関係者外秘

PJ関係者限り

サンプルプロジェクト

JavaScriptコーディング規約

第１．０版

2018年8月24日

[会社名]

[部門名]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| No. | 版数 | 変更日 | 区分 | 変更箇所 | | 変更内容 | 担当者 |
| 頁 | 項番 |
| 1 | 1.0版 | 2018/8/24 | 新規 | - | - | (新規作成) | TIS |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

[1． 本規約について 1](#_Toc339959627)

[2． コードスタイルに関する規約 2](#_Toc339959628)

[2.1. ソースコードレイアウト 2](#_Toc339959629)

[2.1.1. 文字コード ★ 2](#_Toc339959630)

[2.1.2. 改行 ★ 2](#_Toc339959631)

[2.1.3. インデント ★ 2](#_Toc339959632)

[2.1.4. 行の長さ ★ 2](#_Toc339959633)

[2.1.5. ステートメント ★ 3](#_Toc339959634)

[2.1.6. 演算子 ★ 4](#_Toc339959635)

[2.1.7. 括弧の位置 ★ 5](#_Toc339959636)

[2.1.8. 変数宣言 ★ 7](#_Toc339959637)

[2.1.9. 制御構文中の括弧の省略 ★ 9](#_Toc339959638)

[2.2. 命名規約 10](#_Toc339959639)

[2.2.1. 変数 ★ 10](#_Toc339959640)

[2.2.2. 定数 ★ 10](#_Toc339959641)

[2.2.3. ループ変数 ★ 11](#_Toc339959642)

[2.2.4. 変数(真偽値) ★ 11](#_Toc339959643)

[2.2.5. 変数(関数名) ★ 12](#_Toc339959644)

[2.2.6. 変数(コンストラクタ関数名) ★ 13](#_Toc339959645)

[2.2.7. 変数(jQueryオブジェクト) ★ 14](#_Toc339959646)

[2.3. リテラルの記述 15](#_Toc339959647)

[2.3.1. 文字リテラルの記述 ★ 15](#_Toc339959648)

[2.3.2. 配列リテラルの記述 ★ 16](#_Toc339959649)

[2.3.3. オブジェクトリテラルの記述 ★ 17](#_Toc339959650)

[2.3.4. 正規表現リテラルの記述 ★ 17](#_Toc339959651)

[3． セキュリティに関する規約 18](#_Toc339959652)

[3.1. JavaScript一般 18](#_Toc339959653)

[3.1.1. ソースコードの動的評価(eval()/Functionコンストラクタ)を使用しない。 ★ 18](#_Toc339959654)

[3.2. jQuery関連 18](#_Toc339959655)

[3.2.1. 文字列からDOMノードを生成する関数の引数に変数を連結しない。 ★ 18](#_Toc339959656)

[4． パフォーマンスに関する規約 20](#_Toc339959657)

[4.1. JavaScript一般 20](#_Toc339959658)

[4.1.1. for文開始前に走査対象となる配列の長さを取得しておく。 ★ 20](#_Toc339959659)

[4.1.2. ループ内での文字連結処理には Array.prototype.join() を使用する。 ★ 21](#_Toc339959660)

[4.1.3. with句を使用しない。★ 22](#_Toc339959661)

[4.1.4. script要素はページの終端付近に配置する。 23](#_Toc339959662)

[4.2. jQuery関連 23](#_Toc339959663)

[4.2.1. セレクタの処理結果をキャッシュする。 ★ 23](#_Toc339959664)

[4.2.2. ブラウザが直接処理できるセレクタをセレクタ式の先頭に記述する。 24](#_Toc339959665)

[5． プログラムの堅牢性に関する規約 25](#_Toc339959666)

[5.1. JavaScript一般 25](#_Toc339959667)

[5.1.1. strictディレクティブをグローバルスコープで使用しない。 25](#_Toc339959668)

[5.1.2. 暗黙的な型変換をともなう比較演算子 (==, !=) は使用しない。 ★ 26](#_Toc339959669)

[5.1.3. 例外を捕捉しない。(try-catch-finallyブロックを使用しない。) 27](#_Toc339959670)

[5.1.4. for-in ループ内では Object.prototype.hasOwnProperty() でフィルタをかける。★ 28](#_Toc339959671)

[5.1.5. グローバル変数の定義を行なわない。 ★ 29](#_Toc339959672)

[5.1.6. プロトタイプの拡張を行わない。 ★ 29](#_Toc339959673)

[5.1.7. JavaScriptをHTMLに直接記述する場合、「-」（ハイフン）を2つ以上連続して記述しない。★ 30](#_Toc339959674)

[5.2. jQuery関連 31](#_Toc339959675)

[5.2.1. DOM関連APIを直接使用することを避ける。 31](#_Toc339959676)

[5.2.2. :enabled 擬似セレクタを使用しない。 ★ 31](#_Toc339959677)

# 本規約について

本ドキュメントは、アプリケーションプログラマがJavaScriptをコーディングする際に準拠しなければならない標準ルールについて記述している。

本ドキュメントに記載したルールのうち、必ず守るべきものについては、タイトルの文末に「★」マークをつけている。

「★」マークがついていないルールは、画一的に適用できるものではなく、どちらかというと JavaScriptをコーディングする際に知っておくべき心得や注意事項に近い。

内容をよく読み、ルールの適用可否を慎重に判断すること。

【注意事項】

本規約はECMAScript5（以下ES5）をベースとして、ECMAScript 2015（以下ES2015）の新機能であるconstとletを取り入れて作成した。

constとletはInternet Explorer 11を含む主要ブラウザでサポートされており、便利なため取り入れた（詳細は「2.1.8.変数宣言」を参照）。

ES2015には言及しているが、ES2015以降をベースとした先進的な規約ではない点は注意すること。

※ プログラム内で使用できるAPI については、「使用可能API一覧」を参照すること。

* JavaScriptを記述する場面や用途は、プロジェクトごとに方針が異なり、業務画面でJavaScriptを記述できる箇所を制限している場合も多い。

本ドキュメントでは、JavaScriptを記述する場合に従うべき規約についてのみ述べ、JavaScript自体の使用可否については各プロジェクトの方針に従うものとする。

※ 「静的検査ツールで検査を行う。」と記載のある規約については、一般的に利用されている[JSLint](http://www.jslint.com/)などの静的検査ツールを利用して検査を行うこと。

# コードスタイルに関する規約

## ソースコードレイアウト

### 文字コード ★

ソースコードの文字コードは全て **UTF-8** で記述する。

### 改行 ★

改行には**LF(0x0A)** を使用すること。

### インデント ★

インデントは **半角スペース** を**2つ**使用すること。タブの使用は認めない。

※静的検査ツールで検査を行う。

### 行の長さ ★

1行あたりの文字数は最大で **120字** を上限とする。通常は**80字** 以内とすることが望ましい。

※静的検査ツールで検査を行う。

### ステートメント ★

複数のステートメントを1行に記述することは避ける。

各ステートメントの末尾には必ず セミコロン **";"** を記述すること。

* + 良い例

doSomething();

doOtherThings();

* + 悪い例

doSomething() // !! セミコロンを省略している。

doSomething(); doOtherThings(); // !! 1行に複数のステートメントを記述している。

### 演算子 ★

2項演算子は前後のオペランドとの間に空白1文字を挿む。

* + 良い例

let nodeTop = parent.top + offset.top;

* + 悪い例

let price = basePrice\*discountRate; // !! 2項演算子とオペランドの間が隣接している。

一方、単項演算子の場合はオペランドとの間を隣接させる。

ただし、同じ単行演算子でも、**typeof** と **instanceof** については(当然ながら)オペランドとの間に空白が必要である。

* + 良い例

for (let i = 0, len = array.length; i < len; i++) { // 単行演算子(**++**)とオペランド(**i**)の間は詰める。

if (typeof array[i] === 'string') { // ただし、typeof の後はスペースを挟む。

// ...

}

}

### 括弧の位置 ★

ブロック等で使用する括弧の記述位置は、いわゆる **K&R記法** に従うものとする。

すなわち、ステートメントと同じ行の末尾に開き括弧を配置し、かつ、閉じ括弧の位置はステートメントの先頭列に合わせる。

また、 **for**, **if**, **switch** 等のキーワードと括弧が並ぶ場合は、空白1文字を挿む。

一方、関数の呼び出しや、配列添字のアクセスのように、識別子と括弧が並ぶ場合は、間をあけずに隣接させる。

* + 良い例

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

let item = array[i];

doSomethingWith(item);

}

* + 悪い例

for( let i = 0, len = array.length; i < len; i++) // !! キーワード(for)と括弧が隣接している。

// また、括弧と識別子(i)の間が空いている。

{

let item = array [i]; // !! 変数名と括弧の間があいている。

doSomethingWith (item); // !! 関数名と括弧の間が空いている。

}

**(注)** JavaScriptでは、必要に応じて行末にセミコロンを自動的に補完するという仕様があるため、

**K&R** 以外のスタイルでは、本来意図しない挙動となる場合がある。

例えばオブジェクトを生成してリターンする以下の関数の例を考える。

* + 良い例 (K&Rスタイル)

function createObj() {

return {

key1: 'val1',

key2: 'val2'

};

}

この関数を仮に **BSDスタイル** で記述した場合は、本来の意図とはことなる結果(**undefined**)が返る。

* + 悪い例 (BSDスタイル)

function createObj()

{

return

{

key1: 'val1',

key2: 'val2'

};

}

これは、セミコロンの補完によって、以下の様なソースコードと解釈されてしまうためである。

* + 悪い例 (BSDスタイル)

function createObj()

{

return; // ここで undefined がリターンされる。

{

key1: 'val1',

key2: 'val2'

};

}

### 変数宣言 ★

変数宣言はletとconstのいずれかを使用する。

値を再代入しない場合、または定数はconstを使用する。それ以外はletを使用する。

varはES2015が登場する前に使われていたキーワードであり、スコープが関数単位という特徴がある。

可能な限りvarは使わず、より狭いブロック単位のスコープで変数を宣言できるletやconstを使うようにすること。

* + 良い例

const MAX\_LENGTH = 100; //定数なのでconstを使用している

function doSomething() {

const records = getRecords(); //再代入しないのでconstを使用している

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

let record = records[i];

// ....

}

}

* + 悪い例

function doSomething() {

var records = getRecords(), //varを使用して関数の先頭ですべての変数を宣言するレガシーなスタイル

i,

len = records.length,

record;

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

record = records[i];

// ....

}

}

}

**(注)** **let、const、var** をつけ忘れた場合、外部スコープの定義済み変数、もしくは、グローバルオブジェクトのプロパティに対する代入と解釈されるため、

構文エラーとはならずにそのまま動作してしまう。

これは、非常に検出しづらい不具合の原因となるので、特に注意すること。

### 制御構文中の括弧の省略 ★

言語仕様上、省略が許されている括弧についても、必ず明示的に記述する。

次のコードはif文の括弧を省略している。

* + 悪い例

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

このコードは次のように記述すべきである。

* + 良い例

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

return false;

}

**(注)** この規約の唯一の例外は、for-in文中でObject.hasOwnProperty() を使用するパターンである。

この場合、for文の括弧を省略し、以下の様に記述することが許される。

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

let value = collection[key];

// ...

}

## 命名規約

本節では、プログラム中の各識別子の命名規則について説明する。

### 変数 ★

変数の名前は、その変数に格納する内容を端的に表す **名詞** とし、必要に応じて前後に **形容詞** 、 **形容詞句** を付加する。

また、誤読の心配が無い限り **冠詞** は省略する。

先頭文字は小文字とし、 各単語はキャメル記法に従って連結する。

特に、変数の内容がリストやハッシュテーブルのようなコレクションとして使用される場合は、

変数名は要素を表す名詞の複数形か、そのコレクション自体を表す名詞とする。

以下は本規約に従った命名例である。

let nextNode = null; // (形容詞) + (名詞)

let prevNode = null; // (形容詞) + (名詞)

let totalNumOfNodes = 0; // (形容詞) + (名詞) + (形容詞句)

### 定数 ★

定数は必ずconstを使用して定義する。

変数名は全て大文字かつ単語間の区切りにアンダースコア(**"\_"**)を用いる。

以下は本規約に従った命名例である。

const PI = 3.1416;

const MAX\_THRESHOLD\_FREQUENCY = 40000;

### ループ変数 ★

for文等のループ構文において、カウンタとして使用される変数の名称には、

慣用的に使用される変数名 **(i, j, k ...)** を使用できる。

### 変数(真偽値) ★

真偽値を格納する変数の名前は、**形容詞**、もしくは **節** とする。

節の時制は、基本的に状態動詞の場合は現在形、動作動詞の場合は過去形とし、主語が明確である場合は省略する。

以下は本規約に従った命名例である。

let isValid = false; // (現在形状態動詞:主語省略)

let needsDefault = true; // (現在形状態動詞:主語省略)

let loadingFinished = false; // (過去形動作動詞)

let available = true; // (形容詞)

let inProgress = false; // (形容詞句)

### 変数(関数名) ★

JavaScriptでは関数も単なるオブジェクトであるため、関数名は単に関数オブジェクトを格納する変数の名前である。

関数オブジェクトを格納する変数の名前は、その処理の内容を端的に表す **原型の動作動詞** とする。

関数が引数をとる場合は、その引数の内容を目的語とする **前置詞** を追加し、(**動詞** + **前置詞**) としてもよい。

なお、その関数が、内部状態を変更しない単純な問い合わせ処理(いわゆるアクセサ)である場合は、その関数の戻り値の内容を端的に表す **名詞** としてもよい。

また、関数オブジェクトをスコープの外部で実行することを意図して、引数や戻り値を通じて受け渡す場合も、その目的を表す **名詞** としてよい。

以下は本規約に従った命名例である。

function initialize() { // (原形動作動詞)

// ...

}

function appendTo(parentNode) { // (原形動作動詞) + (前置詞)

// ...

};

function price() { // (名詞)

// ...

return this.attributes.price;

}

function lastRecordOf(resultSet) { // (名詞) + (前置詞)

// ...

return resultSet[resultSet.length];

}

let callback = function() { // (名詞)

// ....

return callback;

};

### 変数(コンストラクタ関数名) ★

**new**演算子とあわせて使用することを前提としている関数(コンストラクタ)の関数名は、

そのコンストラクタが生成するインスタンスの特徴を端的に表す **名詞** とし、先頭の文字は大文字で開始すること。

以下は本規約に従った命名例である。

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) {

//…

}

### 変数(jQueryオブジェクト) ★

jQueryオブジェクトを格納する変数の名前は “**$**" で開始すること。

以下は本規約に従った命名例である。

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

let $button = $(this); // クリックされたbutton要素のjQueryオブジェクト

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

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

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

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

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

});

return false;

});

## リテラルの記述

### 文字リテラルの記述 ★

文字リテラルのクォート文字には、ダブルクォート(**"**)を使用すること。

ただし、文字列中にダブルクォートが多く、エスケープが煩わしい場合は、シングルクォート(**'**)を用いてもよい。

* + 良い例

let prefix = "nablarch";

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

let $field = $('<input type="text" name="searchText" value="" />'); // ダブルクォートが多く含まれる場合は

// シングルクォートを使用してよい。

* + 悪い例

let prefix = 'nablarch'; // !! 通常はダブルクォートを使用しなければならない。

let message = "How to escape single quote(\') in javascript?"; // !! ダブルクォート中のシングルクォートのエスケープは不要

**(注)** JSONフォーマット仕様では、シングルクォートによる文字リテラルのクォートおよび、シングルクォートのエスケープは認められていない。

本規約では、無用な混乱を避けるため、リテラル表記の規約については可読性に問題が無い範囲でJSONフォーマットに合わせるようにしている。

### オブジェクトリテラルの記述 ★

キー文字列のクウォートにはダブルクォートを用いるが、キーが半角英数文字およびアンダースコアのみで構成されている場合は省略することができる。

また、キー名にダブルクォートが含まれ、エスケープが煩わしい場合は、シングルクォート(**'**)を用いてもよい。

* + 記述例

let obj = {

"key1": "val1",

"key2": "val2",

"key3": "val3"

};

let obj2 = {

key1: "val1", // キー文字列のクォートを省略

key2: "val2",

key3: "val3"

};

**(注)** JSONフォーマットでは、オブジェクトリテラルにおけるキー文字列は必ずダブルクォートでクォートしなければならいので注意すること。

### 正規表現リテラルの記述 ★

正規表現を使用する場合は、可能であれば、文字列引数ではなく、正規表現リテラルを使用すること。

ただし、正規表現の内容が動的に変わる場合や、正規表現の内容が複雑でコメントが必要となる場合は、RegExp()コンストラクタを使用すること。

# セキュリティに関する規約

## JavaScript一般

### ソースコードの動的評価(eval()/Functionコンストラクタ)を使用しない。 ★

これらの関数の引数に外部から操作できる文字列を連結している場合は、ブラウザ上で任意のコードを実行できる危険性があるため、原則使用しないこと。

(**利用可能API一覧**を参照)

※静的検査ツールで検査を行う。

**(注)** JavaScriptではクロージャが使用できるのでこれらの関数が必要となることはほぼ無い。

## jQuery関連

### 文字列からDOMノードを生成する関数の引数に変数を連結しない。 ★

jQueryの $()コンストラクタや、$.append() などのメソッドでは、引数にhtmlを表す文字列を渡すことで、そのhtmlの内容に相当するDOMノードを生成することができる。

しかし、これらの文字列に外部から操作できる値を連結している場合は、変数にscript要素等を挿入することにより任意のスクリプトが実行でき、XSS脆弱性の直接の原因となる。

仕様上、そのような処理が必須となる場合は、当該部分について有識者による十分なレビューを受けること。

下記のソースコードはXSS脆弱性が存在する典型的な例である。

* + 悪い例

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

この record.message の内容が外部からの入力値であるときに、XSS攻撃が成立する可能性がある。

例えば、この変数の内容が以下の文字列であったとすると、任意のスクリプトの実行が可能となる。

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

この例であれば、以下のように修正することで脆弱性を回避することができる。

* + 良い例

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

# パフォーマンスに関する規約

## JavaScript一般

### for文開始前に走査対象となる配列の長さを取得しておく。 ★

Array や Arguments オブジェクトような配列型オブジェクトを操作する場合、

以下のようにループ毎に length プロパティにアクセスするとパフォーマンスが劣化する場合がある。

* + 悪い例

let array = getLongArray();

for (let i = 0; i < array.length; i++) { // !! 毎回lengthプロパティにアクセスしている。

let item = array[i];

// ...

}

次の例のように、lengthプロパティはfor文の初期化ブロックで取得したものを終端条件判定で使用すること。

* + 良い例

let array = getLongArray();

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

let item = array[i];

// ...

}

### ループ内での文字連結処理には Array.prototype.join() を使用する。 ★

下記の例のように、ループ内で文字列の連結を行なうような場合、**+** 演算子や String.prototype.concat()を使用すると、

一部のブラウザでは性能が極端に劣化する傾向がある。

* + 悪い例

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));

この様なケースでは、次の例のように、配列に連結したい文字列をpush()しておき、最後にまとめてjoin()することで、

どのブラウザにおいても安定した性能が得られる。

* + 良い例

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("")));

// ...

}

### with句を使用しない。★

with句を使用した場合は、ローカル変数を格納したスコープの内側にスコープを挿入することになり、ローカル変数へのアクセス性能が劣化する。

このため、with句は使用しないこと。

### script要素はページの終端付近に配置する。

ブラウザは、初期表示処理の際、JavaScriptファイルのダウンロードおよびスクリプトの評価を行なっている間は後続の要素のレンダリング処理を停止する。

このため、script要素がhead要素やドキュメントの中間に挿入されていると画面描画の完了が遅れる。

この問題を回避するため、script要素の配置位置はドキュメントの末尾(body要素の終了タグ直前)に配置することを推奨する。

こうすることで、スクリプトファイルのダウンロードおよびスクリプトの評価は、画面のレンダリング処理と並行で行なわれるようになり、

体感的なパフォーマンスが向上する。

## jQuery関連

### セレクタの処理結果をキャッシュする。 ★

**$()**コンストラクタによるセレクタ処理は比較的負荷の高い処理であるので、セレクタの結果を複数回参照する場合は必ずローカル変数にキャッシュすること。

以下の例では、同じ内容のセレクタ処理を個別に呼びだしているため、非効率である。

* + 悪い例

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

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

// ...

});

この場合、次の例のように記述すること。

* + 良い例

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

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

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

// ...

});

### ブラウザが直接処理できるセレクタをセレクタ式の先頭に記述する。

**$()**によるセレクタ処理では、渡されたセレクタ式の内容を解析し、ブラウザのネイティブAPIで実行できる内容であれば、ブラウザに処理を移譲する。

特に、**$("#ID")** 、**$("TAG")** 、**$(".CLASSNAME")** はそれぞれ、ブラウザのgetElementById() / getElementsByTagName() / getElementsByClassName()で直接処理されるため、非常に高速である。

従って、この2つのセレクタを先頭に配置することで、セレクタの性能を向上させることができる。

# プログラムの堅牢性に関する規約

## JavaScript一般

### strictディレクティブをグローバルスコープで使用しない。

strictディレクティブを使用することで、対応するブラウザでは、以下のような問題のあるコードを明示的なエラーとして検出することができる。

1. varステートメントの付け忘れによる意図しないグローバル変数へのアクセス
2. withステートメントの使用
3. オブジェクトリテラルにおけるプロパティの重複定義
4. this変数を経由したグローバル変数へのアクセス(≒コンストラクタ関数呼び出し時のnewステートメントのつけ忘れ)

基本的にstrictディレクティブを使用することを推奨する。

ただし、下の例のようにグローバルスコープ上でstrictディレクティブを使用すると、それ以降に評価される全てのコードに対して適用されてしまい、

外部ライブラリが動作しなくなる可能性があるため禁止する。

* + 悪い例

“use strict”; // !!全スクリプトに対してstrictモードが適用されてしまう。

(function($) {

$(initialize);

function initialize() {

//…

}

})(jQuery)

正しくは、次の例のように、関数スコープ内で個別にstrictディレクティブを指定すること。

* + 良い例

(function($) {

“use strict”;

$(initialize);

function initialize() {

//…

}

})(jQuery)

### 暗黙的な型変換をともなう比較演算子 (==, !=) は使用しない。 ★

論理演算子 **==** と **!==** は両辺の型が異なる場合、暗黙的な型変換を行うため、プログラマが意図しない結果となることがある。

例えば、以下のステートメントは真と判定される。

※静的検査ツールで検査を行う。

* + 悪い例

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

(undefined == null) // -> true

原則としてこれらの演算子は使用せず、以下の例のように、型変換を行わない比較演算子(**===**, **!==** を使用する。)

例えば、以下のステートメントは真と判定される。

* + 良い例

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

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

### 例外を捕捉しない。(try-catch-finallyブロックを使用しない。)

ブラウザ内で発生したエラーを捕捉したとしても、サーバー側のアプリケーションのように 障害ログ出力や復旧処理ができるわけではないので、

エラーオブジェクトの補足は原則行わず、ブラウザのデフォルト動作に任せるものとする。

### オブジェクトのループにはObject.keysを利用する。★

for-in ループでは、プロトタイプチェイン上のプロパティも走査するため、本来、予期しない要素が処理対象となってしまう場合がある。

これを避けるため、**Object.keys()** を使ってループを記述すること。

* + 記述例1

let record = getRecord();

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

console.log(record[key]);

});

### グローバル変数の定義を行なわない。 ★

グローバル変数(windowオブジェクトのプロパティ) を新たに定義することを禁止する。

### プロトタイプの拡張を行わない。 ★

既存のコンストラクタ関数に対するプロトタイプオブジェクトの変更を原則として禁じる。

ただし、jQueryプラグインを作成する場合に限り、jQueryオブジェクトのプロトタイプ定義(jQuery.fn)の拡張を認める。

### JavaScriptをHTMLに直接記述する場合、「-」（ハイフン）を2つ以上連続して記述しない。★

JavaScriptをHTMLに直接記述する場合は、n:scriptカスタムタグを使用する。n:scriptカスタムタグは、JavaScriptに未対応のブラウザが画面上にJavaScriptコードを表示するのを防ぐため、JavaScriptコードをHTMLコメントとして出力する。

* + JSP実装例

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

// ここにJavaScriptを記述する。

</script>

* + HTLM出力例

<script type="text/javascript">

<!--

// ここにJavaScriptを記述する。

-->

</script>

HTMLの仕様では、HTMLコメント内に「-」を2つ以上連続して記載してはならないため、上記のJavaScriptに「-」が2つ以上連続して現れるとHTML構文エラーとなる。

* + 悪い例

let message = "--"; // 文字列で「-」が連続する。

let count = 10;

count--; // デクリメント演算子で「-」が連続する。

JavaScriptを外部ファイルに記述できないか検討する。外部ファイルに記述できない場合は、下記のとおり実装して対応する。

* + 良い例

let message = "\-\-"; // エスケープする。

let count = 10;

count -= 1; // デクリメント演算子を使用しない。

## jQuery関連

### DOM関連APIを直接使用することを避ける。

DOM関連APIは、ブラウザごとにインターフェースの差異が存在するため、直接これらのAPIを使用せず、jQueryの同等の関数を通じて使用すること。

(**利用可能API一覧**を参照)

### :enabled 擬似セレクタを使用しない。 ★

:enabled擬似セレクタにマッチする要素は、「disabled属性が設定されていない、もしくはfalseである入力項目のうち、**type=”hidden”でないもの**。」

と仕様で定められており、disabled属性値の値に関係なく、hidden要素は :enabledにマッチしない。

例えば、次のようなinput要素、<input type=”hidden” name=”name1” value=”val1”/> は :enabled と :disabled の**どちらにもマッチしない**。

この仕様は直感と乖離しており、バグの温床となるため、使用を禁ずる。

* + 悪い例

let $inputs = $("input, select, textarea").filter(":enabled"); // hidden項目が含まれない。

回避策としては、以下の例のように **.not(":disabled")** で代用する。

* + 良い例

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