Google JavaScript Style Guide 和訳

この和訳について

この文章は Google JavaScript Style Guide を非公式に和訳したものです. 内容の正確性は保証しません. ライセンスは原文と同じく CC-By 3.0 とします. フィードバックは Issue への登録 , あるいは Kosei Moriyama (@cou929 または cou929 at gmail.com) へ直接お願いします. この和訳のリポジトリは こちら です.

バージョン

Revision 2.93

著者

- Aaron Whyte
- Bob Jervis
- Dan Pupius
- Erik Arvidsson
- Fritz Schneider
- Robby Walker

背景

JavaScript は Google の オープンソースプロジェクトにおいて使われる, 主要なクライアントサイドスクリプト言語です. このスタイルガイドでは JavaScript プログラムにおいて すべき こと, すべきでない ことをまとめています.

JavaScript Language Rules

var

変数宣言には常に var を付ける.

宣言時に var を付けなかった場合, その変数はグローバルコンテキストに置かれます. このことにより, 既存の変数が汚染される可能性があります. また宣言がない場合は, その変数がどのスコープなのかが分かりづらくなります. よって常に var を付けるべきです.

定数

- 定数 (constant values) は `NAMES_LIKE_THIS` のように名づけます.
- `@const` を定数ポインタ (constant pointers, overwrite できない変数やプロパティ) に使います.

- Internet Explorer でサポートされていないので `const` キーワードは使うべきではありません.

定数 (Constant values)

一定で不変であることを意図した値には `CONSTANT_VALUE_CASE` のように定数としての名前をつけます. 加えて, すべてを大文字にし単語をアンダースコアで区切るようにすることは `@const` (その値は overwrite できない) という意味にもなります..

プリミティブ型 (`number`, `string`, `boolean`) は定数です.

オブジェクトの不変性はより主観的です. オブジェクトは観測できる状態変化がない場合のみ不変であるとみなされます. これはコンパイラによって強制されません.

定数ポインタ (Constant pointers, 変数とプロパティ)

変数やプロパティへの `@const` アノテーションはそれが overwrite できないことを示します. これはビルド時にコンパイラによって強制されます. この挙動は const キーワードと一貫性があるものです. (ただし Internet Explorer がサポートしていないので使用していません.)

メソッドへの `@const` アノテーションはサブクラスからオーバーライドできないということも示します.

コンストラクタの `@const` アノテーションはサブクラスを作成できないということを示します. (Java の `final` と類似です.)

例

`CONSTANT_VALUE_CASE` という名前にした場合 `@const` は必須ではありません. `CONSTANT_VALUE_CONSTANT` という名前を付けることで `@const` であるという意味にもなります.

```
/**
 * Request timeout in milliseconds.
 * @type {number}
 */
goog.example.TIMEOUT_IN_MILLISECONDS = 60;
```

`TIMEOUT_IN_MILLISECONDS` は定数なので変更されることはありません. `ALL_CAPS` という記法なので `@const` であることにもなります. よって overwrite もされません.

`@const` がついていないので, オープンソースのコンパイラの中には overwrite を許すものもあります.

```
/**
 * Map of URL to response string.
 * @const
 */
MyClass.fetchedUrlCache_ = new goog.structs.Map();
```

```
/**
  * サブクラスを作成できないクラス
```

```
  * @const
  * @constructor
*/
sloth.MyFinalClass = function() {};
```

このケースでは, overwrite をすることはできませんが, 値は定数でなく変更可能です. (ALL_CAPS ではなくキャメルケースなため)

セミコロン

常にセミコロンを使います.

コードのセミコロンを省き, セミコロンの挿入を処理系に任せた場合, 非常にデバッグが困難な問題が起こります. 決してセミコロンを省くべきではありません.

以下のコードで, セミコロンの省略が非常に危険である例を示します.

```
// 1.
MyClass.prototype.myMethod = function() {
  return 42;
}  // ここにセミコロンがない

(function() {
  // この一時的なブロックスコープで初期化処理などを行う
})();


var x = {
  'i': 1,
  'j': 2
}  // セミコロンがない

// 2. Internet Explorer や FireFox のために以下のようなコードを書く
// 普通はこんな書き方はしないけど, 例なので
[ffVersion, ieVersion][isIE]();


var THINGS_TO_EAT = [apples, oysters, sprayOnCheese]  // セミコロンがない

// 3. bash 風な条件文
-1 == resultOfOperation() || die();
```

何が起こるか

1. JavaScript Error: はじめの 42 を返している無名関数が, 2つ目の関数を引数にとって実行されてしまい, 42 を関数として呼び出そうとしてエラーになる.
2. おそらく実行時に‘no such property in undefined’エラーとなる. x[ffVersion, ieVersion][isIE]() と解釈されてしまうため.
3. 常に `die()` が呼び出される. なぜなら `配列 - 1` は常に `NaN` で, (`resultOfOperation()` が `NaN` を返したとしても) イコールになることがないため. そして `THINGS_TO_EAT` に `die()` の結果が代入される.

なぜ

JavaScript は, 安全にセミコロンの存在が推測できる場合を除いて, 文の最後にセミコロンを要求します. 上記の例では関数宣言やオブジェクトや配列リテラルが文の中にあります. 閉じ括弧は文の終わりを表現するものではありません. 次のトークンが()演算子などの場合, JavaScript はそれを前の文の続きとみなしてしまいます.

これらの挙動は本当にプログラマを驚かせてしまいます. よってセミコロンを徹底すべきです.

解説: セミコロンと関数

セミコロンは関数式の後ろには必須ですが, 関数宣言には不要です. 以下の例で違いを示します:

```
var foo = function() {
  return true;
};  // semicolon here.

function foo() {
  return true;
}  // no semicolon here.
```

ネストした関数

使っても良い.

ネストした関数は非常に便利です. 例えば, continuation を作り, ヘルパー関数を隠蔽する場合などです. 自由にネストした関数を使ってください.

ブロックの中での関数宣言

してはいけない.

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

ブロック内での関数宣言は多くのスクリプトエンジンでサポートされていますが, これは ECMAScript で標準化されていません (ECMA-262 の 13, 14 節を参照してください). よって各実装や将来の ECMAScript 標準との間での一貫性がとれなくなります. ECMAScript での関数宣言は, スクリプトのルート部分か関数内で許可されています. ブロック内では関数宣言の代わりに関数式を用いてください:

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

例外

使っても良い.

何か通常でないこと (例えばフレームワークを使う場合など) をするときには, 基本的に例外は

避けられません. よって使うべきです.

カスタム例外

使っても良い.

例外を独自に定義しない場合, エラー時の関数の戻り値を工夫せねばならず, エレガントではありません. 以下のような方法がありますが推奨されません.

- エラー情報へのリファレンスを返す
- エラーメンバーを含むオブジェクトを返す

これらは (JavaScript の) 例外処理機構を破壊することと同義です. よって適切な場面では独自の例外を使用すべきです.

標準機能

常に標準の機能を使うべきです.

ポータビリティとコンパチビリティを最大化するために, 常に非標準の機能よりも標準の機能を使うべきです (例えば string[3] ではなく string.charAt(3) を使ったり, アプリケーション特有の省略記法ではなく DOM 関数を使うなど).

プリミティブ型のラッパーオブジェクト

使用すべきでない.

プリミティブな型のラッパーオブジェクトを使う理由はありません. またそれは危険です.

```
var x = new Boolean(false);
if (x) {
  alert('hi');  // 'hi' がアラートされる.
}
```

絶対にやらないでください!

しかし型キャストは問題ありません.

```
var x = Boolean(0);
if (x) {
  alert('hi');  // これはアラートされません
}
typeof Boolean(0) == 'boolean';
typeof new Boolean(0) == 'object';
```

この方法はデータを `number`, `string`, `boolean` にキャストする際に便利です.

多層のプロトタイプヒエラルキー

好ましくありません.

多層のプロトタイプヒエラルキー(Multi-level prototype hierarchies) は JavaScript が継承を実装している方法です. ユーザー定義クラスDがプロトタイプとしてユーザー定義クラスBを持っている場合, 多層のヒエラルキーになります. こうしたヒエラルキーは理解するのが難しくなります.

よって同様のことを実現したい場合は, Closure Library の `goog.inherits()` や類似のライブラリ関数を使うべきです.

```
function D() {
  goog.base(this)
}
goog.inherits(D, B);

D.prototype.method = function() {
  ...
};
```

メソッドとプロパティの定義

```
/** @constructor */
function SomeConstructor() {
  this.someProperty = 1;
}
Foo.prototype.someMethod = function() { ... };
```

メソッドやプロパティを `new` を使ってオブジェクトに付与する方法はいくつかありますが, 推奨されるのは次の方法です:

```
Foo.prototype.bar = function() {
  /* ... */
};
```

またコンストラクタの中でフィールドを初期化するのも推奨されます.

```
/** @constructor */
function Foo() {
  this.bar = value;
}
```

なぜ

現在の JavaScript エンジンはオブジェクトの“形状”に応じて最適化を行います. オブジェクトにプロパティを追加すること (プロトタイプで値をオーバーライドすることも含む) はパフォーマンスの低下につながります.

delete

`this.foo = null` というスタイルが推奨されます.

```
Foo.prototype.dispose = function() {
  this.property_ = null;
};
```

以下のような書き方の代わりに `null` を代入する方式にしてください:

```
Foo.prototype.dispose = function() {
  delete this.property_;
};
```

近年の JavaScript エンジンではオブジェクトのプロパティ数の変更は値の再代入よりもパフォーマンスが低下します. delete キーワードは本当に必要な場合, オブジェクトの keys のリストからそのプロパティを削除したい場合や `if (key in obj)` の結果を変えたい場合など, 以外には使用しないでください.

クロージャ

使っても良い. ただし慎重に.

クロージャは JavaScript の中でも最も便利でよく見る機能です. クロージャについて詳しくはこちらを参照してください.

ただし一点注意すべき点は, クロージャはその閉じたスコープへのポインタを保持しているという点です. そのため, クロージャを DOM 要素に付加すると循環参照が発生する可能性があり, メモリリークの原因となります. 例えば, 以下のコードを見てください:

```
function foo(element, a, b) {
  element.onclick = function() { /* 引数 a と b を使う */ };
}
```

上記の無名関数はそれらを使う・使わないにかかわらず `element`, `a`, `b` への参照をずっと保持しています. `element` はクロージャへの参照を持っているので, 循環が発生していて, gc が回収できなくなっています. この場合, 循環参照が発生しないように以下のような構造にすべきです:

```
function foo(element, a, b) {
  element.onclick = bar(a, b);
}

function bar(a, b) {
  return function() { /* 引数 a と b を使う */ };
}
```

eval()

Code loader か REPL (Read-eval-print loop) を実装するときのみ使用可.

`eval()` はセマンティクスを混乱させやすいし, ユーザーインプットを `eval()` したものは危険です. 通常はもっとクリアで安全な代替手段が存在するので, 大抵の場合には `eval()` は使用すべきではありません. JSON RPC の場合には, `eval()` ではなく `JSON.parse()` で結果を読み取ることができます.

例えば, このようなものを返すサーバがあったとします:

```
{
  name: 'Alice',
```

```
  id: 31502,
  email: 'looking_glass@example.com'
},
```

```
var userInfo = eval(feed);
var email = userInfo['email'];
```

もし `feed` に悪意のあるコードが挿入されており, `eval()` を使った場合は, そのコードが実行されてしまします.

```
var userInfo = JSON.parse(feed);
var email = userInfo['email'];
```

`JSON.parse()` を使うと, (実行可能な JavaScript コードも含め) 不正な JSON の場合は例外が投げられます.

with() {}

使用すべきでない.

`with` によってコードの意味が分かりにくくなります. `with` のオブジェクトはローカル変数と衝突するプロパティを持ちます. これによってプログラムの意味が大きく変わってしまいます. 例えば次のコードはどういう動作をするでしょう?

```
with (foo) {
  var x = 3;
  return x;
}
```

答え: anything. ローカル変数 `x` は `foo` プロパティによって上書きされます. もし `x` がセッターだったとき, 3 を代入することが沢山のコードを実行してしまいます. `with` を使ってはいけません.

this

オブジェクトのコンストラクタ, メソッド, クロージャのセットアップのときのみ使用可.

this の意味はトリッキーです. this はグローバルスコープを指していたり (多くの場合), 呼び出し元を指していたり (`eval`), DOMのノードを指していたり (イベントハンドラを HTML 要素にセットした場合), 新しく作られたオブジェクトを指していたり (コンストラクタ), なにか他のオブジェクトを指している場合 (call(), apply() された関数) もあります.

this の使用は間違えやすいので, 以下の場面以外での使用は制限されています.

- コンストラクタ内での使用
- オブジェクトのメソッド内での使用 (クロージャの作成を含む)

for-in ループ

オブジェクト, map, hash のキーに対してイテレーションする場合のみ使用可.

`for-in` ループは配列のすべての要素を走査する場合などによく誤って利用されています. これはインデックス `0` から `length - 1` までをループするわけではなく, 配列プロトタイプにあるすべてのキーについてループします. 以下は `for-in` ループでの配列の走査を失敗する例です.

```
function printArray(arr) {
  for (var key in arr) {
    print(arr[key]);
  }
}

printArray([0,1,2,3]);  // 正しく動作.

var a = new Array(10);
printArray(a);  // 正しく動かない.

a = document.getElementsByTagName('*');
printArray(a);  // 正しく動かない.

a = [0,1,2,3];
a.buhu = 'wine';
printArray(a);  // 正しく動かない.

a = new Array;
a[3] = 3;
printArray(a);  // 正しく動かない.
```

配列には通常の `for` ループを使用してください.

```
function printArray(arr) {
  var l = arr.length;
  for (var i = 0; i < l; i++) {
    print(arr[i]);
  }
}
```

連想配列

配列を map/hash/連想配列 として使用してはいけません.

連想配列は許可されていません, つまり配列に数値以外のインデックスを使用してはいけません. map/hash を使いたいときは配列でなくオブジェクトを使用してください. なぜならこのような機能はもともと配列ではなくオブジェクトの機能です. 配列はオブジェクトを拡張したものです (そして他の JavaScript のオブジェクト, Data, RegExp, String なども同様です).

複数行の string リテラル

以下のような複数行の文字列は使用してはいけません.

```
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.';
```

各行の先頭の空白はコンパイラによって安全に取り除かれますが, スラッシュの後の空白によってトリッキーなエラーが発生する可能性があります. また多くの JavaScript エンジンはこの記法をサポートしていますが, これは ECMAScript 標準ではありません.

このような場合は, 次のように文字列を結合させます.

```
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.';
```

> Note: 訳注
>
> バックスラッシュによる複数行の string リテラルは ECMAScript 3 では非標準だったのですが, ECMAScript 5 では標準化されたようです.
>
> http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

配列・オブジェクトリテラル

使用して良い.

`Array`, `Object` コンストラクタではなくリテラルを使ってください.

`Array` コンストラクタはその引数の取り方のせいでエラーを引き起こしがちです.

```
// 長さは 3.
var a1 = new Array(x1, x2, x3);

// 長さは 2.
var a2 = new Array(x1, x2);

// もし x1 が数字で, かつ自然数の場合, length は x1 になる.
// もし x1 が数字で, かつ自然数でない場合, 例外が発生する.
// 数字でない場合, 配列は x1 を値として持つ.
var a3 = new Array(x1);

// 長さは 0.
var a4 = new Array();
```

コンストラクタはこのような動作をするので, もし別の人がコードを書き換えて, コンストラクタに2つの引数を与えていたところを1つにすると, その結果できた配列は期待する長さを持っていないかもしれません.

このようなミスを避けるために, 配列のリテラルを使用してください.

```
var a = [x1, x2, x3];
var a2 = [x1, x2];
var a3 = [x1];
var a4 = [];
```

オブジェクトの場合は, コンストラクタに配列のような紛らわしさはないのですが, 可読性と一貫性のためにリテラルを使用してください.

```
var o = new Object();

var o2 = new Object();
o2.a = 0;
o2.b = 1;
o2.c = 2;
o2['strange key'] = 3;
```

上記のようなコードは, 以下のように書くべきです.

```
var o = {};

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

ビルトインオブジェクトのプロトタイプの書き換え

してはいけません.

`Object.prototype` や `Array.prototype` などのビルトインオブジェクトのプロトタイプを変更することは厳密に禁じられています. `Function.prototype` などはそれに比べ比較的安全ですが, デバッグ時に問題を引き起こす可能性があるので, 変更は避けてください.

Internet Explorer の条件付きコメント

使ってはいけない.

次のように書かないでください.

```
var f = function () {
  /*@cc_on if (@_jscript) { return 2* @*/  3; /*@ } @*/
};
```

条件付きコメントはランタイムに JavaScript のシンタックスツリーを変更するので, 自動化されたツールの動作を妨げてしまいます.

JavaScript Style Rules

命名

基本的に次のように命名してください:

- `functionNamesLikeThis`

- `variableNamesLikeThis`
- `ClassNamesLikeThis`
- `EnumNamesLikeThis`
- `methodNamesLikeThis`
- `CONSTANT_VALUES_LIKE_THIS`
- `foo.namespaceNamesLikeThis.bar`
- `filenameslikethis.js`

プロパティとメソッド

- `Private` のプロパティ, メソッドには, 末尾にアンダースコア `_` を付けてください.
- `Protected` のプロパティ, メソッドにはアンダースコアを付けないでください (パブリックなものと同様です).

`Private` と `Protected` に関しては visibility のセクションを参考にしてください.

メソッドと関数パラメータ

オプション引数には `opt_` というプレフィックスをつけてください.

可変長の引数をとる場合, 最後の引数を `var_args` と名づけてください. ただし参照する際は `var_args` ではなく `arguments` を参照するようにしてください.

オプション引数と可変長引数に関しては `@param` アノテーションでもコンパイラは正しく解釈してくれます. 両方を同時に用いることが好ましいです.

getter と setter

ECMAScript 5 ではプロパティへの getter/setter の使用が推奨されていません. やむを得なく使用する場合は, 観測できる状態を変更しないようにする必要があります.

```
/**
 * 間違い -- このようにはしないでください
 */
var foo = { get next() { return this.nextId++; } };
};
```

アクセサ関数

getter, setter は必須ではありません. もし使う場合は `getFoo()`, `setFoo(value)` という名前にしてください. (boolean の getter の場合は `isFoo()` も許可されています. こちら のほうがより自然です.)

名前空間

JavaScript は階層的なパッケージングや名前空間をサポートしていません.

グローバル名前衝突が起こるとデバッグは難しくなり, 2つのプロジェクトの統合も難しくなります. 名前の衝突を避け, 共有できる JavaScript コードをモジュール化するために, 以下のような規約を設けています.

グローバルなコードには名前空間を使う

グローバルスコープに出すものには, プロジェクトやライブラリ名に関連したプレフィックスを常に付けてください. 例えば “Project Sloth” の場合, `sloth.*` という具合です.

```
var sloth = {};

sloth.sleep = function() {
  ...
};
```

Closure Library や Dojo toolkit でも名前空間を定義する関数が提供されています. これらを使う場合は一貫性に注意してください.

```
goog.provide('sloth');

sloth.sleep = function() {
  ...
};
```

名前空間のオーナーシップへの配慮

子の名前空間を作る場合は, 親の名前空間への連絡をしてください. sloth から hats というプロジェクトを始めた場合は, sloth チームに `sloth.hats` という名前を使用する旨を伝えてください.

外部のコードと内部のコードで別の名前空間を使う

“外部のコード (External code)” とはあなたのコードの外から読み込んだもので, 独立してコンパイルされたものです. 内部と外部のコードの名前空間は厳密に分けてください. もし `foo.hats.*` という外部ライブラリを使用した場合, 衝突の可能性があるので, 内部のコードでは `foo.hats.*` に何も定義してはいけません.

```
foo.require('foo.hats');

/**
 * 間違い -- 絶対にこのようにはしないでください.
 * @constructor
 * @extends {foo.hats.RoundHat}
 */
foo.hats.BowlerHat = function() {
};
```

もし外部名前変数に新しい API を定義する必要がある場合は, 明示的に公開 API をエクスポートする必要があります. 一貫性とコンパイラの最適化のために, 内部のコードでは内部の API を内部の名前で呼ぶ必要があります.

```
foo.provide('googleyhats.BowlerHat');
```

```
foo.require('foo.hats');

/**
 * @constructor
 * @extend {foo.hats.RoundHat}
 */
googleyhats.BowlerHat = function() {
  ...
};

goog.exportSymbol('foo.hats.BowlerHat', googleyhats.BowlerHat);
```

長い型名をエイリアスし可読性を向上させる

ローカルのエイリアスを使うことで長い型名の可読性を向上できる場合はそうしてください. エイリアスの名前は型名の最後の部分にしてください.

```
/**
 * @constructor
 */
some.long.namespace.MyClass = function() {
};

/**
 * @param {some.long.namespace.MyClass} a
 */
some.long.namespace.MyClass.staticHelper = function(a) {
  ...
};

myapp.main = function() {
  var MyClass = some.long.namespace.MyClass;
  var staticHelper = some.long.namespace.MyClass.staticHelper;
  staticHelper(new MyClass());
};
```

名前空間のローカルなエイリアスは作成しないでください. `goog.scope` によってのみ, 名前空間のエイリアスを作成すべきです.

```
myapp.main = function() {
  var namespace = some.long.namespace;
  namespace.MyClass.staticHelper(new namespace.MyClass());
};
```

エイリアスした型のプロパティにはアクセスしないでください. ただし列挙型は除きます.

```
// 訳注: エイリアスからのプロパティアクセスが許可される例 (enumであるため)
/** @enum {string} */
some.long.namespace.Fruit = {
  APPLE: 'a',
  BANANA: 'b'
};

myapp.main = function() {
  var Fruit = some.long.namespace.Fruit;
  switch (fruit) {
    case Fruit.APPLE:
      ...
    case Fruit.BANANA:
```

```
      ...
  }
};
```

```
myapp.main = function() {
  var MyClass = some.long.namespace.MyClass;
  MyClass.staticHelper(null);
};
```

グローバルスコープではエイリアスを使用しないでください. エイリアスは関数スコープの中でのみ使用可能です.

ファイル名

ファイル名は case-sensitive なプラットフォームのために, 必ず小文字にしてください. サフィックスは `.js` に, 句読点は `-`, `_` (`_` よりも `-` を使用してください) 以外は使わないでください.

カスタム toString() メソッド

副作用なしに, 必ず動作しないといけません.

`toString()` メソッドを定義して, 独自のオブジェクトがどのように文字列化されるかを定義できます. ただし以下の2点が必ず守られる必要があります.

1. 必ず成功する
2. 副作用がない

これらが守られなかった場合, 簡単に問題が引き起こされてしまいます. 例えば `toString()` が `assert` を呼び出している場合, `assert` はオブジェクト名をアウトプットしようとするので, `toString()` が必要になります.

初期化の延期

しても良い.

必ずしも宣言時に変数の初期化ができるわけではないので, 初期化を延期することは認められています.

明示的なスコープ

常に必要です.

常に明示的なスコープを使用してください. ポータビリティが向上し, またクリアになります. 例えば `window` が content window でないアプリケーションもあるので, `window` に依存するようなコードは書かないでください.

コードのフォーマット

基本的に C++ formatting rules に従います. 以下はそれに追加する項目です.

波括弧

処理系によってセミコロンが暗黙で挿入されるのを防ぐために, 必ず開き波括弧は改行せずに同じ行に書いてください.

```
if (something) {
  // ...
} else {
  // ...
}
```

配列・オブジェクトの初期化

一行に収まる場合は, 初期化を一行で行ってもかまいません.

```
var arr = [1, 2, 3];  // 括弧の前後に空白を入れないでください
var obj = {a: 1, b: 2, c: 3};  // 括弧の前後に空白を入れないでください
```

複数行に渡る初期化の場合は, 普通のブロック同様スペース2つのインデントを行い, かつ括弧だけで一行を使ってください.

```
// オブジェクトの初期化
var inset = {
  top: 10,
  right: 20,
  bottom: 15,
  left: 12
};

// 配列の初期化
this.rows_ = [
  '*Slartibartfast* <fjordmaster@magrathea.com>',
  '*Zaphod Beeblebrox* <theprez@universe.gov>',
  '*Ford Prefect* <ford@theguide.com>',
  '*Arthur Dent* <has.no.tea@gmail.com>',
  '*Marvin the Paranoid Android* <marv@googlemail.com>',
  'the.mice@magrathea.com'
];

// メソッドの引数としてのオブジェクト
goog.dom.createDom(goog.dom.TagName.DIV, {
  id: 'foo',
  className: 'some-css-class',
  style: 'display:none'
}, 'Hello, world!');
```

identifer が長い場合, プロパティを整列させると問題を引き起こす場合があるので, 整列させないようにしてください.

```
CORRECT_Object.prototype = {
  a: 0,
  b: 1,
  lengthyName: 2
};
```

以下のようにはしないでください.

```
WRONG_Object.prototype = {
  a           : 0,
  b           : 1,
  lengthyName: 2
};
```

関数の引数

可能ならば, すべての関数の引数は一行にしてください. もしそれでは80文字の制限を超えてしまう場合は, 読みやすい形で複数行にしてください. スペースの節約のために各行をできるだけ80文字に近づけるように書くか, あるいは可読性のためにひとつの引数に付き一行を割り当てます. インデントは空白4つにするか, 括弧にあわせてください. 以下に典型的な例を示します.

```
// 空白4つのインデント, 80文字近くまでならべる. とても長い関数名で, スペースが少ない場合.
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
  // ...
};

// 空白4つのインデント, 1引数につき1行. とても長い関数名で各引数を強調したい場合
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne,
    veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy,
    artichokeDescriptorAdapterIterator) {
  // ...
};

// 括弧にあわせたインデント, 80文字近くまでならべる. 引数を見やすくまとめて, スペースが少ない場合
function foo(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
             tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
  // ...
}

// 括弧にあわせたインデント, 1引数につき1行. 各引数を強調したい場合.
function bar(veryDescriptiveArgumentNumberOne,
             veryDescriptiveArgumentTwo,
             tableModelEventHandlerProxy,
             artichokeDescriptorAdapterIterator) {
  // ...
}
```

関数呼び出しそのものがインデントされている場合は, オリジナルの文のはじめからスペース4つ分のインデントをあけ引数を記述, 関数呼び出しのはじめからスペース4つ分のインデントをあけ引数を記述, のどちらでもかまいません. 以下はすべて正しいインデント方法です.

```
if (veryLongFunctionNameA(
        veryLongArgumentName) ||
    veryLongFunctionNameB(
    veryLongArgumentName)) {
  veryLongFunctionNameC(veryLongFunctionNameD(
      veryLongFunctioNameE(
          veryLongFunctionNameF)));
}
```

無名関数を渡す場合

関数の引数として無名関数を定義し渡すとき, 無名関数の中身はその分の左端からスペース2つか, あるいは `function` キーワードの左端からスペース2つのインデントを入れます. これは引数の無名関数の可読性を高めるためのルールです (例えばコードが右側に寄りすぎてしまうのを防ぎます).

```
prefix.something.reallyLongFunctionName('whatever', function(a1, a2) {
  if (a1.equals(a2)) {
    someOtherLongFunctionName(a1);
  } else {
    andNowForSomethingCompletelyDifferent(a2.parrot);
  }
});

var names = prefix.something.myExcellentMapFunction(
    verboselyNamedCollectionOfItems,
    function(item) {
      return item.name;
    });
```

goog.scope を用いたエイリアス

Closure Library を使用している場合, goog.scope で名前空間分けされたシンボルへの参照を短くすることができます.

ファイルごとの `goog.scope` の呼び出しは 1 回までです. またそれをグローバルスコープで行う必要があります.

`goog.scope(function() {` という開始行の後に続くのは一行の空行と `goog.provide`, `goog.require` またはトップレベルコメントである必要があります. `goog.scope` 呼び出しの終了はファイルの末尾にしてください. スコープを閉じたところに `// goog.scop` というコメントを追加してください. このコメントはセミコロンから 2 スペースあけて追加します.

C++ と同じように `goog.scop` の定義の中ではインデントする必要はありません. 0 行目から書き始めてください.

他のオブジェクトを再代入されないもの (多くのコンストラクタ, enum, 名前空間など) のみ名前をエイリアスしてください. 次のようにはしないでください:

```
goog.scope(function() {
var Button = goog.ui.Button;

Button = function() { ... };
...
```

エイリアス名はその対象のグローバルでの最後のプロパティ名と同じにしてください

```
goog.provide('my.module');

goog.require('goog.dom');
goog.require('goog.ui.Button');

goog.scope(function() {
```

```
var Button = goog.ui.Button;
var dom = goog.dom;
var module = my.module;

module.button = new Button(dom.$('my-button'));
...
});  // goog.scope
```

More Indentation

配列リテラル・オブジェクトリテラルと無名関数以外は，直前の兄弟の式の左端にあわせるか，親の式よりもスペース4つ (2つではない) 深いインデントにします．(ここで言う兄弟・親とは括弧のネストのレベルです．)

```
someWonderfulHtml = '' +
                    getEvenMoreHtml(someReallyInterestingValues, moreValues,
                                    evenMoreParams, 'a duck', true, 72,
                                    slightlyMoreMonkeys(0xfff)) +
                    '';

thisIsAVeryLongVariableName =
    hereIsAnEvenLongerOtherFunctionNameThatWillNotFitOnPrevLine();

thisIsAVeryLongVariableName = siblingOne + siblingTwo + siblingThree +
    siblingFour + siblingFive + siblingSix + siblingSeven +
    moreSiblingExpressions + allAtTheSameIndentationLevel;

thisIsAVeryLongVariableName = operandOne + operandTwo + operandThree +
    operandFour + operandFive * (
        aNestedChildExpression + shouldBeIndentedMore);

someValue = this.foo(
    shortArg,
    '非常に長い文字列型の引数 - 実際にはこのようなケースはとてもよくあります.',
    shorty2,
    this.bar());

if (searchableCollection(allYourStuff).contains(theStuffYouWant) &&
    !ambientNotification.isActive() && (client.isAmbientSupported() ||
                                        client.alwaysTryAmbientAnyways())) {
  ambientNotification.activate();
}
```

空白行

論理的に関連のある行をまとめるために空白行を使用してください．

```
doSomethingTo(x);
doSomethingElseTo(x);
andThen(x);

nowDoSomethingWith(y);

andNowWith(z);
```

2項・3項演算子

演算子は常に先行する行においてください. 改行とインデントは他の Google Style Guide と同様の規約に従ってください. 当初このルールは, セミコロンの自動挿入を考慮して定められていました. 実際には二項演算子の前にセミコロンは自動挿入されません. しかし過去のコードとの一貫性のため, 新しいコードでもこのルールに従ってください.

```
var x = a ? b : c;  // 可能ならば1行に

// 空白4つのインデント
var y = a ?
    longButSimpleOperandB : longButSimpleOperandC;

// 最初のオペランドにあわせたインデント
var z = a ?
        moreComplicatedB :
        moreComplicatedC;
```

ドット演算子の場合の例.

```
var x = foo.bar().
    doSomething().
    doSomethingElse();
```

丸括弧

必要なところだけで使います.

構文上・意味上不可欠な場面以外では, 丸括弧を使わないようにします.

単項演算子 (`delete`, `typeof`) や `void` に丸括弧を使用してはいけません. また `return` や `throw`, `case`, `new` などのあとにも付けません.

文字列

`"` よりも `'` を使ってください.

ダブルクオートよりもシングルクオートを使ってください. そのほうが HTML を含む文字列を作る際に便利です.

```
var msg = 'なんらかの HTML';
```

Visibility (private, protected 領域)

JSDoc の `@private`, `@protected` アノテーションが推奨されます.

クラス, 関数, プロパティの visibility レベルの指定に, JSDoc の `@private`, `@protected` アノテーションを使うことが推奨されます.

コンパイル時に `--jscomp_warning=visibility` フラグを付けることで, visibility の侵害があった場合コンパイラが警告を出してくれるようにできます. 詳しくは Closure Compiler Warnings を参照してください.

`@private` なグローバル変数と関数は同じファイルのコードからのみアクセスできます.

`@private` なコンストラクタは, 同じファイルの同じインスタンスのメンバーからアクセスできます. また `@private` コンストラクタは同じファイルのパブリックな静的プロパティと `instanceof` 演算子からアクセスできます.

グローバル変数・関数・コンストラクタは `@protected` にはなりません.

```
// File 1.
// AA_PrivateClass_ と AA_init_ はグローバルで同じファイルからなのでアクセスできる

/**
 * @private
 * @constructor
 */
AA_PrivateClass_ = function() {
};

/** @private */
function AA_init_() {
  return new AA_PrivateClass_();
}

AA_init_();
```

`@private` なプロパティは同じファイルのすべてのコードにアクセスできます. 加えて, そのプロパティがクラスに属していた場合, そのプロパティが含まれるクラスの静的メソッドとインスタンスメソッドにもアクセスできます. ただし, 別ファイルのサブクラスからアクセスしたり, オーバーライドすることはできません.

`@protected` なプロパティは同じファイルのすべてのコードにアクセスできます. 加えて, そのプロパティを含むクラスのサブクラスの, 静的メソッドとインスタンスメソッドにもアクセスできます.

ここで, これらのセマンティクスは C++ や Java のものとは異なっていることに注意してください. まずここでの private, protected 指定は同じファイルのすべてのコードにアクセス権を与えていて, C++ や Java のようにクラスの継承関係によってアクセス権が変化するものではありません. また private なプロパティはサブクラスからオーバーライドできないことも, C++ などと異なる点です.

```
// File 1.

/** @constructor */
AA_PublicClass = function() {
  /** @private */
  this.privateProp_ = 2;

  /** @protected */
  this.protectedProp = 4;
};

/** @private */
AA_PublicClass.staticPrivateProp_ = 1;

/** @protected */
AA_PublicClass.staticProtectedProp = 31;

/** @private */
```

```
AA_PublicClass.prototype.privateMethod_ = function() {};

/** @protected */
AA_PublicClass.prototype.protectedMethod = function() {};

// File 2.

/**
 * @return {number} The number of ducks we've arranged in a row (一列にならべるアヒルの数).
 */
AA_PublicClass.prototype.method = function() {
  // これら2つのプロパティへの合法的なアクセス
  return this.privateProp_ + AA_PublicClass.staticPrivateProp_;
};

// File 3.

/**
 * @constructor
 * @extends {AA_PublicClass}
 */
AA_SubClass = function() {
  // protected な静的プロパティへの合法的なアクセス
  AA_PublicClass.staticProtectedProp = this.method();
};
goog.inherits(AA_SubClass, AA_PublicClass);

/**
 * @return {number} The number of ducks we've arranged in a row (一列にならべるアヒルの数).
 */
AA_SubClass.prototype.method = function() {
  // protected なインスタンスプロパティへの合法的なアクセス
  return this.protectedProp;
};
```

注意点として, JavaScript には (例えば `AA_PrivateClass_` のような) 型と, 型のコンストラクタとの間に区別がありません. public な型と private なコンストラクタを説明する方法がありません (なぜならば privacy check を行っても簡単にコンストラクタが呼び出せてしまうためです).

JavaScript の型

コンパイラによって強制されます.

JSDoc で型についてドキュメント化するときはできるだけ型を特定し正確にしてください. サポートしているのは EcmaScript4 です.

JavaScript 型指定言語

ES4 のプロポーサルには JavaScript の型を指定するための言語が記述されています. この言語を使って JSDoc のドキュメントに関数パラメータや返り値の型を記述します.

ES4 のプロポーサルの発展によって, 記法にも変化がありました. コンパイラは古い記法をサポートしていますがそれらは非推奨です.

> Note: 訳注
>
> 省略しました. 詳しくは原文にある表を参照してください. 後日補完します.

http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml?showone=JavaScript_Types#JavaScript_Types

JavaScript の型

Note: 訳注

省略しました. 詳しくは原文にある表を参照してください. 後日補完します.

http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml?showone=JavaScript_Types#JavaScript_Types

型キャスト

ある文の型を正確に推論できない場合, 型キャストのコメントを付加して括弧でくくり付加することができます. 括弧は必ず必要です. コメントと共に括弧でくくります.

```
/** @type {number} */ (x)
(/** @type {number} */ x)
```

nullable vs オプション パラメータとプロパティ

JavaScript は弱い型付けの言語なので, 関数の引数やクラスのプロパティの オプション引数, nullable (ヌルを取りえる), undefine の3つの違いについて知る必要があります.

オブジェクトの型 (あるいは参照型) はデフォルトで nullable です. しかし関数の型はデフォルトで nullable ではありません. オブジェクトは文字列, 数字, 真偽値, undefine 以外のものか null として定義されます. 例として以下のコードを示します.

```
/**
 * コンストラクタの引数 value で初期化されるクラス.
 * @param {Object} value Some value.
 * @constructor
 */
function MyClass(value) {
  /**
   * なんらかの値.
   * @type {Object}
   * @private
   */
  this.myValue_ = value;
}
```

このコードではコンパイラに `myValue_` プロパティはオブジェクトか null をとるように指定しています. もし `myValue_` が null を取りえなくする場合は次のようにします.

```
/**
 * コンストラクタの引数 value (なんらかの null でない値) で初期化されるクラス.
 * @param {!Object} value Some value.
 * @constructor
```

```
 */
function MyClass(value) {
  /**
   * なんらかの値.
   * @type {!Object}
   * @private
   */
  this.myValue_ = value;
}
```

この場合, もし `myClass` が null で初期化されたとき, コンパイラがワーニングを出します.

関数のオプションパラメータは実行時に undefined になり得ます. よってそれらがクラスのプロパティとして使われる場合は, 以下のように定義する必要があります.

```
/**
 * コンストラクタの引数 value (オプション) で初期化されるクラス.
 * @param {Object=} opt_value Some value (optional).
 * @constructor
 */
function MyClass(opt_value) {
  /**
   * なんらかの値.
   * @type {Object|undefined}
   * @private
   */
  this.myValue_ = opt_value;
}
```

この場合 `myValue_` はオブジェクト, null, undefined を取り得ます.

ここで `opt_value` は `{Object|undefined}` ではなく `{Object=}` と定義されていることに注意してください. これはオプションのパラメータは定義上そもそも undefined になりえるためです. 可読性のためわざわざ undefined を取りうることを明示する必要はありません.

最後に, nullable と オプション引数 の指定は直行しています. よって以下の4つの宣言はすべて別の意味です.

```
/**
 * 4つのうち2つは nullable, 2つはオプション
 * @param {!Object} nonNull Mandatory (must not be undefined), must not be null.
 * @param {Object} mayBeNull Mandatory (must not be undefined), may be null.
 * @param {!Object=} opt_nonNull Optional (may be undefined), but if present,
 *     must not be null!
 * @param {Object=} opt_mayBeNull Optional (may be undefined), may be null.
 */
function strangeButTrue(nonNull, mayBeNull, opt_nonNull, opt_mayBeNull) {
  // ...
};
```

Typedef

型が複雑になることもあります. 例えばある要素を引数としてとる関数はこのようになります:

```
/**
 * @Param {string} tagName
 * @param {(string|Element|Text|Array.<Element>|Array.<Text>)} contents
```

```
 * @return {Element}
 */
goog.createElement = function(tagName, contents) {
  ...
};
```

`@typedef` タグで型を定義することができます.

```
/** @typedef {(string|Element|Text|Array.<Element>|Array.<Text>)} */
goog.ElementContent;

/**
* @param {string} tagName
* @param {goog.ElementContent} contents
* @return {Element}
*/
goog.createElement = function(tagName, contents) {
...
};
```

テンプレート型

コンパイラはテンプレート型を不完全にしかサポートできていません. コンパイラは無名関数の中の `this` の型については, `this` 引数の型とそれの有無からしか推論できません.

```
/**
 * @param {function(this:T, ...)} fn
 * @param {T} thisObj
 * @param {...*} var_args
 * @template T
 */
goog.bind = function(fn, thisObj, var_args) {
...
};
// プロパティがないという警告を出すことができる例
goog.bind(function() { this.someProperty; }, new SomeClass());
// undefined this という警告を出す例
goog.bind(function() { this.someProperty; });
```

コメント

JSDoc を使用してください.

C++ style for comments に基本的に従います.

すべてのファイル, クラス, メソッド, プロパティを JSDoc コメントでドキュメンテーションしてください. その際適切なタグ, 型を使用してください. 名前から自明でない場合は, プロパティ, メソッド, 引数, 返り値の説明文を記載すべきです.

インラインコメントには `//` を使用してください.

文章が断片的になることは避けてください. 文頭では適切に語頭を大文字にし, 文末には句点を入れます. 完全な文章を書くことが推奨されますが, 必須ではありません. 完全な文章を書く際は大文字, 句読点を適切に使用してください.

コメントの構文

JSDoc の構文は JavaDoc をベースにしています. 多くのツールは JSDoc のコメントからメタ情報を抽出し, コードのバリデーションや最適化を行います. 次は正しいフォーマットのコメントの例です.

```
/**
 * JSDoc のコメントはスラッシュと 2 つのアスタリスクで始めます.
 * インラインタグは次のように波括弧で囲みます: {@code this}.
 * @desc のように, ブロックタグは常に新しい行から始めます.
 */
```

JSDoc のインデント

ブロックタグの内容が複数行になる場合, コードと同様に扱い, 空白 4 つ分のインデントにします.

```
/**
 * 説明文が長く, 複数行にまたがった場合の例.
 * @param {string} これはとても説明文の長い引数の例です. 複数行にまたがる場合は空白4つ分の
 *     インデントを入れてください.
 * @return {number} これはとても説明文の長い返り値の例です. 複数行にまたがる場合は空白4つ分の
 *     インデントを入れてください.
 */
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

`@fileoverview` のコメントはインデントしてはいけません. `@desc` コマンドをインデントする必要はありません.

文章の左端でそろえる方法も可能ですが, 推奨されません. 変数名が変わったときに毎回対応する必要が出てくるためです.

```
/**
 * これらは推奨されないインデントの例です.
 * @param {string} これはとても説明文の長い引数の例です. 複数行にまたがっていますが, 上の例のよう
 *                     4スペースのインデントではありません.
 * @return {number} これはとても説明文の長い返り値の例です. 複数行にまたがっていますが, 4つの空白
 *                  説明文の開始位置にあわせてインデントしています.
 */
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

JSDoc での HTML

JavaDoc のように JSDoc でも `<code>`, `<pre>`, `<tt>`, `<strong>`, `<ul>`, `<ol>`, `<li>`, `<a>` などの HTML タグがサポートされています.

よってプレインテキスト上のフォーマットは考慮されなくなります. JSDoc では空白に頼ったフォーマットをしないでください.

```
/**
 * 3つの要素から重みを計算する:
 *   items sent
 *   items received
 *   last timestamp
 */
```

このコードは次のように表示されます

```
3つの要素から重みを計算する: items sent items received last timestamp
```

代わりに以下のように記述してください.

```
/**
 * 3つの要素から重みを計算する:
 * <ul>
 * <li>items sent
 * <li>items received
 * <li>last timestamp
 * </ul>
 */
```

より詳細は JavaDoc を参照してください.

トップレベル・ファイルレベルコメント

コピーライト と作者の情報は必須ではありません. 一般的には, 2 つ以上のクラス定義があるファイルの場合は概要を記載することを推奨します. トップレベルコメントはそのコードに詳しくない読者を対象として, そのファイルが何をしているのかを説明するコメントです. ファイルの内容, 互換性の情報などを記述します.

```
/**
 * @fileoverview ファイルの説明, 使用方法や
 * 依存関係の情報など.
 * @author user@google.com (Firstname Lastname)
 */
```

クラスコメント

クラスコメントには説明とコンストラクタを示す型情報を記述します.

```
/**
 * Class making something fun and easy.
 * @param {string} arg1 An argument that makes this more interesting.
 * @param {Array.<number>} arg2 List of numbers to be processed.
 * @constructor
 * @extends {goog.Disposable}
 */
project.MyClass = function(arg1, arg2) {
  // ...
};
goog.inherits(project.MyClass, goog.Disposable);
```

メソッド・関数コメント

パラメータと戻り値のドキュメントを必ず記述します. メソッドの説明はパラメータと戻り値から明らかな場合は省略できます. メソッドの説明文は第三者が宣言している文体で書きます.

```
/**
 * MyClass のインスタンスを処理して何かを返す関数
 * @param {project.MyClass} obj Instance of MyClass which leads to a long
 *     comment that needs to be wrapped to two lines.
 * @return {boolean} Whether something occured.
 */
function PR_someMethod(obj) {
  // ...
}
```

プロパティコメント

```
/** @constructor */
project.MyClass = function() {
  /**
   * 1 pane ごとの最大数.
   * @type {number}
   */
  this.someProperty = 4;
}
```

JSDoc タグリファレンス

> Note: 訳注
>
> 省略しました. 詳しくは原文にある表を参照してください. 後日補完します.
>
> http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml?showone=Comments#Comments

goog.provide での依存の提供

トップレベルのシンボルのみを提供します.

クラスのすべてのメンバは同じクラスにあるべきです. そのためトップレベルのクラスのみを提供してください. 中でメンバとして定義されている enum やインナークラスは提供しないでください.

このようにしてください:

```
goog.provide('namespace.MyClass');
```

このようにはしないでください:

```
goog.provide('namespace.MyClass');
goog.provide('namespace.MyClass.Enum');
goog.provide('namespace.MyClass.InnerClass');
goog.provide('namespace.MyClass.TypeDef');
goog.provide('namespace.MyClass.CONSTANT');
goog.provide('namespace.MyClass.staticMethod');
```

名前空間の中のメンバも提供できます.

```
goog.provide('foo.bar');
goog.provide('foo.bar.method');
goog.provide('foo.bar.CONSTANT');
```

コンパイル

顧客が接するコードには Closure Compiler のような JavaScript コンパイラを使うことが必要です.

Tips や トリック

真偽値表現

以下はすべて boolean 表現では false になります.

- null
- undefined
- ‘’(空の文字列)
- 0 (数字)

以下は true になるので注意してください

- ‘0’(文字列)
- [] (空の配列)
- {} (空のオブジェクト)

以上より, 以下のようなコードの代わりに:

```
while (x != null) {
```

以下のように短く書くことができます (ただし x は 0 や空文字列や false にならないと仮定しています).

```
while (x) {
```

もし文字列が null でも空でもないことをチェックしたいときは:

```
if (y != null && y != '') {
```

こうではなく, 以下のようによりスマートに記述できます.

```
if (y) {
```

ただし, boolean 表現には直感的でないものが多くあるので注意してください.

```
Boolean('0') == true
'0' != true
```

```
0 != null
0 == []
0 == false
Boolean(null) == false
null != true
null != false
Boolean(undefined) == false
undefined != true
undefined != false
Boolean([]) == true
[] != true
[] == false
Boolean({}) == true
{} != true
{} != false
```

条件式と3項演算子

このコードの代わりに:

```
if (val != 0) {
  return foo();
} else {
  return bar();
}
```

以下のように書けます.

```
return val ? foo() : bar();
```

3項演算子は HTML を生成するときにも便利です.

```
var html = '<input type="checkbox"' +
    (isChecked ? ' checked' : '') +
    (isEnabled ? '' : ' disabled') +
    ' name="foo">';
```

&& と ||

2項の boolean 演算子はショートサーキットで, 最後の項まで評価されます.

`||` は “デフォルト演算子” とも呼ばれます. 以下のコードは,

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win;
  if (opt_win) {
    win = opt_win;
  } else {
    win = window;
  }
  // ...
}
```

次のように書き換えられます.

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win = opt_win || window;
  // ...
}
```

同様に `&&` 演算子を使うことでもコードを短縮できます. このようなコードの代わりに:

```
if (node) {
  if (node.kids) {
    if (node.kids[index]) {
      foo(node.kids[index]);
    }
  }
}
```

次のように書けます.

```
if (node && node.kids && node.kids[index]) {
  foo(node.kids[index]);
}
```

あるいは, 次のような書き方も可能です.

```
var kid = node && node.kids && node.kids[index];
if (kid) {
  foo(kid);
}
```

しかしながら, この例はすこしやりすぎでしょう.

```
node && node.kids && node.kids[index] && foo(node.kids[index]);
```

ノードリストのイテレート

ノードリストの多くは, ノードのイテレータとフィルタから実装されています. よって, 例えばリストの長さを取得したい場合は O(n), またリストの要素を走査しそれぞれについて長さをチェックした場合は O(n^2) かかってしまいます.

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0; i < paragraphs.length; i++) {
  doSomething(paragraphs[i]);
}
```

代わりにこう書いたほうがベターです:

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0, paragraph; paragraph = paragraphs[i]; i++) {
  doSomething(paragraph);
}
```

これは, false として扱われる値を含まない, すべてのコレクションや配列に対して問題なく動作します.

Note: 訳注

id:co-sche さんにご指摘いただき修正しました.

http://d.hatena.ne.jp/co-sche/20100729/1280409953

childNodes をたどる場合は, firstChild や nextSibling プロパティを使うことができます.

```
var parentNode = document.getElementById('foo');
for (var child = parentNode.firstChild; child; child = child.nextSibling) {
  doSomething(child);
}
```

あとがき

一貫性をもたせてください

あなたがコードを書くとき, どのようなスタイルで書くかを決める前に, すこしまわりのコードを見るようにしてください. もしまわりのコードが算術演算子の両端にスペースを入れていれば, あなたもそうすべきです. もしまわりのコードのコメントが, ハッシュマーク `#` を使って矩形を描いていたとしたら, あなたもまたそうすべきです.

コーディングスタイルのガイドラインを策定することのポイントは, コーディングの共通の語彙を持って, どう書くか ではなく 何を書くか に集中できるようにすることです. 私たちはここでグローバルなスタイルのルールを提供したので, 人々は共通の語彙を得られたことになります しかしローカルなスタイルもまた重要です. もしあなたが追加したコードがあまりにもまわりのコードと違っていた場合, コードを読む人のリズムが乱されてしまいます. それは避けてください.