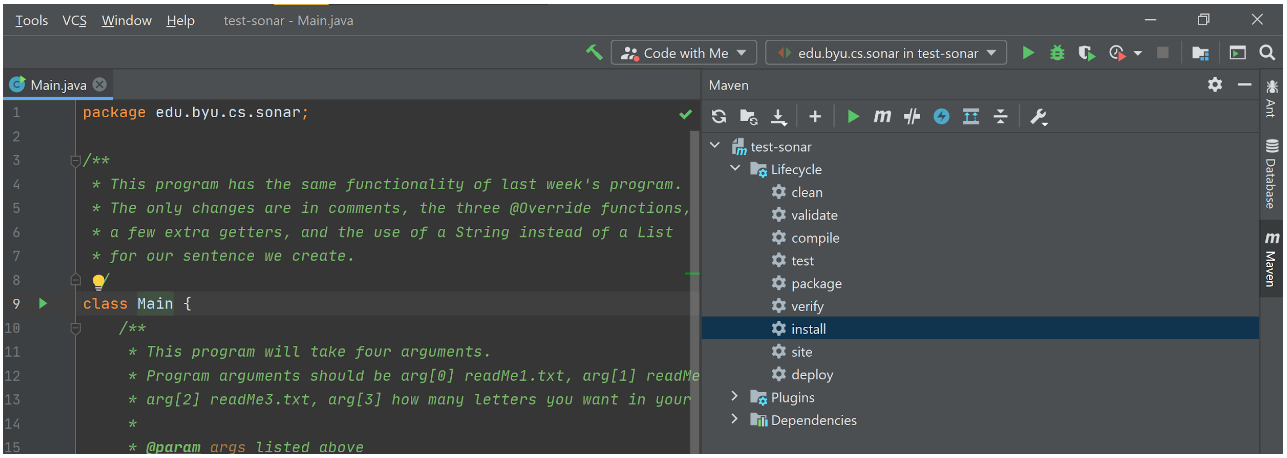
Conditions on New Code

Metric	Operator	Value	Edit	Delete
Coverage	is less than	80.0%	✎	🗑
Duplicated Lines (%)	is greater than	3.0%	✎	🗑
Maintainability Rating	is worse than	A	✎	🗑
Reliability Rating	is worse than	A	✎	🗑
Security Hotspots Reviewed	is less than	100%	✎	🗑
Security Rating	is worse than	A	✎	🗑

Conditions on Overall Code

Metric	Operator	Value	Edit	Delete
Coverage	is less than	65.0%	✎	🗑
Maintainability Rating	is worse than	A	✎	🗑
Reliability Rating	is worse than	A	✎	🗑
Technical Debt	is greater than	10min	✎	🗑

13. There are different ways to initialize a project to connect to SonarQube. We will use Maven to initialize our project. The pom.xml file we provided is configuration to build and send reports directly to your SonarQube server. All you have to do is run the “mvn install” command, which you can do in IntelliJ via the following:



```
package edu.byu.cs.sonar;

/**
 * This program has the same functionality of last week's program.
 * The only changes are in comments, the three @Override functions,
 * a few extra getters, and the use of a String instead of a List
 * for our sentence we create.
 */
class Main {
    /**
     * This program will take four arguments.
     * Program arguments should be arg[0] readMe1.txt, arg[1] readMe
     * arg[2] readMe3.txt, arg[3] how many letters you want in your
     *
     * @param args listed above
```

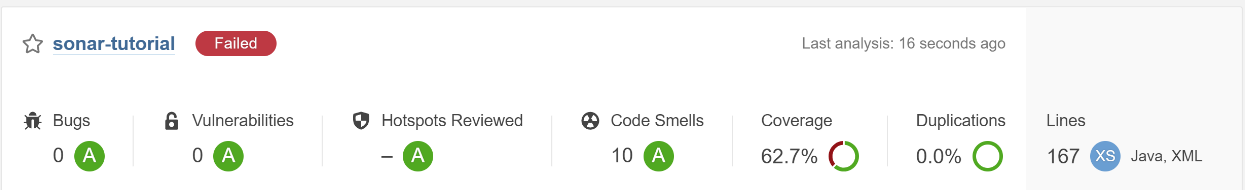
If you imported your project into IntelliJ, you can use the built-in Maven commands by clicking on the Maven side menu on the right, then click on sonar-tutorial > Lifecycle > Install > Run Maven Build. If you do not see the Maven side menu in IntelliJ, you will need to click on View > Tool Windows > Maven. If you didn’t use IntelliJ, you can also run the command “mvn install” manually in the terminal if you had Maven installed on your system.

NOTE: **If running “maven install” reports an error saying that Sonar was not authorized to analyze code, you need to add your login and password to the pom.xml file inside the “properties” section.

<sonar.login>YourLogin</sonar.login>
<sonar.password>YourPassword</sonar.password>

No matter how you run “mvn install”, you should see in the output that a sonar-maven-plugin ran in part of the build. You will use this method to update the SonarQube console for the rest of this assignment.

14. Now go back to your SonarQube instance at http://localhost:9000. You should see the project that was just built in the Projects page like this:



sonar-tutorial Failed

Last analysis: 16 seconds ago

Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Lines
0 A	0 A	– A	10 A	62.7%	0.0%	167 XS Java, XML

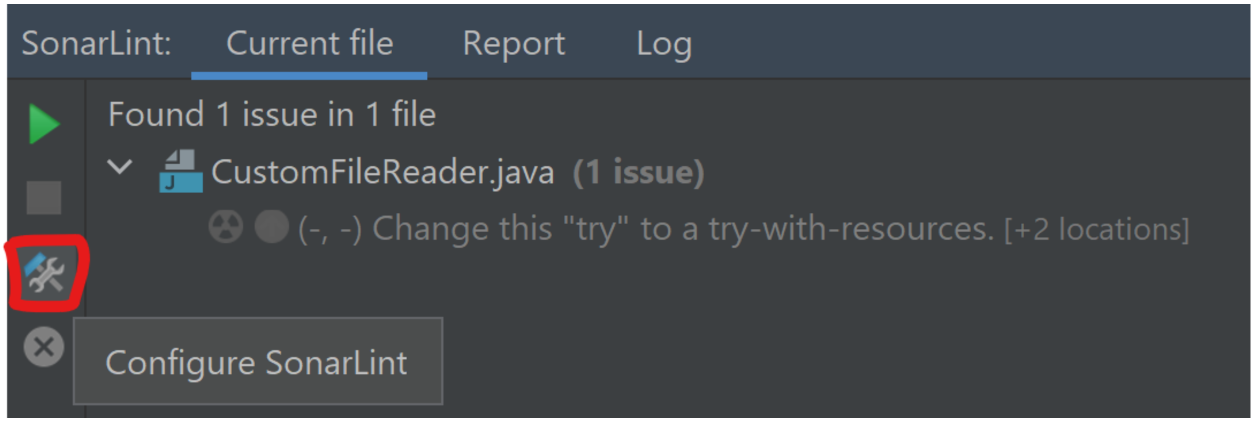
15. Click on the project name to see what failed. On the left, you will see the two reasons why the gate failed. This was what we configured in step 13. You should see that there are two types of measurements for Quality Gates: New code and Overall code. This is

separated because Sonar believes that it is more productive for developers to ensure their new code is good rather than spending a lot of time fixing old code.

16. One of the quality measurements we configured was “Technical Debt”. It is an estimate of how long it takes for a developer to fix an issue that was detected by SonarQube (though it tends to estimate more than it will actually take to fix it). Click on the “Issues” tab under the “sonar-tutorial” project name to see all the issues found by SonarQube. We will now connect SonarLint to point out these issues in our IDE.

SonarLint Connected Mode

17. In IntelliJ, click on the SonarLint tab on the bottom of the IDE. Then click on the Settings icon shown here:

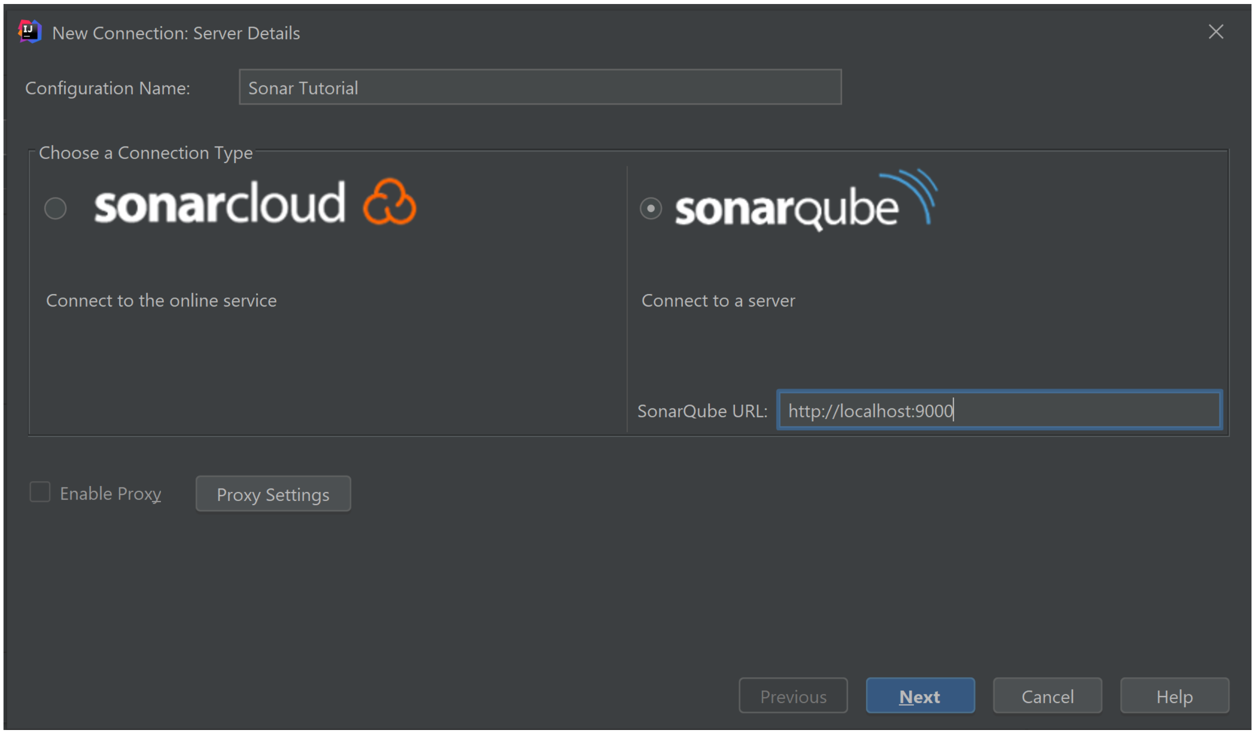


Found 1 issue in 1 file

CustomFileReader.java (1 issue)

(-, -) Change this "try" to a try-with-resources. [+2 locations]

18. In the popup, check the box that says “Bind project to SonarQube / SonarCloud”. Then, click on “Configure the connection”.
19. You should see an empty list. Click on the “Add” button on the right side to see a new pop up asking for a configuration name. Name your configuration “Sonar Tutorial”, then select “SonarQube” as the connection type and enter the SonarQube instance URL: http://localhost:9000. Make sure your configuration is the same as below, then click “Next”.



Configuration Name: Sonar Tutorial

Choose a Connection Type: ○ sonarcloud ● sonarqube

SonarQube URL: http://localhost:9000

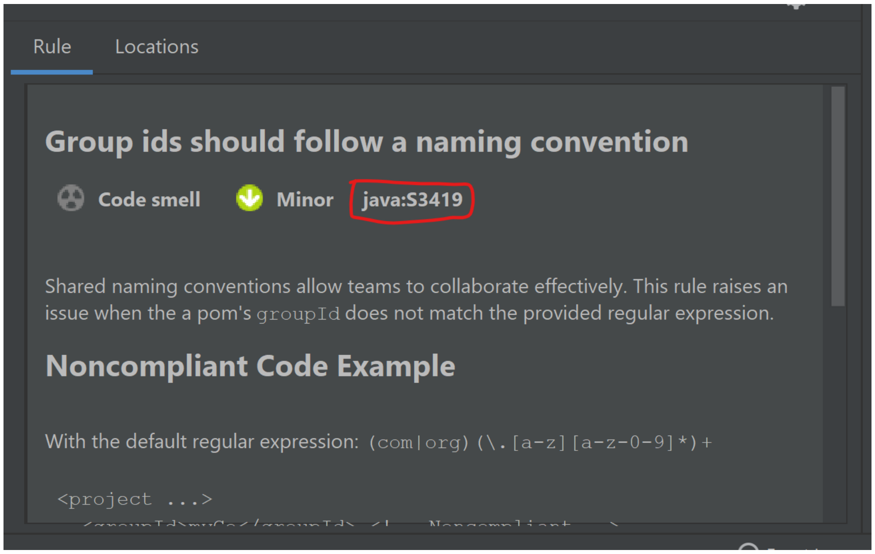
□ Enable Proxy Proxy Settings

20. You are now shown a popup to authenticate your connection to SonarQube. Normally, you may receive an authorization token to login to SonarQube. Since you created your own SonarQube instance, you can just enter your login credentials for SonarQube instead. Open the dropdown for “Connection Type”, then select “Login / Password”. Enter the same login you have used to log in to SonarQube (If you never changed it, the username and password should be “admin”).
21. After clicking OK, click on “Search in list...”, which will show the project that we imported into SonarQube using Maven earlier. Select the project, then click OK. You may see a warning, but to check whether or not the connection was successful, run the SonarLint analysis by following step 4. You should see the issues that you were seeing on the SonarQube dashboard in the SonarLint pane.

We will be doing three things to resolve these issues for this assignment. You are going to modify a rule to not show an issue, suppress an issue, and fix an issue.

Updating a Quality Profile

22. You should have seen an issue pointed out by SonarLint regarding our pom.xml file. Click on this issue to go to it. A “Rule” pane will show on the bottom right corner why this is a violation. Read through the rule to see why this was seen as a violation.
23. It looks like SonarQube only accepts “com” and “org” as valid starting names of group ids, which will not work for us since edu is valid for educational settings. We need to update that rule to allow our future projects to not show that violation. First copy the ID number of the rule here:



Group ids should follow a naming convention

Code smell · Minor · java:S3419

Shared naming conventions allow teams to collaborate effectively. This rule raises an issue when the a pom's `groupId` does not match the provided regular expression.

Noncompliant Code Example

With the default regular expression: `(com|org)(\.[a-z][a-z-0-9]*)+`

```
<project ...>
  <groupId>myCo</groupId> <!-- Noncompliant -->
```

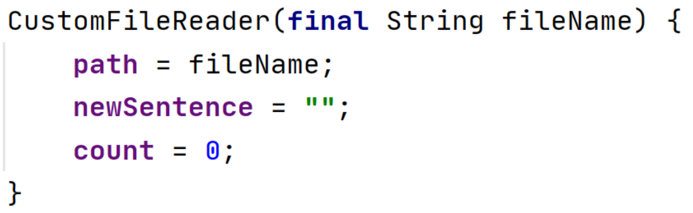
24. Now, open up SonarQube, make sure you are logged in, and go to the “Rules” tab. Search for the rule and open it up. On the bottom, it shows that our quality profile “CS 203 Way” has this rule enabled. On the right of that, you should be able to change the rule by clicking on the “Change” button. We can now add “edu” as part of a valid starting name. Paste this regex in and click “Save”:
    `(com|org|edu)(\.[a-z][a-z-0-9]*)+`
25. Now in IntelliJ, follow step 17 to open the SonarLint settings. Then click on “Configure the connection”, then click on “Update bindings”. Now SonarLint has the latest updates from SonarQube. After running SonarLint on the project by following step 4, you should not see that rule violation anymore.

Suppressing Rules

26. Now open the rule violation “Replace this use of System.out or System.err by a logger” under Main.java. Read the rule to understand why this was pointed out to be a violation.
27. Since our project is a console program, we would need to use System.out. So this rule shouldn’t apply to this specific program. We can suppress this by adding this right above class Main:
    ```
    // A console program that prints to System.out
    @SuppressWarnings("squid:S106")
    ```
28. “S106” is the rule identifier for the violation we want to suppress. Adding a comment above helps other developers know why this is here. After this is added, run SonarLint on your whole project and you would see the rule gone with an info level code smell that lets you track when SuppressWarnings is used in your project. You do not have to worry about this rule in SonarLint.

Fixing Rule Violations

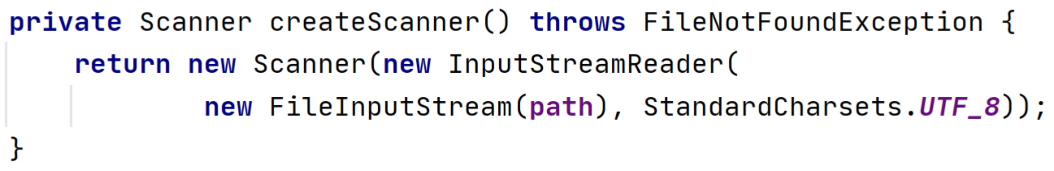
29. Now let's fix the other rule violation in Main.java: "Define the locale to be used in this String operation." Open and read why this is marked as a violation.
30. After you read this section, you should be able to understand that Sonar wants you to specify what language you are trying to use when calling .toLowerCase(). This is because other languages may have different characters than English when converting to lowercase or other things that would cause undefined behavior. We will specify the language we are using to be English by adding Locale.US inside the parameter of our lowercase function, like this .toLowerCase(Locale.US). This will ensure that if we call the lowercase function on any non English words, they will be ignored. Make sure to import the Locale data type in your Source class by moving your cursor prompt over the word, pressing Alt + Enter on your keyboard, and then clicking "import class".
31. Run SonarLint again and that violation should disappear.
32. Now we will fix the three issues found by SonarLint in the constructor of CustomFileReader. These are related to the flaws of the current implementation of our CustomFileReader class. First read through and understand what the three issues mean. Here are some of the issues with our CustomFileReader: It never closed a file properly after reading it; the way the exception is handled would not be clear to the end user; and like the last violation we fixed, does not consider what happens when a user is using a machine with a different encoding. There are different ways to fix these issues with different tradeoffs. We will fix it by removing the Scanner field in CustomFileReader and creating a Scanner every time a method that needs it gets called. Before moving on, run all the tests in CustomFileReaderTest to make sure they all pass.
33. Let's first fix the CustomFileReader. First, let's change the constructor to this:



```
CustomFileReader(final String fileName) {
    path = fileName;
    newSentence = "";
    count = 0;
}
```

This would mean that you would need to replace the Scanner s member variable with a String path variable. Go to the variable declaration of Scanner s. Change it to be "String path" instead. Now change the Scanner getter method to the String path getter method, and finally update the equals method to reflect the changes.

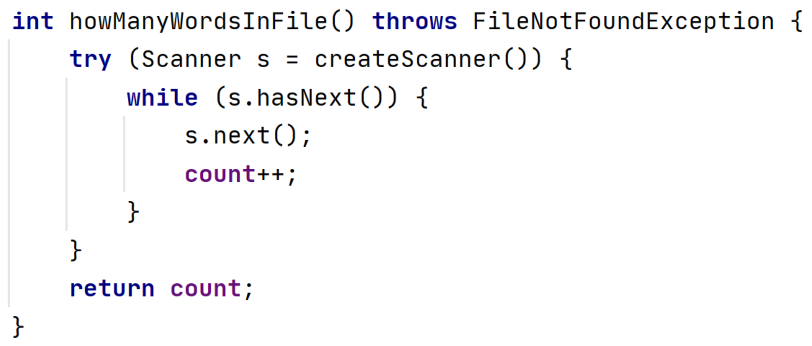
34. Now insert this private method that our other methods can use to instantiate a Scanner:



```
private Scanner createScanner() throws FileNotFoundException {
    return new Scanner(new InputStreamReader(
            new FileInputStream(path), StandardCharsets.UTF_8));
}
```

Note that this is the proper way to instantiate a Scanner to a file specifying a locale (SonarQube issue with New FileReader from Stack OverFlow).

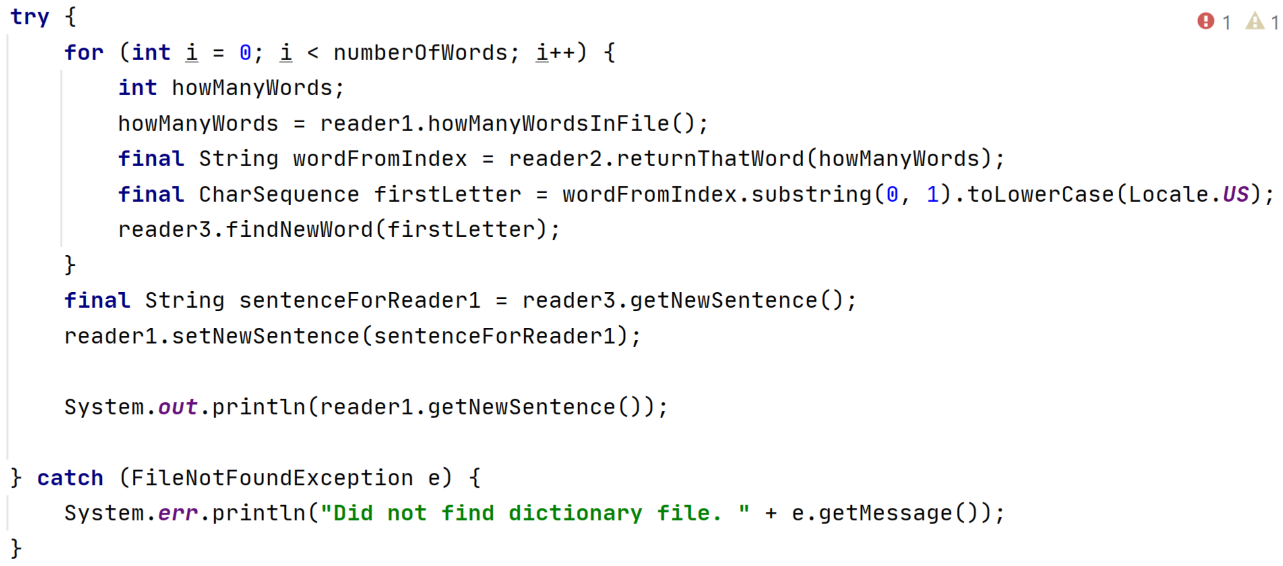
35. Lastly, we will need to change all our methods to access the scanner using try-with-resources. This is how it was done for howManyWordsInFile:



```
int howManyWordsInFile() throws FileNotFoundException {
    try (Scanner s = createScanner()) {
        while (s.hasNext()) {
            s.next();
            count++;
        }
    }
    return count;
}
```

After you change howManyWordsInFile(), follow the same pattern to fix returnThatWord() and findNewWord().

36. Now that Main is calling methods from CustomFileReader that throw FileNotFoundException (which is fine since Java's FileReader class does the same thing), put a try-catch block around the main logic like this:



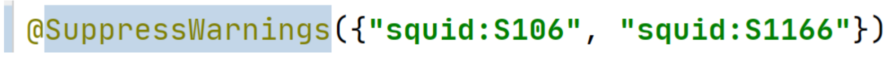
```
try {
    for (int i = 0; i < numberOfWords; i++) {
        int howManyWords;
        howManyWords = reader1.howManyWordsInFile();
        final String wordFromIndex = reader2.returnThatWord(howManyWords);
        final CharSequence firstLetter = wordFromIndex.substring(0, 1).toLowerCase(Locale.US);
        reader3.findNewWord(firstLetter);
    }
    final String sentenceForReader1 = reader3.getNewSentence();
    reader1.setNewSentence(sentenceForReader1);

    System.out.println(reader1.getNewSentence());

} catch (FileNotFoundException e) {
    System.err.println("Did not find dictionary file. " + e.getMessage());
}
```

A nice shortcut for this is to highlight the logic, then do Ctrl + Alt + T, then select "Try/Catch".

37. To suppress the warning that you would see for the catch clause (since it is basically the same issue we suppressed as before), append the rule ID to the @SuppressWarning annotation on the top of Main like this:



```
@SuppressWarnings({"squid:S106", "squid:S1166"})
```

38. Lastly, update the CustomFileReaderTest.java to allow it to run. You can do it easily by clicking on the red squiggles and do Alt + Enter (Option + return for Mac). Then select "Add Exception to Method Signature".

39. All the tests should pass. When they do, run a "mvn install" again and look at the dashboard.

SonarQube Lab

To complete this assignment, fix all the violations in the source code until the SonarQube dashboard quality gate shows the project to pass. Use “mvn install” to update the dashboard with your changes in the code. Pay attention to the quality dashboard for any new issues SonarQube has found. You can refresh SonarLint as mentioned in step 4 to show the new issues in IntelliJ. You may not suppress any warnings other than the ones mentioned in the tutorial.

Remember to use the documentation provided by Sonar as it can be a helpful hint in knowing what each violation means, and how to fix it. You will need to create more tests to increase the code coverage, but only enough to meet the Quality Gate standards that we have set in step 12 of the tutorial.

Make sure that IntelliJ’s static analyzer also doesn’t detect any further violations in your code while making your changes. Also refresh SonarLint, run the program, and run the tests regularly to make sure that new issues did not come up.

Submit all the items listed below in the “Submission” section to Canvas by the due date to receive credit for this assignment.