


-  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 rules632

Vulnerability53

Bug154

Security Hotspot36

Code Smell389

Quick Fix42

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

Standard functional interfaces should not be redefined

Analyze your code

Code Smell

Major ?

java8

Just as there is little justification for writing your own String class, there is no good reason to re-define one of the existing, standard functional interfaces.

Doing so may seem tempting, since it would allow you to specify a little extra context with the name. But in the long run, it will be a source of confusion, because maintenance programmers will wonder what is different between the custom functional interface and the standard one.

Noncompliant Code Example

```
@FunctionalInterface
public interface MyInterface { // Noncompliant
    double toDouble(int a);
}

@FunctionalInterface
public interface ExtendedBooleanSupplier { // Noncompliant
    boolean get();
    default boolean isFalse() {
        return !get();
    }
}

public class MyClass {
    private int a;
    public double myMethod(MyInterface instance){
        return instance.toDouble(a);
    }
}
```

Compliant Solution





```
@FunctionalInterface
public interface ExtendedBooleanSupplier extends BooleanSupplier {
    default boolean isFalse() {
        return !getAsBoolean();
    }
}

public class MyClass {
    private int a;
    public double myMethod(IntToDoubleFunction instance){
        return instance.applyAsDouble(a);
    }
}
```

Available In:

https://rules.sonarsource.com/java/RSPEC-1711

1/2

<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>