Secrets
ABAP
Apex
C
C++
CloudFormation
COBOL
C#
CSS
Flex
Go
HTML
Java
JavaScript
Kotlin
Objective C
PHP
PL/I
PL/SQL
Python
RPG
Ruby
Scala
Swift
Terraform
Text
TypeScript
T-SQL
VB.NET
VB6
XML

Java static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your JAVA code

All rules 632 | Vulnerability 53 | Bug 154 | Security Hotspot 36 | Code Smell 389 | Quick Fix 42

Tags

Search by name...

Abstract class names should comply with a naming convention
Code Smell

Strings literals should be placed on the left side when checking for equality
Code Smell

Files should contain an empty newline at the end
Code Smell

Source code should be indented consistently
Code Smell

A close curly brace should be located at the beginning of a line
Code Smell

Close curly brace and the next "else", "catch" and "finally" keywords should be on two different lines
Code Smell

Close curly brace and the next "else", "catch" and "finally" keywords should be located on the same line
Code Smell

An open curly brace should be located at the beginning of a line
Code Smell

An open curly brace should be located at the end of a line
Code Smell

Tabulation characters should not be used
Code Smell

Functions should not be defined with a variable number of arguments
Code Smell

Floating point numbers should not be tested for equality

Analyze your code

Bug Major

Floating point math is imprecise because of the challenges of storing such values in a binary representation. Even worse, floating point math is not associative; push a `float` or a `double` through a series of simple mathematical operations and the answer will be different based on the order of those operation because of the rounding that takes place at each step.

Even simple floating point assignments are not simple:

```
float f = 0.1; // 0.10000000149011611938476562 5
double d = 0.1; // 0.10000000000000000555111512312578270211 8
```

(Results will vary based on compiler and compiler settings);

Therefore, the use of the equality (`==`) and inequality (`!=`) operators on `float` or `double` values is almost always an error. Instead the best course is to avoid floating point comparisons altogether. When that is not possible, you should consider using one of Java's float-handling `Numbers` such as `BigDecimal` which can properly handle floating point comparisons. A third option is to look not for equality but for whether the value is close enough. I.e. compare the absolute value of the difference between the stored value and the expected value against a margin of acceptable error. Note that this does not cover all cases (`NaN` and `Infinity` for instance).

This rule checks for the use of direct and indirect equality/inequailty tests on floats and doubles.

Noncompliant Code Example

```
float myNumber = 3.146;
if ( myNumber == 3.146f ) { //Noncompliant. Because of float
  // ...
}
if ( myNumber != 3.146f ) { //Noncompliant. Because of float
  // ...
}

if (myNumber < 4 || myNumber > 4) { // Noncompliant; indirec
  // ...
}

float zeroFloat = 0.0f;
if (zeroFloat == 0) {  // Noncompliant. Computations may end
}
```

Exceptions

Since `NaN` is not equal to itself, the specific case of testing a floating point value against itself is a valid test for `NaN` and is therefore ignored. Though using `Double.isNaN` method should be preferred instead, as intent is more explicit.

Local-Variable Type Inference should be used

Code Smell

Migrate your tests from JUnit4 to the new JUnit5 annotations

Code Smell

Track uses of disallowed classes

Code Smell

Track uses of "@SuppressWarnings" annotations

Code Smell

```
float f;
double d;
if(f != f) { // Compliant; test for NaN value
  System.out.println("f is NaN");
} else if (f != d) { // Noncompliant
  // ...
}
```

Available In:

sonarlint | sonarcloud | sonarqube