CONFIDENTIAL

PROJECT PERSONNEL ONLY

Sample Project

JavaScript Coding Protocol

Version 1.0

August 24, 2018

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| No. | Version | Date of change(s) | Category | Area(s) changed (section, etc.) | | Details of change(s) | Personnel |
| 1 | 1.0 | 2018/8/24 | - | - | - | (First version created) | TIS |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

[1． About this protocol 1](#_Toc526511886)

[2． Protocol on code style 2](#_Toc526511887)

[2.1. Source 2](#_Toc526511888)

[2.1.1. Text code ★ 2](#_Toc526511889)

[2.1.2. Line breaks ★ 2](#_Toc526511890)

[2.1.3. Indents ★ 2](#_Toc526511891)

[2.1.4. Line length ★ 2](#_Toc526511892)

[2.1.5. Statements ★ 3](#_Toc526511893)

[2.1.6. Operators ★ 4](#_Toc526511894)

[2.1.7. Placement of brackets ★ 5](#_Toc526511895)

[2.1.8. Variable declarations ★ 7](#_Toc526511896)

[2.1.9. Omission of brackets in control syntax ★ 9](#_Toc526511897)

[2.2. Naming protocol 10](#_Toc526511898)

[2.2.1. Variables ★ 10](#_Toc526511899)

[2.2.2. Constants ★ 10](#_Toc526511900)

[2.2.3. Loop variables ★ 11](#_Toc526511901)

[2.2.4. Variables (Boolean values) ★ 11](#_Toc526511902)

[2.2.5. Variables (function names) ★ 12](#_Toc526511903)

[2.2.6. Variables (constructor function names) ★ 13](#_Toc526511904)

[2.2.7. Variables (jquery objects) ★ 14](#_Toc526511905)

[2.3. Literal descriptors 15](#_Toc526511906)

[2.3.1. String literal descriptors ★ 15](#_Toc526511907)

[2.3.2. Object literal descriptors ★ 16](#_Toc526511908)

[2.3.3. Regular expression literal descriptors ★ 16](#_Toc526511909)

[3． Security protocol 17](#_Toc526511910)

[3.1. General JavaScript protocol 17](#_Toc526511911)

[3.1.1. Do not use dynamic evaluations (eval()/function constructors). ★ 17](#_Toc526511912)

[3.2. jQuery protocol 17](#_Toc526511913)

[3.2.1. Do not link variables to arguments in functions that generate DOM nodes from character strings. ★ 17](#_Toc526511914)

[4． Performance protocol 19](#_Toc526511915)

[4.1. General JavaScript protocol 19](#_Toc526511916)

[4.1.1. Acquire the length of the array for scanning before starting a for statement.★ 19](#_Toc526511917)

[4.1.2. Use Array.prototype.join() as a process to link characters in a loop. ★ 20](#_Toc526511918)

[4.1.3. Do not use with phrases. ★ 21](#_Toc526511919)

[4.1.4. Place script elements near the end of a page. 22](#_Toc526511920)

[4.2. jQuery protocol 22](#_Toc526511921)

[4.2.1. Cache the processing results of selectors. ★ 22](#_Toc526511922)

[4.2.2. Place a selector that can be directly processed by the browser at the beginning of selector expressions. 23](#_Toc526511923)

[5． Protocol concerning the robustness of the program 24](#_Toc526511924)

[5.1. General JavaScript protocol 24](#_Toc526511925)

[5.1.1. Do not use strict directives in global scopes. 24](#_Toc526511926)

[5.1.2. Do not use comparative operators (==, !=) with implicit format conversion ★ 25](#_Toc526511927)

[5.1.3. Do not supplement code with exceptions (do not use try-catch-finally blocks) 26](#_Toc526511928)

[5.1.4. Use Object.keys for object loops. ★ 27](#_Toc526511929)

[5.1.5. Do not define global variables. ★ 27](#_Toc526511930)

[5.1.6. Do not expand prototypes. ★ 27](#_Toc526511931)

[5.1.7. Do not use multiple hyphens consecutively when directly writing JavaScript as HTML. ★ 28](#_Toc526511932)

[5.2. jQuery protocol 29](#_Toc526511933)

[5.2.1. Avoid direct use of DOM-related APIs. 29](#_Toc526511934)

[5.2.2. Do not use :enabled pseudo-selectors. ★ 29](#_Toc526511935)

# About this protocol

This document indicates standard rules that must be followed by application programmers when coding in JavaScript.

“★” at the end of a title indicates a rule that is particularly critical.

Rules without a “★” do not need to be applied uniformly; these are more like useful information or points of caution for JavaScript coding.

Read this information thoroughly, and carefully consider whether each rule can be applied.

NOTE

The new const and let functions introduced in ECMAScript 2015 (hereafter referred to as “ES2015”) are used in this protocol, based on ECMAScript5 (hereafter referred to as "ES5”).

const and let are included because they are supported by major browsers such as Internet Explorer 11 and are convenient. (For details, refer to “2.1.8. Variable declarations”.)

While ES2015 is referenced, please note that this protocol does not serve as advanced protocol based on later versions.

\* Refer to “List of APIs that can be used in Javascript” for information on APIs that can be used in programs.

\* Policies differ from project to project depending on the situation and purpose for using JavaScript, and there are restrictions on where JavaScript can be written in many business screens.

This document only explains protocol that needs to be followed when writing JavaScript. Follow the policies of your project regarding whether JavaScript can be used.

\* For protocol where “Scan with a static scan tool” is indicated, perform a scan using a static scan tool such as [JSLint](http://www.jslint.com/).

# Protocol on code style

## Source code layout

### Text code ★

All text code in source code must be written in **UTF-8**.

### Line breaks ★

Use **LF(0x0A)** for line breaks.

### Indent ★

Use **two single-byte spaces** for indents. Tab usage is not recognized.

\*Scan using a static scan tool.

### Line length ★

The maximum length of each line is **120 characters**. Ideally, lines should usually be **80 characters** or shorter.

\*Scan using a static scan tool.

### Statements ★

Avoid writing multiple statements in one line.

Be sure to write a semicolon (;) at the end of each statement,

* + Acceptable examples

doSomething();

doOtherThings();

* + Unacceptable examples

doSomething() // !! Semicolon is missing.

doSomething(); doOtherThings(); // !! Multiple statements in one line.

### Operators ★

When writing two operators, insert one blank space between each operator and the operands on either side.

* + Acceptable example

let nodeTop = parent.top + offset.top;

* + Unacceptable example

let price = basePrice\*discountRate; // !! No space between the operators and the operands

For single operators, do not insert a blank space between the operator and operand.

The exceptions, of course, are **typeof** and **instanceof**: blank spaces need to be inserted between the operator and operand even for single-line operators.

* + Acceptable example

for (let i = 0, len = array.length; i < len; i++) { // No space between single-line operator (**++**)and the operand (**i**).

if (typeof array[i] === 'string') { // Insert a space after typeof.

// ...

}

}

### Placement of brackets ★

Use the **“K&R”** **writing method** for the position of brackets used for blocks, etc.

Place an open bracket at the end of the line where the statement is written and align the close bracket with the beginning of the statement.

When using brackets with keywords such as **for**, **if** or **switch**, insert one blank space.

When using brackets with identifiers such as function calls or array index access, do not insert a space.

* + Acceptable example

for (let i = 0, len = array.length; i < len; i++) {

let item = array[i];

doSomethingWith(item);

}

* + Unacceptable example

for( let i = 0, len = array.length; i < len; i++) // !! No space between keyword (for) and bracket.

// Space between bracket and identifier (i).

{

let item = array [i]; // !! Space between variable and bracket.

doSomethingWith (item); // !! Space between function name and bracket.

}

**NOTE:** Unintended actions may occur if a style other than **K&R** is used, as JavaScript automatically adds a semicolon to the end of lines if one is missing.

Consider the following example, a function that generates and returns an object.

* + Acceptable example (K&R style)

function createObj() {

return {

key1: 'val1',

key2: 'val2'

};

}

If **BSD style** is used for this function, an unintended result (**undefined**) is returned.

* + Unacceptable example (BSD style)

function createObj()

{

return

{

key1: 'val1',

key2: 'val2'

};

}

This is because a semicolon is automatically added, causing the function to be interpreted as the following source code.

* + Unacceptable example (BSD style)

function createObj()

{

return; // “undefined” is returned here.

{

key1: 'val1',

key2: 'val2'

};

}

### Variable declarations ★

Variable declarations use let or const.

Use const for constants and when values are not resubstituted. Use let in all other cases.

var is a keyword that was used before ES2015 was registered. The scope is in function units.

Where possible, use let and const instead of var, as let and const enable variables to be declared in smaller block units than var.

* + Acceptable example

const MAX\_LENGTH = 100; //const is used as this is a constant

function doSomething() {

const records = getRecords(); //const is used as values are not resubstituted

for (let i = 0, len = records.length; i < len && i < MAX\_LENGTH; i++) {

let record = records[i];

// ....

}

}

* + Unacceptable example

function doSomething() {

var records = getRecords(), //Legacy style in which var is used to declare all variables at the beginning of a function

i,

len = records.length,

record;

for (i = 0; i < len; i++) {

record = records[i];

// ....

}

}

}

**NOTE:** If you forget to add **let, const** or **var**, the variables are interpreted as variables defined by an external scope or as global substitutions for global object properties. Take particular care to avoid this issue, as it is extremely difficult to detect, with the result that the code is operated as-is without being flagged as a syntax error.

### Omission of brackets in control syntax ★

Be sure to explicitly indicate brackets that can be omitted in the language specifications.

In the following code, brackets are omitted for if statements.

* + Unacceptable example

if ($form.has("div.errors")) return false;

The code should be written as follows.

* + Acceptable example

if ($form.has("div.errors")) {

return false;

}

**\*** The only exception in this protocol is patterns where Object.hasOwnProperty() is used in for-in statements.

In this case, as brackets can be omitted for for statements, the code can be written as follows.

for (let key in collection) if (collection.hasOwnProperty(key)) {

let value = collection[key];

// ...

}

## Naming protocol

This section explains rules for naming identifiers in a program.

### Variables ★

Variable names are treated as **nouns** that directly express the contents stored in the variable. **Adjectives** and **adjective phrases** can be added before or after them where necessary.

**Articles** are omitted unless there are concerns about misreading.

Names are written in all lower case, and the Camel format is used to join words.

If variable contents are used as a collection, such as a list or hash table, use the plural form of a noun as the variable name to express the elements, or use a noun expressing the collection itself.

Below are examples of names that follow this protocol.

let nextNode = null; // (adjective + noun)

let prevNode = null; // (adjective + noun)

let totalNumOfNodes = 0; // (adjective + noun + adjective phrase)

### Constants ★

const must be used to define constants.

Capitalize the first letter of every word in variable names or use an underscore (“\_”) to separate each word.

Below are examples of names that follow this protocol.

const PI = 3.1416;

const MAX\_THRESHOLD\_FREQUENCY = 40000;

### Loop variables ★

Commonly used variable names (**i, j, k…**) can be used to name variables used as counters in loop syntax such as for statements.

### Variables (Boolean values) ★

### Use adjectives or clauses for the names of variables storing Boolean values.

As a general rule, use present tense for verbs indicating a state, use past tense for verbs indicating an action and omit the subject when it is obvious.

Below are examples of names that follow this protocol.

let isValid = false; // (verb in present tense indicating status; subject omitted)

let needsDefault = true; // verb in present tense indicating status; subject omitted)

let loadingFinished = false; // (verb in past tense indicating action)

let available = true; // (adjective)

let inProgress = false; // (adjective phrase)

### Variables (function names) ★

In JavaScript, functions are simply objects, so function names should simply be the name of the variable storing the function object.

The names of variables storing function objects should be **root forms of verbs indicating an action** to directly express the nature of the process.

When a function has an argument, a **preposition** can be added with the nature of the argument as an object (**verb** + **preposition**).

When a function is an inquiry process that does not change the contents (an accessor), a **noun** can be used to directly express the nature of the function’s return value.

It is also possible to use a **noun** to express the purpose when the function object will be transferred using arguments or return values with the intent of executing it outside the scope.

Below are examples of names that follow this protocol.

function initialize() { // (verb in root form indicating action)

// ...

}

function appendTo(parentNode) { // (verb in root form indicating action + preposition)

// ...

};

function price() { // (noun)

// ...

return this.attributes.price;

}

function lastRecordOf(resultSet) { // (noun + preposition)

// ...

return resultSet[resultSet.length];

}

let callback = function() { // (noun)

// ....

return callback;

};

### Variables (constructor function names) ★

For functions that are expected to be used with new operators (constructors), use a **noun** to directly express the characteristics of the instances generated by the constructor and capitalize the first letters.

Below are examples of names that follow this protocol.

function BigDecimal(value, /\*optional\*/ scale) {

this.\_value = isNumber(value) ? value.toString() : "0";

this.\_scale = isNumber(scale) ? Math.floor(scale) : getScale(this.\_value);

}

BigDecimal.prototype = {

add : BigDecimal\_add,

subtract: BigDecimal\_subtract,

multiply: BigDecimal\_multiply,

toString: BigDecimal\_toString,

toNumber: BigDecimal\_toNumber

};

function BigDecimal\_add(value) {

//…

}

### Variables (jQuery objects) ★

Names of variables storing jQuery objects should start with “**$**".

Below are examples of names that follow this protocol.

$('button.openMenu').click(function() {

let $button = $(this); // jQuery object for clicked button element

let id = $button.prop('id');

let target = $button.prop('data-target');

$button.prop('disabled', true);

$('#' + target).openMenu(function() {

$button.prop('disabled', false);

});

return false;

});

## Literal descriptors

### String literal descriptors ★

Use double quotes (“) to quote string literals.

Single quotes (‘) can be used if there are many double quotes in a character string and escaping them will be a nuisance.

* + Acceptable example

let prefix = "nablarch";

let message = "How to escape single quote(') in javascript?";

let $field = $('<input type="text" name="searchText" value="" />'); // Single quotes can be used

// if there are many double quotes.

* + Unacceptable example

let prefix = 'nablarch'; // !! Double quotes must be used in normal situations.

let message = "How to escape single quote(\') in javascript?"; // !! Escaping single quotes is not necessary in double quotes.

**NOTE:** In JSON format, single quotes and single quote escapes are not recognized for string literals.

To avoid unnecessary confusion, this protocol’s literal descriptor protocol is designed to fit JSON format as long as this poses no readability issues.

### Object literal descriptors ★

Double quotes are used for key strings, but can be omitted if the key consists only of single-byte alphanumeric characters and underscores.

Single quotes (‘) can be used for key names if there are many double quotes in a character string and escaping them will be a nuisance.

* + Examples of descriptors

let obj = {

"key1": "val1",

"key2": "val2",

"key3": "val3"

};

let obj2 = {

key1: "val1", // Quotes omitted for key strings

key2: "val2",

key3: "val3"

};

**NOTE:** Be aware that in JSON format, double quotes must always be used for key strings of object literals.

### Regular expression literal descriptors ★

When using regular expressions, use regular expression literals instead of string arguments where possible.

Use the RegExp() constructor if the content of a regular expression will dynamically change or is complicated enough to require a comment.

# Security protocol

## General JavaScript protocol

### Do not use dynamic evaluations (eval()/Function constructors) in the source code. ★

They generally should not be used if a string that can be externally operated is linked to an argument in the function, as this may enable users to execute the code of their choice in their browser.

(Refer to “**List of APIs that can be used in Javascript**”.)

\*Scan using a static scan tool.

**NOTE:** These functions are almost never needed as closure can be used in JavaScript.

## jQuery protocol

### Do not link variables to arguments in functions that generate DOM nodes from character strings. ★

For $() constructors and methods such as $.append() in jQuery, DOM nodes equivalent to the content of the HTML can be generated as a string expressing HTML is passed as an argument.

If values that can be externally operated are linked to these strings, this directly causes an XSS vulnerability as users can execute the script of their choice by inserting script elements or similar as variables.

If the specifications require a process of this nature, this area of the code needs to be thoroughly reviewed by an expert.

The source code below is a typical example of an XSS vulnerability.

* + Unacceptable example

$("#messages").append($('<div class="message">'+ record.message + '</div>'));

If this record.message content is an externally input value, an XSS attack may occur.

For example, users can execute the script of their choice if a string like the following is used as the content of this variable

record.message = "<script>alert('a malicious program!!');</" + "script>";

In this example, the vulnerability can be avoided by fixing the code as shown below.

* + Acceptable example

$("#messages").append($('<div class="message">').text(record.message));

# Performance protocol

## General JavaScript protocol

### Acquire the length of the array for scanning before starting a for statement. ★

When operating an array object (array, argument object, etc.), the performance may decline if length properties are accessed for each loop as shown below.

* + Unacceptable example

let array = getLongArray();

for (let i = 0; i < array.length; i++) { // !! The length property is accessed every time.

let item = array[i];

// ...

}

Use length properties acquired by the initialization block of a for statement to judge terminal conditions.

* + Acceptable example

let array = getLongArray();

for (let i = 0, len = array.length; i < len; i++) {

let item = array[i];

// ...

}

### Use Array.prototype.join() as a process to link characters in a loop. ★

Performance severely declines in some browsers when a + operator or String.prototype.concat() is used to join character strings in a loop as shown in the example below.

* + Unacceptable example

let records = getRecords();

let tbody = "<tbody>";

for (let i = 0, len = records.length; i < len; i++) {

let record = array[i];

tbody += "<tr>"

+ "<td>" + record.name + "</td>"

+ "<td>" + record.address + "</td>"

// ...

+ "</tr>";

}

tbody += "</tbody>";

$("#dataTable tbody").replaceWith($(tbody));

In cases like these, reliable performance in all browsers can be achieved by applying push() to the character string to be linked to the array and applying join() collectively at the end.

* + Acceptable example

let records = getRecords();

let tbody = ["<tbody>"];

for (let i = 0, len = records.length; i < len; i++) {

let record = array[i];

tbody.push(

"<tr>"

+ "<td>" + record.name + "</td>"

+ "<td>" + record.address + "</td>"

// ...

+ "</tr>"

);

}

tbody.push("</tbody>");

$("#dataTable tbody").replaceWith($(tbody.join("")));

// ...

}

### Do not use with phrases. ★

This is because using with phrases causes a scope to be inserted inside a scope containing local variables, reducing the accessibility of local variables.

### Place script elements near the end of a page.

During downloading and script evaluation of JavaScript files in the initial display process in the browser, the rendering process of subsequent elements stops.

As a result, completion of image drawing may be delayed if a script element is inserted in a head element or document.

To avoid this issue, it is recommended to place script elements at the end of a document (immediately before the end tag of the body element).

This enables image rendering to be performed parallel to downloading and script evaluation of script files, improving physical performance.

## jQuery protocol

### Cache the processing results of selectors. ★

Selector processes using the **$()** constructor place a comparatively large load on the system, so be sure to cache local variables if selector results will be referenced multiple times.

The following example is inefficient because selector processors with the same content are called individually.

* + Unacceptable example

$("#dataTable tr:odd").addClass("odd");

$("#dataTable tr button.clearField").click(function() {

// ...

});

This should be written as shown in the example below.

* + Acceptable example

let $rows = $("#dataTable tr");

$rows.find(":odd").addClass("odd");

$rows.find("button.clearField").click(function() {

// ...

});

### Place a selector that can be directly processed by the browser at the beginning of selector expressions.

For selector processes using **$()**, the content of the passed selector expression is analyzed and then passed to the browser if it can be executed by the native API of the browser.

In particular, **$("#ID")**, **$("TAG")** and **$(".CLASSNAME")** are directly processed in the browser by getElementById(), getElementsByTagName() and getElementsByClassName() respectively, which makes the process extremely fast.

As a result, the performance of selectors can be improved by placing these two selectors at the beginning.

# Protocol concerning the robustness of the program

## General JavaScript protocol

### Do not use strict directives in global scopes.

Using strict directives enables code with the following issues to be detected as an explicit error in browsers where this is supported.

1. Unintended access to global variables when a programmer forgets to add a var statement
2. Use of with statements
3. Duplicate definitions of properties in object literals
4. Access to global variables via this variables (≒ forgetting to add a new statement when calling a constructor function)

It is generally recommended to use strict directives.

The exception is global scopes. Use of strict directives here is prohibited, because using a strict directive in a global scope as shown in the example below causes the directive to be applied to all code that is evaluated after that point, which may cause the external library to become inoperable.

* + Unacceptable

“use strict”; // !! The strict mode is applied to all of the script.

(function($) {

$(initialize);

function initialize() {

//…

}

})(jQuery)

The correct usage is to specify strict directives individually in function scopes as shown below.

* + Acceptable example

(function($) {

“use strict”;

$(initialize);

function initialize() {

//…

}

})(jQuery)

### Do not use comparative operators (==, !=) with implicit format conversion ★

If different formats are used on either side of the logical operators = and !==, an implicit conversion occurs, which may lead to results that were not intended by the programmer.

For example, the following statement is judged as true

\*Scan using a static scan tool.

* + Unacceptable example

(" " == false) // -> true

(undefined == null) // -> true

As a general rule, do not use these operators. Instead, use comparative operators that do not convert the format (==, !==) as shown below.

For example, the following statement is judged as true.

* + Acceptable example

(" " !== false) // -> false

(undefined !== null) // -> false

### Do not supplement code with exceptions (do not use try-catch-finally blocks).

Supplementing an error that occurs in a browser will not enable fault log output or a recovery process the way it does in server-side applications. Error objects should generally not be supplemented, and depend on the default operations of the browser.

### Use Object.keys for object loops.★

for-in loops may cause processing of items that are not expected to be processed, as the properties in a prototype chain are included in scans.

To avoid this, use **Object.keys()** to write loops.

* + Example 1

let record = getRecord();

Object.keys(record).forEach(function(key) {

console.log(record[key]);

});

### Do not define global variables. ★

It is prohibited to define new global variables (window object properties).

### Do not expand prototypes. ★

It is generally prohibited to change prototype objects for existing constructor functions.

The exception is cases where a jQuery plugin is created. iQuery object prototype definitions (jQuery.fn) can be expanded in these cases.

### Do not use multiple hyphens consecutively when directly writing JavaScript as HTML.★

Use an n:script custom tag when directly writing JavaScript as HTML. N:script custom tags output JavaScript code as HTML comments to prevent JavaScript code from being displayed on the screen in browsers that do not support JavaScript.

* + Example of JSP implementation

<n:script type="text/javascript">

// Write JavaScript here.

</script>

* + Example of HTML output

<script type="text/javascript">

<!--

// Write JavaScript here.

-->

</script>

Due to the nature of HTML’s specifications, multiple “-“ cannot be written consecutively in HTML comments. An HTML syntax error occurs if multiple “-“ are written consecutively in the above JavaScript,

* + Unacceptable example

let message = "--"; // Consecutive “-“ in character string.

let count = 10;

count--; // Consecutive “-“ in decrement operators.

Consider whether JavaScript can be written to external files. If it cannot, handle this by implementing the following.

* + Acceptable example

let message = "\-\-"; // Escapes.

let count = 10;

count -= 1; // Does not use decrement operator.

## jQuery protocol

### Avoid direct use of DOM-related APIs.

DOM-related APIs have different interface depending on the browser. Use these APIs via a function equivalent to jQuery instead of using them directly.

(Refer to **List of APIs that can be used in Javascript**)

### Do not use :enabled pseudo-selectors. ★

Elements matching :enabled pseudo-selectors are indicated as “false input items or elements for which no disabled attributes are set, for which **type=”hidden”** is not applied.” Hidden elements do not match :enabled regardless of whether disabled attributes are set.

For example, the input element <input type=”hidden” name=”name1” value=”val1”/> **matches nether :enabled nor :disabled**.

This specification is unintuitive and a hotbed for bugs, so usage is prohibited.

* + Unacceptable example

let $inputs = $("input, select, textarea").filter(":enabled"); // No hidden items included.

To avoid this, substitute it with **.not(":disabled")** as shown in the example below.

* + Acceptable example

let $inputs = $("input, select, textarea").not(":disabled");