Tutorial & Lab: IntelliJ Static Analyzer

Using the static analyzer to write better code

Introduction

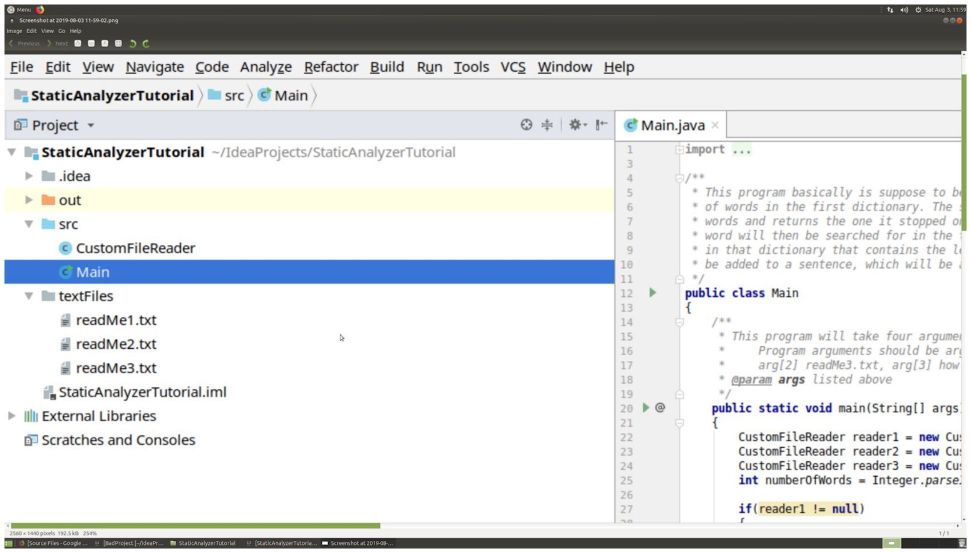
The Static Analyzer is a powerful tool that helps you write better, cleaner, and sometimes faster code without changing what your code actually does. With the help of a static analyzer you can avoid many run time errors that would otherwise cost you hours of debugging. Keeping your code simple and reliable is a skill that every programmer should have as they begin to work on larger projects.

In this tutorial, we will be using the pre-installed static analyzer that comes with Intellij to find and fix any anomalies it finds. A quick description of the program is as follows, you can also find the same description in the code itself:

> *This program uses three instances of a CustomFileReader class to interact with three separate dictionaries. The first reader counts the number of words in the first dictionary. It then passes this number to the second reader which reads that many words from the second dictionary and gives the word it stopped on to the third reader. The third reader will use the first letter of this word to find the first word in the third dictionary that contains this letter and add it to a list. The sentence will be as long as the user specifies in the program arguments. The program continues this pattern until the number of words we want for our sentence is met. In the end we give the sentence constructed by the third reader to the first reader and display it.*

Tutorial

1. Start by creating an empty project in Intellij called StaticAnalyzerTutorial, and add the two .java classes provided for you in the src directory of your new project (the files you need for this tutorial are in this zip file). Then right-click on the top-level folder of your project, and go to New > Directory and name it textFiles. Copy the three .txt files into the new directory you just created. Your project structure should look something like this.

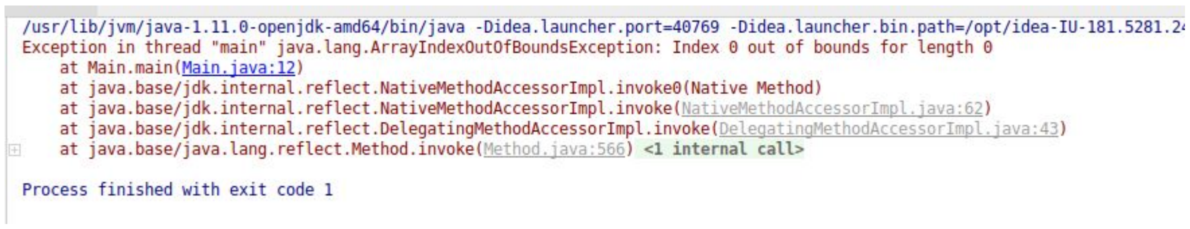

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
StaticAnalyzerTutorial 〉 src 〉 Main
Project
StaticAnalyzerTutorial ~/IdeaProjects/StaticAnalyzerTutorial
.idea
out
src
CustomFileReader
Main
textFiles
readMe1.txt
readMe2.txt
readMe3.txt
StaticAnalyzerTutorial.iml
External Libraries
Scratches and Consoles
Main.java

```
import ...

/**
 * This program basically is suppose to b
 * of words in the first dictionary. The
 * words and returns the one it stopped o
 * word will then be searched for in the
 * in that dictionary that contains the l
 * be added to a sentence, which will be
 */
public class Main
{
    /**
     * This program will take four argume
     *     Program arguments should be ar
     *     arg[2] readMe3.txt, arg[3] how
     * @param args listed above
     */
    public static void main(String[] args
    {
        CustomFileReader reader1 = new Cu
        CustomFileReader reader2 = new Cu
        CustomFileReader reader3 = new Cu
        int numberOfWords = Integer.parse

        if(reader1 != null)
```

2. To get your project to run, you can press the green play(run) button next to the main method in the Main.java file. Notice that when you pressed the green play button, the console window showed that an IndexOutOfBoundsException was thrown.



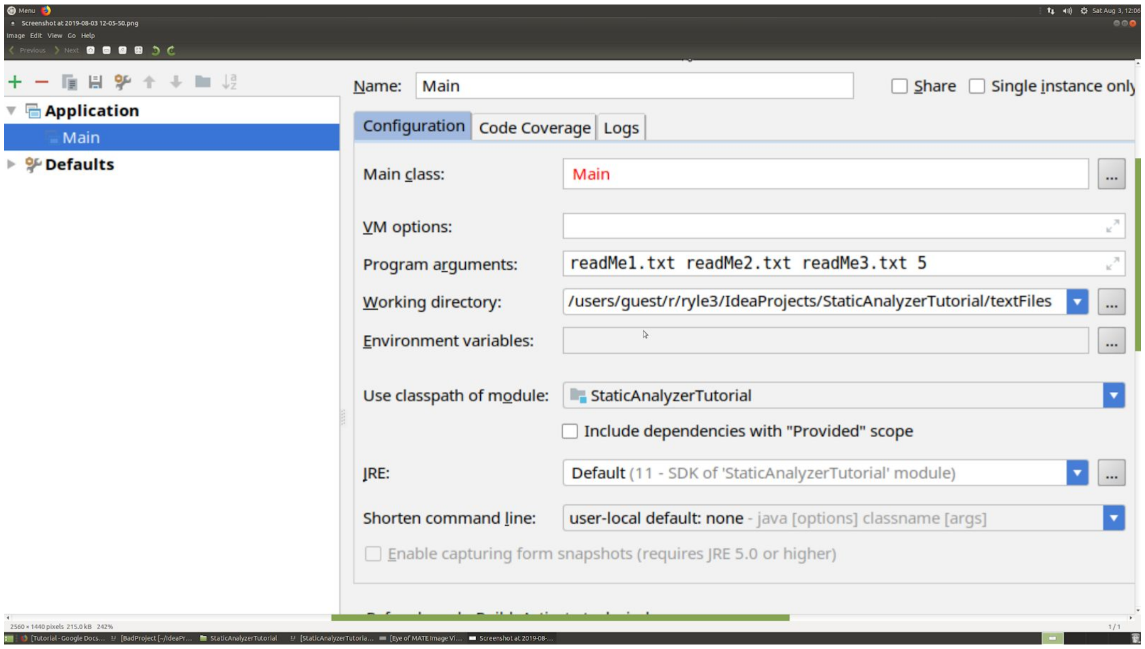
```
/usr/lib/jvm/java-1.11.0-openjdk-amd64/bin/java -Didea.launcher.port=40769 -Didea.launcher.bin.path=/opt/idea-IU-181.5281.2
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
    at Main.main(Main.java:12)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:566) <1 internal call>

Process finished with exit code 1
```

This is because your Main class is trying to read program arguments that have not been provided yet.

To solve this, click on the drop down menu at the top right, which should now say Main, then click on "Edit Configurations...".This will bring up the Run/Debug Configurations window, with a tab called Configuration. Make sure in that tab the Program Arguments line is filled in with the appropriate program arguments specified at the top of main, ie- "readMe1.txt readMe2.txt readMe3.txt 6" (the arguments should not be enclosed in quotes). For this tutorial, we will just keep the sentence length at 6.

Also in your configurations window, make sure the Working Directory line contains the path all the way to (and includes) the testFiles directory. To do this you will need to add /textFiles to the end of the path. This is important because it tells the program where to look for the .txt files. Here is an example of a correctly configured run configuration.


Name: Main

Main class: Main

VM options:

Program arguments: readMe1.txt readMe2.txt readMe3.txt 5

Working directory: /users/guest/r/ryle3/IdeaProjects/StaticAnalyzerTutorial/textFiles

Environment variables:

Use classpath of module: StaticAnalyzerTutorial

Include dependencies with "Provided" scope

JRE: Default (11 - SDK of 'StaticAnalyzerTutorial' module)

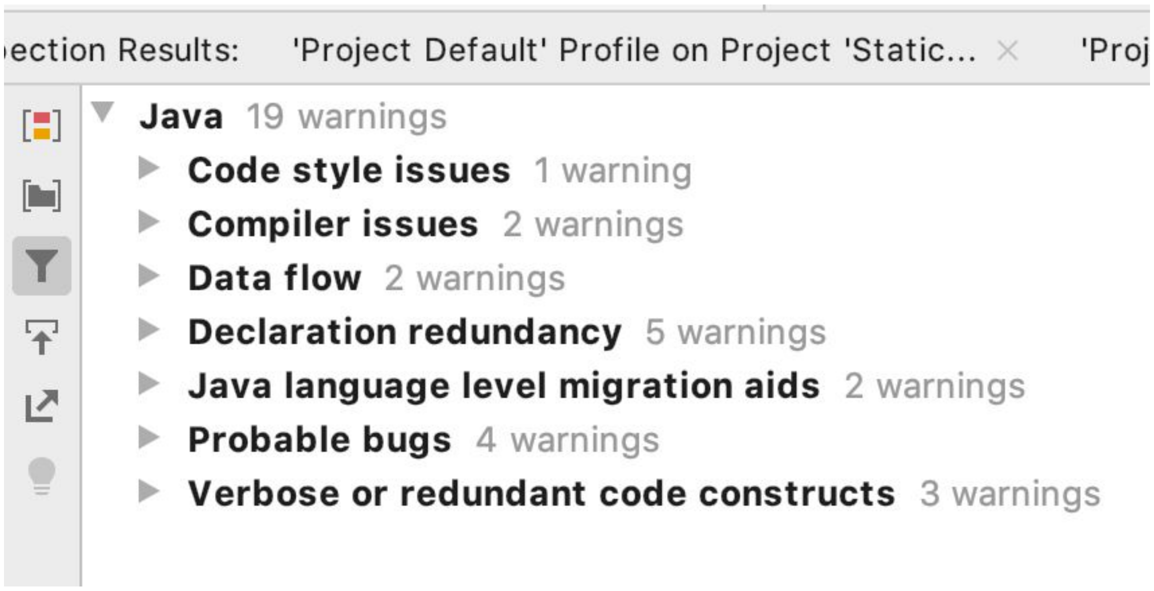
Shorten command line: user-local default: none - java [options] classname [args]

Enable capturing form snapshots (requires JRE 5.0 or higher)

If you set everything up correctly, after you have pressed the green play button, you should see the sentence “[was, times, was, the, worst, belief]” displayed in the console window. This means your program is working.

3. Now lets run the static analyzer on this code and see if there are any warnings/anomalies. Go to the top menu bar and click on Analyze > Inspect Code. Make sure the Inspection Scope is set to the whole project then click ok to run the analyzer. You should now see a window reporting that there are 19 java warnings we should fix.

   Note: Depending on your Java version and editor settings, you may see slightly more or fewer warnings. If we ask you to fix a warning you don’t see, just skip that step.



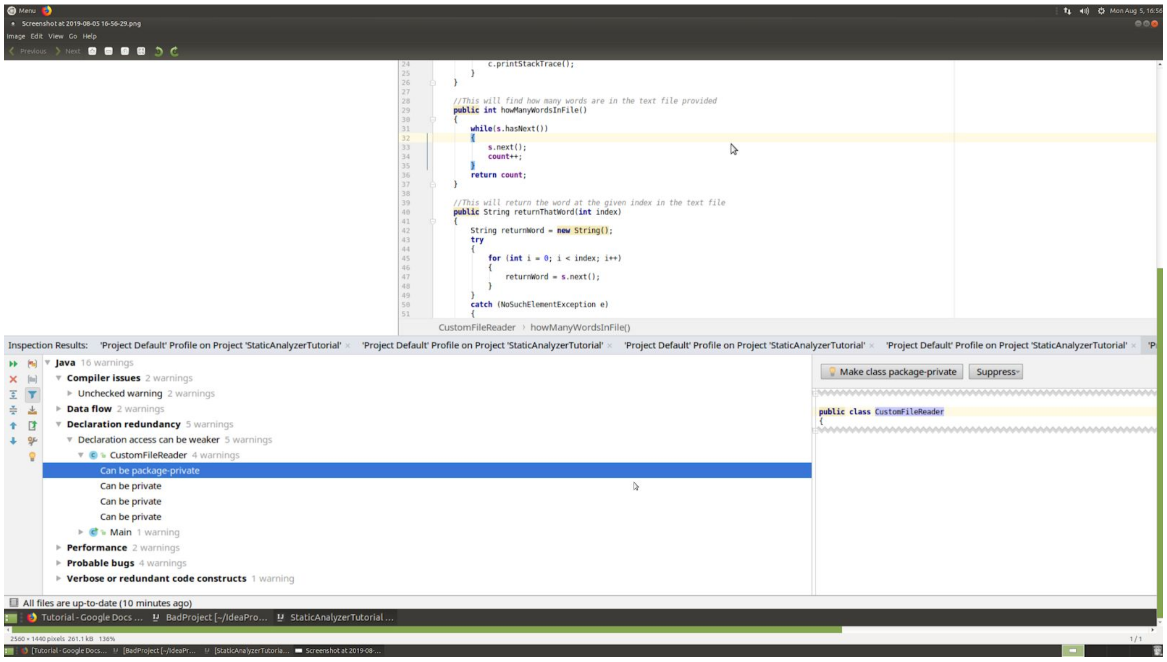
ection Results: 'Project Default' Profile on Project 'Static... 'Proj

- Java 19 warnings
  - Code style issues 1 warning
  - Compiler issues 2 warnings
  - Data flow 2 warnings
  - Declaration redundancy 5 warnings
  - Java language level migration aids 2 warnings
  - Probable bugs 4 warnings
  - Verbose or redundant code constructs 3 warnings

4. The first warning is under a heading called "**Code style issues**". Expand the view and you will see a description of the first warning that says "Class 'CustomFileReader' explicitly extends 'java.lang.Object'". Double-click this warning and the file screen you are looking at will take you to the exact place of the issue. In Java, every class we create extends from the Object base class by default. Including this extension ourselves is redundant and possibly more confusing. To fix this, erase the portion of the code that says "extends Object" or in the window to the right, click the button that says "Remove redundant extends object".

5. Looking back at the Inspection Results, there should now only be 18 warnings and the title "**Code style issues**" is gone because we have resolved everything under that category.

6. The next warning we will handle is under the title "**Declaration redundancy**". If you expand the view you will see that in total there are 5 warnings, 4 in CustomFileReader and 1 in Main.



```
            c.printStackTrace();
        }
    }

    //This will find how many words are in the text file provided
    public int howManyWordsInFile()
    {
        while(s.hasNext())
        {
            s.next();
            count++;
        }
        return count;
    }

    //This will return the word at the given index in the text file
    public String returnThatWord(int index)
    {
        String returnWord = new String();
        try
        {
            for (int i = 0; i < index; i++)
            {
                returnWord = s.next();
            }
        }
        catch (NoSuchElementException e)
        {
```

```
Inspection Results: 'Project Default' Profile on Project 'StaticAnalyzerTutorial'
Java 16 warnings
  Compiler issues 2 warnings
    Unchecked warning 2 warnings
  Data flow 2 warnings
  Declaration redundancy 5 warnings
    Declaration access can be weaker 5 warnings
      CustomFileReader 4 warnings
        Can be package-private
        Can be private
        Can be private
        Can be private
      Main 1 warning
  Performance 2 warnings
  Probable bugs 4 warnings
  Verbose or redundant code constructs 1 warning
```

```
public class CustomFileReader
{
```

Make class package-private
Suppress
All files are up-to-date (10 minutes ago)

When programming, we should limit the accessibility of our code as much as possible. This prevents unnecessary code dependencies and results in better, more maintainable code. In Java we use packages to hold related code. Classes in the same package can declare their methods in a way that gives access to only the other classes in the same package.

Since all of our classes reside in one package, we can give them access to each other without declaring them public (which would allow other classes in a different package to access them).

To do this, we want to look at the two warnings that say “Can be package-private”. You can fix these by clicking on the button to auto fix them, or you can remove the keyword “public” in the appropriate places. Double-click on the message to make sure you understand where the warning is located.
For these warnings, make sure that you don’t replace “public” with “private” when trying to fix them. Even though our classes are in the same package, the keyword “private” allows only the class itself to access the portions of code labeled as private.

You should now be down to 16 warnings. We will fix the other warnings under “**Declaration redundancy**” in the next step.

7. Run the static analyzer again. You should still have warnings about our three variables inside CustomFileReader. That is because these variables are only accessed within that class, so in this instance it is safe, and actually preferred for those member variables to

be “private”. Make them private and run the static analyzer again. You should now be down to 13 warnings.

8. The last warnings we will be looking at in the tutorial portion of this assignment are listed under the title “**Compiler issues**”. These issues are a little more complicated to explain, but, like most warnings, they come with a fairly easy fix once we understand the problem.

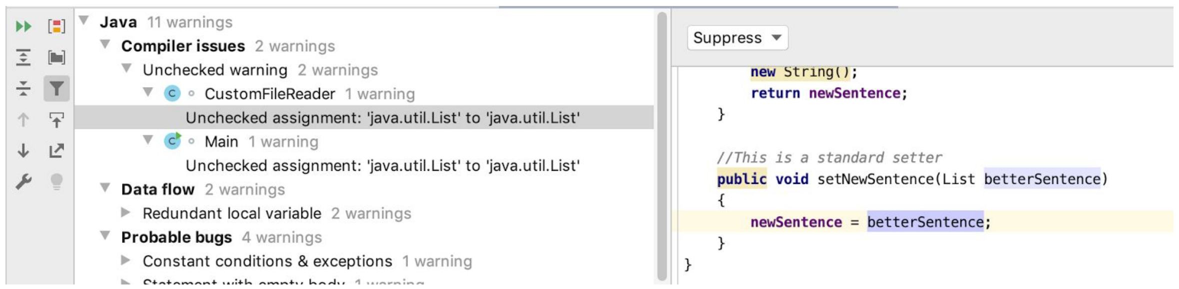
   If this is your first time learning about Java generics, it’s okay if you don’t completely understand everything in the following explanation.

   Java has something called generics, which allows us to restrict the types of objects placed in a data structure. For example, if we declare an ArrayList like this: `List myList = new ArrayList();` we are creating a list that can hold any type of object. However, that is usually more general than we need and can lead to bugs.
   Most of the time we only want to put one type of object in a data structure. In these cases, we can use Java generics when we create a data structure to specify what type of objects it can hold. If we change our declaration to this: `List<String> myList = new ArrayList<>();` we are using Java generics to specify that only Strings can be placed in our ArrayList. Now if we try to put something else in the ArrayList (like a Person object) we will get a compile error.

   Also notice in the above example that we declared the ‘myList’ variable to be of type List and created an instance of ArrayList to assign to it. The specifics are beyond the scope of this class, but just know this is okay to do because ArrayList is a subtype (child type) of List.

   In the CustomFileReader class, notice that on line 10 we declare a variable to hold our ArrayList, but we don’t create an ArrayList to assign to it until line 16. This is a common practice.

   Now we are ready to understand the next warning found in our static analysis output, which is the one highlighted in the following image:



```
Java 11 warnings
  Compiler issues 2 warnings
    Unchecked warning 2 warnings
      CustomFileReader 1 warning
        Unchecked assignment: 'java.util.List' to 'java.util.List'
      Main 1 warning
        Unchecked assignment: 'java.util.List' to 'java.util.List'
  Data flow 2 warnings
    Redundant local variable 2 warnings
  Probable bugs 4 warnings
    Constant conditions & exceptions 1 warning
```

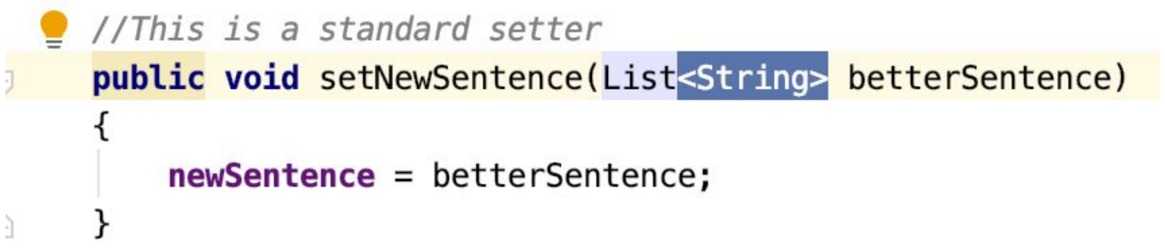
```
        new String();
        return newSentence;
    }

    //This is a standard setter
    public void setNewSentence(List betterSentence)
    {
        newSentence = betterSentence;
    }
}
```

9. Double-click this warning to go to the setNewSentence(List betterSentence) method. Notice that the parameter in this method accepts a List that does not specify a generic data type, meaning we could place any data type in that list. Line 74 assigns this list to the variable declared with the generic type of String on line 10. If we make this assignment, we will be setting the newSentence list to a list that allows any type of object to be inserted in the list. Based on how we declared the List on line 10, we intended to restrict the data type of the list to Strings.

   The static analyzer provides a warning whenever it detects a case where we declare or create a List, or any other class, that was created to use generic type information without specifying the type information. This warning is intended to prevent the above scenario, where we intended to have a List containing only Strings but the setNewSentence method doesn't match that.

10. This was a long explanation to help you understand that changing the setNewSentence method's parameter to include a generic type of String (as shown in the image below), will correct the warning. Make that correction as shown below.



```
//This is a standard setter
public void setNewSentence(List<String> betterSentence)
{
    newSentence = betterSentence;
}
```

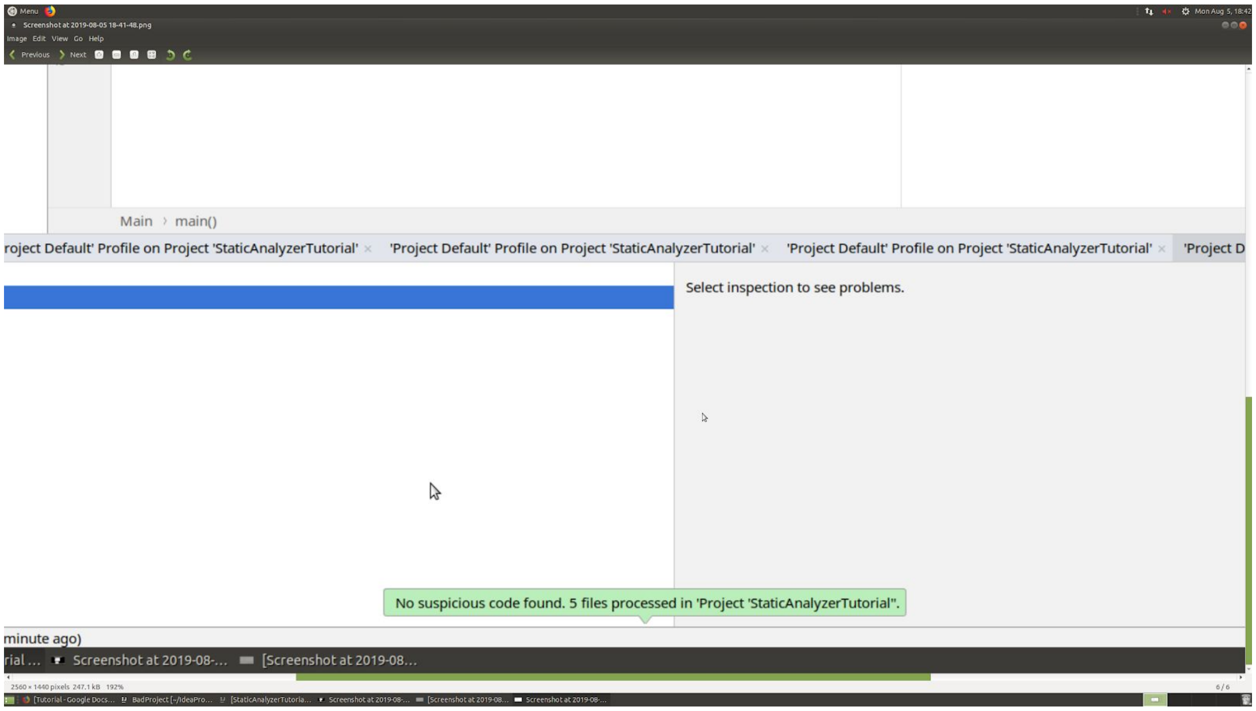
11. The other warning under "**Compiler issues**" is also about a missing generic type declaration. Double-click the warning and you will be taken to line 42 of Main.java. Here we assign a List<String> variable to the result of calling the method getNewSentence(). Right-click on getNewSentence on this line of code and select 'Go to > Declaration or Uses' and you will be taken to the declaration of the method. Notice that this method returns a list without a generic type declaration. Correct that by replacing the return type in the method signature with List<String>.

12. Run the Static Analyzer again and you should be down to 9 warnings. We resolved the most complicated warnings together. You will now resolve the remaining warnings on your own by completing the lab described in the next section.

Lab

Now that you have learned how to use the static analyzer in the tutorial, you are ready to resolve the rest of the warnings on your own. Make sure as you look at each remaining issue that you consider why each issue identified by the static analyzer is a problem, and why your code will be better when you resolve it.

As you fix each item, keep track of what changes you make and why it was an issue. You will report this information when you submit your work at the end of the lab. Running the analyzer after you have fixed all issues will result in a green bar telling you that there is no longer anything suspicious about your code. You will need a screen shot of this bar to submit for full credit.


Main › main()
roject Default' Profile on Project 'StaticAnalyzerTutorial'
'Project Default' Profile on Project 'StaticAnalyzerTutorial'
'Project Default' Profile on Project 'StaticAnalyzerTutorial'
'Project D
minute ago)
Screenshot at 2019-08-...
[Screenshot at 2019-08...

Select inspection to see problems.

No suspicious code found. 5 files processed in 'Project 'StaticAnalyzerTutorial''.

Make sure after you have fixed all the warnings that the program still runs the same way as it did in the beginning. Remember to submit all the items listed below in the “Submission” section by the due date to receive credit for this assignment.

Submission

Electronically submit (upload) the following files:

- Your revised Main.java and CustomFileReader.java classes,
- A screenshot showing the results of running the static analyzer after you have made your corrections (this should be the green bar that shows up after running the static analyzer with no warnings),
- Your list of revisions and reasons why they were necessary.