# Règles SonarLint à appliquer en ILU1

# Syntaxe

## A close curly brace should be located at the beginning of a line (java:S1109)

Code smell Minor

Shared coding conventions make it possible for a team to efficiently collaborate. This rule makes it mandatory to place a close curly brace at the beginning of a line.

### Noncompliant Code Example

```
if(condition) {
  doSomething();}
```

### Compliant Solution

```
if(condition) {
  doSomething();
}
```

### Exceptions

When blocks are inlined (open and close curly braces on the same line), no issue is triggered.

```
if(condition) {doSomething();}
```

---

```
if (b == 0) {  // Noncompliant
  doOneMoreThing();
} else {
  doOneMoreThing();
}

int b = a > 12 ? 4 : 4;  // Noncompliant

switch (i) {  // Noncompliant
  case 1:
    doSomething();
    break;
  case 2:
    doSomething();
    break;
  case 3:
```

```
    doSomething();
    break;
  default:
    doSomething();
}
```

## Exceptions

This rule does not apply to if chains without else-s, or to switch-es without default clauses.

```
if(b == 0) {    //no issue, this could have been done on purpose to make the code more readable
  doSomething();
} else if(b == 1) {
  doSomething();
}
```

---

# Class names should comply with a naming convention (java:S101)

Code smell Minor

Shared coding conventions allow teams to collaborate effectively. This rule allows to check that all class names match a provided regular expression.

### Noncompliant Code Example

With default provided regular expression ^[A-Z][a-zA-Z0-9]*$:

```
class my_class {...}
```

### Compliant Solution

```
class MyClass {...}
```

### Parameters

Following parameter values can be set in Rules Configuration.

| | |
|---|---|
| **format** | Regular expression used to check the class names against.<br><br>Current value: ^[A-Z][a-zA-Z0-9]*$<br><br>Default value: ^[A-Z][a-zA-Z0-9]*$ |

---

# Constant names should comply with a naming convention (java:S115)

Code smell Critical

Shared coding conventions allow teams to collaborate efficiently. This rule checks that all constant names match a provided regular expression.

### Noncompliant Code Example

With the default regular expression ^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$:

```
public class MyClass {
  public static final int first = 1;
}

public enum MyEnum {
  first;
}
```

### Compliant Solution

```
public class MyClass {
  public static final int FIRST = 1;
}

public enum MyEnum {
  FIRST;
}
```

### Parameters

Following parameter values can be set in Rules Configuration.

| | |
|---|---|
| **format** | Regular expression used to check the constant names against.<br><br>Current value: ^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$<br><br>Default value: ^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$ |

---

# Field names should comply with a naming convention (java:S116)

Code smell Minor

Sharing some naming conventions is a key point to make it possible for a team to efficiently collaborate. This rule allows to check that field names match a provided regular expression.

### Noncompliant Code Example

With the default regular expression ^[a-z][a-zA-Z0-9]*$:

```
class MyClass {
  private int my_field;
}
```

### Compliant Solution

```
class MyClass {
  private int myField;
}
```

### Parameters

Following parameter values can be set in Rules Configuration.

---

# Package names should comply with a naming convention (java:S120)

Code smell Minor

Shared coding conventions allow teams to collaborate efficiently. This rule checks that all package names match a provided regular expression.

### Noncompliant Code Example

With the default regular expression ^[a-z_]+(\.[a-z_][a-z0-9_]*)*$:

```
package org.exAmple; // Noncompliant
```

### Compliant Solution

```
package org.example;
```

### Parameters

Following parameter values can be set in Rules Configuration.

| | |
|---|---|
| **format** | Regular expression used to check the package names against.<br><br>Current value: ^[a-z_]+(\.[a-z_][a-z0-9_]*)*$<br><br>Default value: ^[a-z_]+(\.[a-z_][a-z0-9_]*)*$ |

---

# Multiple variables should not be declared on the same line (java:S1659)

Code smell Minor

Declaring multiple variables on one line is difficult to read.

### Noncompliant Code Example

```
class MyClass {

  private int a, b;

  public void method(){
    int c; int d;
  }
}
```

### Compliant Solution

```
class MyClass {

  private int a;
```

```
  private int b;

  public void method(){
    int c;
    int d;
  }
}
```

## See

- CERT, DCL52-J. - Do not declare more than one variable per declaration
- CERT, DCL04-C. - Do not declare more than one variable per declaration

---

# Méthodes

## Local variables should not be declared and then immediately returned or thrown (java:S1488)

Code smell Minor

Declaring a variable only to immediately return or throw it is a bad practice.

Some developers argue that the practice improves code readability, because it enables them to explicitly name what is being returned. However, this variable is an internal implementation detail that is not exposed to the callers of the method. The method name should be sufficient for callers to know exactly what will be returned.

### Noncompliant Code Example

```
public long computeDurationInMilliseconds() {
  long duration = (((hours * 60) + minutes) * 60 + seconds ) * 1000 ;
  return duration;
}

public void doSomething() {
  RuntimeException myException = new RuntimeException();
  throw myException;
}
```

### Compliant Solution

```
public long computeDurationInMilliseconds() {
  return (((hours * 60) + minutes) * 60 + seconds ) * 1000 ;
}

public void doSomething() {
  throw new RuntimeException();
}
```

---

## Local variables should not shadow class fields (java:S1117)

Code smell Major

Overriding or shadowing a variable declared in an outer scope can strongly impact the readability, and therefore the maintainability, of a piece of code. Further, it could lead maintainers to introduce bugs because they think they're using one variable but are really using another.

### Noncompliant Code Example

```
class Foo {
  public int myField;

  public void doSomething() {
    int myField = 0;
    ...
  }
}
```

### See

- CERT, DCL01-C. - Do not reuse variable names in subscopes
- CERT, DCL51-J. - Do not shadow or obscure identifiers in subscopes

---

## Method names should comply with a naming convention (java:S100)

Code smell Minor

Shared naming conventions allow teams to collaborate efficiently. This rule checks that all method names match a provided regular expression.

### Noncompliant Code Example

With default provided regular expression ^[a-z][a-zA-Z0-9]*$:

```
public int DoSomething(){...}
```

### Compliant Solution

```
public int doSomething(){...}
```

### Exceptions

Overriding methods are excluded.

```
@Override
public int Do_Something(){...}
```

### Parameters

Following parameter values can be set in Rules Configuration.

| | |
|---|---|
| **format** | Regular expression used to check the method names against.<br><br>Current value: ^[a-z][a-zA-Z0-9]*$<br><br>Default value: ^[a-z][a-zA-Z0-9]*$ |

---

# Methods should not have too many parameters (java:S107)

 Code smell  Major

A long parameter list can indicate that a new structure should be created to wrap the numerous parameters or that the function is doing too many things.

### Noncompliant Code Example

With a maximum number of 4 parameters:

```
public void doSomething(int param1, int param2, int param3, String param4, long param5) {
...
}
```

### Compliant Solution

```
public void doSomething(int param1, int param2, int param3, String param4) {
...
}
```

### Exceptions

Methods annotated with :

- Spring's @RequestMapping (and related shortcut annotations, like @GetRequest)
- JAX-RS API annotations (like @javax.ws.rs.GET)

- Bean constructor injection with @org.springframework.beans.factory.annotation.Autowired
- CDI constructor injection with @javax.inject.Inject
- @com.fasterxml.jackson.annotation.JsonCreator

may have a lot of parameters, encapsulation being possible. Such methods are therefore ignored.

### Parameters

Following parameter values can be set in Rules Configuration.

| | |
|---|---|
| **max** | Maximum authorized number of parameters<br><br>Current value: 7<br><br>Default value: 7 |
| **constructorMax** | Maximum authorized number of parameters for a constructor<br><br>Current value: 7<br><br>Default value: 7 |

---

# Methods should not return constants (java:S3400)

Code smell Minor

There's no point in forcing the overhead of a method call for a method that always returns the same constant value. Even worse, the fact that a method call must be made will likely mislead developers who call the method thinking that something more is done. Declare a constant instead.

This rule raises an issue if on methods that contain only one statement: the return of a constant value.

### Noncompliant Code Example

```
int getBestNumber() {
  return 12;  // Noncompliant
}
```

### Compliant Solution

```
static final int BEST_NUMBER = 12;
```

### Exceptions

Methods with annotations, such as @Override and Spring's @RequestMapping, are ignored.

---

# Variables should not be self-assigned (java:S1656)

Bug Major

There is no reason to re-assign a variable to itself. Either this statement is redundant and should be removed, or the re-assignment is a mistake and some other value or variable was intended for the assignment instead.

### Noncompliant Code Example

```
public void setName(String name) {
  name = name;
}
```

### Compliant Solution

```
public void setName(String name) {
  this.name = name;
}
```

### See

- CERT, MSC12-C. - Detect and remove code that has no effect or is never executed

---

# Private fields only used as local variables in methods should become local variables (java:S1450)

Code smell Minor

When the value of a private field is always assigned to in a class' methods before being read, then it is not being used to store class information. Therefore, it should become a local variable in the relevant methods to prevent any misunderstanding.

### Noncompliant Code Example

```
public class Foo {
  private int a;
  private int b;

  public void doSomething(int y) {
    a = y + 5;
    ...
    if(a == 0) {
      ...
    }
    ...
  }

  public void doSomethingElse(int y) {
    b = y + 3;
    ...
  }
}
```

### Compliant Solution

```
public class Foo {

  public void doSomething(int y) {
    int a = y + 5;
    ...
    if(a == 0) {
      ...
    }
  }

  public void doSomethingElse(int y) {
    int b = y + 3;
    ...
  }
}
```

### Exceptions

This rule doesn’t raise any issue on annotated field.

---

# Structure

## "else" statements should be clearly matched with an "if" (java:S5261)

Code smell Major

The dangling else problem appears when nested if/else statements are written without curly braces. In this case, else is associated with the nearest if but that is not always obvious and sometimes the indentation can also be misleading.

This rules reports else statements that are difficult to understand, because they are inside nested if statements without curly braces.

Adding curly braces can generally make the code clearer (see rule {rule:java:S121} ), and in this situation of dangling else, it really clarifies the intention of the code.

### Noncompliant Code Example

```
if (a)
  if (b)
    d++;
 else     // Noncompliant, is the "else" associated with "if(a)" or "if (b)"? (the answer is "if(b)")
  e++;
```

### Compliant Solution

```
if (a) {
  if (b) {
    d++;
  }
 } else { // Compliant, there is no doubt the "else" is associated with "if(a)"
  e++;
 }
```

### See

- https://en.wikipedia.org/wiki/Dangling_else

---

# "if ... else if" constructs should end with "else" clauses (java:S126)

Code smell Critical

This rule applies whenever an if statement is followed by one or more else if statements; the final else if should be followed by an else statement.

The requirement for a final else statement is defensive programming.

The else statement should either take appropriate action or contain a suitable comment as to why no action is taken. This is consistent with the requirement to have a final default clause in a switch statement.

### Noncompliant Code Example

```
if (x == 0) {
  doSomething();
} else if (x == 1) {
  doSomethingElse();
}
```

### Compliant Solution

```
if (x == 0) {
  doSomething();
} else if (x == 1) {
  doSomethingElse();
} else {
  throw new IllegalStateException();
}
```

### See

- CERT, MSC01-C. - Strive for logical completeness
- CERT, MSC57-J. - Strive for logical completeness

---

# Collapsible "if" statements should be merged (java:S1066)

Code smell Major

Merging collapsible if statements increases the code's readability.

### Noncompliant Code Example

```
if (file != null) {
  if (file.isFile() || file.isDirectory()) {
    /* ... */
  }
}
```

### Compliant Solution

```
if (file != null && isFileOrDirectory(file)) {
  /* ... */
}

private static boolean isFileOrDirectory(File file) {
  return file.isFile() || file.isDirectory();
}
```

---

# Conditionals should start on new lines (java:S3972)

Code smell Critical

Code is clearest when each statement has its own line. Nonetheless, it is a common pattern to combine on the same line an if and its resulting *then* statement. However, when an if is placed on the same line as the closing } from a preceding *then*, *else* or *else if* part, it is either an error - else is missing - or the invitation to a future error as maintainers fail to understand that the two statements are unconnected.

### Noncompliant Code Example

```
if (condition1) {
  // ...
} if (condition2) {  // Noncompliant
  //...
}
```

### Compliant Solution

```
if (condition1) {
  // ...
} else if (condition2) {
  //...
}
```

Or

```
if (condition1) {
  // ...
}

if (condition2) {
  //...
}
```

---

# "switch" statements should have "default" clauses (java:S131)

Code smell Critical

The requirement for a final default clause is defensive programming. The clause should either take appropriate action, or contain a suitable comment as to why no action is taken.

### Noncompliant Code Example

```
switch (param) {  //missing default clause
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
}

switch (param) {
  default: // default clause should be the last one
    error();
    break;
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
}
```

### Compliant Solution

```
switch (param) {
  case 0:
    doSomething();
```

```
    break;
  case 1:
    doSomethingElse();
    break;
  default:
    error();
    break;
}
```

### Exceptions

If the switch parameter is an Enum and if all the constants of this enum are used in the case statements, then no default clause is expected.

Example:

```
public enum Day {
    SUNDAY, MONDAY
}
...
switch(day) {
  case SUNDAY:
    doSomething();
    break;
  case MONDAY:
    doSomethingElse();
    break;
}
```

### See

- MITRE, CWE-478 - Missing Default Case in Switch Statement
- CERT, MSC01-C. - Strive for logical completeness

---

# "switch" statements should have at least 3 "case" clauses (java:S1301)

Code smell Minor

switch statements are useful when there are many different cases depending on the value of the same expression.

For just one or two cases however, the code will be more readable with if statements.

### Noncompliant Code Example

```
switch (variable) {
  case 0:
    doSomething();
    break;
  default:
    doSomethingElse();
    break;
}
```

### Compliant Solution

```
if (variable == 0) {
  doSomething();
} else {
  doSomethingElse();
}
```

---

# "switch" statements should not have too many "case" clauses (java:S1479)

 Code smell  Major

When switch statements have large sets of case clauses, it is usually an attempt to map two sets of data. A real map structure would be more readable and maintainable, and should be used instead.

### Exceptions

This rule ignores switches over Enums and empty, fall-through cases.

### Parameters

Following parameter values can be set in Rules Configuration.

| | |
|---|---|
| **maximum** | Maximum number of case<br><br>Current value: 30<br><br>Default value: 30 |

---

A COCHER :

# "switch case" clauses should not have too many lines of code (java:S1151)

Code smell Major

The switch statement should be used only to clearly define some new branches in the control flow. As soon as a case clause contains too many statements this highly decreases the readability of the overall control flow statement. In such case, the content of the case clause should be extracted into a dedicated method.

### Noncompliant Code Example

With the default threshold of 5:

```
switch (myVariable) {
  case 0: // Noncompliant: 6 lines till next case
    methodCall1("");
    methodCall2("");
    methodCall3("");
    methodCall4("");
    break;
  case 1:
  ...
}
```

### Compliant Solution

```
switch (myVariable) {
  case 0:
    doSomething()
    break;
  case 1:
  ...
}
...
private void doSomething(){
    methodCall1("");
    methodCall2("");
    methodCall3("");
    methodCall4("");
}
```

### Parameters

Following parameter values can be set in Rules Configuration.

max
Maximum number of lines
Current value: 5

Default value: 5

---

# Switch cases should end with an unconditional "break" statement (java:S128)

Code smell Blocker

When the execution is not explicitly terminated at the end of a switch case, it continues to execute the statements of the following case. While this is sometimes intentional, it often is a mistake which leads to unexpected behavior.

### Noncompliant Code Example

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:  // Both 'doSomething()' and 'doSomethingElse()' will be executed. Is it on purpose
?
    doSomething();
  default:
    doSomethingElse();
    break;
}
```

### Compliant Solution

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:
    doSomething();
    break;
  default:
```

```
    doSomethingElse();
    break;
}
```

### Exceptions

This rule is relaxed in the following cases:

```
switch (myVariable) {
  case 0:                       // Empty case used to specify the same behavior for a group of
cases.
  case 1:
    doSomething();
    break;
  case 2:                       // Use of a fallthrough comment
    // fallthrough
  case 3:                       // Use of return statement
    return;
  case 4:                       // Use of throw statement
    throw new IllegalStateException();
  case 5:                       // Use of continue statement
    continue;
  default:                      // For the last case, use of break statement is optional
    doSomethingElse();
}
```

### See

- MITRE, CWE-484 - Omitted Break Statement in Switch
- CERT, MSC17-C. - Finish every set of statements associated with a case label with a break statement
- CERT, MSC52-J. - Finish every set of statements associated with a case label with a break statement

---

# Loop conditions should be true at least once (java:S2252)

Bug Major

If a for loop's condition is false before the first loop iteration, the loop will never be executed. Such loops are almost always bugs, particularly when the initial value and stop conditions are hard-coded.

### Noncompliant Code Example

```
for (int i = 10; i < 10; i++) {  // Noncompliant
  // ...
```

---

## "for" loop increment clauses should modify the loops' counters (java:S1994)

Code smell Critical

It can be extremely confusing when a for loop's counter is incremented outside of its increment clause. In such cases, the increment should be moved to the loop's increment clause if at all possible.

### Noncompliant Code Example

```
for (i = 0; i < 10; j++) { // Noncompliant
  // ...
  i++;
}
```

### Compliant Solution

```
for (i = 0; i < 10; i++, j++) {
  // ...
}
```

Or

```
for (i = 0; i < 10; i++) {
  // ...
  j++;
}
```

---

## "for" loop stop conditions should be invariant (java:S127)

Code smell Major

A for loop stop condition should test the loop counter against an invariant value (i.e. one that is true at both the beginning and ending of every loop iteration). Ideally, this means that the stop condition is set to a local variable just before the loop begins.

Stop conditions that are not invariant are slightly less efficient, as well as being difficult to understand and maintain, and likely lead to the introduction of errors in the future.

This rule tracks three types of non-invariant stop conditions:

- When the loop counters are updated in the body of the for loop
- When the stop condition depend upon a method call
- When the stop condition depends on an object property, since such properties could change during the execution of the loop.

### Noncompliant Code Example

```
for (int i = 0; i < 10; i++) {
  ...
  i = i - 1; // Noncompliant; counter updated in the body of the loop
  ...
}
```

### Compliant Solution

```
for (int i = 0; i < 10; i++) {...}
```

---

# A "while" loop should be used instead of a "for" loop (java:S1264)

Code smell Minor

When only the condition expression is defined in a for loop, and the initialization and increment expressions are missing, a while loop should be used instead to increase readability.

### Noncompliant Code Example

```
for (;condition;) { /*...*/ }
```

### Compliant Solution

```
while (condition) { /*...*/ }
```

---

# Jump statements should not be redundant (java:S3626)

Code smell Minor

Jump statements such as return and continue let you change the default flow of program execution, but jump statements that direct the control flow to the original direction are just a waste of keystrokes.

### Noncompliant Code Example

```
public void foo() {
  while (condition1) {
    if (condition2) {
      continue; // Noncompliant
    } else {
      doTheThing();
    }
  }
  return; // Noncompliant; this is a void method
}
```

### Compliant Solution

```
public void foo() {
  while (condition1) {
    if (!condition2) {
      doTheThing();
    }
  }
}
```

---

# Tableau

## Array designators "[]" should be located after the type in method signatures (java:S1195)

Code smell Minor

According to the Java Language Specification:

> For compatibility with older versions of the Java SE platform,
>
> the declaration of a method that returns an array is allowed to place (some or all of) the empty bracket pairs that form the declaration of the array type after the formal parameter list.
>
> This obsolescent syntax should not be used in new code.

### Noncompliant Code Example

```
public int getVector()[] { /* ... */ }    // Noncompliant

public int[] getMatrix()[] { /* ... */ }  // Noncompliant
```

### Compliant Solution

```
public int[] getVector() { /* ... */ }

public int[][] getMatrix() { /* ... */ }
```

---

## Array designators "[]" should be on the type, not the variable (java:S1197)

Code smell Minor

Array designators should always be located on the type for better code readability. Otherwise, developers must look both at the type and the variable name to know whether or not a variable is an array.

### Noncompliant Code Example

```
int matrix[][];   // Noncompliant
int[] matrix[];   // Noncompliant
```

### Compliant Solution

```
int[][] matrix;   // Compliant
```

---

# Assert

## Asserts should not be used to check the parameters of a public method (java:S4274)

Code smell Major

An assert is inappropriate for parameter validation because assertions can be disabled at runtime in the JVM, meaning that a bad operational setting would completely eliminate the intended checks. Further, asserts that fail throw AssertionErrors, rather than throwing some type of Exception. Throwing Errors is completely outside of the normal realm of expected catch/throw behavior in normal programs.

This rule raises an issue when a public method uses one or more of its parameters with asserts.

### Noncompliant Code Example

```
public void setPrice(int price) {
  assert price >= 0 && price <= MAX_PRICE;
  // Set the price
 }
```

### Compliant Solution

```
public void setPrice(int price) {
  if (price < 0 || price > MAX_PRICE) {
    throw new IllegalArgumentException("Invalid price: " + price);
  }
  // Set the price
 }
```

### See

Programming With Assertions

---

## Expressions used in "assert" should not produce side effects (java:S3346)

Bug Major

Since assert statements aren't executed by default (they must be enabled with JVM flags) developers should never rely on their execution the evaluation of any logic required for correct program function.

### Noncompliant Code Example

```
assert myList.remove(myList.get(0));  // Noncompliant
```

### Compliant Solution

```
boolean removed = myList.remove(myList.get(0));
assert removed;
```

### See

- CERT, EXP06-J. - Expressions used in assertions must not produce side effects

---

# Méthodes JavaDoc

## "Random" objects should be reused (java:S2119)

Bug Critical

Creating a new Random object each time a random value is needed is inefficient and may produce numbers which are not random depending on the JDK. For better efficiency and randomness, create a single Random, then store, and reuse it.

The Random() constructor tries to set the seed with a distinct value every time. However there is no guarantee that the seed will be random or even uniformly distributed. Some JDK will use the current time as seed, which makes the generated numbers not random at all.

This rule finds cases where a new Random is created each time a method is invoked.

### Noncompliant Code Example

```
public void doSomethingCommon() {
  Random rand = new Random();  // Noncompliant; new instance created with each invocation
  int rValue = rand.nextInt();
  //...
```

### Compliant Solution

```
private Random rand = SecureRandom.getInstanceStrong();  // SecureRandom is preferred to Random

public void doSomethingCommon() {
  int rValue = this.rand.nextInt();
  //...
```

### Exceptions

A class which uses a Random in its constructor or in a static main function and nowhere else will be ignored by this rule.

### See

- OWASP Top 10 2017 Category A6 - Security Misconfiguration

---

"default" clauses should be last (java:S4524)

CODE_SMELLCode smellCRITICALCritical

switch can contain a default clause for various reasons: to handle unexpected values, to show that all the cases were properly considered.

For readability purpose, to help a developer to quickly find the default behavior of a switch statement, it is recommended to put the default clause at the end of the switch statement. This rule raises an issue if the default clause is not the last one of the switch's cases.

Noncompliant Code Example

```
switch (param) {
  case 0:
    doSomething();
    break;
  default: // default clause should be the last one
    error();
    break;
  case 1:
    doSomethingElse();
    break;
}
```

Compliant Solution

```
switch (param) {
  case 0:
    doSomething();
    break;
  case 1:
    doSomethingElse();
    break;
  default:
    error();
    break;
}
```

---

# "toString()" and "clone()" methods should not return null (java:S2225)

Bug Major

Calling toString() or clone() on an object should always return a string or an object. Returning null instead contravenes the method’s implicit contract.

### Noncompliant Code Example

```
public String toString () {
  if (this.collection.isEmpty()) {
    return null; // Noncompliant
  } else {
    // ...
```

### Compliant Solution

```
public String toString () {
  if (this.collection.isEmpty()) {
    return "";
  } else {
    // ...
```

### See

- MITRE, CWE-476 - NULL Pointer Dereference
- CERT, EXP01-J. - Do not use a null in a case where an object is required

---

# "toString()" should never be called on a String object (java:S1858)

Code smell Minor

Invoking a method designed to return a string representation of an object which is already a string is a waste of keystrokes. This redundant construction may be optimized by the compiler, but will be confusing in the meantime.

### Noncompliant Code Example

```
String message = "hello world";
System.out.println(message.toString()); // Noncompliant;
```

### Compliant Solution

```
String message = "hello world";
System.out.println(message);
```

---

# Tag

## "@NonNull" values should not be set to null (java:S2637)

Bug Minor

Fields, parameters and return values marked @NotNull, @NonNull, or @Nonnull are assumed to have non-null values and are not typically null-checked before use. Therefore setting one of these values to null, or failing to set such a class field in a constructor, could cause NullPointerExceptions at runtime.

### Noncompliant Code Example

```
public class MainClass {

  @Nonnull
  private String primary;
  private String secondary;

  public MainClass(String color) {
    if (color != null) {
      secondary = null;
    }
    primary = color;  // Noncompliant; "primary" is Nonnull but could be set to null here
  }

  public MainClass() { // Noncompliant; "primary" is Nonnull but is not initialized
  }

  @Nonnull
  public String indirectMix() {
    String mix = null;
    return mix;  // Noncompliant; return value is Nonnull, but null is returned.
  }
```

### See

- MITRE, CWE-476 - NULL Pointer Dereference
- CERT, EXP01-J. - Do not use a null in a case where an object is required

---

# "@Override" should be used on overriding and implementing methods (java:S1161)

Code smell Major

Using the @Override annotation is useful for two reasons :

- It elicits a warning from the compiler if the annotated method doesn't actually override anything, as in the case of a misspelling.
- It improves the readability of the source code by making it obvious that methods are overridden.

### Noncompliant Code Example

```
class ParentClass {
  public boolean doSomething(){...}
}
class FirstChildClass extends ParentClass {
  public boolean doSomething(){...}  // Noncompliant
}
```

### Compliant Solution

```
class ParentClass {
  public boolean doSomething(){...}
}
class FirstChildClass extends ParentClass {
  @Override
  public boolean doSomething(){...}  // Compliant
}
```

### Exceptions

This rule is relaxed when overriding a method from the Object class like toString(), hashCode(), …

---

# Track uses of "TODO" tags (java:S1135)

Code smell Info

TODO tags are commonly used to mark places where some more code is required, but which the developer wants to implement later.

Sometimes the developer will not have the time or will simply forget to get back to that tag.

This rule is meant to track those tags and to ensure that they do not go unnoticed.

### Noncompliant Code Example

```
void doSomething() {
  // TODO
}
```

### See

- MITRE, CWE-546 - Suspicious Comment

# Modifieurs

## "static" members should be accessed statically (java:S2209)

Code smell Major

While it is *possible* to access static members from a class instance, it's bad form, and considered by most to be misleading because it implies to the readers of your code that there's an instance of the member per class instance.

### Noncompliant Code Example

```
public class A {
  public static int counter = 0;
}

public class B {
  private A first = new A();
  private A second = new A();

  public void runUpTheCount() {
    first.counter ++;  // Noncompliant
    second.counter ++;  // Noncompliant. A.counter is now 2, which is perhaps contrary to
expectations
  }
}
```

### Compliant Solution

```
public class A {
  public static int counter = 0;
}

public class B {
  private A first = new A();
  private A second = new A();

  public void runUpTheCount() {
    A.counter ++;  // Compliant
    A.counter ++;  // Compliant
  }
}
```

---

# Class variable fields should not have public accessibility (java:S1104)

Code smell Minor

Public class variable fields do not respect the encapsulation principle and has three main disadvantages:

- Additional behavior such as validation cannot be added.
- The internal representation is exposed, and cannot be changed afterwards.
- Member values are subject to change from anywhere in the code and may not meet the programmer's assumptions.

By using private attributes and accessor methods (set and get), unauthorized modifications are prevented.

### Noncompliant Code Example

```
public class MyClass {

  public static final int SOME_CONSTANT = 0;     // Compliant - constants are not checked

  public String firstName;                       // Noncompliant

}
```

### Compliant Solution

```
public class MyClass {

  public static final int SOME_CONSTANT = 0;     // Compliant - constants are not checked

  private String firstName;                      // Compliant

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

}
```

### Exceptions

Because they are not modifiable, this rule ignores public final fields. Also, annotated fields, whatever the annotation(s) will be ignored, as annotations are often used by injection frameworks, which in exchange require having public fields.

### See

- MITRE, CWE-493 - Critical Public Variable Without Final Modifier

---

# Public constants and fields initialized at declaration should be "static final" rather than merely "final" (java:S1170)

 Code smell  Minor

Making a public constant just final as opposed to static final leads to duplicating its value for every instance of the class, uselessly increasing the amount of memory required to execute the application.

Further, when a non-public, final field isn't also static, it implies that different instances can have different values. However, initializing a non-static final field in its declaration forces every instance to have the same value. So such fields should either be made static or initialized in the constructor.

### Noncompliant Code Example

```
public class Myclass {
  public final int THRESHOLD = 3;
}
```

### Compliant Solution

```
public class Myclass {
  public static final int THRESHOLD = 3;    // Compliant
}
```

### Exceptions

No issues are reported on final fields of inner classes whose type is not a primitive or a String. Indeed according to the Java specification:

> An inner class is a nested class that is not explicitly or implicitly declared static. Inner classes may not declare static initializers (§8.7) or member interfaces. Inner classes may not declare static members, unless they are compile-time constant fields (§15.28).

---

# Static fields should not be updated in constructors (java:S3010)

Code smell Major

Assigning a value to a static field in a constructor could cause unreliable behavior at runtime since it will change the value for all instances of the class.

Instead remove the field's static modifier, or initialize it statically.

### Noncompliant Code Example

```
public class Person {
  static Date dateOfBirth;
  static int expectedFingers;

  public Person(date birthday) {
    dateOfBirth = birthday;  // Noncompliant; now everyone has this birthday
    expectedFingers = 10;  // Noncompliant
  }
}
```

### Compliant Solution

```
public class Person {
  Date dateOfBirth;
  static int expectedFingers = 10;

  public Person(date birthday) {
    dateOfBirth = birthday;
  }
}
```

---

# Boolean

## Boolean checks should not be inverted (java:S1940)

Code smell Minor

It is needlessly complex to invert the result of a boolean comparison. The opposite comparison should be made instead.

### Noncompliant Code Example

```
if ( !(a == 2)) { ...}  // Noncompliant
boolean b = !(i < 10);  // Noncompliant
```

### Compliant Solution

```
if (a != 2) { ...}
boolean b = (i >= 10);
```

---

## Boolean expressions should not be gratuitous (java:S2589)

Code smell Major

If a boolean expression doesn't change the evaluation of the condition, then it is entirely unnecessary, and can be removed. If it is gratuitous because it does not match the programmer's intent, then it's a bug and the expression should be fixed.

### Noncompliant Code Example

```
a = true;
if (a) { // Noncompliant
  doSomething();
}

if (b && a) { // Noncompliant; "a" is always "true"
  doSomething();
}

if (c || !a) { // Noncompliant; "!a" is always "false"
  doSomething();
}
```

### Compliant Solution

```
a = true;
if (foo(a)) {
  doSomething();
}

if (b) {
  doSomething();
}

if (c) {
  doSomething();
}
```

### See

- MITRE, CWE-571 - Expression is Always True
- MITRE, CWE-570 - Expression is Always False

---

# Boolean literals should not be redundant (java:S1125)

Code smell Minor

Redundant Boolean literals should be removed from expressions to improve readability.

### Noncompliant Code Example

```
if (booleanMethod() == true) { /* ... */ }
if (booleanMethod() == false) { /* ... */ }
if (booleanMethod() || false) { /* ... */ }
doSomething(!false);
doSomething(booleanMethod() == true);

booleanVariable = booleanMethod() ? true : false;
booleanVariable = booleanMethod() ? true : exp;
booleanVariable = booleanMethod() ? false : exp;
booleanVariable = booleanMethod() ? exp : true;
booleanVariable = booleanMethod() ? exp : false;
```

### Compliant Solution

```
if (booleanMethod()) { /* ... */ }
if (!booleanMethod()) { /* ... */ }
```

```
if (booleanMethod()) { /* ... */ }
doSomething(true);
doSomething(booleanMethod());

booleanVariable = booleanMethod();
booleanVariable = booleanMethod() || exp;
booleanVariable = !booleanMethod() && exp;
booleanVariable = !booleanMethod() || exp;
booleanVariable = booleanMethod() && exp;
```

---

# Null should not be returned from a "Boolean" method (java:S2447)

Code smell Critical

While null is technically a valid Boolean value, that fact, and the distinction between Boolean and boolean is easy to forget. So returning null from a Boolean method is likely to cause problems with callers' code.

### Noncompliant Code Example

```
public Boolean isUsable() {
  // ...
  return null;  // Noncompliant
}
```

### See

- MITRE, CWE-476 - NULL Pointer Dereference
- CERT, EXP01-J. - Do not use a null in a case where an object is required

---

# Heritage

## Child class fields should not shadow parent class fields (java:S2387)

Code smell Blocker

Having a variable with the same name in two unrelated classes is fine, but do the same thing within a class hierarchy and you'll get confusion at best, chaos at worst.

### Noncompliant Code Example

```
public class Fruit {
  protected Season ripe;
  protected Color flesh;

  // ...
}

public class Raspberry extends Fruit {
  private boolean ripe;  // Noncompliant
  private static Color FLESH; // Noncompliant
}
```

### Compliant Solution

```
public class Fruit {
  protected Season ripe;
  protected Color flesh;

  // ...
}

public class Raspberry extends Fruit {
  private boolean ripened;
  private static Color FLESH_COLOR;

}
```

### Exceptions

This rule ignores same-name fields that are static in both the parent and child classes. This rule ignores private parent class fields, but in all other such cases, the child class field should be renamed.

```
public class Fruit {
  private Season ripe;
  // ...
}

public class Raspberry extends Fruit {
  private Season ripe;  // Compliant as parent field 'ripe' is anyway not visible from Raspberry
  // ...
}
```

---

# Child class methods named for parent class methods should be overrides (java:S2177)

Bug Major

When a method in a child class has the same signature as a method in a parent class, it is assumed to be an override. However, that's not the case when:

- the parent class method is static and the child class method is not.
- the arguments or return types of the child method are in different packages than those of the parent method.
- the parent class method is private.

Typically, these things are done unintentionally; the private parent class method is overlooked, the static keyword in the parent declaration is overlooked, or the wrong class is imported in the child. But if the intent is truly for the child class method to be different, then the method should be renamed to prevent confusion.

### Noncompliant Code Example

```
// Parent.java
import computer.Pear;
public class Parent {

  public void doSomething(Pear p) {
    //,,,
  }

  public static void doSomethingElse() {
    //...
  }
}

// Child.java
import fruit.Pear;
```

```
public class Child extends Parent {

  public void doSomething(Pear p) {  // Noncompliant; this is not an override
    // ...
  }


  public void doSomethingElse() {  // Noncompliant; parent method is static
    //...
  }
}
```

### Compliant Solution

```
// Parent.java
import computer.Pear;
public class Parent {

  public void doSomething(Pear p) {
    //,,,
  }

  public static void doSomethingElse() {
    //...
  }
}

// Child.java
import computer.Pear;  // import corrected
public class Child extends Parent {

  public void doSomething(Pear p) {  // true override (see import)
    //,,,
  }

  public static void doSomethingElse() {
    //...
  }
}
```

---

# Classes should not access their own subclasses during initialization (java:S2390)

Code smell Critical

When a parent class references a member of a subclass during its own initialization, the results might not be what you expect because the child class might not have been initialized yet. This could create what is known as an "initialisation cycle", or even a deadlock in some extreme cases.

To make things worse, these issues are very hard to diagnose so it is highly recommended you avoid creating this kind of dependencies.

### Noncompliant Code Example

```
class Parent {
  static int field1 = Child.method(); // Noncompliant
  static int field2 = 42;

  public static void main(String[] args) {
    System.out.println(Parent.field1); // will display "0" instead of "42"
  }
}

class Child extends Parent {
  static int method() {
    return Parent.field2;
  }
}
```

### See

- CERT, DCL00-J. - Prevent class initialization cycles
- Java Language Specifications - Section 12.4: Initialization of Classes and Interfaces

---

# Autres

## "=+" should not be used instead of "+=" (java:S2757)

Bug Major

The use of operators pairs ( =+, =- or =! ) where the reversed, single operator was meant (+=, -= or !=) will compile and run, but not produce the expected results.

This rule raises an issue when =+, =-, or =! is used without any spacing between the two operators and when there is at least one whitespace character after.

### Noncompliant Code Example

```
int target = -5;
int num = 3;

target =- num;  // Noncompliant; target = -3. Is that really what's meant?
target =+ num; // Noncompliant; target = 3
```

### Compliant Solution

```
int target = -5;
int num = 3;

target = -num;  // Compliant; intent to assign inverse value of num is clear
target += num;
```

---

## "==" and "!=" should not be used when "equals" is overridden (java:S1698)

Code smell Minor

It is equivalent to use the equality == operator and the equals method to compare two objects if the equals method inherited from Object has not been overridden. In this case both checks compare the object references.

But as soon as equals is overridden, two objects not having the same reference but having the same value can be equal. This rule spots suspicious uses of == and != operators on objects whose equals methods are overridden.

### Noncompliant Code Example

```
String firstName = getFirstName(); // String overrides equals
String lastName = getLastName();

if (firstName == lastName) { ... }; // Non-compliant; false even if the strings have the same value
```

### Compliant Solution

```
String firstName = getFirstName();
String lastName = getLastName();

if (firstName != null && firstName.equals(lastName)) { ... };
```

### Exceptions

Comparing two instances of the Class object will not raise an issue:

```
Class c;
if(c == Integer.class) { // No issue raised
}
```

Comparing Enum will not raise an issue:

```
public enum Fruit {
   APPLE, BANANA, GRAPE
}
public boolean isFruitGrape(Fruit candidateFruit) {
  return candidateFruit == Fruit.GRAPE; // it's recommended to activate S4551 to enforce comparison of Enums using ==
}
```

Comparing with final reference will not raise an issue:

```
private static final Type DEFAULT = new Type();

void foo(Type other) {
  if (other == DEFAULT) { // Compliant
  //...
  }
}
```

Comparing with this will not raise an issue:

```
 public boolean equals(Object other) {
```

```
  if (this == other) {  // Compliant
    return false;
  }
}
```

Comparing with java.lang.String and boxed types java.lang.Integer, … will not raise an issue.

**See**

- {rule:java:S4973} - Strings and Boxed types should be compared using "equals()"
- MITRE, CWE-595 - Comparison of Object References Instead of Object Contents
- MITRE, CWE-597 - Use of Wrong Operator in String Comparison
- CERT, EXP03-J. - Do not use the equality operators when comparing values of boxed primitives
- CERT, EXP50-J. - Do not confuse abstract object equality with reference equality

---

# Null checks should not be used with "instanceof" (java:S4201)

Code smell Minor

There's no need to null test in conjunction with an instanceof test. null is not an instanceof anything, so a null check is redundant.

### Noncompliant Code Example

```
if (x != null && x instanceof MyClass) { ... }  // Noncompliant

if (x == null || ! x instanceof MyClass) { ... } // Noncompliant
```

### Compliant Solution

```
if (x instanceof MyClass) { ... }

if (! x instanceof MyClass) { ... }
```

---

# Sections of code should not be commented out (java:S125)

Code smell Major

Programmers should not comment out code as it bloats programs and reduces readability.

Unused code should be deleted and can be retrieved from source control history if required.