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

Regex patterns should not be created needlessly

Analyze your code

Code Smell Major regex performance

The `java.util.regex.Pattern.compile()` methods have a significant performance cost, and therefore should be used sensibly.

Moreover they are the only mechanism available to create instances of the Pattern class, which are necessary to do any pattern matching using regular expressions. Unfortunately that can be hidden behind convenience methods like `String.matches()` or `String.split()`.

It is therefore somewhat easy to inadvertently repeatedly compile the same regular expression at great performance cost with no valid reason.

This rule raises an issue when:

- A `Pattern` is compiled from a `String` literal or constant and is not stored in a static final reference.
- `String.matches`, `String.split`, `String.replaceAll` or `String.replaceFirst` are invoked with a `String` literal or constant. In which case the code should be refactored to use a `java.util.regex.Pattern` while respecting the previous rule.

Noncompliant Code Example

```
public void doingSomething(String stringToMatch) {
  Pattern regex = Pattern.compile("myRegex");  // Noncomplia
  Matcher matcher = regex.matcher("s");
  // ...
  if (stringToMatch.matches("myRegex2")) {  // Noncompliant
    // ...
  }
}
```

Compliant Solution

```
private static final Pattern myRegex = Pattern.compile("myRe
private static final Pattern myRegex2 = Pattern.compile("myR

public void doingSomething(String stringToMatch) {
  Matcher matcher = myRegex.matcher("s");
  // ...
  if (myRegex2.matcher(stringToMatch).matches()) {
    // ...
  }
}
```

Exceptions

`String.split` doesn't create a regex when the string passed as argument meets either of these 2 conditions:

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

- It is a one-char String and this character is not one of the RegEx's meta characters ".$|()[{^?*+\"
- It is a two-char String and the first char is the backslash and the second is not the ascii digit or ascii letter.

In which case no issue will be raised.

Available In:

sonarlint sonarcloud sonarqube