

 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

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

Tags ▾

Search by name... 🔍

Lazy initialization of "static" fields should be "synchronized"

Analyze your code

- Code Smell
- Critical
- multi-threading

In a multi-threaded situation, un-synchronized lazy initialization of static fields could mean that a second thread has access to a half-initialized object while the first thread is still creating it. Allowing such access could cause serious bugs. Instead, the initialization block should be synchronized.

Similarly, updates of such fields should also be synchronized.

This rule raises an issue whenever a lazy static initialization is done on a class with at least one synchronized method, indicating intended usage in multi-threaded applications.

Noncompliant Code Example

```
private static Properties fPreferences = null;

private static Properties getPreferences() {
    if (fPreferences == null) {
        fPreferences = new Properties(); // Noncompliant
        fPreferences.put("loading", "true");
        fPreferences.put("filterstack", "true");
        readPreferences();
    }
    return fPreferences;
}
```

Compliant Solution

```
private static Properties fPreferences = null;

private static synchronized Properties getPreferences() {
    if (fPreferences == null) {
        fPreferences = new Properties();
        fPreferences.put("loading", "true");
        fPreferences.put("filterstack", "true");
        readPreferences();
    }
    return fPreferences;
}
```

Available In:

sonarlint | sonarcloud | sonarqube

<div>Local-Variable Type Inference should be used</div> <div> Code Smell</div>
<div>Migrate your tests from JUnit4 to the new JUnit5 annotations</div> <div> Code Smell</div>
<div>Track uses of disallowed classes</div> <div> Code Smell</div>
<div>Track uses of "@SuppressWarnings" annotations</div> <div> Code Smell</div>

SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.
[Privacy Policy](#)