Lab 3.1 Using splint for C static analysis

Overview

The learning objective of this lab is for students to gain the first-hand experience on using static code analysis tools to check c program for security vulnerabilities and coding mistakes.

Splint link is a tool for statically checking C programs for security vulnerabilities and programming mistakes. Splint does many of the traditional lint checks including unused declarations, type inconsistencies, use before definition, unreachable code, ignored return values, execution paths with no return, likely infinite loops, and fall through cases. More powerful checks are made possible by additional information given in source code annotations. Annotations are stylized comments that document assumptions about functions, variables, parameters and types. In addition to the checks specifically enabled by annotations, many of the traditional lint checks are improved by exploiting this additional information.

11 kinds of problems detected by Splint include:

- Dereferencing a possibly null pointer;
- Using possibly undefined storage or returning storage that is not properly defined;
- Type mismatches, with greater precision and flexibility than provided by C compilers;
- Violations of information hiding;
- Memory management errors including uses of dangling references and memory leaks;
- Dangerous aliasing;
- Modifications and global variable uses that are inconsistent with specified interfaces;
- Problematic control flow such as likely infinite loops, fall through cases or incomplete switches, and suspicious statements;
- Buffer overflow vulnerabilities;
- Dangerous macro implementations or invocations;
- Violations of customized naming conventions.

More details you can get from Splint User's Manual link.

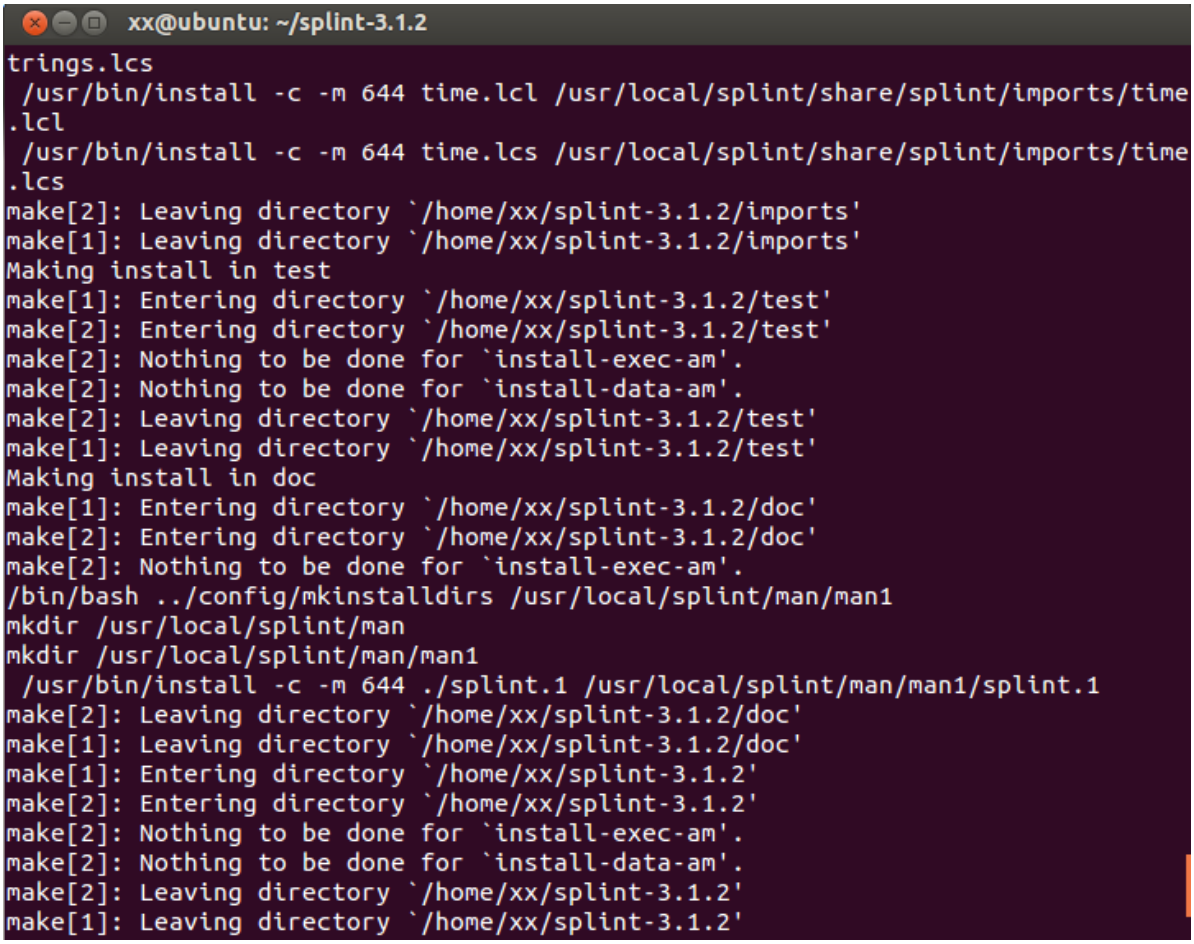
With such knowledge, your goal is to achieve the followings:

- Install splint;
- Finish code samples with 2 different kinds of problems which can be detected by Splint. You can choose any 2 of 11 problems as above.
- Use splint to detect the 2 kinds of problems. Descibe your observations in your report.

Steps

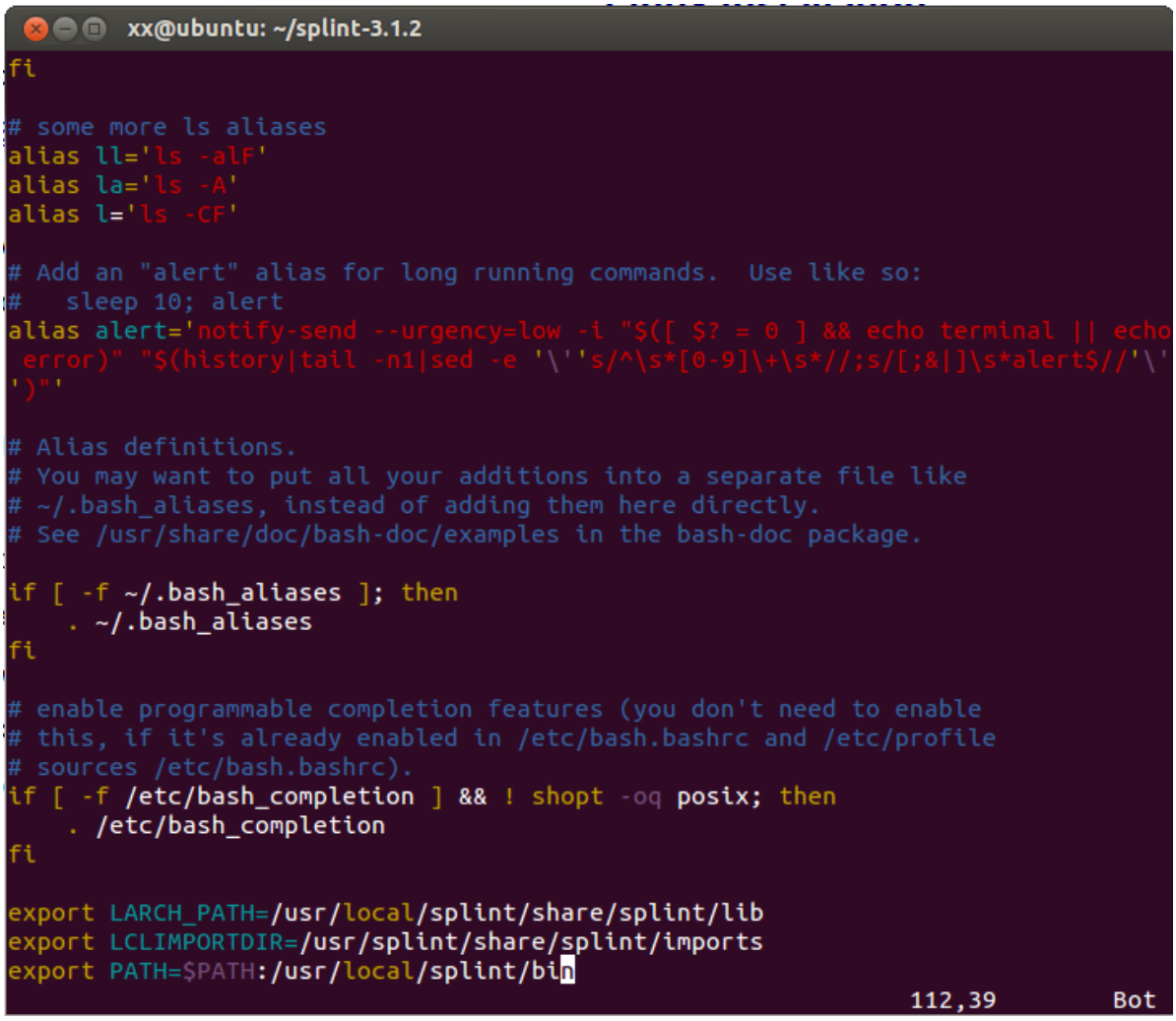
1. Download Splint source code distribution link.
2. Extract and setup Splint.

```
$ tar -zxvf splint-3.1.2.linux.tgz
$ sudo mkdir /usr/local/splint
  [sudo]password for ***: (enter password)
$ cd splint-3.1.2
$ ./configure --prefix=/usr/local/splint
$ sudo apt-get install flex
$ make
$ sudo make install
```



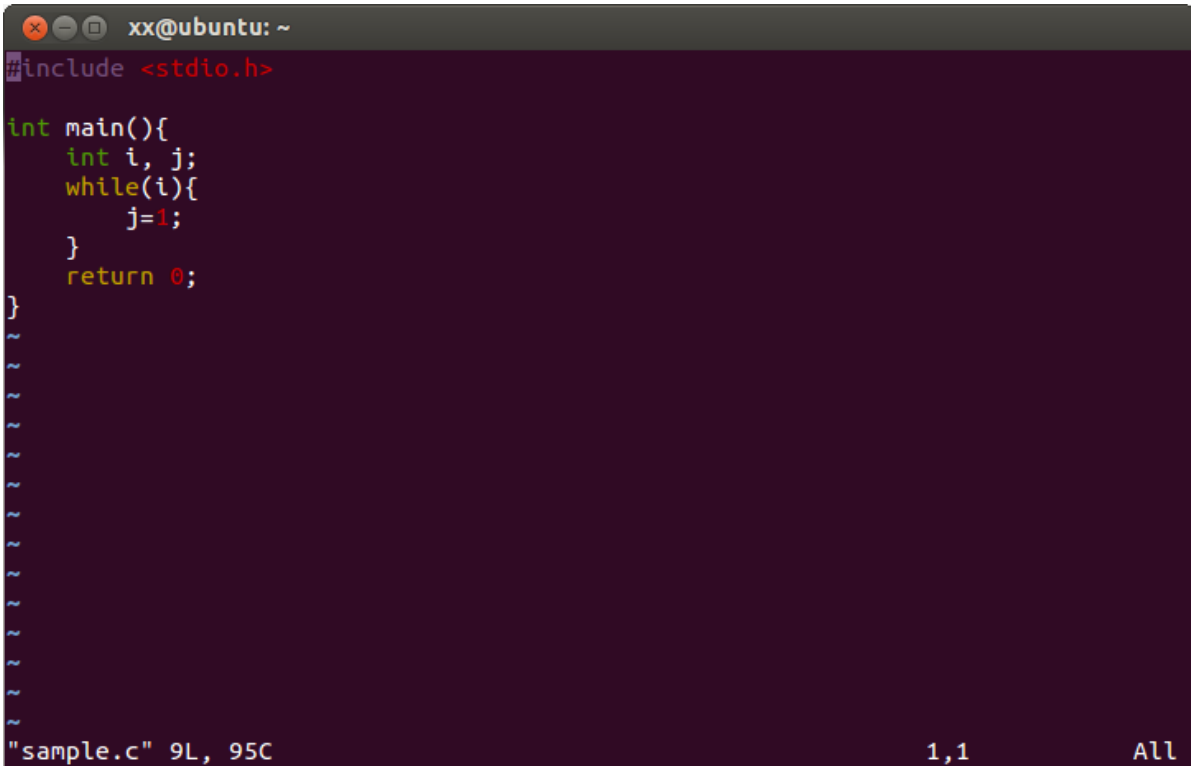
3. Use vi Text Editor to add the following to the environment variable:

```
$ vi ~/.bashrc

(add the following to the environment variable in .bashrc)
export LARCH_PATH=/usr/local/splint/share/splint/lib
export LCLIMPORTDIR=/usr/splint/share/splint/imports
export PATH=$PATH:/usr/local/splint/bin

$ source ~/.bashrc
```



4. Finish code samples with 2 different kinds of problems.

```c
#include <stdio.h>

int main(){
      int i, j;
      while(i){
            j=1;
     }
      return 0;
}
```



5. Use Splint to detect problems in code samples.

```
1 $ splint sample.c
```

```
xx@ubuntu:~$ splint sample.c
Splint 3.1.2 --- 10 Jun 2021

sample.c: (in function main)
sample.c:5:11: Test expression for while not boolean, type int: i
  Test expression type is not boolean or int. (Use -predboolint to inhibit
  warning)
sample.c:5:11: Variable i used before definition
  An rvalue is used that may not be initialized to a value on some execution
  path. (Use -usedef to inhibit warning)
sample.c:5:11: Suspected infinite loop.  No value used in loop test (i) is
                 modified by test or loop body.
  This appears to be an infinite loop. Nothing in the body of the loop or the
  loop test modifies the value of the loop test. Perhaps the specification of a
  function called in the loop body is missing a modification. (Use -infloops to
  inhibit warning)

Finished checking --- 3 code warnings
```

There are 3 warnings:

- Test expression for while not boolean, type int: i
- Variable i used before definition.
- Suspected infinite loop.

Lab 3.2 Using eclipse for java static analysis

Overview

The learning objective of this lab is for students to gain the first-hand experience on using static code analyzers in Eclipse to check Java program for security vulnerabilities and coding mistakes.

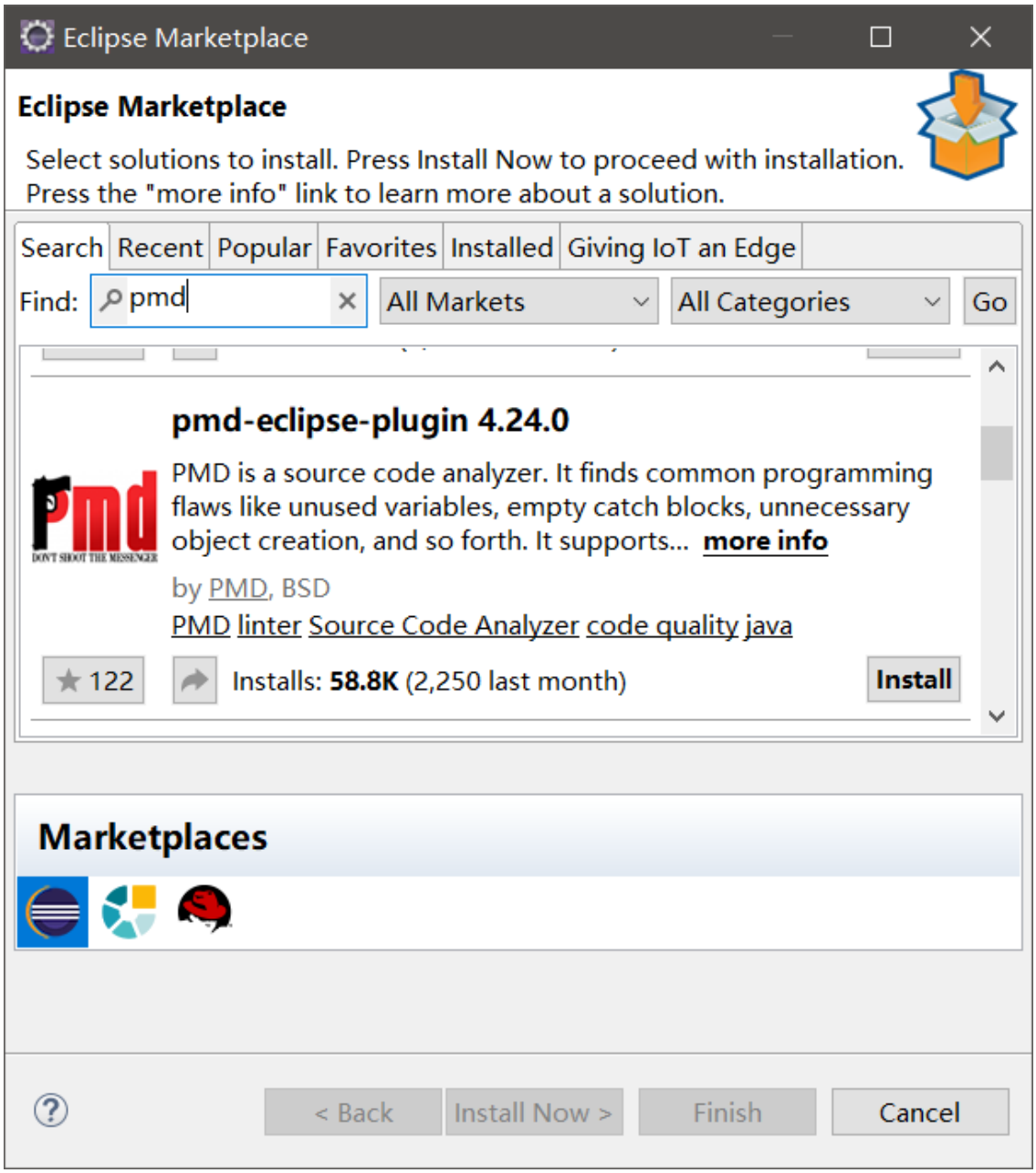
In this Lab, your goal is to achieve the followings:

- Install plugins in Java;

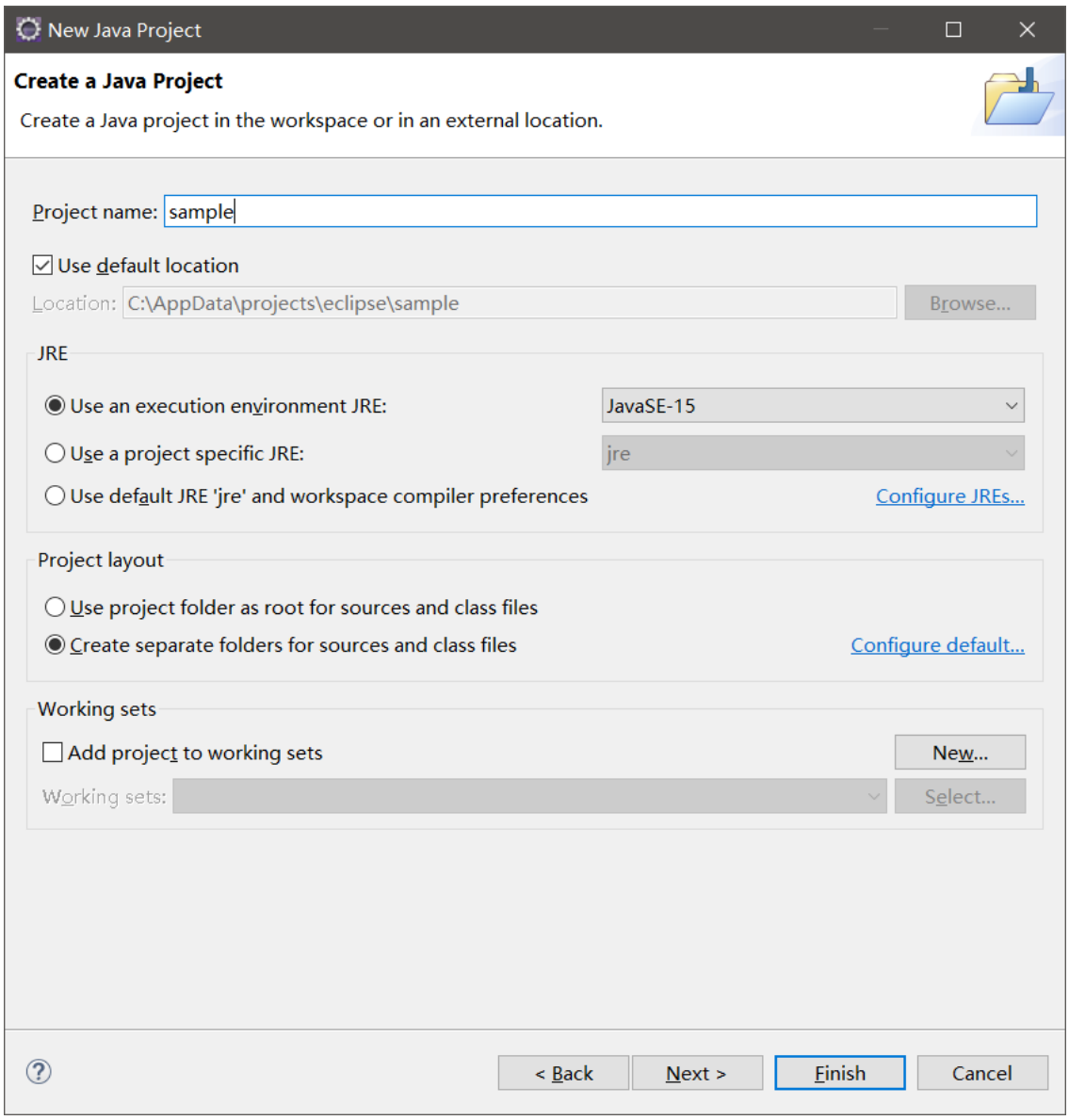
- Learn to check Java code by using static code analyzers in Eclipse. Descibe your observations in your report.

Steps

1. Install PMD plugins in Eclipse.



2. Create arbitrary sample Java project.

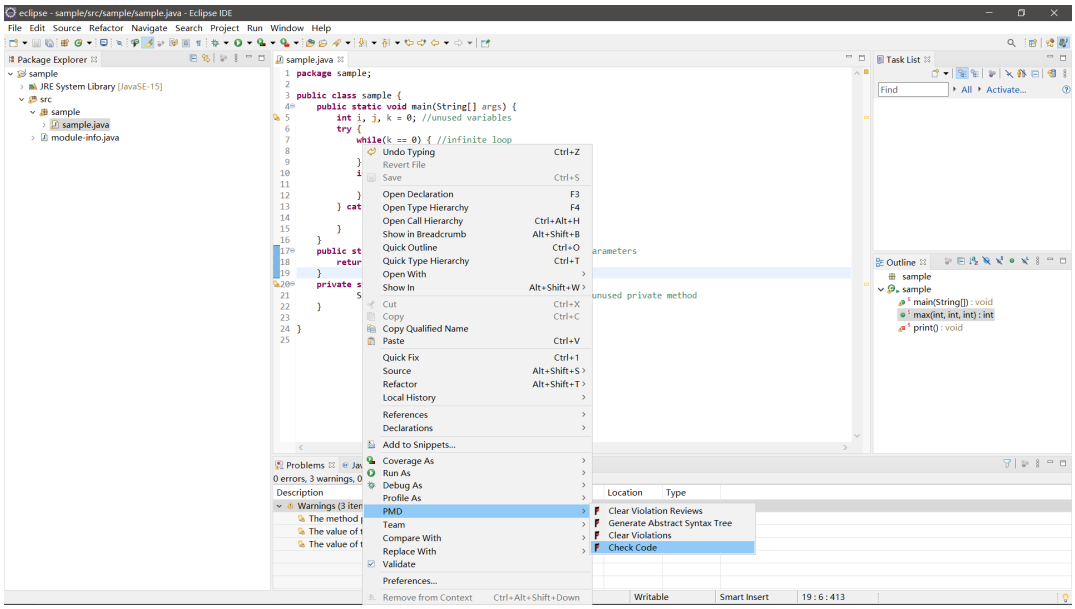


Create sample.java

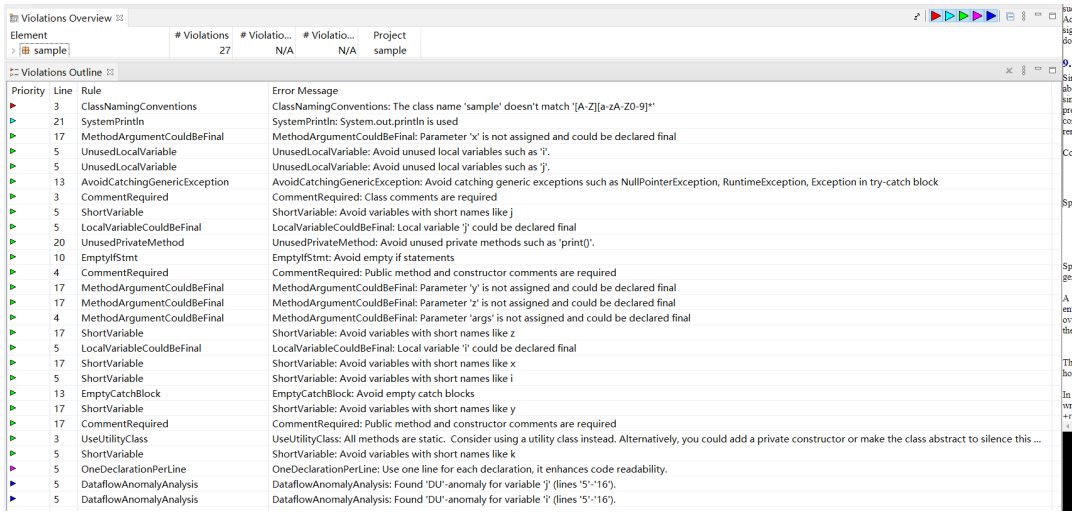
```java
package sample;

public class sample {
    public static void main(String[] args) {
        int i, j, k = 0; // unused variables
        try {
            while(k == 0) { // infinite loop
                k = k + 1 - 1;
            }
            if(max(k, k , k) > 0){ // empty if statements

            }
        } catch (Exception e) { // empty catch blocks

        }
    }
    public static int max(int x, int y, int z) { // unused parameters
        return x > y ? x : y;
    }
    private static void print() {
            System.out.println("unused private method"); // unused private method
    }
```

```
23
24  }
```

3. Begin to check Java code.
    - Right-click the entire project or some java file selected.
    - Select the check option and click.



The PWD plugin gives the violations overview and outline as follows:



Violations Overview

Element	# Violations	# Violatio...	# Violatio...	Project
sample	27	N/A	N/A	sample

Violations Outline

Priority	Line	Rule	Error Message
	3	ClassNamingConventions	ClassNamingConventions: The class name 'sample' doesn't match '[A-Z][a-zA-Z0-9]*'
	21	SystemPrintln	SystemPrintln: System.out.println is used
	17	MethodArgumentCouldBeFinal	MethodArgumentCouldBeFinal: Parameter 'x' is not assigned and could be declared final
	5	UnusedLocalVariable	UnusedLocalVariable: Avoid unused local variables such as 'i'.
	5	UnusedLocalVariable	UnusedLocalVariable: Avoid unused local variables such as 'j'.
	13	AvoidCatchingGenericException	AvoidCatchingGenericException: Avoid catching generic exceptions such as NullPointerException, RuntimeException, Exception in try-catch block
	3	CommentRequired	CommentRequired: Class comments are required
	5	ShortVariable	ShortVariable: Avoid variables with short names like j
	5	LocalVariableCouldBeFinal	LocalVariableCouldBeFinal: Local variable 'j' could be declared final
	20	UnusedPrivateMethod	UnusedPrivateMethod: Avoid unused private methods such as 'print()'.
	10	EmptyIfStmt	EmptyIfStmt: Avoid empty if statements
	4	CommentRequired	CommentRequired: Public method and constructor comments are required
	17	MethodArgumentCouldBeFinal	MethodArgumentCouldBeFinal: Parameter 'y' is not assigned and could be declared final
	17	MethodArgumentCouldBeFinal	MethodArgumentCouldBeFinal: Parameter 'z' is not assigned and could be declared final
	4	MethodArgumentCouldBeFinal	MethodArgumentCouldBeFinal: Parameter 'args' is not assigned and could be declared final
	17	ShortVariable	ShortVariable: Avoid variables with short names like z
	5	LocalVariableCouldBeFinal	LocalVariableCouldBeFinal: Local variable 'i' could be declared final
	17	ShortVariable	ShortVariable: Avoid variables with short names like x
	5	ShortVariable	ShortVariable: Avoid variables with short names like i
	13	EmptyCatchBlock	EmptyCatchBlock: Avoid empty catch blocks
	17	ShortVariable	ShortVariable: Avoid variables with short names like y
	17	CommentRequired	CommentRequired: Public method and constructor comments are required
	3	UseUtilityClass	UseUtilityClass: All methods are static. Consider using a utility class instead. Alternatively, you could add a private constructor or make the class abstract to silence this ...
	5	ShortVariable	ShortVariable: Avoid variables with short names like k
	5	OneDeclarationPerLine	OneDeclarationPerLine: Use one line for each declaration, it enhances code readability.
	5	DataflowAnomalyAnalysis	DataflowAnomalyAnalysis: Found 'DU'-anomaly for variable 'j' (lines '5'-'16').
	5	DataflowAnomalyAnalysis	DataflowAnomalyAnalysis: Found 'DU'-anomaly for variable 'i' (lines '5'-'16').

It points out the problems of unused variables, empty if statements, empty catch blocks, unused parameters, unused private method except infinite loop.