ESLint Javascript Sonar Rules Based

For the control of best practices, conventions or possible errors in Javascript.

EqEqEq

The == and != operators do type coercion before comparing values. This is bad because it can mask type errors. For example, it evaluates:

```
* [] == false
* [] == ![]
* 3 == "03"
```

as true.

It is best to always use the side-effect-less === and !== operators instead.

Noncompliant Code Example

```
if (var == 'howdy') {...} // Noncompliant
Compliant Solution

if (var === 'howdy') {...}
```

Exceptions

Even if testing the equality of a variable against null doesn't do exactly what most JavaScript developers believe, usage of == or != is tolerated in such context. In the following case, if foo hasn't been initialized, its default value is not null but undefined. Nevertheless undefined == null, so JavaScript developers get the expected behavior.

```
if(foo == null) {...}
```

Eval

The eval function is a way to run arbitrary code at run-time. Generally it is considered to be very dangerous because it allows execution of arbitrary code. Its use is thus discouraged. If you have carefully verified that there is no other option than to use this construct, pay special attention not to pass any user-provided data into it without properly validating it beforehand.

Noncompliant Code Example

```
eval(code_to_be_dynamically_executed)
```

One statement per line

For better readability, do not put more than one statement on a single line.

Noncompliant Code Example

```
if(someCondition) doSomething();
```

Compliant Solution

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

Exceptions

Anonymous functions containing a single statement are ignored.

```
onEvent(function() { doSomething(); });               // Compliant
onEvent(function(p) { doSomething(); return p %2 ;}); // Noncompliant
```

Semicolon

In JavaScript, the semicolon ";" is optional as a statement separator, but omitting semicolons can be confusing, and lead to unexpected results because a semicolon is implicitly inserted at the end of each line.

Noncompliant Code Example

```
function fun() {
  return  // Noncompliant. ';' implicitly inserted at end of line
       5   // Noncompliant. ';' implicitly inserted at end of line
}
print(fun());  // prints "undefined", not "5"
```

Compliant Solution

```
function fun() {
  return 5;
}
print(fun());
```

Curly braces

While not technically incorrect, the omission of curly braces can be misleading, and may lead to the introduction of errors during maintenance.

Noncompliant Code Example

```
// the two statements seems to be attached to the if statement, but that is only true for the first
one:
if (condition)
  executeSomething();
  checkSomething();
Compliant Solution

if (condition) {
  executeSomething();
  checkSomething();
}
```

JavaScript does not have an integer type, but it does have bitwise operators '<<', '>>', '>>>','~', '&', '|'. These operators convert their operands from floating point values to integers and back, so they are not as efficient as in C or other languages. Further, they are rarely useful in browser applications, and the similarity to the logical operators can mask some programming errors.

Noncompliant Code Example

```
if (a & b) { ... } // Noncompliant; & used in error
var oppositeSigns = ((x ^ y) < 0); // Noncompliant; there's a clearer way to test for this
```

Compliant Solution

```
if (a && b) { ... }
var oppositeSigns = false;
if ( (x < 0 && y > 0) || (x > 0 && y < 0) ) {
  oppositeSigns = true;
}
```

Array and Object literals should always be preferred to Array and Object constructors.

Array constructors are error-prone due to the way their arguments are interpreted. If more than one argument is used, the array length will be equal to the number of arguments. However, using a single argument will have one of three consequences:

If the argument is a number and it is a natural number the length will be equal to the value of the argument. If the argument is a number, but not a natural number an exception will be thrown. Otherwise the array will have one element with the argument as its value. For these reasons, if someone changes the code to pass 1 argument instead of 2 arguments, the array might not have the expected length. To avoid these kinds of weird cases, always use the more readable array.

Object constructors don't have the same problems, but for readability and consistency object literals should be used.

Noncompliant Code Example

```
var a1 = new Array(x1, x2, x3);  // Noncompliant. Results in 3-element array.
var a2 = new Array(x1); // Noncompliant and variable in results
var a3 = new Array();  // Noncompliant. Results in 0-element array.

var o = new Object(); // Noncompliant

var o2 = new Object(); // Noncompliant
o2.a = 0;
```

```
o2.b = 1;
o2.c = 2;
o2['strange key'] = 3;
```

Compliant Solution

```
var a1 = [x1, x2, x3];
var a2 = [x1];
var a3 = [];

var o = {};

var o2 = {
  a: 0,
  b: 1,
  c: 2,
  'strange key': 3
};
```

Debugger statement

By definition such statement must absolutely be removed from the source code to prevent any unexpected behavior or added vulnerability to attacks in production.

Noncompliant Code Example

```
for (i = 1; i<5; i++) {
  // Print i to the Output window.
  Debug.write("loop index is " + i);
  // Wait for user to resume.
  debugger;
}
```

Compliant Solution

```
for (i = 1; i<5; i++) {
  // Print i to the Output window.
  Debug.write("loop index is " + i);
}
```

Primitive Wrappers

The use of wrapper objects for primitive types is gratuitous, confusing and dangerous. Simple literals should be used instead.

Noncompliant Code Example

```
var x = new Boolean(false);
if (x) {
  alert('hi');  // Shows 'hi'.
}
```

Compliant Solution

```
var x = false;
if (x) {
  alert('hi');
}
```

For In

The for...in statement allows you to loop through the names of all of the properties of an object. Unfortunately, the list of properties includes all those properties that were inherited through the prototype chain. This has the bad side effect of serving up functions when the interest is in data properties. Programs that don't take this into account can fail.

Therefore, the body of every for...in statement should be wrapped in an if statement that filters which properties are acted upon. It can select for a particular type or range of values, or it can exclude functions, or it can exclude properties from the prototype.

Noncompliant Code Example

```
for (name in object) {
    doSomething(name);  // Noncompliant
}
```

Compliant Solution

```
for (name in object) {
  if (object.hasOwnProperty(name)) {
    doSomething(name);
  }
}
```

Exceptions

Loops used to clone objects are ignored.

```
for (prop in obj) {
  a[prop] = obj[prop];  // Compliant by exception
}
```

Function Declarations Within Blocks

While most script engines support function declarations within blocks, it is not part of ECMAScript 5 and below, and from browser to browser the implementations are inconsistent with each other. ECMAScript 5 and below only allow function declarations in the root statement list of a script or function. If you are targeting browsers that don't support ECMAScript 6, use a variable initialized with a function expression to define a function within a block :

Noncompliant Code Example

```
if (x) {
  function foo() {}
}
```

Compliant Solution

```
if (x) {
  var foo = function() {}
}
```

Trailing Comma

Most browsers parse and discard a meaningless, trailing comma. Unfortunately, that's not the case for Internet Explorer below version 9, which throws a meaningless error. Therefore trailing commas should be eliminated.

Noncompliant Code Example

```
var settings = {
    'foo'  : oof,
    'bar' : rab,    // Noncompliant - trailing comma
};
```

Compliant Solution

```
var settings = {
    'foo'  : oof,
    'bar' : rab
};
```

Assignment With in Condition

Assignments within sub-expressions are hard to spot and therefore make the code less readable.

It is also a common mistake to write = when == was meant.

Ideally, sub-expressions should not have side-effects.

Noncompliant Code Example

```
doSomething(i = 42);
Compliant Solution

i = 42;
doSomething(i);
// or
doSomething(i == 42);  // Perhaps in fact the comparison operator was expected`
```

Exceptions

Assignments in while statement conditions, and assignments enclosed in relational expressions are allowed.

```
while ((line = nextLine()) != null) {...}  // Compliant

while (line = nextLine()) {...}  // Compliant

if (line = nextLine()) {...}  // Noncompliant
```

Label Placement

Any statement or block of statements can be identified by a label, but those labels should be used only on while, do-while and for statements. Using labels in any other context leads to unstructured, confusing code.

Noncompliant Code Example

```
myLabel:if (i % 2 == 0) {  // Noncompliant
 if (i == 12) {
   print("12");
   break myLabel;
 }
 print("Odd number, but not 12");
}
```

Compliant Solution

```
myLabel:for (i = 0; i < 10; i++) {   // Compliant
  print("Loop");
  break myLabel;
}
```

Switch Without Default

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;
}
```

Non Empty Case Without Break

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 return statement
    return;
  case 3:                               // Ends with comment when fall-through is intentional
    console.log("this case falls through")
    // fall through
  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();
}
```

No Empty Block

Most of the time a block of code is empty when a piece of code is really missing. So such empty block must be either filled or removed.

Noncompliant Code Example

```
for (var i = 0; i < length; i++) {}  // Empty on purpose or missing piece of code ?
```

Exceptions

When a block contains a comment, this block is not considered to be empty.

Tab Character

Developers should not need to configure the tab width of their text editors in order to be able to read source code.

So the use of tabulation character must be banned.

Bound Or Assigned Eval Or Arguments

Eval is used to evaluate a string as JavaScript code, and arguments is used to access function arguments through indexed properties. As a consequence, eval and arguments should not be bound or assigned, because doing so would overwrite the original definitions of those two reserved words.

What's more, using either of those two names to assign or bind will generate an error in JavaScript strict mode code.

Noncompliant Code Example

```
eval = 17; // Noncompliant
arguments++; // Noncompliant
++eval; // Noncompliant
var obj = { set p(arguments) { } }; // Noncompliant
var eval; // Noncompliant
try { } catch (arguments) { } // Noncompliant
function x(eval) { } // Noncompliant
function arguments() { } // Noncompliant
var y = function eval() { }; // Noncompliant
var f = new Function("arguments", "return 17;"); // Noncompliant

function fun() {
  if (arguments.length == 0) { // Compliant
    // do something
  }
}
```

Compliant Solution

```
result = 17;
args++;
++result;
var obj = { set p(arg) { } };
var result;
try { } catch (args) { }
function x(arg) { }
function args() { }
var y = function fun() { };
var f = new Function("args", "return 17;");

function fun() {
  if (arguments.length == 0) {
    // do something
  }
}
```

Variable Declaration After Usage

One of the biggest sources of confusion for JavaScript beginners is scoping. The reason scoping is so confusing in JavaScript is because JavaScript looks like a C-family language but doesn't act like one. C-family languages have block-level scope, meaning that when control enters a block, such as an if statement, new variables can be declared within that scope without affecting the outer scope. However, this is not the case in JavaScript.

To minimize confusion as much as possible, variables should always be declared before they are used.

Noncompliant Code Example

```
var x = 1;

function fun(){
  alert(x); // Noncompliant as x is declared later in the same scope
  if(something) {
    var x = 42; // Declaration in function scope (not block scope!) shadows global variable
  }
}

fun(); // Unexpectedly alerts "undefined" instead of "1"
```

Compliant Solution

```
var x = 1;

function fun() {
  print(x);
  if (something) {
    x = 42;
  }
}

fun(); // Print "1"
```

Redundant Boolean literals

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

Noncompliant Code Example

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

Compliant Solution

if (booleanVariable) { /* ... / } *if (!booleanVariable) { / ... / } if (booleanVariable) { / ...* */ } doSomething(true);

Exceptions

The use of literal booleans in comparisons which use identity operators (=== and !==) are ignored.

Redeclared Variable

This rule checks that the var keyword is not used to declare a variable with a name that is already in use. It applies to already defined variables as well as to function parameters.

This use of duplicate name is often unwanted and can lead to bugs and more generally to confusing code :

```
var a = 'foo';
var a = 'bar'; // Non-Compliant

function f(e) {
  var e = "event"; // Non-Compliant
}
```

Trailing Whitespace

Trailing whitespaces are simply useless and should not stay in code. They may generate noise when comparing different versions of the same file.

If you encounter issues from this rule, this probably means that you are not using an automated code formatter - which you should if you have the opportunity to do so.

Noncompliant Code Example

// The following string will error if there is a whitespace after '\'

```
var str = "Hello \
World";
```

No alert

alert(...) can be useful for debugging during development, but in production mode this kind of pop-up could expose sensitive information to attackers, and should never be displayed.

Noncompliant Code Example

```
if(unexpectedCondition)
{
  alert("Unexpected Condition");
}
```

Max lines per files

A source file that grows too much tends to aggregate too many responsibilities and inevitably becomes harder to understand and therefore to maintain. Above a specific threshold, it is strongly advised to refactor it into smaller pieces of code which focus on well defined tasks. Those smaller files will not only be easier to understand but also probably easier to test.

With Statement

The use of the with keyword produces an error in JavaScript strict mode code. However, that's not the worst that can be said against with.

Using with allows a short-hand access to an object's properties - assuming they're already set. But use with to access some property not already set in the object, and suddenly you're catapulted out of the object scope and into the global scope, creating or overwriting variables there. Since the effects of with are entirely dependent on the object passed to it, with can be dangerously unpredictable, and should never be used.

Noncompliant Code Example

```
var x = 'a';

var foo = {
  y: 1
}

with (foo) {  // Noncompliant
  y = 4;  // updates foo.x
  x = 3;  // does NOT add a foo.x property; updates x var in outer scope
}
print(foo.x + " " + x); // shows: undefined 3
```

Compliant Solution

```
var x = 'a';

var foo = {
  y: 1
}

foo.y = 4;
foo.x = 3;

print(foo.x + " " + x); // shows: 3 a
```

Multiline String Literals

Continuing a string across a linebreak is supported in most script engines, but it is not a part of ECMAScript. Additionally, the whitespace at the beginning of each line can't be safely stripped at compile time, and any whitespace after the slash will result in tricky errors.

Noncompliant Code Example

```
var myString = 'A rather long string of English text, an error message \
                actually that just keeps going and going -- an error \
                message to make the Energizer bunny blush (right through \
                those Schwarzenegger shades)! Where was I? Oh yes, \
                you\'ve got an error and all the extraneous whitespace is \
                just gravy.  Have a nice day.';  // Noncompliant
```

Compliant Solution

```
var myString = 'A rather long string of English text, an error message ' +
    'actually that just keeps going and going -- an error ' +
    'message to make the Energizer bunny blush (right through ' +
    'those Schwarzenegger shades)! Where was I? Oh yes, ' +
    'you\'ve got an error and all the extraneous whitespace is ' +
    'just gravy.  Have a nice day.';
    ```
Function Definition Inside Loop

Defining a function inside of a loop can yield unexpected results. Such a function keeps references
to the variables which are defined in outer scopes. All function instances created inside the loop
therefore see the same values for these variables, which is probably not expected.

Noncompliant Code Example
```

var funs = []; for (var i = 0; i < 13; i++) { funs[i] = function() { // Non-Compliant return i; }; } console.log(funs0); // 13 instead of 0 console.log(funs1); // 13 instead of 1 console.log(funs2); // 13 instead of 2 console.log(funs3); // 13 instead of 3 ...

```

Don't forget remove TODO

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

function doSomething() {
  // TODO
}

## Trailing Comment

This rule verifies that single-line comments are not located at the ends of lines of code. The main
idea behind this rule is that in order to be really readable, trailing comments would have to be
properly written and formatted (correct alignment, no interference with the visual structure of the
code, not too long to be visible) but most often, automatic code formatters would not handle this
correctly: the code would end up less readable. Comments are far better placed on the previous empty
line of code, where they will always be visible and properly formatted.

Noncompliant Code Example
```

var a1 = b + c; // This is a trailing comment that can be very very long

```


Compliant Solution
```

// This very long comment is better placed before the line of code var a2 = b + c;

```


## Don't forget remove FIXME

FIXME tags are commonly used to mark places where a bug is suspected, but which the developer wants
to deal with 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
```

function divide(numerator, denominator) { return numerator / denominator; // FIXME denominator value might be 0 }

```


The comma operator takes two expressions, executes them from left to right and returns the result of
the second one. Use of this operator is generally detrimental to the readability and reliability of
code, and the same effect can be achieved by other means.

Noncompliant Code Example
```

i = a += 2, a + b; // What's the value of i ? Compliant Solution

a += 2; i = a + b;

```
### Exceptions

Use of comma operator is tolerated in initialization and increment expressions of for loops.
```

for(i = 0, j = 5; i < 6; i++, j++) { ... }

```
## Comma operator
The comma operator takes two expressions, executes them from left to right and returns the result of
the second one. Use of this operator is generally detrimental to the readability and reliability of
code, and the same effect can be achieved by other means.

Noncompliant Code Example
```

i = a += 2, a + b; // What's the value of i ?

```
Compliant Solution
```

a += 2; i = a + b;

```
### Exceptions

Use of comma operator is tolerated in initialization and increment expressions of for loops.
```

for(i = 0, j = 5; i < 6; i++, j++) { ... }

```
## Nested If Depth

Nested if, for, while, switch, and try statements is a key ingredient for making what's known as
"Spaghetti code".

Such code is hard to read, refactor and therefore maintain.

Noncompliant Code Example

With the default threshold of 3:
```

if (condition1) { // Compliant - depth = 1 /* ... *\/ if (condition2) { // Compliant - depth = 2 /* ... */ for(int i = 0; i < 10; i++) { // Compliant - depth = 3, not exceeding the limit /* ... */ if (condition4) { // Non-Compliant - depth = 4 if (condition5) { // Depth = 5, exceeding the limit, but issues are only reported on depth = 4 /* ... */ } return; } } }

```
## Unreachable Code

Jump statements (return, break and continue) and throw expressions move control flow out of the
current code block. Typically, any statements in a block that come after a jump or throw are simply
wasted keystrokes lying in wait to confuse the unwary.

Noncompliant Code Example
```

fun(a) { var i = 10; return i + a;# i++; // Noncompliant; this is never executed }

```
Compliant Solution
```

int fun(int a) { int i = 10; return i + a; }

```
## Duplicate Function Argument

Function arguments should all have different names to prevent any ambiguity. Indeed, if arguments
have the same name, the last duplicated argument hides all the previous arguments with the same name
(those previous arguments remain available through arguments[i], so they're not completely
inaccessible).

This hiding makes no sense, reduces understandability and maintainability, and obviously can be error
prone. Furthermore, in strict mode, declaring arguments with the same name produces an error.

Noncompliant Code Example
```

function compute(a, a, c) { // Noncompliant } Compliant Solution

function compute(a, b, c) { // Compliant }

```
## Duplicate Property Name

Function arguments should all have different names to prevent any ambiguity. Indeed, if arguments
have the same name, the last duplicated argument hides all the previous arguments with the same name
(those previous arguments remain available through arguments[i], so they're not completely
inaccessible).

This hiding makes no sense, reduces understandability and maintainability, and obviously can be error
prone. Furthermore, in strict mode, declaring arguments with the same name produces an error.


Noncompliant Code Example
```

function compute(a, a, c) { // Noncompliant }

```
Compliant Solution
```

function compute(a, b, c) { // Compliant }

```
## Octal Number

Integer literals starting with a zero are octal rather than decimal values. While using octal values
is fully supported, most developers do not have experience with them. They may not recognize octal
values as such, mistaking them instead for decimal values.

Noncompliant Code Example
```

var myNumber = 010; // Noncompliant. myNumber will hold 8, not 10 - was this really expected?

```
Compliant Solution
```

var myNumber = 8;

```
## Strict Mode

Even thought it may be a good practice to enforce JavaScript strict mode, doing so could result in
unexpected behaviors on browsers that do not support it yet. Using this feature should therefore be
done with caution and with full knowledge of the potential consequences on browsers that do not
support it.

Noncompliant Code Example
```

function strict() { 'use strict'; }

```
## Unused Variable

If a local variable or a local function is declared but not used, it is dead code and should be
removed. Doing so will improve maintainability because developers will not wonder what the variable
or function is used for.

Noncompliant Code Example
```

function numberOfMinutes(hours) { var seconds = 0; // seconds is never used return hours * 60; }

```
Compliant Solution
```

function numberOfMinutes(hours) { return hours * 60; }

```
## Future Reserved Words

The following words may be used as keywords in future evolutions of the language, so using them as
identifiers should be avoided to allow an easier adoption of those potential future versions:

await
class
const
enum
export
extends
implements
import
interface
let
package
private
protected
public
static
super
yield
Use of these words as identifiers would produce an error in JavaScript strict mode code.

Noncompliant Code Example
```

var package = document.getElementsByName("foo"); // Noncompliant var someData = { package: true }; // Compliant, as it is not used as an identifier here

```
Compliant Solution
```

var elements = document.getElementsByName("foo"); // Compliant