번역작업한 곳**dfdf**

https://docs.google.com/document/d/1Ngvhp0m16cCQEdDAIwL7MNdtdGnQZ7-eucSRyluEHDU/edit

번역작업한 곳
작업방식
Todo
Todo . 요 윗부분은 뭔가 내용이 다른 듯.. 원문이 바뀐 건지.. 일단 패스
요기까지 정리했어요~
여기까지 작업했음.... 0311
일단 넘어가기.. 원문에서 못 찾겠음.
Todo . 링크를 걸어야 함.
여기까지 작업했어요~ - nassol 0311

작업방식

남길 것은 파랗게!
번역작업한 문서에서 내용을 복사해서 붙여넣기.
#, * 등을 원문에서 옮겨 쓰기..
애매하면 Todo 표시, 제목2 서식 설정하고 넘어가기

Principles of Writing Consistent, Idiomatic JavaScript
자연스럽고 일관성 있게 자바스크립트 코딩하는 원칙

이 문서는 계속 바뀝니다. 주변의 코드를 더 낫게 만들기 위한 좋은 아이디어가 있으시면 알려주세요. Github에서 fork, clone, branch, commit, push, pull request 하는 방법으로 참여해 보세요.

* Rick Waldron [@rwaldron](http://twitter.com/rwaldron), [github](https://github.com/rwldrn)
* Mathias Bynens [@mathias](http://twitter.com/mathias), [github](https://github.com/mathiasbynens)
* Schalk Neethling [@ossreleasefeed](http://twitter.com/ossreleasefeed), [github](https://github.com/ossreleasefeed/)
* Kit Cambridge [@kitcambridge](http://twitter.com/kitcambridge),

ㅌ[github](https://github.com/kitcambridge)
* Raynos [github](https://github.com/Raynos)
* Matias Arriola [@MatiasArriola](https://twitter.com/MatiasArriola), [github](https://github.com/MatiasArriola/)
* Idan Gazit [@idangazit](http://twitter.com/idangazit), [github](https://github.com/idangazit)
* Leo Balter [@leobalter](http://twitter.com/leobalter), [github](https://github.com/leobalter)
* Breno Oliveira [@garu_rj](http://twitter.com/garu_rj), [github](https://github.com/garu)
* Leo Beto Souza [@leobetosouza](http://twitter.com/leobetosouza), [github](https://github.com/leobetosouza)

All code in any code-base should look like a single person typed it, no matter how many people contributed.

코드는 마치 한 사드 기반에 있는 모든 코람이 작성한 것처럼 보여야 합니다. 많은 사람이 코드 작성에 참여했더라도 말이죠.

The following list outlines the practices that I use in all code that I am the original author of; contributions to projects that I have created should follow these guidelines.

제가 원저자인 프로젝트의 코드를 작성할 때는 몇 가지 원칙을 따르는데요, 아래의 목록에 그 원칙들을 나열하였습니다. 이 프로젝트의 코드작성에 참여할 때에는 다음의 가이드라인을 따아야 합니다.

I do not intend to impose my style preferences on other people's code; if they have an existing common style - this should be respected.

다른 사람들에게 제 코드 작성 스타일을 따르라고 강요하려는 의도는 아닙니다. 이미 준수하는 코드작성 스타일이 있다면, 그것을 따라야 하겠지요.

> "Part of being a good steward to a successful project is realizing that writing code for yourself is a Bad Idea™. If thousands of people are using your code, then write your code for maximum clarity, not your personal preference of how to get clever within the spec." - Idan Gazit

> “성공적인 프로젝트의 멋진 일원이 되기 위해서는 여러분 마음대로 코드를 작성하는 것이 나쁜 생각™임을 깨닫는 것이지요. 수천만 사람들이 여러분의 코드를 사용한다면, 가장 명확하게 코딩해야 합니다. 언어 명세가 허용하는 한 가장 똑똑한 코드를 만들겠다는 당신의 개인적인 선호에 따라서가 아니라” - Idan Gazit

번역

* [프랑스어](https://github.com/jfroffice/idiomatic.js/)
* [스페인어](https://github.com/MatiasArriola/idiomatic.js/)
* [포르투갈어 - 브라질](https://github.com/leobalter/idiomatic.js/)

* [한국어](http://https://github.com/EngForDev/idiomatic.js/) - by EngForDev.com : 이대엽, rkJun, nassol, 망고, Hwan Min Hong, 심태현, leoinsigh
Important, Non-Idiomatic Stuff:

관련은 없지만, 중요한

Code Quality: Awesome Tools, Resources & References

코드 품질: 멋진 툴들, 리소스 & 참고자료

 * [jsPerf](http://jsperf.com/)
 * [jsFiddle](http://jsfiddle.net/)
 * [jsbin](http://jsbin.com/)
 * [JavaScript Lint (JSL)](http://javascriptlint.com/)
 * [jshint](http://jshint.com/)
 * [jslint](http://jslint.org/)

[Leveraging Code Quality Tools by Anton Kovalyov](http://anton.kovalyov.net/slides/gothamjs/)

Get Smart
똑똑해지기
http://es5.github.com/

The following should be considered 1) incomplete, and 2) *REQUIRED READING*. I don't always agree with the style written by the authors below, but one thing is certain: They are consistent. Furthermore, these are authorities on the language.

아래 리스트는

1) 이 주제의 모든 지식들을 포괄하지는 않습니다.
2) 반드시 읽기를 권합니다. 저는 아래 저자들이 말하는 코딩 스타일에 대해 항상 찬성하는 것은 아니지만, 분명한 것은 그들이 일관성 있단 겁니다. 게다가 이들은 프로그래밍 언어의 권위자들입니다.

Todo

 * [Eloquent JavaScript](http://eloquentjavascript.net/)
 * [JavaScript, JavaScript](http://javascriptweblog.wordpress.com/)
 * [Rebecca Murphey](http://www.rebeccamurphey.com/) or [Adventures in JavaScript Development](http://rmurphey.com/)

* [Perfection Kills](http://perfectionkills.com/)
* [Douglas Crockford's Wrrrld Wide Web](http://www.crockford.com)

Build & Deployment Process

빌드 & 배포 프로세스

Projects should always attempt to include some generic means by which source can be linted, tested and compressed in preparation for production use. For this task, [grunt](https://github.com/cowboy/grunt) by Ben Alman is second to none and has officially replaced the "kits/" directory of this repo.

Todo . 요 윗부분은 뭔가 내용이 다른 듯.. 원문이 바뀐 건지.. 일단 패스

https://docs.google.com/document/d/1Ngvhp0m16cCQEdDAIwL7MNdtdGnQZ7-eucSRyluEHDU/edit#bookmark=id.oot2a11nhrzd

Test Facility

Projects _must_ include some form of unit, reference, implementation or functional testing. Use case demos DO NOT QUALIFY as "tests". The following is a list of test frameworks, none of which are endorsed more than the other.

테스트 Facility

프로젝트에는 unit, reference, implementation이나 기능에 관한 테스트 부분이 들어가 있어야 합니다. functional testing. 시범 데모인 DO NOT QUALIFY를 사용해서 테스트하세요. 아래에는 테스트를 하는 프레임워크의 목록을 나열해 두었습니다. 이 중에 어느 하나를 더욱 추천하는 것은 아닙니다.

* [QUnit](http://github.com/jquery/qunit)
* [Jasmine](https://github.com/pivotal/jasmine)
* [Vows](https://github.com/cloudhead/vows)
* [Mocha](https://github.com/visionmedia/mocha)
* [Hiro](http://hirojs.com/)
* [JsTestDriver](https://code.google.com/p/js-test-driver/)

Table of Contents

목차

Idiomatic Style Manifesto
자연스러운 스타일에 관한 선언문

1. 화이트 스페이스

* Never mix spaces and tabs.
* When beginning a project, before you write any code, choose between soft indents (spaces) or real tabs — this is law.
* For readability, I always recommend setting your editor's indent size to two characters — this means two spaces or two spaces representing a real tab.
* If your editor supports it, always work with the "show invisibles" setting turned on. The benefits of this practice are:
* Enforced consistency
* Eliminating end of line whitespace
* Eliminating blank line whitespace
* Commits and diffs that are easier to read

* 절대 스페이스와 탭을 섞어 쓰지 마세요.
* 프로젝트를 시작할 때, 코드를 작성하기 전에 먼저 스페이스와 탭 중의 하나를 선택하여야 합니다. — 이것이 규칙입니다.
* 가독성을 위해서 저는 항상 편집기의 들여쓰기 크기를 2 문자로 설정하기를 권장합니다. — 2 문자란 2 스페이스 또는 탭으로 표현되는 2칸 공간을 의미합니다.
* 에디터가 들여쓰기 설정을 지원한다면, 저는 "show invisibles" 설정을 켜고 작업합니다. 이렇게 했을 때의 장점은 다음과 같아요:
* 일관성을 강제할 수 있어요.
* 줄 끝의 공백문자 제거하기 좋아요.
* 빈 줄 공백문자를 제거하기 좋아요.

* 커밋하고 비교(diff)할 때 읽기 편해요.

요기까지 정리했어요~

2. Beautiful Syntax
2. 아름답게 구문 작성하기

A. Parens, Braces, Linebreaks
A. 중괄호{}, 괄호(), 줄 바꾸기

```javascript

// if/else/for/while/try always have spaces, braces and span multiple lines
// this encourages readability

// if나 else, for while, try를 쓸 때에는 항상 빈 칸을 띠우고, 괄호를 사용하고,
// 여러 줄로 나누어 쓰세요.
// 이렇게 하면 가독성이 더 좋아집니다..

// 2.A.1.1
// Examples of really cramped syntax
//2.A.1.1
//빼곡해서 알아보기 어려운 구문의 예

if(condition) doSomething();

while(condition) iterating++;중ㄱ
for(var i=0;i<100;i++) someIterativeFn();


// 2.A.1.1
// Use whitespace to promote readability

//2.A.1.1
//가독성이 높아지도록 빈 칸을 띠워주세요.

if ( condition ) {
        // statements
        // 코드
}
```

```
while ( condition ) {
        //코드
}

for ( var i = 0; i < 100; i++ ) {
        // 코드
}

// Even better:
//아래처럼 하면 더 좋습니다.

var i,
        length = 100;

for ( i = 0; i < length; i++ ) {
        // 코드
}

// Or...
// 아니면 이렇게 할 수도 있죠...

var i = 0,
        length = 100;

for ( ; i < length; i++ ) {
        // 코드
}

var prop;

for ( prop in object ) {
        // 코드
}


if ( true ) {
        // 코드
} else {
        // 코드
}
```

B. Assignments, Declarations, Functions (Named, Expression, Constructor)
B. 할당, 선언, 함수(일반, 표현식, 생성자)

```javascript

// 2.B.1.1
// Variables
// 변수
var foo = "bar",
        num = 1,
        undef;

// Literal notations:
// 리터럴 표기법
var array = [],
        object = {};


// 2.B.1.2
// Using only one `var` per scope (function) promotes readability
// and keeps your declaration list free of clutter (also saves a few keystrokes)

// 함수 같은 유효 범위에 *'var'*를 하나만 사용하면 가독성이 높아집니다*.*
// 이렇게 하면 선언 목록도 깔끔해집니다*(*아울러 키보드로 입력해야 할 양도 줄어들지요*)*

// Bad
// 나쁜 스타일
var foo = "";
var bar = "";
var qux;

// Good
// 좋은 스타일
var foo = "",
        bar = "",
        quux;

// or..
// 또는..
var // Comment on these
**var** // 아래 변수에 대한 설명
foo = "",
```

```
    bar = "",
    quux;
```

여기까지 작업했음**.... 0311**

일단 넘어가기**..** 원문에서 못 찾겠음**.**

```
// 2.B.1.3
// var statements should always be in the beginning of their respective scope (function).
// Same goes for const and let from ECMAScript 6.



// Bad
function foo() {

    // some statements here

    var bar = "",
        qux;
}

// Good
function foo() {
    var bar = "",
        qux;

    // all statements after the variables declarations.
}
```

```javascript

// 2.B.2.1
// Named Function Declaration
// 일반 함수 선언
function foo( arg1, argN ) {

}
```

```
// Usage
// 사용법
foo( arg1, argN );


// 2.B.2.2
// Named Function Declaration
// 일반 함수 선언
function square( number ) {
        return number * number;
}

// Usage
// 사용법
square( 10 );

// Really contrived continuation passing style
// 아주 부자연스러운 연속 전달 스타일(continuation passing style)
// (CPS에 대해서는 http://goo.gl/TA32o를 참고)
```

Todo . 링크를 걸어야 함**.**

```
function square( number, callback ) {
        callback( number * number );
}

square( 10, function( square ) {
        // 콜백 문장

});


// 2.B.2.3
// Function Expression
// 함수 표현식
var square = function( number ) {
        // Return something valuable and relevant
        // 가치 있고 의미 있는 뭔가를 반환합니다
        return number * number;
};
```

```
// Function Expression with Identifier
// This preferred form has the added value of being
// able to call itself and have an identity in stack traces:
// 식별자를 지닌 함수 표현식
// 아래와 같은 형태는 자기 자신을 호출할 수 있으면서
// 스택 트레이스상에서 식별할 수 있다는 부가적인 장점이 있습니다.

var factorial = function factorial( number ) {
        if ( number < 2 ) {
                return 1;
        }

        return number * factorial( number-1 );
};
```

여기까지 작업했어요~ **- nassol 0311**

```
// 2.B.2.4
// Constructor Declaration
function FooBar( options ) {

        this.options = options;
}

// Usage
var fooBar = new FooBar({ a: "alpha" });

fooBar.options;
// { a: "alpha" }

```

C. Exceptions, Slight Deviations

```javascript

// 2.C.1.1
// Functions with callbacks
foo(function() {
```

```
        // Note there is no extra space between the first paren
        // of the executing function call and the word "function"
});

// Function accepting an array, no space
foo([ "alpha", "beta" ]);

// 2.C.1.2
// Function accepting an object, no space
foo({
        a: "alpha",
        b: "beta"
});

// Inner grouping parens, no space
if ( !("foo" in obj) ) {

}
```

D. Consistency Always Wins

In sections 2.A-2.C, the whitespace rules are set forth as a recommendation with a simpler, higher purpose: consistency.

It's important to note that formatting preferences, such as "inner whitespace" should be considered optional, but only one style should exist across the entire source of your project.

```javascript
// 2.D.1.1

if (condition) {
        // statements
}

while (condition) {
        // statements
}

for (var i = 0; i < 100; i++) {
        // statements
}
```

```
if (true) {
	// statements
} else {
	// statements
}
```

E. End of Lines and Empty Lines

Whitespace can ruin diffs and make changesets impossible to read. Consider incorporating a pre-commit hook that removes end-of-line whitespace and blanks spaces on empty lines automatically.

3. Type Checking (Courtesy jQuery Core Style Guidelines)

3.A Actual Types

* String:

	`typeof variable === "string"`

* Number:

	`typeof variable === "number"`

* Boolean:

	`typeof variable === "boolean"`

* Object:

	`typeof variable === "object"`

* Array:

	`Array.isArray(arrayObject)`
	(wherever possible)

* null:

	`variable === null`

* null or undefined:

    `variable == null`

* undefined:

    * Global Variables:

        * `typeof variable === "undefined"`

    * Local Variables:

        * `variable === undefined`

    * Properties:
        * `object.prop === undefined`
        * `object.hasOwnProperty( prop )`
        * `"prop" in object`

JavaScript is a dynamically typed language - which can be your best friend or worst enemy, so: Always respect `type`, as recommended.

3.B Coerced Types

Consider the implications of the following...

Given this HTML:

```html

<input type="text" id="foo-input" value="1">

```

```js

// 3.B.1.1

// `foo` has been declared with the value `0` and its type is `number`
```

```
var foo = 0;

// typeof foo;
// "number"
...

// Somewhere later in your code, you need to update `foo`
// with a new value derived from an input element

foo = document.getElementById("foo-input").value;

// If you were to test `typeof foo` now, the result would be `string`
// This means that if you had logic that tested `foo` like:

if ( foo === 1 ) {

        importantTask();

}

// `importantTask()` would never be evaluated, even though `foo` has a value of "1"


// 3.B.1.2

// You can preempt issues by using smart coercion with unary + or - operators:

foo = +document.getElementById("foo-input").value;
    ^ unary + operator will convert its right side operand to a number

// typeof foo;
// "number"

if ( foo === 1 ) {

        importantTask();

}

// `importantTask()` will be called
```

Here are some common cases along with coercions:

```javascript

// 3.B.2.1

var number = 1,
          string = "1",
          bool = false;

number;
// 1

number + "";
// "1"

string;
 // "1"

+string;
// 1

+string++;
// 1

string;
// 2

bool;
// false

+bool;
// 0

bool + "";
// "false"
```


```javascript
// 3.B.2.2

var number = 1,
```

```
            string = "1",
            bool = true;

string === number;
// false

string === number + "";
// true

+string === number;
// true

bool === number;
// false

+bool === number;
// true

bool === string;
// false

bool === !!string;
// true
```

```javascript
// 3.B.2.3

var array = [ "a", "b", "c" ];

!!~array.indexOf( "a" );
// true

!!~array.indexOf( "b" );
// true

!!~array.indexOf( "c" );
// true

!!~array.indexOf( "d" );
// false
```

```
var num = 2.5;

parseInt( num, 10 );

// is the same as...

~~num;
```

4. Conditional Evaluation

```javascript
// 4.1.1
// When only evaluating that an array has length,
// instead of this:
if ( array.length > 0 ) ...

// ...evaluate truthiness, like this:
if ( array.length ) ...


// 4.1.2
// When only evaluating that an array is empty,
// instead of this:
if ( array.length === 0 ) ...

// ...evaluate truthiness, like this:
if ( !array.length ) ...


// 4.1.3
// When only evaluating that a string is not empty,
// instead of this:
if ( string !== "" ) ...

// ...evaluate truthiness, like this:
if ( string ) ...


// 4.1.4
// When only evaluating that a string _is_ empty,
```

```
// instead of this:
if ( string === "" ) ...

// ...evaluate falsy-ness, like this:
if ( !string ) ...


// 4.1.5
// When only evaluating that a reference is true,
// instead of this:
if ( foo === true ) ...

// ...evaluate like you mean it, take advantage of it's primitive capabilities:
if ( foo ) ...


// 4.1.6
// When evaluating that a reference is false,
// instead of this:
if ( foo === false ) ...

// ...use negation to coerce a true evaluation
if ( !foo ) ...

// ...Be careful, this will also match: 0, "", null, undefined, NaN
// If you _MUST_ test for a boolean false, then use
if ( foo === false ) ...


// 4.1.7
// When only evaluating a ref that might be null or undefined, but NOT false, "" or 0,
// instead of this:
if ( foo === null || foo === undefined ) ...

// ...take advantage of == type coercion, like this:
if ( foo == null ) ...

// Remember, using == will match a `null` to BOTH `null` and `undefined`
// but not `false`, "" or 0
null == undefined

```
ALWAYS evaluate for the best, most accurate result - the above is a guideline, not a

dogma.

```javascript

// 4.2.1
// Type coercion and evaluation notes

Prefer `===` over `==` (unless the case requires loose type evaluation)

=== does not coerce type, which means that:

"1" === 1;
// false

== does coerce type, which means that:

"1" == 1;
// true


// 4.2.2
// Booleans, Truthies & Falsies

Booleans: true, false

Truthy are: "foo", 1

Falsy are: "", 0, null, undefined, NaN, void 0

```


5. Practical Style

```javascript

// 5.1.1
// A Practical Module

(function( global ) {
	var Module = (function() {

		var data = "secret";
```

```
                return {
                        // This is some boolean property
                        bool: true,
                        // Some string value
                        string: "a string",
                        // An array property
                        array: [ 1, 2, 3, 4 ],
                        // An object property
                        object: {
                                lang: "en-Us"
                        },
                        getData: function() {
                                // get the current value of `data`
                                return data;
                        },
                        setData: function( value ) {
                                // set the value of `data` and return it
                                return ( data = value );
                        }
                };
        })();

        // Other things might happen here

        // expose our module to the global object
        global.Module = Module;

})( this );

```

```javascript

// 5.2.1
// A Practical Constructor

(function( global ) {

        function Ctor( foo ) {

                this.foo = foo;
```

```
        return this;
    }

    Ctor.prototype.getFoo = function() {
        return this.foo;
    };

    Ctor.prototype.setFoo = function( val ) {
        return ( this.foo = val );
    };


    // To call constructor's without `new`, you might do this:
    var ctor = function( foo ) {
        return new Ctor( foo );
    };


    // expose our constructor to the global object
    global.ctor = ctor;

})( this );
```

6. Naming

    You are not a human code compiler/compressor, so don't try to be one.

    The following code is an example of egregious naming:

    ```javascript

    // 6.1.1
    // Example of code with poor names

    function q(s) {
        return document.querySelectorAll(s);
    }
    var i,a=[],els=q("#foo");
    ```

```
for(i=0;i<els.length;i++){a.push(els[i]);}
```

Without a doubt, you've written code like this - hopefully that ends today.

Here's the same piece of logic, but with kinder, more thoughtful naming (and a readable structure):

```javascript

// 6.2.1
// Example of code with improved names

function query( selector ) {
	return document.querySelectorAll( selector );
}

var idx = 0,
	elements = [],
	matches = query("#foo"),
	length = matches.length;

for( ; idx < length; idx++ ){
	elements.push( matches[ idx ] );
}

```

A few additional naming pointers:

```javascript

// 6.3.1
// Naming strings

`dog` is a string


// 6.3.2
// Naming arrays

`dogs` is an array of `dog` strings
```

```
// 6.3.3
// Naming functions, objects, instances, etc

camelCase; function and var declarations


// 6.3.4
// Naming constructors, prototypes, etc.

PascalCase; constructor function


// 6.3.5
// Naming regular expressions

rDesc = //;


// 6.3.6
// From the Google Closure Library Style Guide

functionNamesLikeThis;
variableNamesLikeThis;
ConstructorNamesLikeThis;
EnumNamesLikeThis;
methodNamesLikeThis;
SYMBOLIC_CONSTANTS_LIKE_THIS;



```

7. Misc

This section will serve to illustrate ideas and concepts that should not be considered dogma, but instead exists to encourage questioning practices in an attempt to find better ways to do common JavaScript programming tasks.

A. Using `switch` should be avoided, modern method tracing will blacklist functions with switch statements

There seems to be drastic improvements to the execution of `switch` statements in latest

releases of Firefox and Chrome.
http://jsperf.com/switch-vs-object-literal-vs-module

Notable improvements can be witnesses here as well:
https://github.com/rwldrn/idiomatic.js/issues/13

```javascript

// 7.A.1.1
// An example switch statement

switch( foo ) {
	case "alpha":
		alpha();
		break;
	case "beta":
		beta();
		break;
	default:
		// something to default to
		break;
}

// 7.A.1.2
// A better approach would be to use an object literal or even a module:

var switchObj = {
	alpha: function() {
		// statements
		// a return
	},
	beta: function() {
		// statements
		// a return
	},
	_default: function() {
		// statements
		// a return
	}
};

var switchModule = (function () {
	return {
```

```
                alpha: function() {
                        // statements
                        // a return
                },
                beta: function() {
                        // statements
                        // a return
                },
                _default: function() {
                        // statements
                        // a return
                }
        };
})();


// 7.A.1.3
// If `foo` is a property of `switchObj` or `switchModule`, execute as a method...

( Object.hasOwnProperty.call( switchObj, foo ) && switchObj[ foo ] || switchObj._default )( args );

( Object.hasOwnProperty.call( switchObj, foo ) && switchModule[ foo ] || switchModule._default )( args );

// If you know and trust the value of `foo`, you could even omit the OR check
// leaving only the execution:

switchObj[ foo ]( args );

switchModule[ foo ]( args );


// This pattern also promotes code reusability.

```

B. Early returns promote code readability with negligible performance difference

```javascript

// 7.B.1.1
// Bad:
```

```
function returnLate( foo ) {
	var ret;

	if ( foo ) {
		ret = "foo";
	} else {
		ret = "quux";
	}
	return ret;
}

// Good:

function returnEarly( foo ) {

	if ( foo ) {
		return "foo";
	}
	return "quux";
}
```

8. Native & Host Objects

The basic principal here is:

Don't do stupid shit and everything will be ok.

To reinforce this concept, please watch the following presentation:

“Everything is Permitted: Extending Built-ins” by Andrew Dupont (JSConf2011, Portland, Oregon)

<iframe src="http://blip.tv/play/g_Mngr6LegI.html" width="480" height="346" frameborder="0" allowfullscreen></iframe><embed type="application/x-shockwave-flash" src="http://a.blip.tv/api.swf#g_Mngr6LegI" style="display:none"></embed>

http://blip.tv/jsconf/jsconf2011-andrew-dupont-everything-is-permitted-extending-built-ins-5211542

9. Comments

* JSDoc style is good (Closure Compiler type hints++)
* Single line above the code that is subject
* Multiline is good
* End of line comments are prohibited!

10. One Language Code

Programs should be written in one language, whatever that language may be, as dictated by the maintainer or maintainers.

Appendix

Comma First.

Any project that cites this document as its base style guide will not accept comma first code formatting unless explicitly specified otherwise.

See: https://mail.mozilla.org/pipermail/es-discuss/2011-September/016805.html
Notable: "That is horrible, and a reason to reject comma first.", "comma-first still is to be avoided"