




 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... 🔍


Value-based classes should not be used for locking

 Bug


Expressions used in "assert" should not produce side effects

 Bug


"volatile" variables should not be used with compound operators

 Bug


"getClass" should not be used for synchronization

 Bug


Min and max used in combination should not always return the same value

 Bug


Assignment of lazy-initialized members should be the last step with double-checked locking

 Bug


"String" calls should not go beyond their bounds

 Bug


Raw byte values should not be used in bitwise operations in combination with shifts

 Bug


Getters and setters should be synchronized in pairs

 Bug

Non-thread-safe fields should not be static


 Bug


"null" should not be used with "Optional"


 Bug

### Mocking all non-private methods of a class should be avoided

Analyze your code

 Code Smell

 Critical



tests mockito

If you end up mocking every non-private method of a class in order to write tests, it is a strong sign that your test became too complex, or that you misunderstood the way you are supposed to use the mocking mechanism.

You should either refactor the test code into multiple units, or consider using the class itself, by either directly instantiating it, or creating a new one inheriting from it, with the expected behavior.

This rule reports an issue when every member of a given class are mocked.

#### Noncompliant Code Example

```
@Test
void test_requiring_MyClass() {
    MyClass myClassMock = mock(MyClass.class); // Noncompliant
    when(myClassMock.f()).thenReturn(1);
    when(myClassMock.g()).thenReturn(2);
    //...
}

abstract class MyClass {
    abstract int f();
    abstract int g();
}
```

#### Compliant Solution




```
@Test
void test_requiring_MyClass() {
    MyClass myClass = new MyClassForTest();
    //...
}

class MyClassForTest extends MyClass {

    @Override
    int f() {
        return 1;
    }

    @Override
    int g() {
        return 2;
    }
}
```

or

<div>Unary prefix operators should not be repeated</div> <div> Bug</div>
<div>"=+" should not be used instead of "+="</div> <div> Bug</div>
<div>"read" and "readLine" return values should be used</div> <div> Bug</div>
<div>Inappropriate regular expressions should not be used</div> <div> Bug</div>

```
@Test
void test_requiring_f() {
    MyClass myClassMock = mock(MyClass.class);
    when(myClassMock.f()).thenReturn(1);
    //...
}

@Test
void test_requiring_g() {
    MyClass myClassMock = mock(MyClass.class);
    when(myClassMock.g()).thenReturn(2);
    //...
}

abstract class MyClass {
    abstract int f();
    abstract int g();
}
```

Available In:

 |  | 