sonar RULES

Products

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

Functional Interfaces should be as specialised as possible

Analyze your code

Code Smell Minor performance

The `java.util.function` package provides a large array of functional interface definitions for use in lambda expressions and method references. In general it is recommended to use the more specialised form to avoid auto-boxing. For instance `IntFunction<Foo>` should be preferred over `Function<Integer, Foo>`.

This rule raises an issue when any of the following substitution is possible:

Current Interface	Preferred Interface
`Function<Integer, R>`	`IntFunction<R>`
`Function<Long, R>`	`LongFunction<R>`
`Function<Double, R>`	`DoubleFunction<R>`
`Function<Double,Integer>`	`DoubleToIntFunction`
`Function<Double,Long>`	`DoubleToLongFunction`
`Function<Long,Double>`	`LongToDoubleFunction`
`Function<Long,Integer>`	`LongToIntFunction`
`Function<R,Integer>`	`ToIntFunction<R>`
`Function<R,Long>`	`ToLongFunction<R>`
`Function<R,Double>`	`ToDoubleFunction<R>`
`Function<T,T>`	`UnaryOperator<T>`
`BiFunction<T,T,T>`	`BinaryOperator<T>`
`Consumer<Integer>`	`IntConsumer`
`Consumer<Double>`	`DoubleConsumer`
`Consumer<Long>`	`LongConsumer`
`BiConsumer<T,Integer>`	`ObjIntConsumer<T>`
`BiConsumer<T,Long>`	`ObjLongConsumer<T>`
`BiConsumer<T,Double>`	`ObjDoubleConsumer<T>`
`Predicate<Integer>`	`IntPredicate`
`Predicate<Double>`	`DoublePredicate`
`Predicate<Long>`	`LongPredicate`
`Supplier<Integer>`	`IntSupplier`
`Supplier<Double>`	`DoubleSupplier`
`Supplier<Long>`	`LongSupplier`
`Supplier<Boolean>`	`BooleanSupplier`
`UnaryOperator<Integer>`	`IntUnaryOperator`
`UnaryOperator<Double>`	`DoubleUnaryOperator`
`UnaryOperator<Long>`	`LongUnaryOperator`
`BinaryOperator<Integer>`	`IntBinaryOperator`
`BinaryOperator<Long>`	`LongBinaryOperator`
`BinaryOperator<Double>`	`DoubleBinaryOperator`
`Function<T, Boolean>`	`Predicate<T>`
`BiFunction<T,U,Boolean>`	`BiPredicate<T,U>`

Noncompliant Code Example

```
public class Foo implements Supplier<Integer> {  // Noncompl
    @Override
    public Integer get() {
```

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

```
      // ...
    }
}
```

Compliant Solution

```
public class Foo implements IntSupplier {

  @Override
  public int getAsInt() {
    // ...
  }
}
```

Available In:

sonarlint sonarcloud sonarqube