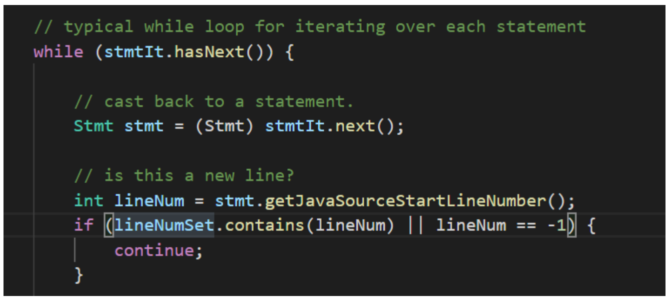
Assignment 2

In this assignment, you will be given Java programs of an open source project and the associated tests. We expect to report the statement coverage, and branch coverage achieved by executing test cases. More specifically, you need to report the ***statement coverage*** and ***branch coverage*** for ***each method*** in the target package. This assignment will help you understand how test coverage is calculated.

You can count the statement coverage as follows:

$$statement\ coverage = \frac{the\ statements\ that\ have\ been\ executed\ \boldsymbol{dynamically}}{the\ total\ number\ of\ statements\ in\ this\ method}$$

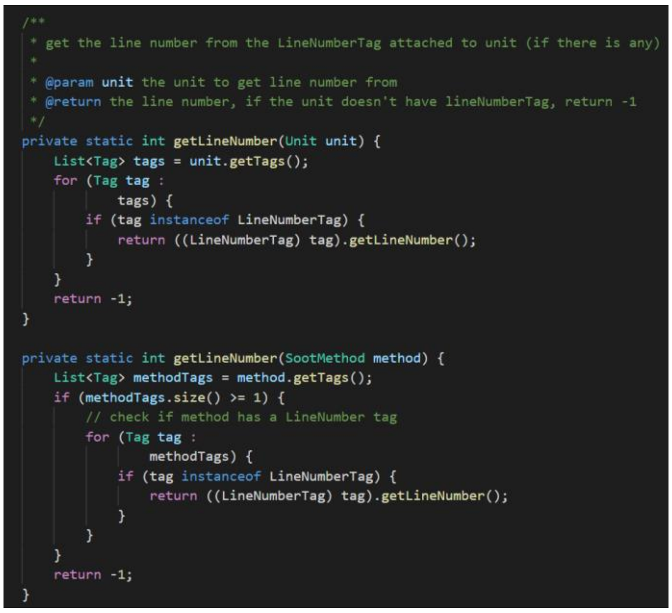
To count the covered statement and the total number of statements, we need to identify the mapping between the line number of source code and that of the Jimple code. Actually, we can easily obtain the line number of each Jimple code. The following shows two examples:



```java
// typical while loop for iterating over each statement
while (stmtIt.hasNext()) {

    // cast back to a statement.
    Stmt stmt = (Stmt) stmtIt.next();

    // is this a new line?
    int lineNum = stmt.getJavaSourceStartLineNumber();
    if (lineNumSet.contains(lineNum) || lineNum == -1) {
        continue;
    }
```



```java
/**
 * get the line number from the LineNumberTag attached to unit (if there is any)
 *
 * @param unit the unit to get line number from
 * @return the line number, if the unit doesn't have lineNumberTag, return -1
 */
private static int getLineNumber(Unit unit) {
    List<Tag> tags = unit.getTags();
    for (Tag tag :
            tags) {
        if (tag instanceof LineNumberTag) {
            return ((LineNumberTag) tag).getLineNumber();
        }
    }
    return -1;
}

private static int getLineNumber(SootMethod method) {
    List<Tag> methodTags = method.getTags();
    if (methodTags.size() >= 1) {
        // check if method has a LineNumber tag
        for (Tag tag :
                methodTags) {
            if (tag instanceof LineNumberTag) {
                return ((LineNumberTag) tag).getLineNumber();
            }
        }
    }
    return -1;
}
```

It is easy to obtain the total number of statements. In order to achieve such a goal, you might need to ***compile the Java source code with the debug information enabled.***

However, ***if finding the mapping between source lines is too challenging for you***, you can simply count the total number of statements as the total number of Jimple code, and simply check how many Jimple code has been executed. Then the statement coverage is computed as the covered Jimple code divided by the total number of Jimple code.

You can count the branch coverage as follows:

$$branch\ coverage = \frac{the\ branches\ that\ have\ been\ executed\ \boldsymbol{dynamically}}{the\ total\ number\ of\ branches\ in\ this\ method}$$

To count the total number of branches and instrument branches, we need to construct the control flow graph first. Soot provides several different control flow graphs (CFG) in the package ***soot.toolkits.graph***. For example, we can obtain the CFG of a method through $\boldsymbol{UnitGraph\ g = new\ BriefUnitGraph(body)}$. For each node (statement) in the graph, we can get the successor of the node through $\boldsymbol{g.getSuccsOf(stmt)}$**.** Similarly, we can obtain the predecessors of a node through $\boldsymbol{g.getPredsOf(stmt)}$. More details could be found at:
https://www.sable.mcgill.ca/soot/doc/soot/toolkits/graph/UnitGraph.html#getSuccsOf(soot.Unit)

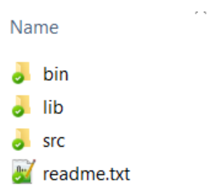
If a node has multiple successors, it indicates that there are multiple branches. Therefore, we can count the total number of branches based on such information. For those nodes with multiple successors, we can instrument at the beginning of each successor, thus to record if the branch has been executed dynamically. Usually, ***if*** statements and ***switch*** statements will have multiple successors.

Objectives

a) To get familiar with software testing and the test coverage criteria, such as statement coverage and branch coverage.
b) Learn how to instrument programs for Java.
c) Learn how to use basic functionalities of Soot, which is a code static analysis and optimization framework for Java. More details could be found in https://github.com/soot-oss/soot

Steps

1. Download the provided zip files ***assignment2.zip***, the structure of which is as follows:


Name

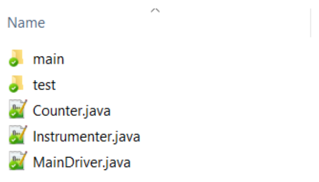
bin

lib

src

readme.txt

Please go to this folder, and all the ***commands should be run under this folder.***
Under the `src` folder, you can find the following files or directories, and the meaning of them are as follows:


Name

main

test

Counter.java

Instrumenter.java

MainDriver.java

a) `main.` It contains all the source files of an Open Source Project Apache Math. These are also the source files that we are going to instrument.
b) `test.` It contains all the test files associated with the above main source files. We are not going to instrument such files, but will execute them to record the dynamic coverage information.
You can read the above source files, but you are not required to understand them or modify them.
c) `MainDriver.java.` The main driver of our instrumentation. We have already implemented this class for you. You do not need to change it, and you can run this file to launch Soot.
d) `Counter.java.` The helper class for instrumentation. You are required to implement necessary functionalities of this class (***In this assignment, we provide a simpler version of Counter.java than that in Assignment#1, we expect you to implement most of the logics by yourself.***).
e) `Instrumenter.java.` The main class for instrumentation. You are required to implement necessary functionalities of this class in order to track statement coverage and branch coverage (***In this assignment, we provide a simpler version of Counter.java than that in Assignment#1, we expect you to implement most of the logics by yourself.***).

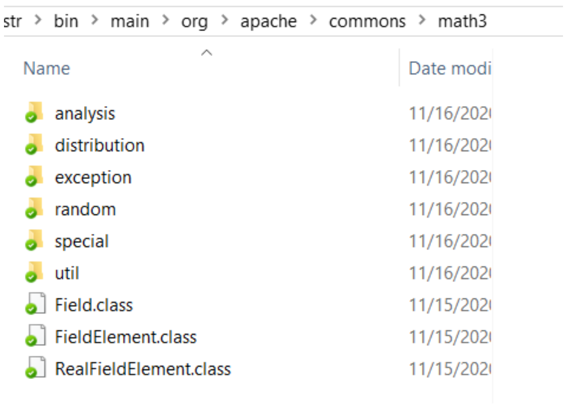
2. Implement the required functionalities in `Count.java` and `Instrumenter.java`. Please refer to the tutorial and PPT provided in the previous assignment and also your code of assignment#1. If necessary, you can also add other classes or source files by yourself.
3. After implementing the required two classes, test your implemented functionalities via run the following commands:
   a) Compile:
```
javac -cp "lib/*;src/main/" -d bin/main
src\main\org\apache\commons\math3\util\*
```
compile the target main source files. (Please copy the above command in one line)
```
javac -cp "lib/*;src/main/;src/test/" -d bin/test
src\test\org\apache\commons\math3\util\*
```

compile the test source files of the associated main source files. (Please copy the above command in one line)

`javac -cp "lib/*;src/" -d bin/ src\*.java`

compile our implemented Java source files.

if successfully, you will see the compiled class file under folder "bin". The target files will be compiled into folders of **`bin/main`** as follows:



```
str > bin > main > org > apache > commons > math3

Name                                  Date modi
analysis                              11/16/202
distribution                          11/16/202
exception                             11/16/202
random                                11/16/202
special                               11/16/202
util                                  11/16/202
Field.class                           11/15/202
FieldElement.class                    11/15/202
RealFieldElement.class                11/15/202
```

b) Instrumentation:

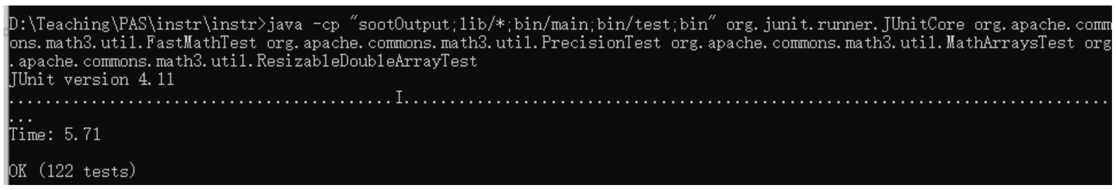
`java -cp "bin;lib/*" MainDriver`

Through this command, we can implement all the classes under folder **`bin/main`**. if successfully, you will see the instrumented class file under folder "sootOutput".

c) Run Test:

```
java -cp "sootOutput;lib/*;bin/main;bin/test;bin"
org.junit.runner.JUnitCore
org.apache.commons.math3.util.FastMathTest
org.apache.commons.math3.util.PrecisionTest
org.apache.commons.math3.util.MathArraysTest
```

Please run the above command in one line as follows



```
D:\Teaching\PAS\instr\instr>java -cp "sootOutput;lib/*;bin/main;bin/test;bin" org.junit.runner.JUnitCore org.apache.comm
ons.math3.util.FastMathTest org.apache.commons.math3.util.PrecisionTest org.apache.commons.math3.util.MathArraysTest org
.apache.commons.math3.util.ResizableDoubleArrayTest
JUnit version 4.11
.........................................I...............................................................................
...
Time: 5.71

OK (122 tests)
```

We can add as many as test classes at the end of this command as possible. In the above example, we only provide 3 test class, but we can more such as 10 test classes. The more tests included, the higher coverage should be reported.

We will check the results after running this command. If you have successfully finished this assignment, you should generate a file "**`result.txt`**" under the folder "**`report`**". We will then check and compare the results in this file.

Requirement

- Linux/Mac/Windows
- Java 1.8 (you need to install it on your computer)
- Soot 4.2.1 (we have provided it to you in “lib”. You need to do nothing)
- Junit 4.1.1 (we have provided it to you in “lib”. You need to do nothing)

Assessment

Output the statement coverage and branch coverage for each method that has been executed to the file *“report/result.txt”*. The file should be generated after running command as shown in Steps. In our assignment, please output the coverage information in the following format. Please to use the ***className*** plus the ***methodName*** (FastMath.cosh) to denote a method. There are six numbers following each method, which is shown as follows, among which:

CoveredStatement: the statements that have been executed by the tests in the method.
TotalStatement: the total number of statements in the given method.

$$\text{StatementCoverage} = \frac{CoveredStatement}{TotalStatement}$$

CoveredBranch: the branches that have been executed by the tests in the method.
TotalBranch: the total number of statements in the given method.

$$\text{BranchCoverage} = \frac{CoveredBranch}{TotalBranch}$$

```
MethodName \t CoveredStatement \ TotalStatement \t StatementCoverage
CoveredBranch \t TotalBranch \t BranchCoverage \n
FastMath. cosh \t [number] \t [number] \t [number] \t [number] \t
[number] \t [number] \n
Precision. compareTo \t [number] \t [number] \t [number] \t [number]
\t [number] \t [number]
```

…

We will compare the correctness of the outputted numbers to assess your assignment.
If you have implemented the required source files but unable to produce any results, you will get at least 60% of the score.
If you can generate results which are not nonsense (if any), you can get 80% of the score.
We will give the other 20% of the score based on the correctness of your results 😉

Submission:

After implementing the required two classes, please zip the folder “assignment2” and submit it through “微助教”.