**JavaScript Style Guide**

Contents

1. 소개

1.1 용어 참고

1.2 가이드 노트

2. 소스 파일 기본 사항

2.1 파일 이름

2.2 파일 인코딩 : UTF – 8

2.3 특수 문자

3. 소스 파일 구조

3.1 라이선스 또는 저작권 정보 (있는 경우)

3.2 @fileoverview JSDoc (있는 경우)

3.3 ES 모듈

3.4 파일의 구현

4. 서식

4.1 교정기

4.2 블록 들여 쓰기 : +2 공백

4.3 진술

4.4 열 제한 : 80

4.5 줄 바꿈

4.6 공백

4.7 그룹화 괄호 : 권장

4.8 댓글

5. 언어 기능

5.1 지역 변수 선언

5.2 배열 리터럴

5.3 객체 리터럴

5.4 클래스

5.5 기능

5.6 문자열 리터럴

5.7 숫자 리터럴

5.8 제어 구조

5.9 this

5.10 평등 검사

5.11 허용되지 않는 기능

6. 명명규칙

6.1 모든 식별자에 공통적인 규칙

6.2 식별자 유형별 규칙

6.3 카멜 케이스

7. JSDoc

1.1 용어 참고

1.2 가이드 노트

8. 정책

8.1 Google 스타일에서 지정하지 않은 문제 : 일관성 유지!

8.2 컴파일러 경고

8.3 지원 중단

8.4 Google 스타일에없는 코드

8.5 지역 스타일 규칙

8.6 생성 된 코드 : 대부분 면제

9. 부록

9.1 JSDoc 태그 참조

9.2 일반적으로 오해되는 스타일 규칙

9.3 스타일 관련 도구

9.4 레거시 플랫폼에 대한 예외

**1. 소개**

이 문서는 JavaScript 프로그래밍 언어의 소스 코드에 대한 코딩 표준을 정의합니다. 다른 프로그래밍 스타일 가이드와 마찬가지로 다루는 문제는 형식화의 미적 문제 뿐 아니라 다른 유형의 규칙이나 코딩 표준에도 적용됩니다. 그러나이 문서는 주로 우리가 보편적으로 따르는 엄격하고 빠른 규칙에 초점을 맞추고 명확하게 시행 할 수없는 조언 (인간 또는 도구)을 제공하지 않습니다.

1.1 용어 참고

이 문서에서 달리 명시되지 않는 한 :

1. 주석 이라는 용어 는 항상 구현 주석을 의미합니다 . Google은 문서 주석 이라는 문구를 사용하지 않고 대신 사람이 읽을 수있는 텍스트와 /\*\* … \*/.

2. 이 스타일 가이드는 must , must not , should , should not , may 라는 문구 를 사용할 때 RFC 2119 용어를 사용합니다 . 조건은 선호 하고 회피 대응 해야 하고 해야하지 , 각각. 명령문 및 선언문은 규범 적이며 must에 해당 합니다 .

문서 전체에 다른 용어 참고 사항 이 가끔 표시됩니다.

1.2 가이드 노트

이 문서의 예제 코드는 **표준**이 **아닙니다** .  즉, 예제는 단순히 스타일이지만 코드를 표현하는 유일한 세련된 방법을 설명하지 못할 수 있습니다.  예제에서 선택한 선택적 형식 지정은 규칙으로 적용되지 않아야합니다.

**2. 소스 파일 기본 사항**

2.1 파일 이름

파일 이름은 모두 소문자 여야하며 밑줄 ( \_) 또는 대시 ( -)를 포함 할 수 있지만 추가 구두점은 없습니다. 프로젝트에서 사용하는 규칙을 따르십시오. 파일 이름의 확장자는 .js

2.2 파일 인코딩 : UTF-8

소스 파일은 **UTF-8** 로 인코딩됩니다.

2.3 특수 문자

2.3.1 공백 문자

줄 종결 자 시퀀스를 제외하고 ASCII 가로 공백 문자 (0x20)는 소스 파일의 모든 위치에 나타나는 유일한 공백 문자입니다. 이것은

- 문자열 리터럴의 다른 모든 공백 문자는 이스케이프됩니다.

- 탭 문자는 들여 쓰기에 사용 되지 않습니다 .

2.3.2 특수 이스케이프 시퀀스

특별한 이스케이프 시퀀스가 있는 모든 문자 ( \', \", \\, \b, \f, \n, \r, \t, \v), 그 시퀀스 (예 : 해당 숫자 탈출보다는 사용 \x0a, \u000a또는 \u{a}). 레거시 8 진 이스케이프는 사용되지 않습니다.

2.3.3 비 ASCII 문자

나머지 비 ASCII 문자의 경우 실제 유니 코드 문자 (예 ∞:) 또는 동등한 16 진수 또는 유니 코드 이스케이프 (예 :) \u221e가 사용됩니다. 이는 코드 를 읽고 이해하기 쉽게 만드는 요소에 따라 다릅니다 .

팁 : 유니 코드 이스케이프 케이스 및 실제 유니 코드 문자가 사용되는 경우에도 설명 주석이 매우 유용 할 수 있습니다.

/ \* 최고 : 댓글이 없어도 완벽하게 선명합니다. \* / const 단위 = 'μs' ;

/ \* 허용 : 그러나 μ는 인쇄 가능한 문자이므로 불필요합니다. \* / const 단위 = '\ u03bcs' ; // 'μs'

/ \* 좋음 : 명확성을 위해 주석과 함께 인쇄 할 수없는 문자에는 이스케이프를 사용합니다. \* / return '\ ufeff' + content ; // 바이트 순서 표시를 앞에 추가합니다.

/ \* 나쁨 : 독자는 이것이 어떤 캐릭터인지 모릅니다. \* / const 단위 = '\ u03bcs' ;

팁 : 일부 프로그램이 비 ASCII 문자를 제대로 처리하지 못할 수 있다는 두려움 때문에 코드의 가독성을 낮추지 마십시오. 이런 일이 발생하면 해당 프로그램이 **중단** 되고 **수정** 해야합니다 .

## 3. 소스 파일 구조

모든 새 소스 파일은 goog.module파일 ( goog.module호출을 포함하는 파일 ) 또는 ECMAScript (ES) 모듈 (사용 import및 export 문) 이어야합니다 . 파일은 다음 순서 로 구성됩니다 .

1. 라이센스 또는 저작권 정보 (있는 경우)

2. @fileoverview JSDoc (있는 경우)

3. ES import문 (ES 모듈 인 경우)

4. 파일의 구현

하나 또는 두 개의 빈 줄이 앞에 올 수있는 파일 구현을 제외하고 정확히 하나의 빈 줄 이 존재하는 각 섹션을 구분합니다.

**3.1 라이선스 또는 저작권 정보 (있는 경우)**

라이선스 또는 저작권 정보가 파일에 속하면 여기에 속합니다.

**3.2 @fileoverviewJSDoc (있는 경우)**

서식 규칙 은 7.5 최상위 / 파일 수준 주석 을 참조하십시오 .

**3.3 import**

import문은 줄 바꿈되어서는 안되므로 80 열 제한에 대한 예외입니다.

3.3.1.1 import 경로

ES 모듈 파일은 import명령문을 사용하여 다른 ES 모듈 파일을 가져와야합니다.

import './sideeffects.js' ;

import \* as goog from '../closure/goog/goog.js' ; import \* as parent from '../parent.js' ;

import { name } from './sibling.js' ;

3.3.1.1.1 import 경로의 파일 확장자

.js파일 확장자는 import경로에서 선택하지 않고 항상 포함해야합니다.

import '../directory/file' ; (x)

import '../directory/file.js' ;

3.3.1.2 동일한 파일을 여러번 import

동일한 파일을 여러번 import하지 마십시오. 이로 인해 파일의 모든 import를 결정하기가 어려울 수 있습니다.

// import는 동일한 경로를 갖지만 정렬되지 않으므로보기 어려울 수 있습니다. import { short } from './long/path/to/a/file.js' ; import { aLongNameThatBreaksAlignment } from './long/path/to/a/file.js' ;

3.3.1.3 이름 지정 import

3.3.1.3.1 이름 지정 모듈 import

모듈 import 이름 지정 ( import \* as name)은 lowerCamelCase가져온 파일 이름에서 파생된 이름입니다.

import \* as fileOne from '../file-one.js' ; import \* as fileTwo from '../file\_two.js' ; import \* as fileThree from '../filethree.js' ;

import \* as libString from './lib/string.js' ; import \* as math from './math/math.js' ; import \* as vectorMath from './vector/math.js' ;

3.3.1.3.2 기본 import 이름 지정

기본 import 이름은 가져온 파일이름에서 파생되며 6.2 규칙 별 식별자 유형 의 규칙을 따릅니다 .

import MyClass from '../my-class.js' ; import myFunction from '../my\_function.js' ; import SOME\_CONSTANT from '../someconstant.js' ;

참고 : 일반적으로이 스타일 가이드에 의해 기본 내보내기가 금지되어 있으므로 이는 발생하지 않아야합니다 . 3.4.2.1 명명 된 내보내기와 기본 내보내기를 참조하세요 . 기본 가져 오기는이 스타일 가이드를 따르지 않는 모듈을 가져 오는 데만 사용됩니다.

3.3.1.3.3 명명된 import 이름 지정

일반적으로 명명된 import ( import {name}) 를 통해 가져온 심볼은 동일한 이름을 유지해야합니다. import 앨리어싱 ( import {SomeThing as SomeOtherThing})을 피하십시오 . 모듈 가져 오기 ( import \*)를 사용하거나 내보내기 자체의 이름을 변경 하여 이름 충돌을 수정하는 것을 선호합니다.

import \* as bigAnimals from './biganimals.js' ; import \* as domesticatedAnimals from './domesticatedanimals.js' ;

new bigAnimals.cat ();

new domesticatedAnimals.cat();

명명 된 가져 오기의 이름을 변경해야하는 경우 가져온 모듈의 파일 이름 또는 결과 별칭에서 경로의 구성 요소를 사용합니다.

import { Cat as BigCat } from './biganimals.js' ; import { Cat as DomesticatedCat } from './domesticatedanimals.js' ;

new BigCat ();

new DomesticatedCat ();

3.3.2 export

심볼은 모듈 외부에서 사용되는 경우에만 내보내집니다. 내보내지 않은 모듈 로컬 기호는 선언 @private되지 않으며 이름이 밑줄로 끝나지 않습니다. 내보낸 기호 및 모듈 로컬 기호에 대해 규정 된 순서는 없습니다.

3.3.2.1 named exports vs default exports

모든 코드에서 named exports 사용합니다. export선언에 키워드를 적용 하거나 export {name};구문을 사용할 수 있습니다.

default exports를 사용하지 마십시오. 모듈을 가져오면 이러한 값에 이름을 지정해야하므로 모듈 간의 이름 지정에 불일치가 발생할 수 있습니다.

// default exports 사용 안 함 :

export default class Foo { ... } // BAD!

// named exports 사용 :

export class Foo { ... }

// exports라는 대체 스타일 :

class Foo { ... }

export { Foo };

3.3.2.2 정적 컨테이너 클래스 및 개체 내보내기

네임 스페이스를 위해 컨테이너 클래스 또는 정적 메서드 또는 속성이있는 개체를 내보내지 마십시오.

// container.js

// Bad : Container는 정적 메서드와 필드만 있는 export 클래스입니다.

export class Container {

/ \*\* @return {number} \* /

static bar () {

return 1 ;

}

}

/ \*\* @const {number} \* /

Container. FOO = 1 ;

대신 개별 상수 및 함수를 내보내십시오.

/ \*\* @return {number} \* /

export function bar () {

return 1 ;

}

export const / \*\* 숫자 \* / FOO = 1 ;

3.3.2.3 export의 가변성

내보낸 변수는 모듈 초기화 외부에서 변경해서는 안됩니다.

변경이 필요한 경우 변경 가능한 필드가있는 개체에 대한 상수 참조를 내보내거나 변경 가능한 데이터에 대한 접근자 함수를 내보내는 등의 대안이 있습니다.

// Bad : foo와 mutateFoo가 모두 내보내지고 변경됩니다.

export let / \*\* number \* / foo = 0 ;

/ \*\*

\* foo를 변형합니다.

\* /

export function mutateFoo () {

++ foo ;

}

/ \*\*

\* @param {function (number) : number} newMutateFoo

\* /

export function setMutateFoo ( newMutateFoo ) {

// 내보낸 클래스와 함수를 변경할 수 있습니다!

mutateFoo = () => {

foo = newMutateFoo ( foo );

};

}

// 좋음 : 변경 가능한 변수 foo 및 mutateFoo를 직접 내보내는 대신

// 모듈 범위를 지정하고 foo에 대한 getter와 mutateFooFunc에 대한 래퍼를 // 내 보냅니다. let / \*\* number \* / foo = 0 ;

let / \*\* function (number) : number \* / mutateFooFunc = foo => foo + 1 ;

/ \*\* @return {number} \* / export function getFoo () { return foo ; }

export function mutateFoo () {

foo = mutateFooFunc ( foo );

}

/ \*\* @param {function (number) : number} mutateFoo \* /

export function setMutateFoo ( mutateFoo ) {

mutateFooFunc = mutateFoo ;

}

3.3.2.4 export from

export from문은 줄 바꿈되어서는 안되며 따라서 80 열 제한에 대한 예외입니다. 이것은 두 export from가지 맛 에 모두 적용됩니다 .

export {specificName} from './other.js';

export \* from './another.js';

3.3.3 ES 모듈의 순환 종속성

ECMAScript 사양이이를 허용하더라도 ES 모듈간에주기를 생성하지 마십시오. import및 export문 을 모두 사용하여주기를 만들 수 있습니다 .

// a.js

import './b.js' ;

// b.js

import './a.js' ;

// ʻexport from`도 순환 종속성을 유발할 수 있습니다! '

export {x} from './c.js';

// c.js import './b.js' ;

export let x ;

**3.4 파일의 구현**

실제 구현은 모든 종속성 정보가 선언 된 후에 수행됩니다 (최소 하나의 빈 줄로 구분).

이는 모듈 로컬 선언 (상수, 변수, 클래스, 함수 등)과 내 보낸 기호로 구성 될 수 있습니다.

**4 서식**

용어 참고 : 블록형 구조는 클래스, 함수, 메서드 또는 중괄호로 구분된 코드 블록의 본문을 나타냅니다. 하여, 그 중 5.2 배열 리터럴 및 5.3 개체 리터럴은 블록 형 구조인 것처럼, 어떤 어레이 또는 객체 리터럴 임의로 처리 될 수있다.

**4.1 중괄호(brace)**

4.1.1 모든 제어 구조에 **brace**가 사용됩니다.

**brace**(즉 모든 제어 구조에 필요한 if, else, for, do, while,뿐만 아니라 다른 사람), 주요 부분이 단 하나의 문으로 이루어졌을 경우에도 마찬가지입니다. 비어 있지 않은 블록의 첫번째 문은 새로운 줄에서 시작해야합니다.

허용되지 않음 :

if ( someVeryLongCondition ())

doSomething ();

for ( let i = 0 ; i < foo . length ; i ++) bar ( foo [ i ]);

예외 : 줄 바꿈없이 한 줄에 완전히 맞출 수 있고 else가없는 간단한 if 문은 가독성을 향상시킬 때 중괄호없이 한 줄에 보관할 수 있습니다. 이것은 제어 구조가 중괄호와 줄 바꿈을 생략 할 수있는 유일한 경우입니다.

if ( shortCondition ()) foo ();

4.1.2 비어 있지 않은 블록 : K & R 스타일

중괄호는 비어 있지 않은 블록 및 블록 유사 구성에 대해 Kernighan 및 Ritchie 스타일 (이집트 괄호)을 따릅니다.

- 여는 중괄호 앞에 줄 바꿈이 없습니다.

- 여는 중괄호 뒤의 줄 바꿈.

- 닫는 중괄호 앞의 줄 바꿈.

- 해당 중괄호가 명령문이나 함수 또는 클래스 문의 본문 또는 클래스 메서드를 종료하는 경우 닫는 중괄호 뒤의 줄 바꿈 . 구체적 없다 더 가 하였다 경우 브레이스 후 개행 else, catch, while, 또는 콤마, 콜론, 또는 오른쪽 괄호.

예:

class InnerClass {

constructor() {}

/\*\* @param {number} foo \*/

method(foo) {

if (condition(foo)) {

try {

// Note: this might fail.

something();

} catch (err) {

recover();

}

}

}

}

4.1.3 빈 블록 : 간결 할 수 있음

빈 블록 또는 블록과 유사한 구조는 문자나 공백 줄바꿈이 없을시에 바로 닫을 수 있습니다.

(if / else / try / catch /finally)

예:

function doNothing () {}

허용되지 않음 :

if ( condition ) {

//…

} else if ( otherCondition ) {} else {

//…

}

try {

// …

} catch (e) {}

**4.2 블록 들여 쓰기 : +2 공백**

새 블록 또는 블록과 유사한 구조가 열릴 때마다 들여 쓰기가 두 칸씩 증가합니다. 블록이 끝나면 들여 쓰기는 이전 들여 쓰기 수준으로 돌아갑니다. 들여 쓰기 수준은 블록 전체의 코드와 주석 모두에 적용됩니다. ( 4.1.2 비어 있지 않은 블록 : K & R 스타일 의 예 참조 ).

4.2.1 배열 리터럴 : 선택적으로 블록 유사

모든 배열 리터럴은 "블록과 유사한 구조"인 것처럼 선택적으로 형식화 될 수 있습니다

ex)

const a = [

0 ,

1 ,

2 ,

];

const b =

[ 0 , 1 , 2 ];

const c = [ 0 , 1 , 2 ];

someMethod ( foo , [

0 , 1 , 2 ,

], bar );

특히 요소 간의 의미 그룹화를 강조 할 때 다른 조합이 허용되지만 더 큰 배열의 수직 크기를 줄이는 데만 사용해서는 안됩니다.

4.2.2 객체 리터럴 : 선택적으로 블록 유사

모든 객체 리터럴은 선택적으로 "블록 유사 구조"인 것처럼 형식화 될 수 있습니다. 동일한 예가 4.2.1 배열 리터럴에 적용됩니다 : 선택적으로 block-like . 예를 들어, 다음 (모든 유효 하지 완전한 목록) :

const a = {

a : 0 ,

b : 1 ,

};

const b =

{ a : 0 , b : 1 };

const c = { a : 0 , b : 1 };

someMethod ( foo , {

a : 0 , b : 1 ,

}, bar );

4.2.3 클래스 리터럴

클래스 리터럴 (선언이든 표현식이든)은 블록으로 들여 쓰기됩니다. 메서드 뒤에 또는 클래스 선언 의 닫는 중괄호 뒤에 세미콜론을 추가하지 마십시오 (예 : 클래스 식 을 포함하는문 은 여전히 ​​세미콜론으로 종료됩니다). extends키워드를 사용 하되 @extends클래스가 템플릿화 된 유형을 확장하지 않는 한 JSDoc 주석은 사용하지 마십시오 .

예:

class Foo {

constructor() {

/\*\* @type {number} \*/

this.x = 42;

}

/\*\* @return {number} \*/

method() {

return this.x;

}

}

Foo.Empty = class {};

/\*\* @extends {Foo<string>} \*/

foo.Bar = class extends Foo {

/\*\* @override \*/

method() {

return super.method() / 2;

}

};

/\*\* @interface \*/

class Frobnicator {

/\*\* @param {string} message \*/

frobnicate(message) {}

}

4.2.4 함수 표현식

함수 호출에 대한 인수 목록에서 익명 함수를 선언 할 때 함수 본문은 이전 들여 쓰기 깊이보다 두 칸 더 들여 쓰기됩니다.

예:

prefix.something.reallyLongFunctionName('whatever', (a1, a2) => {

// Indent the function body +2 relative to indentation depth

// of the 'prefix' statement one line above.

if (a1.equals(a2)) {

someOtherLongFunctionName(a1);

} else {

andNowForSomethingCompletelyDifferent(a2.parrot);

}

});

some.reallyLongFunctionCall(arg1, arg2, arg3)

.thatsWrapped()

.then((result) => {

// Indent the function body +2 relative to the indentation depth

// of the '.then()' call.

if (result) {

result.use();

}

});

4.2.5 Switch 문

다른 블록과 마찬가지로 스위치 블록의 내용은 +2로 들여 쓰기됩니다.

스위치 레이블 뒤에 개행 문자가 나타나고 블록이 열려있는 것처럼 들여 쓰기 수준이 +2로 증가합니다. 어휘 범위 지정에 필요한 경우 명시 적 블록을 사용할 수 있습니다. 다음 스위치 레이블은 마치 블록이 닫힌 것처럼 이전 들여 쓰기 수준으로 돌아갑니다.

break와 다음 대소 문자 사이에 빈 줄은 선택 사항 입니다.

예:

switch (animal) {

case Animal.BANDERSNATCH:

handleBandersnatch();

break;

case Animal.JABBERWOCK:

handleJabberwock();

break;

default:

throw new Error('Unknown animal');

}

**4.3 Statements**

4.3.1 한 줄에 하나의 문

각 문 뒤에는 줄 바꿈이 있습니다.

4.3.2 세미콜론이 필요합니다.

모든 문은 세미콜론으로 종료되어야합니다. 자동 세미콜론 삽입에 의존하는 것은 금지되어 있습니다.

**4.4 열 제한 : 80**

JavaScript 코드의 열 제한은 80 자입니다. 아래에 명시된 경우를 제외하고이 제한을 초과하는 모든 줄은 4.5 줄 바꿈에 설명 된대로 줄 바꿈해야합니다 .

예외 :

1. ES 모듈 import및 export from명령문 (에서 3.4.1 가져 오기 및 3.4.2.4 내보내기 참조 ).

2. 열 제한을 준수 할 수 없거나 검색 가능성을 방해하는 줄입니다. 예는 다음과 같습니다.

- 소스에서 클릭 할 수 있어야하는 긴 URL입니다.

- 복사하여 붙여 넣을 셸 명령입니다.

- 전체적으로 복사하거나 검색해야하는 긴 문자열 리터럴 (예 : 긴 파일 경로).

**4.5 Line-wrapping**

용어 참고 : Line-wrapping은 열 제한을 준수하기 위해 코드 청크를 여러 줄로 나누는 것입니다. 그렇지 않으면 청크가 한 줄에 합법적으로 들어갈 수 있습니다.

모든 상황에서 줄 바꿈하는 방법을 정확히 보여주는 포괄적이고 결정적인 공식은 없습니다 . 동일한 코드를 줄 바꿈하는 여러 가지 유효한 방법이 매우 자주 있습니다.

참고 : 줄 바꿈의 일반적인 이유는 열 제한이 넘치지 않도록하는 것이지만 실제로 열 제한에 맞는 코드도 작성자의 재량에 따라 줄 바꿈 될 수 있습니다.

팁 : 메서드 또는 지역 변수를 추출하면 줄 바꿈없이 문제를 해결할 수 있습니다.

4.5.1 어디서 끊어야 하는지

줄 바꿈의 주요 지침은 더 높은 구문 수준 에서 중단하는 것을 선호하는 것 입니다.

선호 :

currentEstimate =

calc(currentEstimate + x \* currentEstimate) /

2.0;

비선호 :

currentEstimate = calc(currentEstimate + x \*

currentEstimate) / 2.0;

앞의 예에서 가장 높은 것부터 가장 낮은 것까지의 구문 수준은 할당, 나누기, 함수 호출, 매개 변수, 숫자 상수입니다.

연산자는 다음과 같이 래핑됩니다.

1. 연산자에서 선이 끊어지면 기호 뒤에 끊어집니다. (이것은 Java 용 Google 스타일에서 사용되는 것과 동일한 관행이 아닙니다.) 실제로 연산자가 아닌 점 ( .) 에는 적용되지 않습니다 .

2. 메서드 또는 생성자 이름은 그 뒤에 오는 여는 괄호 ( ()에 연결된 상태로 유지 됩니다.

3. 쉼표 ( ,)는 그 앞에있는 토큰에 계속 붙어 있습니다.

참고 : 줄 바꿈의 기본 목표는 최소한의 줄에 맞는 코드가 아니라 명확한 코드를 만드는 것입니다.

4.5.2 연속 줄을 최소 +4 공백 들여 쓰기

줄 바꿈시 첫 번째 줄 (각 연속 줄 ) 이후의 각 줄 은 블록 들여 쓰기 규칙에 속하지 않는 한 원래 줄에서 최소한 +4만큼 들여 쓰기됩니다.

연속 행이 여러 개있는 경우 들여 쓰기는 적절하게 +4 이상으로 변경 될 수 있습니다. 일반적으로 더 깊은 구문 수준의 연속 줄은 더 큰 4의 배수로 들여 쓰기되며 두 줄은 구문 적으로 평행 한 요소로 시작하는 경우에만 동일한 들여 쓰기 수준을 사용합니다.

4.6.3 수평 정렬 : disouraged는 특정 토큰을 이전 줄과 정렬하기 위해 다양한 수의 공백을 사용하는 권장되지 않는 관행을 해결합니다.

4.6 공백

4.6.1 세로 공백

하나의 빈 줄이 나타납니다.

클래스 또는 개체 리터럴의 연속 메서드 간

예외 : 개체 리터럴에서 두 개의 연속 속성 정의 사이의 빈 줄 (사이에 다른 코드 없음)은 선택 사항입니다. 이러한 빈 줄은 필요에 따라 필드의 논리적 그룹 을 만드는 데 사용됩니다 .

메서드 본문 내에서 문을 논리적으로 그룹화 하는 일이 거의 없습니다 . 함수 본문의 시작 또는 끝에 빈 줄이 허용되지 않습니다.

선택적 으로 클래스 또는 객체 리터럴의 첫 번째 또는 마지막 메서드 앞 또는 뒤 (권장되거나 권장되지 않음).

이 문서의 다른 섹션에서 요구하는대로 (예 : 3.6 goog.require 및 goog.requireType 문 ).

여러 개의 연속 된 빈 줄이 허용되지만 필수 사항은 아닙니다 (권장되지 않음).

4.6.2 수평 공백

수평 공백의 사용은 위치에 따라 다르며 행간 (행 시작), 후행 (행 끝) 및 내부의 세 가지 범주로 분류됩니다 . 선행 공백 (즉, 들여 쓰기)은 다른 곳에서 처리됩니다. 후행 공백은 금지됩니다.

언어 또는 기타 스타일 규칙에서 요구하는 곳을 넘어, 리터럴, 주석 및 JSDoc을 제외하고 단일 내부 ASCII 공간도 다음 위치 에만 나타납니다 .

어떤 예약어 (같은 분리 if, for또는 catch제외)을 function하고 super열린 괄호 (에서, (그 라인을 다음).

해당 행에서 앞에 오는 닫는 중괄호 ( ) 에서 예약어 (예 : else또는 catch)를 분리합니다 }.

여는 중괄호 ( {) 앞 , 두 가지 예외 :

함수의 첫 번째 인수 또는 배열 리터럴의 첫 번째 요소 인 객체 리터럴 앞 (예 :) foo({a: [{c: d}]}).

템플릿 확장에서 언어에 의해 금지되어 있습니다 (예 : valid : `ab${1 + 2}cd`, invalid :) `xy$ {3}z`.

이항 또는 삼항 연산자의 양쪽에 있습니다.

쉼표 ( ,) 또는 세미콜론 ( ;) 뒤 . 이러한 문자 앞에는 공백이 허용 되지 않습니다.

:개체 리터럴에서 콜론 ( ) 뒤 .

//줄 끝 주석을 시작하는 이중 슬래시 ( ) 의 양쪽 . 여기에서는 여러 개의 공백이 허용되지만 필수는 아닙니다.

오픈 블록 주석 문자 뒤와 닫는 문자의 양쪽에 있습니다 (예 : 짧은 형식 선언, 캐스트 및 매개 변수 이름 주석의 경우 : this.foo = /\*\* @type {number} \*/ (bar); 또는 function(/\*\* string \*/ foo) {; 또는 baz(/\* buzz= \*/ true)).

4.6.3 수평 정렬 : 권장하지 않음

용어 참고 : 수평 정렬 은 특정 토큰이 이전 줄의 다른 특정 토큰 바로 아래에 표시되도록 코드에 다양한 수의 추가 공백을 추가하는 방법입니다.

이 관행은 허용되지만 일반적으로 Google 스타일에서는 권장하지 않습니다 . 이미 사용 된 장소에서 수평 정렬 을 유지할 필요조차 없습니다 .

다음은 정렬이없는 예제와 정렬이있는 예제입니다. 둘 다 허용되지만 후자는 권장되지 않습니다.

{

tiny : 42 , // 이건

더 길어요 : 435 , // 이것도 };

{

tiny : 42 , // 허용되지만 향후 수정이

더 길어짐 : 435 , // 정렬되지 않은 상태로 둘 수 있음 };

팁 : 정렬은 가독성에 도움이 될 수 있지만 향후 유지 관리에 문제가됩니다. 한 줄만 터치하면되는 미래의 변화를 생각해보십시오. 이 변경으로 이전에 만족 스러웠던 서식이 망가질 수 있으며 허용됩니다. 더 자주 코더 (아마도 사용자)에게 근처 줄의 공백을 조정하라는 메시지를 표시하여 계단식 일련의 재 포맷을 트리거 할 수 있습니다. 이제 그 단선 변경에는 폭발 반경이 있습니다. 이것은 최악의 경우 무의미한 작업을 초래할 수 있지만 기껏해야 버전 기록 정보를 손상시키고 검토 자의 속도를 늦추고 병합 충돌을 악화시킵니다.

4.6.4 함수 인수

모든 함수 인수를 함수 이름과 같은 줄에 배치하는 것이 좋습니다. 이렇게하면 80 열 제한을 초과하는 경우 인수를 읽을 수있는 방식으로 줄 바꿈해야합니다. 공간을 절약하기 위해 가능한 한 80 개에 가깝게 줄 바꿈하거나 가독성을 높이기 위해 각 인수를 자체 행에 배치 할 수 있습니다. 들여 쓰기는 공백 4 개 여야합니다. 괄호에 맞추는 것은 허용되지만 권장되지 않습니다. 다음은 인수 래핑의 가장 일반적인 패턴입니다.

// 인수는 새 줄에서 시작하며 공백 4 개를 들여 씁니다. // 인수가 함수 이름 (또는 키워드 // "함수") 과 같은 줄에 맞지 않지만 두 번째 줄에 완전히 맞을 때 선호됩니다 . // 매우 긴 함수 이름으로 작동하고 , 다시 들여 쓰기없이 이름을 변경해도 유지되며 공간이 부족합니다.

doSomething (

descriptiveArgumentOne , descriptiveArgumentTwo , descriptiveArgumentThree ) { //… }

// 인수 목록이 더 길면 80으로 감 쌉니다. 수직 공간을 덜 사용 하지만 // 직사각형 규칙을 위반하므로 권장되지 않습니다.

doSomething ( veryDescriptiveArgumentNumberOne , veryDescriptiveArgumentTwo ,

tableModelEventHandlerProxy , artichokeDescriptorAdapterIterator ) { //… }

// 네 공백, 한 줄에 하나의 인수. 긴 함수 이름과 함께 작동하고 // 이름이 바뀌더라도 각 인수를 강조합니다.

doSomething (

veryDescriptiveArgumentNumberOne ,

veryDescriptiveArgumentTwo ,

tableModelEventHandlerProxy ,

artichokeDescriptorAdapterIterator ) { //… }

4.7 그룹화 괄호 : 권장

선택적 그룹화 괄호는 작성자와 검토자가 코드가 없으면 코드가 잘못 해석 될 가능성이 없으며 코드를 읽기 쉽게 만들지 않았다는 데 동의하는 경우에만 생략됩니다. 이다 없는 모든 독자가 기억 전체 연산자 우선 순위 테이블이 있다고 가정하는 것이 합리적.

전체 표현식 주위에 불필요한 괄호를 사용하지 마십시오 다음 delete, typeof, void, return, throw, case, in, of, 또는 yield.

유형 캐스트에는 괄호가 필요합니다 /\*\* @type {!Foo} \*/ (foo)..

4.8 댓글

이 섹션에서는 구현 의견을 다룹니다 . JSDoc은 7 JSDoc 에서 별도로 처리됩니다 .

4.8.1 블록 주석 스타일

블록 주석은 주변 코드와 동일한 수준에서 들여 쓰기됩니다. 그들은 /\* … \*/또는 //스타일 일 수 있습니다 . 여러 줄 /\* … \*/주석의 경우 \*추가 컨텍스트없이 주석을 명확하게하기 위해 다음 줄은 이전 줄에 정렬 된 \*로 시작해야합니다 .

/ \*

\* 이것은

\* 괜찮아.

\* /

// 그래서 // 이것입니다.

/ \* 이것도 괜찮습니다. \* /

주석은 별표 또는 기타 문자로 그려진 상자에 포함되지 않습니다.

/\*\* … \*/구현 주석에 JSDoc ( )을 사용하지 마십시오 .

4.8.2 매개 변수 이름 설명

"매개 변수 이름"주석은 값과 메서드 이름이 의미를 충분히 전달하지 못할 때마다 사용해야하며 메서드를 더 명확하게 리팩토링하는 것이 불가능합니다. 선호하는 형식은 = 값 앞에 있습니다 .

someFunction ( obviousParam , / \* shouldRender = \* / true , / \* name = \* / 'hello' );

주변 코드와의 일관성을 위해 = 없는 값 뒤에 넣을 수 있습니다 .

someFunction ( obviousParam , true / \* shouldRender \* / , 'hello' / \* name \* / );