**JavaScript 스타일 가이드**

**목차**

**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 goog.module 문

3.3.3 goog.module 내보내기

3.4 ES 모듈

3.5 goog.setTestOnly

3.6 goog.require 및 goog.requireType 문

3.7 파일의 구현

**4 서식**

4.1 교정기

4.2 블록 들여쓰기: +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

5.9이

5.10 평등 검사

5.11 허용되지 않는 기능

**6 명명**

6.1 모든 식별자에 공통적 인 규칙

6.2 식별자 유형별 규칙

6.3 카멜 케이스: 정의됨

**7 JSDoc**

7.1 일반 형식

7.2 마크 다운

7.3 JSDoc 태그

7.4 줄 바꿈

7.5 최상위/파일 수준 주석

7.6 수업 댓글

7.7 열거 형 및 typedef 주석

7.8 방법 및 기능 설명

7.9 부동산 설명

7.10 유형 주석

7.11 가시성 주석

**8 정책**

8.1 Google 스타일에서 지정하지 않은 문제: 일관성 유지!

8.2 컴파일러 경고

8.3 지원 중단

8.4 Google 스타일에 없는 코드

8.5 지역 스타일 규칙

8.6 생성된 코드: 대부분 면제

**9 부록**

9.1 JSDoc 태그 참조

9.2 일반적으로 오해되는 스타일 규칙

9.3 스타일 관련 도구

9.4 레거시 플랫폼의 예외

**1. 소개**

이 문서는 JavaScript 프로그래밍 언어의 소스 코드에 대한 코딩 표준에 대한 완전한 정의 역할을 합니다. JavaScript 소스 파일은 여기에 있는 규칙을 준수하는 경우에만 설명됩니다.

다른 프로그래밍 스타일 가이드와 마찬가지로 다루는 문제는 형식화의 미적 문제뿐 아니라 다른 유형의 규칙이나 코딩 표준에도 적용됩니다. 그러나이 문서는 주로 우리가 보편적으로 따르는 엄격하고 빠른 규칙에 초점을 맞추고 명확하게 시행할 수 없는 조언 (인간 또는 도구)을 제공하지 않습니다.

**1.1 용어 참고**

이 문서에서 달리 명시되지 않는 한:

1. 주석이라는 용어는 항상 구현 주석을 의미합니다. 문서 주석이라는 문구를 사용하지 않고 대신 사람이 읽을 수 있는 텍스트에서 컴퓨터가 읽을 수 있는 주석 모두에 대해 공통 용어 'JSDoc'를 사용합니다/\*\* … \*/.

2. 이 스타일 가이드에서는 must, must not, should, should not, may 라는 문구를 사용할 때 RFC 2119 용어를 사용합니다. 조건은 선호 하고 회피 대응 해야 하고 해야하지 , 각각. 명령문 및 선언문은 규범적이며 must에 해당합니다.

문서 전체에 다른 용어 참고 사항 이 가끔 표시됩니다.

**1.2 가이드 노트**

이 문서의 예제 코드는 표준이 아닙니다. 즉, 예제는 Google 스타일이지만 코드를 표현하는 유일한 세련된 방법을 설명하지 못할 수 있습니다. 예제에서 선택한 선택적 형식 지정은 규칙으로 적용되지 않아야 합니다.

**2 소스 파일 기본 사항**

**2.1 파일 이름**

파일 이름은 모두 소문자이어야 하고 밑줄 (\_) 또는 대시 (-)를 포함할 수 있지만 추가 구두점은 없습니다. 프로젝트에서 사용하는 규칙을 따르십시오. 파일 이름의 확장자는 .js.

**2.2 파일 인코딩: UTF-8**

소스 파일은 UTF-8 로 인코딩합니다.

**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

1. 문자열 리터럴의 다른 모든 공백 문자는 이스케이프됩니다.

2. 탭 문자는 들여 쓰기에 사용되지 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 유니코드 문자 (예 ∞:) 또는 동등한 16 진수 또는 유니코드 이스케이프 (예 :) \u221e가 사용됩니다. 이는 코드를 더 쉽게 읽고 이해할 수 있도록 합니다.

팁: 유니코드 이스케이프 케이스 및 실제 유니코드 문자가 사용되는 경우에도 설명 주석이 매우 유용할 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 인쇄할 수 없는 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 중단되고 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 다음과 같은 순서 로 구성됩니다.

1. 라이센스 또는 저작권 정보 (있는 경우)

2. @fileoverview JSDoc (있는 경우)

3. goog.module문, goog.module파일 인 경우

4. ES import문 (ES 모듈 인 경우)

5. goog.require및 goog.requireType진술

6. 파일의 구현

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

**3.1 라이선스 또는 저작권 정보 (있는 경우)**

라이센스 또는 저작권 정보가 파일에 속하면 여기에 속합니다.

**3.2 @fileoverviewJSDoc (있는 경우)**

서식 규칙 은 7.5 최상위/파일 수준 주석을 참조하십시오.

**3.3 goog.module성명**

모든 goog.module파일은 goog.module한 줄에 정확히 하나의 이름을 goog.module선언해야합니다 . 선언을 포함하는 줄은 래핑해서는 안되므로 80 열 제한에 대한 예외입니다.

goog.module에 대한 전체 인수는 네임 스페이스를 정의합니다. 패키지 이름 (코드가있는 디렉토리 구조의 조각을 반영하는 식별자)과 선택적으로 끝까지 연결되어 정의하는 기본 클래스/열거 형 / 인터페이스입니다.

예

goog. 모듈 ('search.urlHistory.UrlHistoryService');

**3.3.1 계층**

모듈 네임 스페이스는 다른 모듈 네임 스페이스의 직접 자식으로 이름을 지정할 수 없습니다.

허용되지 않음:

goog. 모듈 ('foo.bar');//'foo.bar.qux'는 괜찮지만

goog. 모듈 ('foo.bar.baz’);

디렉터리 계층 구조는 네임 스페이스 계층 구조를 반영하므로 더 깊은 중첩된 자식은 상위 수준 부모 디렉터리의 하위 디렉터리입니다. 이는 "부모"네임 스페이스 그룹의 소유자가 동일한 디렉토리에 존재하기 때문에 반드시 모든 자식 네임 스페이스를 인식하고 있음을 의미합니다.

**3.3.2 goog.module.declareLegacyNamespace**

단일 goog.module명령문 뒤에 선택적으로를 호출 할 수 goog.module.declareLegacyNamespace();있습니다. goog.module.declareLegacyNamespace()가능하면 피하십시오.

예:

goog. 모듈 ('my.test.helpers');

goog. 모듈. declareLegacyNamespace ();

goog. setTestOnly ();

goog.module.declareLegacyNamespace기존의 개체 계층 기반 네임 스페이스에서 쉽게 전환할 수 있도록 존재하지만 일부 명명 제한이 있습니다. 아이 모듈 이름이 부모 네임 스페이스 후 작성해야합니다, 이 이름은 안 다른의 자식 또는 부모 goog.module(예를 들어, goog.module('parent');하고 goog.module('parent.child');안전하게 할 수 없는 두 존재도 수 goog.module('parent');와 goog.module('parent.child.grandchild');).

**3.3.3 goog.module export**

클래스, 열거 형, 함수, 상수 및 기타 기호는 exports객체를 사용하여 내보내집니다. 내보낸 심볼은 exports 객체에서 직접 정의하거나 로컬로 선언하고 별도로 내보낼 수 있습니다. 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 규정된 순서는 없습니다.

예:

const / \*\*! Array <number> \* / exportArray = [1, 2, 3];

const / \*\*! Array <번호>\* / moduleLocalArray = [4, 5, 6];

/ \*\* @return {number} \* / function moduleLocalFunction () {return moduleLocalArray. 길이;}

/ \*\* @return {NUMBER} \* / 기능 exportedFunction () { 반환 moduleLocalFunction () \* 2 ; }

exports = {exportArray, exportFunction};

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

내보내기. CONSTANT\_ONE = 1;

/ \*\* @const {string} \* /

exports. CONSTANT\_TWO = '다른 상수';

컴파일러에서 이미 상수로 처리되었으므로 exports객체에 주석을 달지 마십시오 @const.

/ \*\* @const \* /

수출 = {exportedFunction};

**3.4 ES 모듈**

**3.4.1 import**

가져오기 문은 줄 바꿈되어서는 안되므로 80 열 제한에 대한 예외입니다.

**3.4.1.1 가져오기 경로**

ES 모듈 파일은 import명령문을 사용하여 다른 ES 모듈 파일을 가져와야합니다. goog.require다른 ES 모듈을 사용 하지 마십시오 .

import './sideeffects.js';

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

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

**3.4.1.1.1 가져오기 경로의 파일 확장자**

.js파일 확장자는 수입 경로에서 선택하지 않고 항상 포함해야합니다.

import '../directory/file';

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

**3.4.1.2 동일한 파일을 여러 번 가져오기**

동일한 파일을 여러 번 가져오지 마십시오. 이로 인해 파일의 총 가져오기를 결정하기가 어려울 수 있습니다.

//가져오기는 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';

**3.4.1.3 이름 지정 가져오기**

**3.4.1.3.1 이름 지정 모듈 가져오기**

모듈 가져오기 이름 (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';

**3.4.1.3.2 기본 가져오기 이름 지정**

기본 가져오기 이름은 가져온 파일 이름에서 파생되며 6.2 규칙 별 식별자 유형의 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 명명된 내보내기와 기본 내보내기를 참조하세요. 기본 가져 오기는이 스타일 가이드를 따르지 않는 모듈을 가져오는 데만 사용됩니다.

**3.4.1.3.3 명명된 가져오기 이름 지정**

일반적으로 명명된 가져오기 (import {name})를 통해 가져온 심볼 은 동일한 이름을 유지해야합니다. 가져오기 별칭 지정 (import {SomeThing as SomeOtherThing})을 피하십시오. 모듈 가져오기 (import \*)를 사용하거나 내보내기 자체의 이름을 변경하여 이름 충돌을 수정하는 것을 선호합니다.

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

새로운 bigAnimals. 고양이 (); new domesticatedAnimals. 고양이 ();

명명된 가져오기의 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';

새로운 BigCat (); 새로운 DomesticatedCat ();

**3.4.2 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 규정된 순서는 없습니다.

3.4.2.1 명명된 내보내기와 기본 내보내기

모든 코드에서 명명된 내보내기를 사용합니다. export선언에 키워드를 적용하거나 export {name}; 구문을 사용할 수 있습니다.

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

//기본 내보내기 사용 안 함: export default class Foo {...}// BAD!

//명명된 내보내기 사용: export class Foo {...}

//exports라는 대체 스타일: class Foo {...}

수출 {Foo};

**3.4.2.2 정적 컨테이너 클래스 및 개체 내보내기**

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

//container.js// 나쁨: 컨테이너는 정적 메서드와 필드만 있는 내보낸 클래스입니다. export class Container {/ \*\* @return {number} \* / static bar () {return 1;}}

/\*\* @const {number} \*/ 컨테이너. FOO = 1;

대신 개별 상수 및 함수를 내보내십시오.

/ \*\* @return {number} \* / export function bar () {return 1;}

내보내기 const / \*\* 숫자 \* / FOO = 1 ;

**3.4.2.3 수출의 가변성**

내보낸 변수는 모듈 초기화 외부에서 변경해서는 안됩니다.

변경이 필요한 경우 변경 가능한 필드가 있는 개체에 대한 상수 참조를 내보내거나 변경 가능한 데이터에 대한 접근 자 함수를 내보내는 등의 대안이 있습니다.

//나쁨 : foo와 mutateFoo가 모두 내보내지고 변경됩니다. export let / \*\* number \* / foo = 0 ;

/ \*\*

\* foo를 변형합니다.

\* / export function mutateFoo () { ++ foo ; }

/ \*\*

\* @param {함수 (숫자) : 숫자} newMutateFoo

\* / export function setMutateFoo ( newMutateFoo ) { // 내 보낸 클래스와 함수를 변경할 수 있습니다!

mutateFoo = () => {

foo = newMutateFoo ( foo ); }; }

// 좋음 : 가변 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 ; }

내보내기 함수 mutateFoo () {

foo = mutateFooFunc ( foo ); }

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

mutateFooFunc = mutateFoo ; }

**3.4.2.4 내보내기**

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

'./other.js'에서 {specificName} 내보내기;

'./another.js'에서 \* 내보내기;

**3.4.3 ES 모듈의 순환 종속성**

ECMAScript 사양이 이를 허용하더라도 ES 모듈간에 주기를 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! './c.js' 에서 { x } 내보내기 ;

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

내보내기 let x ;

**3.4.4 클로저와의 상호 운용**

**3.4.4.1 goog 참조**

Closure goog네임 스페이스 를 참조하려면 Closure의 goog.js.

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

const 이름 = goog . 필요 ( 'a.name' );

export const CONSTANT = name . 계산 ();

goog.jsgoogES 모듈에서 사용할 수있는 전역 속성의 하위 집합 만 내 보냅니다 .

**3.4.4.2 ES 모듈의 goog.require**

goog.requireES 모듈에서는 goog.module파일 에서와 같이 작동 합니다. 당신은 (즉, 기호가 만든 폐쇄 공간 기호 요구할 수 있습니다 goog.provide또는 goog.module)과 goog.require값을 반환됩니다.

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

const GoogPromise = goog . 필요 ( 'goog.Promise' ); const myNamespace = goog . 필요 ( 'my.namespace' );

**3.4.4.3 ES 모듈에서 클로저 모듈 ID 선언**

goog.declareModuleIdES 모듈 내에서 goog.module유사한 모듈 ID 를 선언하는 데 사용할 수 있습니다 . 이는이 모듈 ID goog.require가를 호출하지 않은 a 인 것처럼 d, goog.module.getd, goog.forwardDeclare'd 등이 될 수 있음을 의미합니다 . 전역 적으로 사용 가능한 JavaScript 심볼로 모듈 ID를 생성하지 않습니다.goog.modulegoog.module.declareLegacyNamespace

의 모듈 ID에 대한 A goog.require(또는 goog.module.get) goog.declareModuleId는 항상 모듈 객체를 반환합니다 (예 : import \*'d). 따라서 to 인수 goog.declareModuleId는 항상 lowerCamelCaseName.

참고 : goog.module.declareLegacyNamespaceES 모듈에서 호출하는 것은 오류 이며 goog.module파일 에서만 호출 할 수 있습니다. 레거시 네임 스페이스를 ES 모듈과 직접 연결하는 방법은 없습니다 .

goog.declareModuleId Closure 파일을 명명 된 내보내기가 사용되는 ES 모듈로 업그레이드하는 데만 사용해야합니다.

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

goog . declareModuleId ( 'my.esm' );

내보내기 클래스 클래스 {};

**3.5 goog.setTestOnly**

A의 goog.module파일 goog.module문을 선택적으로의 호출에 의해 올 수 있습니다 goog.setTestOnly().

ES 모듈에서 import명령문 뒤에 선택적으로 goog.setTestOnly().

**3.6 goog.require및 goog.requireType진술**

가져 오기는 goog.require및 goog.requireType문 으로 수행됩니다 . goog.require명령문에서 가져온 이름 은 코드 및 유형 주석 모두에서 사용할 수있는 반면, a에서 가져온 이름 goog.requireType은 유형 주석에서만 사용할 수 있습니다.

goog.require및 goog.requireType문이없는 빈 줄 연속 블록을 형성한다. 이 블록 은 하나의 빈 줄로goog.module 구분 된 선언을 따릅니다 . goog.require또는에 대한 전체 인수 는 별도의 파일에서에 goog.requireType의해 정의 된 네임 스페이스 goog.module입니다. goog.require및 goog.requireType명령문은 파일의 다른 곳에 표시되지 않을 수 있습니다.

goog.require또는 각각 goog.requireType은 단일 상수 별칭에 할당되거나 여러 상수 별칭으로 분해됩니다. 이러한 별칭은 유형 주석 또는 코드에서 종속성을 참조 할 수있는 유일한 방법입니다. goog.require또는에 대한 인수로를 제외하고는 정규화 된 네임 스페이스를 어디에도 사용해서는 안됩니다 goog.requireType.

예외 : externs 파일에 선언 된 유형, 변수 및 함수는 유형 주석 및 코드에서 정규화 된 이름을 사용해야합니다.

별칭은 가져온 모듈 네임 스페이스의 마지막 점으로 구분 된 구성 요소와 일치해야합니다.

예외 : 경우에 따라 네임 스페이스의 추가 구성 요소를 사용하여 더 긴 별칭을 만들 수 있습니다. 결과 별칭은 해당 유형을 올바르게 식별 할 수 있도록 원래 식별자의 대 / 소문자를 유지해야합니다. 동일한 별칭을 명확하게하거나 가독성을 크게 향상시키는 경우 더 긴 별칭을 사용할 수 있습니다. 또한, 이상 별칭이 같은 기본 유형을 마스킹 방지하기 위해 사용되어야한다 Element, Event, Error, Map, 그리고 Promise(더 완전한 목록은 참조 표준 내장 객체 및 웹 API를 MDN에서). 구조화되지 않은 별칭의 이름을 바꿀 때 4.6.2 수평 공백 에서 요구하는대로 콜론 뒤에 공백이 있어야합니다 .

파일은 동일한 네임 스페이스에 대해 a goog.require와 goog.requireType문을 모두 포함해서는 안됩니다 . 가져온 이름이 코드 및 유형 주석 모두에서 사용되는 경우 단일 goog.require명령문 으로 가져와야 합니다.

부작용을 위해서만 모듈을 가져 오는 경우 호출은 a goog.require(a goog.requireType아님) 여야 하며 할당을 생략 할 수 있습니다. 이것이 필요한 이유를 설명하고 컴파일러 경고를 억제하려면 주석이 필요합니다.

행은 다음 규칙에 따라 정렬됩니다. 왼쪽에 이름이있는 모든 요구 사항이 먼저 나오고 해당 이름에 따라 알파벳순으로 정렬됩니다. 그런 다음 디스트 럭처링이 필요하며, 다시 왼쪽에있는 이름으로 정렬됩니다. 마지막으로, 독립형 요구 호출 (일반적으로 부작용을 위해 가져온 모듈 용)입니다.

도움말 :이 주문을 기억하고 수동으로 시행 할 필요가 없습니다. IDE를 사용하여 올바르게 정렬되지 않은 요구 사항을보고 할 수 있습니다.

긴 별칭 또는 모듈 이름으로 인해 행이 80 개 열 제한을 초과하는 경우 줄 바꿈 되지 않아야 합니다. require 줄은 80 개 열 제한에 대한 예외입니다.

예:

// 표준 별칭 스타일. const MyClass = goog . 필요 ( 'some.package.MyClass' ); const MyType = goog . requireType ( 'some.package.MyType' ); // 명확하게하기 위해 사용되는 네임 스페이스 기반 별칭입니다. const NsMyClass = goog . 필요 ( 'other.ns.MyClass' ); // 네이티브 유형 마스킹을 방지하는 데 사용되는 네임 스페이스 기반 별칭입니다. const RendererElement = goog . 필요 ( 'web.renderer.Element' );

// 가독성을 높이기 위해 사용되는 비 순차적 네임 스페이스 기반 별칭입니다. // 또한 80 개 이상의 열을 요구하는 줄은 래핑해서는 안됩니다. const SomeDataStructureModel = goog . requireType ( 'identical.package.identifiers.models.SomeDataStructure' ); const SomeDataStructureProto = goog . 필요 ( 'proto.identical.package.identifiers.SomeDataStructure' ); // 표준 별칭 스타일. const asserts = goog . 필요 ( 'goog.asserts' ); // 명확하게하기 위해 사용되는 네임 스페이스 기반 별칭입니다.

const testingAsserts = goog . 필요 ( 'goog.testing.asserts' ); // 별칭으로 표준 분해. const { clear , clone } = goog . 필요 ( 'goog.array' ); const { Rgb } = goog . 필요 ( 'goog.color' ); // 명확성을 위해 네임 스페이스 기반의 별칭으로 구조 해제. const { SomeType : FooSomeType } = goog

. requireType ( 'foo.types' ); const { clear : objectClear , clone : objectClone } = goog . 필요 ( 'goog.object' ); // 부작용을 유발하기 위해 별칭없이 goog.require. / \*\* @suppress {extraRequire} MyFramework를 초기화합니다. \* /

goog . 필요 ( 'my.framework.initialization' );

낙심 :

// 명확하게하기 위해 필요한 경우 모듈 이름의 형식에 더 가깝기 때문에 SomeClass보다 PackageClass를 선호합니다 . const SomeClass = goog . 필요 ( 'some.package.Class' );

허용되지 않음 :

// 추가 용어는 네임 스페이스에서 가져와야합니다. const MyClassForBizzing = goog . 필요 ( 'some.package.MyClass' ); // 별칭에는 전체 최종 네임 스페이스 구성 요소가 포함되어야합니다. const MyClass = goog . 필요 ( 'some.package.MyClassForBizzing' ); // 별칭은 네이티브 유형을 가리지 않아야합니다 (여기서는 'const JspbMap'이어야 함). const Map = goog . 필요 ( 'jspb.Map' ); // goog.require 줄을 80 개 이상의 열로 나누지 마세요. const SomeDataStructure =

goog . 필요 ( 'proto.identical.package.identifiers.SomeDataStructure' ); // 별칭은 네임 스페이스를 기반으로해야합니다. const randomName = goog . 필요 ( 'something.else' ); // 콜론 뒤에 공백이 없습니다. const { Foo : FooProto } = goog . 필요 ( 'some.package.proto.Foo' ); // 별칭이없는 goog.requireType.

goog . requireType ( 'some.package.with.a.Type' );

/ \*\*

\* @param {! some.unimported.Dependency} param JSDoc에서 사용되는 모든 외부 유형

\* 주석은 externs에서 선언되지 않는 한 goog.require이어야합니다.

\* / function someFunction ( param ) { // goog.require 줄은 다른 코드보다 먼저 최상위 수준에 있어야합니다. const 별칭 = goog . 필요 ( 'my.long.name.alias' ); // ... }

**3.7 파일의 구현**

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

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

**4 서식**

용어 참고 : 블록 형 구조 는 클래스, 함수, 메서드 또는 중괄호로 구분 된 코드 블록의 본문을 나타냅니다. 하여, 그 주 5.2 배열 상수 및 5.3 개체 리터럴 은 블록 형 구조 인 것처럼, 어떤 어레이 또는 객체 리터럴 임의로 처리 될 수있다.

팁 : clang-format. JavaScript 커뮤니티는 clang-format 이 JavaScript 파일 에서 올바른 작업을 수행하도록하기 위해 노력 했습니다 . clang-format여러 유명 편집자와 통합되었습니다.

**4.1 교정기**

**4.1.1 모든 제어 구조에 브레이스가 사용됩니다.**

교정기 (즉 모든 제어 구조에 필요한 if, else, for, do, while,뿐만 아니라 다른 사람), 몸은 단 하나의 문이 포함 된 경우에도 마찬가지입니다. 비어 있지 않은 블록의 첫 번째 문은 자체 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**

중괄호 는 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:

클래스 InnerClass { 생성자 () {}

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

method ( foo ) { if ( condition ( foo )) { try { // 참고 : 실패 할 수 있습니다.

무언가 (); } catch ( err ) {

recover (); } } } }

**4.1.3 빈 블록 : 간결 할 수 있음**

빈 블록 또는 블록과 유사한 구조 는 다중 블록 명령문 (여러 블록을 직접 포함하는 명령문 )의 일부가 아닌 경우 사이에 문자, 공백 또는 줄 바꿈없이 (예 {}) 열린 직후 닫을 수 있습니다 . : / 또는 / / ).ifelsetrycatchfinally

예:

function doNothing () {}

허용되지 않음 :

if ( condition ) { //… } else if ( otherCondition ) {} else { //… }

시도 { //… } 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. 예를 들어, 다음 (모든 유효 하지 완전한 목록) :

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 줄이기 위해서만 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 포함하는 문 은 여전히 세미콜론으로 종료됩니다). extends키워드를 사용 하되 @extends클래스가 템플릿 화 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 . 비어 있음 = 클래스 {};

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

foo . Bar = class extends Foo { / \*\* @override \* /

method () { return super . 방법 () / 2 ; } };

/ \*\* @interface \* / class Frobnicator { / \*\* @param {string} 메시지 \* /

frobnicate ( 메시지 ) {} }

**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:

접두사 . 뭔가 . reallyLongFunctionName ( 'whatever' , ( a1 , a2 ) => { // 'prefix'문의 들여 쓰기 깊이를 기준으로 함수 본문 +2를 들여 씁니다. // 'prefix'문의 한 줄 위. if ( a1 . equals ( a2 )) {

someOtherLongFunctionName ( a1 ); } else {

andNowForSomethingCompletelyDifferent ( a2 . parrot ); } });

일부 . reallyLongFunctionCall ( arg1 , arg2 , arg3 ) . thatsWrapped () . then (( result ) => { // '.then ()'호출 의 들여 쓰기 깊이를 기준으로 // 함수 본문을 들여 씁니다 . 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.

a break와 다음 대소 문자 사이에 빈 줄은 선택 사항 입니다.

예:

스위치 ( 동물 ) { 케이스 동물 . BANDERSNATCH :

handleBandersnatch (); 휴식 ;

케이스 동물 . JABBERWOCK :

handleJabberwock (); 휴식 ;

default : throw new Error ( '알 수없는 동물' ); }

**4.3 진술**

**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. goog.module, goog.require및 goog.requireType문 ( 3.3 goog.module 문 및 3.6 goog.require 및 goog.requireType 문 참조 ).

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

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

o 소스에서 클릭 할 수 있어야하는 긴 URL입니다.

o 복사하여 붙여 넣을 셸 명령입니다.

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

**4.5 줄 바꿈**

용어 참고 : 줄 바꿈 은 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 제한 오버플로를 방지하기위한 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 휴식 장소**

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

선호 :

currentEstimate =

calc ( currentEstimate + x \* currentEstimate ) / 2.0 ;

낙심 :

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.)

a. 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.

1. 클래스 또는 개체 리터럴의 연속 메서드 간

a. 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 .

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

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

4. 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 경우 외에 리터럴, 주석 및 JSDoc을 제외하고 단일 내부 ASCII 공간도 다음 위치 에만 나타납니다 .

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

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

3. 여는 중괄호 ( {) 앞 , 두 가지 예외 :

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

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

4. 이항 또는 삼항 연산자의 양쪽에 있습니다.

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

6. :개체 리터럴에서 콜론 ( ) 뒤 .

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

8. 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 , //

더 길어집니다 . 435 , // 이것도 };

{

tiny : 42 , // 허용되지만 향후 편집이

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 인해 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 4 개의 공백이어야합니다. 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 줄에는 맞지 않지만 두 번째 줄에는 완전히 맞을 때 선호됩니다 . // 매우 긴 함수 이름 과 함께 작동하고 , 다시 들여 쓰기없이 이름을 바꿀 수 있으며 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 다루어 집니다 .

**4.8.1 블록 주석 스타일**

블록 주석은 주변 코드와 동일한 수준에서 들여 쓰기됩니다. /\* … \*/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 \* / );

**5 언어 기능**

JavaScript에는 모호한 (심지어 위험한) 기능이 많이 포함되어 있습니다. 이 섹션에서는 사용하거나 사용하지 않을 수있는 기능과 사용에 대한 추가 제한 사항을 설명합니다.

**5.1 지역 변수 선언**

**5.1.1 사용 const및let**

const또는로 모든 지역 변수를 선언하십시오 let. 변수를 다시 할당해야하는 경우가 아니면 기본적으로 const를 사용하십시오. var 키워드는 사용할 수 없습니다.

**5.1.2 선언 당 하나의 변수**

모든 지역 변수 선언은 하나의 변수 만 선언합니다 let a = 1, b = 2;.

**5.1.3 필요할 때 선언하고 가능한 한 빨리 초기화**

지역 변수는 포함 블록이나 블록과 유사한 구조의 시작 부분에서 습관적으로 선언 되지 않습니다 . 대신 지역 변수는 범위를 최소화하기 위해 처음 사용되는 지점 (이유 내에서)에 가깝게 선언됩니다.

**5.1.4 필요에 따라 유형 선언**

JSDoc 유형 주석은 선언 위의 줄에 추가하거나 다른 JSDoc이없는 경우 변수 이름 앞에 인라인으로 추가 할 수 있습니다.

예:

const / \*\*! Array <번호> \* / data = [];

/ \*\*

\* 일부 설명.

\* @type {! Array <number>}

\* / const 데이터 = [];

인라인 스타일과 JSDoc 스타일을 혼합하는 것은 허용되지 않습니다. 컴파일러는 첫 번째 JsDoc 만 처리하고 인라인 주석은 손실됩니다.

/ \*\* 설명. \* / const / \*\*! Array <number> \* / data = [];

팁 : 컴파일러가 매개 변수가 아닌 템플릿 화 된 유형을 추론 할 수있는 경우가 많이 있습니다. 이는 특히 초기화 리터럴 또는 생성자 호출에 템플릿 매개 변수 유형의 값 (예 : 빈 배열, 객체, Maps 또는 Sets)이 포함되지 않거나 변수가 클로저에서 수정 된 경우에 해당합니다. 지역 변수 유형 주석은 이러한 경우에 특히 유용합니다. 그렇지 않으면 컴파일러가 템플릿 매개 변수를 알 수없는 것으로 추론하기 때문입니다.

**5.2 배열 리터럴**

**5.2.1 후행 쉼표 사용**

마지막 요소와 닫는 대괄호 사이에 줄 바꿈이있을 때마다 후행 쉼표를 포함하십시오.

예:

const 값 = [ '첫 번째 값' , '두 번째 값' , ];

**5.2.2 가변 Array생성자를 사용하지 마십시오.**

생성자는 인수가 추가되거나 제거되면 오류가 발생하기 쉽습니다. 대신 리터럴을 사용하십시오.

허용되지 않음 :

const a1 = 새 배열 ( x1 , x2 , x3 ); const a2 = 새 배열 ( x1 , x2 ); const a3 = 새 배열 ( x1 ); const a4 = 새 배열 ();

이것은 세 번째 경우를 제외하고 예상대로 작동합니다. if x1가 정수이면 모든 요소가 a3인 크기의 배열입니다 . 경우 다른 번호는 다음 예외가 발생되고, 그것이 다른 어떤 경우 그것은 단일 요소 배열 될 것입니다.x1undefinedx1

대신 쓰기

const a1 = [ x1 , x2 , x3 ]; const a2 = [ x1 , x2 ]; const a3 = [ x1 ]; const a4 = [];

new Array(length)적절한 경우를 사용하여 지정된 길이의 배열을 명시 적으로 할당 할 수 있습니다.

**5.2.3 숫자가 아닌 속성**

배열에서 숫자가 아닌 속성을 정의하거나 사용하지 마십시오 (제외 length). 용도 Map(또는 Object) 대신.

**5.2.4 구조 해제**

배열 리터럴은 할당의 왼쪽에서 구조화를 수행하는 데 사용할 수 있습니다 (예 : 단일 배열 또는 반복 가능에서 여러 값을 압축 해제 할 때). 마지막 나머지 요소가 포함될 수 있습니다 (와 ...변수 이름 사이에 공백 없음 ). 사용하지 않는 요소는 생략해야합니다.

const [ a , b , c , ... rest ] = generateResults (); let [, b ,, d ] = someArray ;

Destructuring은 함수 매개 변수에도 사용할 수 있습니다 (매개 변수 이름은 필수이지만 무시 됨). []destructured 배열 매개 변수가 선택 사항 인 경우 항상 기본값으로 지정 하고 왼쪽에 기본값을 제공하십시오.

/ \*\* @param {! Array <number> =} param1 \* / function optionalDestructuring ([ a = 4 , b = 2 ] = []) { … };

허용되지 않음 :

function badDestructuring ([ a , b ] = [ 4 , 2 ]) { … };

팁 : 여러 값을 함수의 매개 변수 또는 반환으로 (풀기) 패키징하려면 개별 요소의 이름을 지정하고 각각에 대해 다른 유형을 지정할 수 있으므로 가능한 경우 배열 분해보다 객체 분해를 선호합니다.

**5.2.5 스프레드 연산자**

배열 리터럴에는 ...하나 이상의 다른 이터 러블에서 요소를 평면화 하는 확산 연산자 ( ) 가 포함될 수 있습니다 . 더 어색한 구문 대신 확산 연산자를 사용해야합니다 Array.prototype. 뒤에 공백이 없습니다 ....

예:

[... foo ] // Array.prototype.slice.call (foo)보다 선호 됨 [... foo , ... bar ] // foo.concat (bar)보다 선호 됨

**5.3 객체 리터럴**

**5.3.1 후행 쉼표 사용**

최종 속성과 닫는 중괄호 사이에 줄 바꿈이있을 때마다 후행 쉼표를 포함합니다.

**5.3.2 Object생성자를 사용하지 마십시오**

반면 Object같은 문제로하지 않습니다 Array, 그것은 여전히 일관성을 위해 허용되지 않습니다. 대신 개체 리터럴 ( {}또는 {a: 0, b: 1, c: 2})을 사용합니다.

**5.3.3 인용 된 키와 인용되지 않은 키를 혼용하지 마십시오.**

객체 리터럴은 구조체 (따옴표없는 키 및 / 또는 기호 포함) 또는 사전 (따옴표 및 / 또는 계산 된 키 포함)을 나타낼 수 있습니다 . 단일 객체 리터럴에서 이러한 키 유형을 혼합하지 마십시오.

허용되지 않음 :

{

width : 42 , // struct-style unquoted key 'maxWidth' : 43 , // dict-style quoted key }

이것은 또한 속성 이름을 같은 함수에 전달하는 것으로 확장됩니다 hasOwnProperty. 특히 이렇게하면 컴파일러가 문자열 리터럴의 이름을 바꾸거나 난독화할 수 없기 때문에 컴파일 된 코드에서 중단됩니다.

허용되지 않음 :

/ \*\* @type {{너비 : 숫자, maxWidth : (숫자 | 정의되지 않음)}} \* / const o = { 너비 : 42 }; if ( o . hasOwnProperty ( 'maxWidth' )) { ... }

이것은 다음과 같이 가장 잘 구현됩니다.

/ \*\* @type {{너비 : 숫자, maxWidth : (숫자 | 정의되지 않음)}} \* / const o = { 너비 : 42 }; if ( o . maxWidth ! = null ) { ... }

**5.3.4 계산 된 속성 이름**

계산 된 속성 이름 (예 :) {['key' + foo()]: 42}이 허용되며 계산 된 속성이 기호 (예 :)가 아닌 한 dict-style (인용 된) 키로 간주됩니다 (즉, 인용되지 않은 키와 혼합되어서는 안 됨 [Symbol.iterator]). 열거 형 값은 계산 된 키에도 사용할 수 있지만 동일한 리터럴에서 비 열거 형 키와 혼합해서는 안됩니다.

**5.3.5 방법 속기**

또는 화살표 함수 리터럴 {method() {… }}바로 뒤에 오는 콜론 대신 약식 ( ) 메서드를 사용하여 객체 리터럴에 메서드를 정의 할 수 있습니다 function.

예:

return {

stuff : 'candy' ,

method () { return this . 물건 ; // 'candy'반환 }, };

주의 this또는 속기 방법에 function반해 자체 리터럴 객체를 참조 this화살표 함수는 객체 리터럴 외부 범위를 말한다.

예:

class {

getObjectLiteral () { this . stuff = '과일' ; return {

stuff : 'candy' ,

method : () => this . stuff , // 'fruit'반환 }; } }

**5.3.6 속기 속성**

개체 리터럴에는 속기 속성이 허용됩니다.

예:

const foo = 1 ; const bar = 2 ; const obj = {

foo ,

bar ,

method () { return this . foo + this . 바 ; }, };

assertEquals ( 3 , obj . method ());

**5.3.7 해체**

객체 분해 패턴은 할당의 왼쪽에서 분해를 수행하고 단일 객체에서 여러 값의 압축을 푸는 데 사용할 수 있습니다.

해체 된 객체는 함수 매개 변수로도 사용할 수 있지만 가능한 한 단순하게 유지해야합니다. 인용되지 않은 속기 속성의 단일 수준입니다. 더 깊은 수준의 중첩 및 계산 된 속성은 매개 변수 구조화에 사용되지 않을 수 있습니다. destructured 매개 변수의 왼쪽에 기본값을 지정하고 ( {str = 'some default'} = {}대신 {str} = {str: 'some default'}), destructured 객체 자체가 선택 사항 인 경우 기본값은 {}. destructured 매개 변수에 대한 JSDoc에는 임의의 이름을 지정할 수 있습니다 (이름은 사용되지 않지만 컴파일러에서 필요함).

예:

/ \*\*

\* @param {string} 보통

\* @param {{num : (number | undefined), str : (string | undefined)} =} param1

\* num : 작업을 수행 한 횟수입니다.

\* str : 작업을 수행 할 문자열.

\* / destructured 함수 ( 일반 , { num , str = '일부 기본값' } = {})

허용되지 않음 :

/ \*\* @param {{x : {num : (number | undefined), str : (string | undefined)}}} param1 \* / function nestedTooDeeply ({ x : { num , str }}) {}; / \*\* @param {{num : (number | undefined), str : (string | undefined)} =} param1 \* / function nonShorthandProperty ({ num : a , str : b } = {}) {}; / \*\* @param {{a : number, b : number}} param1 \* / function computedKey ({ a , b , [ a + b ] :

c }) {}; / \*\* @param {{a : 숫자, b : 문자열} =} param1 \* / function nontrivialDefault ({ a , b } = { a : 2 , b : 4 }) {};

Destructuring은 goog.require문 에도 사용될 수 있으며이 경우에는 래핑되어서는 안됩니다. 전체 문은 길이에 관계없이 한 줄을 차지합니다 ( 3.6 goog.require 및 goog.requireType 문 참조 ).

**5.3.8 열거 형**

열거 형은 @enum객체 리터럴에 주석을 추가하여 정의됩니다 . 추가 속성은 정의 된 후에 열거 형에 추가 할 수 없습니다. 열거 형은 상수 여야하며 모든 열거 형 값은 변경 불가능해야합니다.

/ \*\*

\* 지원되는 온도 눈금.

\* @enum {string}

\* / const TemperatureScale = {

CELSIUS : '섭씨' ,

FAHRENHEIT : '화씨' , };

/ \*\*

\* 두 가지 옵션이있는 열거 형.

\* @enum {number}

\* / const Option = { / \*\* 사용 된 옵션이 첫 번째 여야합니다. \* /

FIRST\_OPTION : 1 , / \*\* 두 옵션 중 두 번째. \* /

SECOND\_OPTION : 2 , };

**5.4 클래스**

**5.4.1 생성자**

생성자는 선택 사항입니다. 하위 클래스 생성자는 super()필드를 설정하거나 this. 인터페이스는 생성자에서 메서드가 아닌 속성을 선언해야합니다.

**5.4.2 필드**

생성자에서 구체적인 객체의 모든 필드 (즉, 메서드를 제외한 모든 속성)를 설정합니다. 재 할당되지 않은 필드에 주석을 추가합니다 @const (깊이 변경할 필요는 없음). 주석 비공개 적절한 가시성 주석 (과 필드 @private, @protected, @package), 그리고 모든 끝 @private밑줄로 필드의 이름을. 필드는 구체적인 클래스에 설정되지 않습니다 prototype. ' .

예:

class Foo { constructor () { / \*\* @private @const {! Bar} \* / this . bar\_ = computeBar ();

/ \*\* @protected @const {! Baz} \* / this . baz = computeBaz (); } }

팁 : 생성자가 완료된 후에는 인스턴스에 속성을 추가하거나 인스턴스에서 제거하면 안됩니다. VM의 최적화 기능을 크게 방해하기 때문입니다. 필요한 경우 나중에 초기화되는 필드는 undefined나중에 모양이 변경되지 않도록 생성자에서 로 명시 적으로 설정해야합니다 . @struct객체에 추가하면 선언되지 않은 속성이 추가 / 액세스되지 않는지 확인합니다. 클래스에는 기본적으로 추가됩니다.

**5.4.3 계산 된 속성**

계산 된 속성은 속성이 심볼 인 경우에만 클래스에서 사용할 수 있습니다. 사전 스타일 속성 (즉, 5.3.3에 정의 된 인용 또는 계산 된 비 기호 키 )는 허용되지 않습니다. [Symbol.iterator]방법은 논리적으로 반복 가능한있는 모든 클래스를 정의해야합니다. 이 Symbol외에는 드물게 사용해야합니다.

팁 : 다른 내장 심볼 (예 Symbol.isConcatSpreadable:)은 컴파일러에 의해 폴리 필되지 않으므로 이전 브라우저에서 작동하지 않으므로주의하십시오.

**5.4.4 정적 메서드**

가독성을 방해하지 않는 경우 개인 정적 메서드보다 모듈 로컬 함수를 선호합니다.

정적 메서드는 기본 클래스 자체에서만 호출해야합니다. 정적 메서드는 생성자 또는 하위 클래스 생성자 일 수있는 동적 인스턴스를 포함하는 변수에서 호출되어서는 안되며 ( @nocollapse이 작업이 수행되면 로 정의되어야 함 ) 메서드를 정의하지 않는 하위 클래스에서 직접 호출되어서는 안됩니다. 그 자체.

허용되지 않음 :

class Base { / \*\* @nocollapse \* / static foo () {} } class Sub extends Base {} function callFoo ( cls ) { cls . foo (); } // 권장하지 않음 : 정적 메서드를 동적으로 호출하지 마십시오 Sub . foo (); // 허용되지 않음 : 스스로 정의하지 않는 하위 클래스에서 정적 메서드를 호출하지 마십시오.

**5.4.5 구식 클래스 선언**

ES6 수업이 선호되지만 ES6 수업이 불가능한 경우가 있습니다. 예를 들면 :

1. 하위 클래스를 만드는 프레임 워크를 포함하여 하위 클래스가 있거나 존재할 경우 ES6 클래스 구문을 사용하도록 즉시 변경할 수 없습니다. 이러한 클래스가 ES6 구문을 사용하는 경우 ES6 클래스 구문을 사용하지 않는 모든 다운 스트림 하위 클래스를 수정해야합니다.

2. thisES6 슈퍼 클래스가있는 생성자 this는 호출이 super반환 될 때까지 인스턴스 값에 액세스 할 수 없기 때문에 슈퍼 클래스 생성자를 호출하기 전에 알려진 값 이 필요한 프레임 워크입니다 .

다른 모든면에서 스타일 가이드는 여전히이 코드에 적용 let, const기본 매개 변수, 휴식 및 화살표 기능이 모든 경우에 적절한 사용되어야한다.

goog.defineClass ES6 클래스 구문과 유사한 클래스와 유사한 정의를 허용합니다.

let C = goog.defineClass (S, {

/ \*\*

\* @param {string} 값

\* /

생성자 (값) {

S.call (this, 2);

/ \*\* @const \* /

this.prop = 값;

},

/ \*\*

\* @param {string} 매개 변수

\* @return {number}

\* /

method (param) {

반환 0;

},

});

또는 goog.defineClass모든 새 코드에 대해 선호되어야 하지만 더 전통적인 구문도 허용됩니다.

/ \*\*

\* @constructor @extends {S}

\* @param {string} 값

\* /

function C (value) {

S.call (this, 2);

/ \*\* @const \* /

this.prop = 값;

}

goog.inherits (C, S);

/ \*\*

\* @param {string} 매개 변수

\* @return {number}

\* /

C.prototype.method = function (param) {

반환 0;

};

인스턴스 별 속성은 슈퍼 클래스가있는 경우 슈퍼 클래스 생성자를 호출 한 후 생성자에서 정의해야합니다. 생성자의 프로토 타입에 메서드를 정의해야합니다.

생성자 프로토 타입 계층을 올바르게 정의하는 것은 처음 나타나는 것보다 어렵습니다! 따라서 Closure Librarygoog.inherits 에서 사용하는 것이 가장 좋습니다 .

**5.4.6 prototypes를 직접 조작하지 마십시오.**

이 class키워드는 prototype속성을 정의하는 것보다 더 명확하고 읽기 쉬운 클래스 정의를 허용 합니다. 일반 구현 코드는 이러한 객체를 조작하는 비즈니스가 없지만 5.4.5 이전 스타일 클래스 선언에 정의 된대로 클래스를 정의하는 데 여전히 유용합니다 . 믹스 인 및 내장 객체의 프로토 타입 수정은 명시 적으로 금지되어 있습니다.

예외 : 프레임 워크 코드 (예 : Polymer 또는 Angular)는 prototypes 를 사용해야 할 수 있으며 그렇게하지 않기 위해 더 나쁜 해결 방법을 사용 해서는 안됩니다.

**5.4.7 게터 및 세터**

JavaScript getter 및 setter 속성을 사용하지 마십시오 . 잠재적으로 놀랍고 추론하기 어렵고 컴파일러에서 제한된 지원을 제공합니다. 대신 일반적인 방법을 제공하십시오.

예외 : getter 또는 setter를 정의하는 것이 불가피한 상황이 있습니다 (예 : Angular 및 Polymer와 같은 데이터 바인딩 프레임 워크 또는 조정할 수없는 외부 API와의 호환성). 이러한 경우에만 getter 및 setter가 및 속기 메서드 키워드 로 정의 되거나 속성 이름 변경을 방해하는 (not) 으로 정의 된 경우 주의해서 사용할 수 있습니다 . 게터 는 관찰 가능한 상태를 변경 해서는 안됩니다 .getsetObject.definePropertiesObject.defineProperty

허용되지 않음 :

class Foo { get next () { return this . nextId ++; } }

**5.4.8 toString 재정의**

이 toString메서드는 재정의 될 수 있지만 항상 성공해야하며 눈에 보이는 부작용이 없어야합니다.

팁 : 예외적 인 조건으로 인해 무한 루프가 발생할 수 있으므로 특히 toString에서 다른 메서드를 호출하지 않도록주의하십시오.

**5.4.9 인터페이스**

인터페이스는 @interface또는 로 선언 할 수 있습니다 @record. 로 선언 된 인터페이스는 @record명시 적으로 (즉를 통해 @implements) 또는 클래스 또는 객체 리터럴에 의해 암시 적으로 구현 될 수 있습니다 .

인터페이스의 모든 비 정적 메서드 본문은 빈 블록이어야합니다. 필드는 클래스 생성자에서 초기화되지 않은 멤버로 선언되어야합니다.

예:

/ \*\*

\* 화를 낼 수있는 것.

\* @record

\* / class Frobnicator { constructor () { / \*\* @type {number} 포기하기 전에 시도한 횟수. \* / 이 . 시도 ; }

/ \*\*

\* 주어진 전략에 따라 frobnication을 수행합니다.

\* @param {! FrobnicationStrategy} 전략

\* /

frobnicate ( 전략 ) {} }

**5.4.10 추상 클래스**

적절한 경우 추상 클래스를 사용하십시오. 추상 클래스와 메서드는로 주석을 달아야합니다 @abstract. 사용하지 마십시오 goog.abstractMethod. 추상 클래스 및 메서드를 참조하십시오 .

**5.5 기능**

**5.5.1 최상위 기능**

최상위 함수는 exports객체에서 직접 정의 하거나 로컬로 선언하고 선택적으로 내보낼 수 있습니다. 내보내기에 대한 자세한 내용은 3.3.3 goog.module 내보내기 를 참조하십시오 .

예 :

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

exports . processString = ( str ) => { // 문자열을 처리합니다. };

/ \*\* @param {string} str \* / const processString = ( str ) => { // 문자열을 처리합니다. };

수출 = { processString };

**5.5.2 중첩 함수와 클로저**

함수는 중첩 된 함수 정의를 포함 할 수 있습니다. 함수에 이름을 지정하는 것이 유용하면 local에 할당해야합니다 const.

**5.5.3 화살표 기능**

화살표 함수는 간결한 함수 구문을 제공하고 this 중첩 함수의 범위 지정 을 단순화 합니다. function특히 중첩 함수의 경우 키워드 보다 화살표 함수를 선호 합니다 (단, 5.3.5 메서드 속기 참조 ).

다른 이상 기능을 화살표 선호 this범위 지정하는 등의 접근 f.bind(this), goog.bind(f, this)및 const self = this. 화살표 함수는 콜백에 전달할 매개 변수를 명시 적으로 지정하는 것을 허용하는 반면 바인딩은 모든 매개 변수를 맹목적으로 전달하므로 콜백 호출에 특히 유용합니다.

화살표의 왼쪽에는 0 개 이상의 매개 변수가 있습니다. 구조화되지 않은 매개 변수가 하나만있는 경우 매개 변수 주위의 괄호는 선택 사항입니다. 괄호를 사용하면 인라인 매개 변수 유형을 지정할 수 있습니다 ( 7.8 메소드 및 함수 주석 참조 ).

팁 : 단일 매개 변수 화살표 함수에도 항상 괄호를 사용하면 매개 변수를 추가하는 상황을 피할 수 있지만 괄호를 추가하는 것을 잊으면 더 이상 의도 한대로 작동하지 않는 구문 분석 가능한 코드가 생성 될 수 있습니다.

화살표의 오른쪽에는 함수의 본문이 포함됩니다. 기본적으로 본문은 블록 문 (중괄호로 묶인 0 개 이상의 문)입니다. 본문은 또한 다음과 같은 경우 암시 적으로 반환 된 단일 표현식이 될 수 있습니다. 프로그램 논리가 값을 반환해야하거나 void연산자가 단일 함수 또는 메서드 호출 앞에 오는 경우 ( voidsures undefined가 반환되고 값이 누출되는 것을 방지하며 의도를 전달 함). 가독성이 향상되는 경우 (예 : 짧거나 간단한 표현) 단일 표현 형식이 선호됩니다.

예 :

/ \*\*

\* 화살표 기능은 일반 기능과 마찬가지로 문서화 할 수 있습니다.

\* @param {number} numParam 더할 숫자.

\* @param {string} strParam 추가 할 또 다른 숫자는 문자열입니다.

\* @return {number} 두 매개 변수의 합.

\* / const moduleLocalFunc = ( numParam , strParam ) => numParam + Number ( strParam );

// 프로그램 논리가 값을 반환 할 필요 가 없기 때문에 // 'void'와 함께 단일 표현식 구문을 사용합니다 .

getValue (( result ) => void alert (` Got $ { result }`));

class CallbackExample { constructor () { / \*\* @private {number} \* / this . cachedValue\_ = 0 ;

// 인라인 콜백의 경우 매개 변수에 인라인 입력을 사용할 수 있습니다. // 단일 표현식의 값 이 반환되지 않아야하고 // 표현식이 단일 함수 호출이 아니기 때문에 블록 문을 사용합니다 .

getNullableValue (( / \*\*? number \* / result ) => { this . cachedValue\_ = result == null ? 0 : result ; }); } }

허용되지 않음 :

/ \*\*

\* 매개 변수가없고 반환 된 값이없는 함수.

\*이 단일 표현식 본문 사용은 프로그램 논리가

\* 값을 반환 할 필요가 없으며`void` 연산자가 없습니다.

\* / const moduleLocalFunc = () => anotherFunction ();

**5.5.4 발전기**

생성기는 여러 가지 유용한 추상화를 가능하게하며 필요에 따라 사용할 수 있습니다.

생성 함수를 정의 할 때, 부착 \*받는 function경우 본 키워드 및 함수의 이름에서 공간을 분리한다. 위임 수익률을 사용할 때 키워드에 \*를 첨부하십시오 yield.

예:

/ \*\* @return {! Iterator <number>} \* / function \* gen1 () { yield 42 ; }

/ \*\* @return {! Iterator <number>} \* / const gen2 = function \* () { yield \* gen1 (); }

class SomeClass { / \*\* @return {! Iterator <number>} \* / \* gen () { yield 42 ; } }

**5.5.5 매개 변수 및 반환 유형**

함수 매개 변수와 반환 유형은 일반적으로 JSDoc 주석으로 문서화되어야합니다. 자세한 내용은 7.8 메서드 및 기능 설명 을 참조하십시오.

**5.5.5.1 기본 매개 변수**

매개 변수 목록에서 같음 연산자를 사용하여 선택적 매개 변수가 허용됩니다. 선택적 매개 변수는 같음 연산자의 양쪽에 공백을 포함해야하며, 필수 매개 변수와 정확히 같은 이름을 지정해야하며 (예 : 접두사가 붙지 않음) JSDoc 유형에 접미사를 opt\_사용하고 =, 필수 매개 변수 뒤에 오며, 관찰 가능한 부작용을 생성하는 이니셜 라이저를 사용하지 않아야합니다. 구체적인 함수에 대한 모든 선택적 매개 변수는 해당 값이 인 경우에도 기본값을 가져야합니다 undefined. 구체적인 함수와 달리 추상 및 인터페이스 메소드는 기본 매개 변수 값을 생략해야합니다.

예:

/ \*\*

\* @param {string} 필수이 매개 변수는 항상 필요합니다.

\* @param {string =} 선택 사항이 매개 변수는 생략 할 수 있습니다.

\* @param {! Node =} node 또 다른 선택적 매개 변수입니다.

\* / function maybeDoSomething ( 필수 , 선택 = '' , 노드 = 정의되지 않음 ) {}

/ \*\* @interface \* / class MyInterface { / \*\*

\* 인터페이스 및 추상 메소드는 기본 매개 변수 값을 생략해야합니다.

\* @param {string =} 선택 사항

\* /

someMethod ( 선택 사항 ) {} }

기본 매개 변수를 조금만 사용하십시오. 자연 순서가없는 소수의 선택적 매개 변수가 많을 때 읽을 수있는 API를 생성하려면 구조 해체 ( 5.3.7 Destructuring 과 같이 )를 선호 합니다.

참고 : Python의 기본 매개 변수와 달리 , 초기화 프로그램은 기본값이 사용될 때마다 평가되므로 단일 객체가 호출간에 공유되지 않으므로 새로운 변경 가능한 객체 (예 : {}또는 []) 를 반환하는 초기화 프로그램을 사용하는 것이 좋습니다 .

팁 : 함수 호출을 포함한 임의의 표현식을 이니셜 라이저로 사용할 수 있지만 가능한 한 간단하게 유지해야합니다. 공유 된 변경 가능 상태를 노출하는 이니셜 라이저는 함수 호출간에 의도하지 않은 결합을 쉽게 도입 할 수 있으므로 피하십시오.

**5.5.5.2 나머지 매개 변수**

에 액세스하는 대신 나머지 매개 변수를 사용하십시오 arguments. 나머지 매개 변수는 ...JSDoc에 접두어로 입력됩니다 . 나머지 매개 변수는 목록의 마지막 매개 변수 여야합니다. ...와 매개 변수 이름 사이에는 공백이 없습니다 . 나머지 매개 변수의 이름을 지정하지 마십시오 var\_args. arguments내장 이름을 혼동스럽게 숨기는 로컬 변수 또는 매개 변수의 이름을 지정하지 마십시오 .

예:

/ \*\*

\* @param {! Array <string>} array 이것은 일반적인 매개 변수입니다.

\* @param {... number} 숫자 나머지 인수는 모두 숫자입니다.

\* / 가변 함수 ( 배열 , ... 숫자 ) {}

**5.5.6 제네릭**

@template TYPE함수 또는 메서드 정의 위에 JSDoc에서 필요한 경우 일반 함수 및 메서드를 선언 합니다.

**5.5.7 스프레드 연산자**

함수 호출은 스프레드 연산자 ( ...)를 사용할 수 있습니다 . Function.prototype.apply배열 또는 반복 가능 항목이 가변 함수의 여러 매개 변수로 풀릴 때보 다 확산 연산자를 선호합니다 . 뒤에 공백이 없습니다 ....

예:

function myFunction (... elements ) {}

myFunction (... array , ... iterable , ... generator ());

**5.6 문자열 리터럴**

**5.6.1 작은 따옴표 사용**

일반 문자열 리터럴은 '큰 따옴표 ( ") 가 아닌 작은 따옴표 ( )로 구분됩니다.

팁 : 문자열에 작은 따옴표 문자가 포함 된 경우 따옴표를 이스케이프하지 않도록 템플릿 문자열을 사용하는 것이 좋습니다.

일반 문자열 리터럴은 여러 줄에 걸쳐있을 수 없습니다.

**5.6.2 템플릿 리터럴**

`특히 여러 문자열 리터럴이 관련된 경우 복잡한 문자열 연결에 템플릿 리터럴 (으로 구분 )을 사용합니다. 템플릿 리터럴은 여러 줄에 걸쳐있을 수 있습니다.

템플릿 리터럴이 여러 줄에 걸쳐있는 경우 추가 된 공백이 중요하지 않을 수 있지만 둘러싸는 블록의 들여 쓰기를 따를 필요가 없습니다.

예:

함수 연산부 ( , B ) { 복귀 ` 여기서 표이다 의 연산 :

$ { } + $ { B } = $ { +의 B }

$ { } - $ { B } = $ { - B }

$ { a } \* $ { b }

= $ { a \* b }

$ { a } / $ { b } = $ { a / b }`; }

**5.6.3 라인 연속 없음**

일반 또는 템플릿 문자열 리터럴에서 줄 연속 (즉, 백 슬래시로 문자열 리터럴 내부의 줄 끝)을 사용하지 마십시오 . ES5가 이것을 허용하더라도 슬래시 뒤에 후행 공백이 오면 까다로운 오류가 발생할 수 있으며 독자에게 명확하지 않습니다.

허용되지 않음 :

CONST longString은 = ' 이것은 매우이다 긴 까지가 초과하는 문자열 (80) \

열 제한 . 그것은 불행하게도 포함 긴 뻗기 의 인해 공간을 \

연속 된 줄을 들여 쓰는 방법 . ';

대신 쓰기

const longString = '이것은 80' + '열 제한 을 훨씬 초과하는 매우 긴 문자열입니다 . ' + '연결된 문자열이 더 깨끗하기 때문에 긴 공백을 포함하지 않습니다 . ;

**5.7 숫자 리터럴**

숫자는 10 진수, 16 진수, 8 진수 또는 2 진수로 지정할 수 있습니다. 16 진수, 8 진수 및 2 진수에 대해 각각 소문자와 함께 정확히 0x, 0o및 0b접두사를 사용합니다 . 그것은 바로 다음에하지 않는 한 절대로 앞에 0을 포함하지 x, o또는 b.

**5.8 제어 구조**

**5.8.1 For 루프**

ES6에서는 이제 언어에 세 가지 종류의 for루프가 있습니다. 하지만 모두 사용할 수있다 for- of루프가 가능한 경우 선호한다.

for- in루프는 dict 스타일 객체에서만 사용할 수 있으며 ( 5.3.3 인용 및 인용되지 않은 키를 혼합하지 마십시오 참조 ) 배열을 반복하는 데 사용해서는 안됩니다. 원하지 않는 프로토 타입 속성을 제외하려면 - 루프 Object.prototype.hasOwnProperty에서 사용해야합니다 . 선호 - 과 이상 - 가능한 경우.forinforofObject.keysforin

**5.8.2 예외**

예외는 언어의 중요한 부분이며 예외적 인 경우가 발생할 때마다 사용해야합니다. 항상 Error의 s 또는 하위 클래스를 Error던지십시오. 문자열 리터럴이나 다른 객체를 던지지 마십시오. 항상 사용 new를 구성 할 때 Error.

이 처리 는 비동기 함수 와 동일한 Promise거부 값으로 확장됩니다 .Promise.reject(obj)throw obj;

사용자 정의 예외는 함수에서 추가 오류 정보를 전달하는 좋은 방법을 제공합니다. 네이티브 Error 유형이 불충분 할 때마다 정의하고 사용해야합니다 .

임시 오류 처리 방식 (예 : 오류 컨테이너 참조 유형 전달 또는 오류 속성이있는 개체 반환)보다 예외를 throw하는 것을 선호합니다.

5.8.2.1 빈 캐치 블록

포착 된 예외에 대한 응답으로 아무것도하지 않는 것은 매우 드뭅니다. catch 블록에서 아무 조치도 취하지 않는 것이 진정으로 적절할 때 이것이 정당화되는 이유는 주석에 설명되어 있습니다.

try { return handleNumericResponse ( response ); } catch ( ok ) { // 숫자가 아닙니다. 괜찮습니다. 계속하십시오 } return handleTextResponse ( response );

허용되지 않음 :

시도 {

shouldFail ();

실패 ( '오류가 예상 됨' ); } catch ( 예상 ) { }

팁 : 일부 다른 언어와 달리 위와 같은 패턴은에서 발생하는 오류를 포착하므로 작동하지 않습니다 fail. assertThrows()대신 사용하십시오 .

**5.8.3 스위치 문**

용어 참고 : 스위치 블록의 중괄호 안에는 하나 이상의 명령문 그룹이 있습니다. 각 문 그룹은 하나 이상의 스위치 레이블 ( case FOO:또는 default:)과 하나 이상의 문으로 구성됩니다.

**5.8.3.1 폴 스루 : 주석 처리됨**

스위치 블록 내에서 각 문 그룹 중 하나 (A와 함께 돌연 종료 break, return또는 thrown은 예외), 또는 또는 다음 명령문 그룹에 계속 한 것 실행을 나타 내기 위해 코멘트와 함께 표시됩니다. 폴 스루 아이디어를 전달하는 모든 의견으로 충분합니다 (일반적으로 // fall through). 이 특수 주석은 스위치 블록의 마지막 명령문 그룹에는 필요하지 않습니다.

예:

스위치 ( 입력 ) { 케이스 1 : 케이스 2 :

prepareOneOrTwo (); // 케이스 3 통과 :

handleOneTwoOrThree (); 휴식 ; 기본값 :

handleLargeNumber ( 입력 ); }

**5.8.3.2 default케이스가 있습니다.**

각 switch 문에는 default코드가없는 경우에도 문 그룹이 포함됩니다. default문 그룹은 지난 있어야합니다.

**5.9 this**

만 사용하는 this클래스 생성자와 메소드 내에서 정의 된 화살표 기능에, 또는 명시 적으로 가지고있는 기능에, 클래스 생성자와 방법에 @this바로-둘러싸는 함수의 JSDoc에 선언했다.

this전역 개체,의 컨텍스트 eval, 이벤트 대상 또는 불필요하게 call()ed 또는 apply()ed 함수 를 참조하는 데 사용하지 마십시오 .

**5.10 평등 검사**

아래에 설명 된 경우를 제외하고 ID 연산자 ( ===/ !==)를 사용하십시오 .

**5.10.1 강제 변환이 바람직한 예외**

null와 undefined가치 모두 잡기 :

if ( someObjectOrPrimitive == null ) { // null을 확인하면 객체와 기본 요소에 대해 null과 undefined를 모두 포착 하지만 // 0 또는 빈 문자열 과 같은 다른 잘못된 값은 포착하지 않습니다 . }

**5.11 허용되지 않는 기능**

**5.11.1**

with키워드를 사용하지 마십시오 . 코드를 이해하기 어렵게 만들고 ES5 이후 엄격 모드에서 금지되었습니다.

**5.11.2 동적 코드 평가**

eval또는 Function(...string)생성자를 사용하지 마십시오 (코드 로더 제외). 이러한 기능은 잠재적으로 위험하며 CSP 환경에서는 작동하지 않습니다.

**5.11.3 자동 세미콜론 삽입**

항상 세미콜론으로 문을 종료합니다 (위에서 언급 한 함수 및 클래스 선언 제외).

**5.11.4 비표준 기능**

비표준 기능을 사용하지 마십시오. 여기에는 제거 된 이전 기능 (예 WeakMap.clear:), 아직 표준화되지 않은 새로운 기능 (예 : 현재 TC39 작업 초안, 모든 단계의 제안 또는 제안되었지만 아직 완전하지 않은 웹 표준) 또는 다음과 같은 독점 기능이 포함됩니다. 일부 브라우저에서만 구현됩니다. 현재 ECMA-262 또는 WHATWG 표준에 정의 된 기능 만 사용하십시오. (Chrome 확장 프로그램 또는 Node.js와 같은 특정 API에 대해 작성하는 프로젝트는 분명히 해당 API를 사용할 수 있습니다.) 비표준 언어 "확장자"(예 : 일부 외부 트랜스 파일러가 제공하는 확장)는 금지됩니다.

**5.11.5 기본 유형에 대한 래퍼 객체**

사용하지 마십시오 new원시 객체 래퍼 (에 Boolean, Number, String, Symbol),도 유형 약어에 포함.

허용되지 않음 :

const / \*\* Boolean \* / x = new Boolean ( false ); if ( x ) alert ( typeof x ); // '객체'경고-WAT?

래퍼는 강제 ( +빈 문자열 을 사용 하거나 연결 하는 것보다 선호 됨 ) 또는 기호 생성을 위한 함수로 호출 될 수 있습니다 .

예:

const / \*\* 부울 \* / x = 부울 ( 0 ); if (! x ) alert ( typeof x ); // 예상대로 '부울'경고

**5.11.6 내장 객체 수정**

생성 자나 프로토 타입에 메소드를 추가하여 내장 유형을 수정하지 마십시오. 이를 수행하는 라이브러리에 의존하지 마십시오. JSCompiler의 런타임 라이브러리는 가능한 경우 표준을 준수하는 폴리 필을 제공합니다. 다른 어떤 것도 내장 객체를 수정할 수 없습니다.

절대적으로 필요한 경우가 아니면 (예 : 타사 API에서 요구하는 경우) 전역 개체에 기호를 추가하지 마십시오.

**5.11.7 ()생성자를 호출 할 때 생략**

new괄호를 사용하지 않고 명령문 에서 생성자를 호출하지 마십시오 ().

허용되지 않음 :

새로운 Foo ;

대신 사용 :

새로운 Foo ();

괄호를 생략하면 미묘한 실수가 발생할 수 있습니다. 이 두 줄은 동일하지 않습니다.

새로운 Foo (). 바 (); 새로운 Foo . 바 ();

**6 명명**

**6.1 모든 식별자에 공통적 인 규칙**

식별자는 ASCII 문자와 숫자 만 사용하며, 아래에 언급 된 소수의 경우 밑줄을 사용하고 매우 드물게 (Angular와 같은 프레임 워크에서 요구하는 경우) 달러 기호를 사용합니다.

이유 내에서 가능한 한 설명적인 이름을 지정하십시오. 새로운 독자가 코드를 즉시 이해할 수 있도록 만드는 것이 훨씬 더 중요하므로 수평 공간을 절약하는 것에 대해 걱정하지 마십시오. 프로젝트 외부의 독자에게 모호하거나 익숙하지 않은 약어를 사용하지 말고, 단어 내의 문자를 삭제하여 축약하지 마십시오.

errorCount // 약어가 없습니다.

dnsConnectionIndex // 대부분의 사람들은 "DNS"가 무엇을 의미하는지 알고 있습니다.

referrerUrl // 'URL'도 마찬가지입니다.

customerId // "Id"는 어디에나 있고 오해의 소지가 없습니다.

허용되지 않음 :

n // 의미 없음.

nErr // 약어가 모호합니다.

nCompConns // 약어가 모호합니다.

wgcConnections // 이것이 의미하는 바를 그룹 만이 알고 있습니다.

pcReader // 많은 것들이 "pc"로 축약 될 수 있습니다.

cstmrId // 내부 문자를 삭제합니다.

kSecondsPerDay // 헝가리 표기법을 사용하지 않습니다.

**6.2 식별자 유형별 규칙**

**6.2.1 패키지 이름**

패키지 이름은 모두 lowerCamelCase. 예를 들어 my.exampleCode.deepSpace,하지만 my.examplecode.deepspace나 my.example\_code.deep\_space.

**6.2.2 클래스 이름**

클래스, 인터페이스, 레코드 및 typedef 이름은 UpperCamelCase. 내 보내지 않은 클래스는 단순히 로컬입니다. 표시되지 @private않으므로 후행 밑줄로 이름이 지정되지 않습니다.

유형 이름은 일반적으로 명사 또는 명사 구입니다. 예를 들어 Request, ImmutableList또는 VisibilityMode. 또한 인터페이스 이름은 때때로 대신 형용사 또는 형용사 구문이 될 수 있습니다 (예 :) Readable.

**6.2.3 메서드 이름**

메서드 이름은 lowerCamelCase. @private메서드 이름 은 후행 밑줄로 끝나야합니다.

메서드 이름은 일반적으로 동사 또는 동사 구입니다. 예를 들어, sendMessage또는 stop\_. 속성에 대한 Getter 및 setter 메서드는 필수가 아니지만 사용하는 경우 이름 getFoo(또는 선택적으로 isFoo또는 hasFoo 부울에 대해) 또는 setFoo(value)setter에 대해 지정해야합니다 .

이름의 논리적 구성 요소를 구분하기 위해 JsUnit 테스트 메서드 이름에 밑줄이 표시 될 수도 있습니다. 한 가지 전형적인 패턴은 test<MethodUnderTest>\_<state>\_<expectedOutcome>예를 들면 다음과 같습니다 testPop\_emptyStack\_throws. 테스트 방법의 이름을 정하는 올바른 방법은 없습니다.

**6.2.4 열거 형 이름**

열거 형 이름은 UpperCamelCase클래스와 유사하게로 작성되며 일반적으로 단수 명사 여야합니다. 열거 형 내의 개별 항목은 CONSTANT\_CASE.

**6.2.5 상수 이름**

상수 이름 사용 CONSTANT\_CASE: 밑줄로 구분 된 단어가있는 모두 대문자입니다. private static 속성은 (암시 적으로 private) 모듈 로컬로 대체 될 수 있기 때문에 상수에 후행 밑줄로 이름을 지정할 이유가 없습니다.

**6.2.5.1 "상수"의 정의**

모든 상수는 @const정적 속성 또는 모듈 로컬 const 선언이지만 모든 @const정적 속성과 모듈 로컬 const이 상수 인 것은 아닙니다 . 상수 케이스를 선택하기 전에 필드가 정말 불변의 상수 처럼 느껴지는 지 고려하십시오 . 예를 들어 해당 인스턴스의 관찰 가능 상태가 변경 될 수 있다면 거의 확실하게 상수가 아닙니다. 단순히 객체를 변경하지 않으려는 것만으로는 일반적으로 충분하지 않습니다.

예 :

// 상수 const NUMBER = 5 ; / \*\* @const \* / 내보내기 . 이름 = ImmutableList . of ( 'Ed' , 'Ann' ); / \*\* @enum \* / 내보내기 . SomeEnum = { ENUM\_CONSTANT : '값' };

// 상수 아님 let letVariable = 'non-const' ; class MyClass { constructor () { / \*\* @const {string} \* / this . nonStatic = '비 정적' ; } }; / \*\* @type {string} \* / MyClass . staticButMutable = '@const가 아님, 재 할당 가능' ; const / \*\* Set <문자열> \* / mutableCollection = new Set (); const / \*\* ImmutableSet <SomeMutableType> \* / mutableElements = ImmutableSet

. 의 ( 변경 가능 ); const Foo = goog . 필요 ( 'my.Foo' ); // 가져온 이름을 미러링 합니다. const logger = log . getLogger ( 'loggers.are.not.immutable' );

상수의 이름은 일반적으로 명사 또는 명사구입니다.

**6.2.5.2 로컬 별칭**

정규화 된 이름보다 가독성이 향상 될 때마다 로컬 별칭을 사용해야합니다. 별칭이 지정된 이름의 마지막 부분을 유지하면서 goog.requires ( 3.6 goog.require 및 goog.requireType 문 ) 와 동일한 규칙을 따릅니다 . 함수 내에서도 별칭을 사용할 수 있습니다. 별칭은 const.

예 :

const staticHelper = importedNamespace . staticHelper ; const CONSTANT\_NAME = ImportedClass . CONSTANT\_NAME ; const { assert , assertInstanceof } = 주장 ;

**6.2.6 상수가 아닌 필드 이름**

상수가 아닌 필드 이름 (정적 또는 기타)은으로 작성되며 lowerCamelCase개인 필드의 경우 뒤에 밑줄이 표시됩니다.

이러한 이름은 일반적으로 명사 또는 명사 구입니다. 예를 들어, computedValues 또는 index\_.

**6.2.7 매개 변수 이름**

매개 변수 이름은 lowerCamelCase. 이것은 매개 변수가 생성자를 예상하는 경우에도 적용됩니다.

공용 메소드에서는 한 문자 매개 변수 이름을 사용해서는 안됩니다.

예외 : 타사 프레임 워크에서 요구하는 경우 매개 변수 이름은 $. 이 예외는 다른 식별자 (예 : 지역 변수 또는 속성)에는 적용되지 않습니다.

**6.2.8 지역 변수 이름**

lowerCamelCase위에서 설명한대로 모듈 로컬 (최상위) 상수를 제외하고 로컬 변수 이름은로 작성됩니다 . 함수 범위의 상수는 여전히 lowerCamelCase. 참고 lowerCamelCase변수는 생성자를 보유하는 경우에도 사용된다.

**6.2.9 템플릿 매개 변수 이름**

템플릿 매개 변수 이름은 간결하고, 한 단어 또는 단일 문자 식별자해야하고, 같은 모든 모자,해야 TYPE하거나 THIS.

**6.2.10 모듈 로컬 이름**

내 보내지 않은 모듈 로컬 이름은 암시 적으로 비공개입니다. 표시 @private되지 않고 밑줄로 끝나지 않습니다. 이것은 클래스, 함수, 변수, 상수, 열거 형 및 기타 모듈 로컬 식별자에 적용됩니다.

**6.3 카멜 케이스 : 정의 됨**

때로는 두문자어 또는 IPv6 또는 iOS 와 같은 비정상적인 구조가있는 경우와 같이 영어 구를 낙타 대문자로 변환하는 합리적인 방법이 여러 가지 있습니다. 예측 가능성을 높이기 위해 Google Style은 다음과 같은 (거의) 결정적 체계를 지정합니다.

이름의 산문 형식으로 시작 :

1. 구를 일반 ASCII로 변환하고 아포스트로피를 제거하십시오. 예를 들어 Müller의 알고리즘 은 Muellers 알고리즘 이 될 수 있습니다 .

2. 이 결과를 단어로 나누고 공백과 나머지 구두점 (일반적으로 하이픈)으로 분리합니다.

a. 권장 : 일반적으로 사용되는 일반적인 낙타 케이스 모양이 이미있는 단어가있는 경우이를 구성 부분으로 분할합니다 (예 : 애드워즈 가 광고 단어가 됨 ). iOS 와 같은 단어 는 실제로 낙타 대문자 자체가 아닙니다. 어떤 규칙도 위반하므로이 권장 사항은 적용되지 않습니다.

3. 이제 모든 것을 소문자 (약어 포함)하고 다음의 첫 번째 문자 만 대문자로 표시합니다.

a. … 각 단어, 낙타 대문자를 산출하거나

b. … 첫 번째 단어를 제외한 각 단어는 낮은 낙타 케이스를 생성합니다.

4. 마지막으로 모든 단어를 단일 식별자로 결합합니다.

원래 단어의 대소 문자는 거의 전적으로 무시됩니다.

예 :

산문 형식 옳은 틀렸다

XML HTTP 요청 XmlHttpRequest XMLHTTPRequest

새 고객 ID newCustomerId newCustomerID

내부 스톱워치 innerStopwatch innerStopWatch

iOS에서 IPv6를 지원합니까? supportsIpv6OnIos 지원 IPv6OnIOS

YouTube 가져 오기 YouTubeImporter YoutubeImporter \*

\* 허용되지만 권장되지는 않습니다.

참고 : 어떤 단어가 모호하게 영어로 하이픈 있습니다 : 예에 대한 비어 있지 않은 및 비어 있지 않은이 있습니다 모두 올바른 방법 이름 checkNonempty 및 checkNonEmpty 마찬가지로 모두 올바른지 있도록.

**7 JSDoc**

JSDoc 은 모든 클래스, 필드 및 메서드에 사용됩니다.

**7.1 일반 형식**

JSDoc 블록의 기본 형식은 다음 예와 같습니다.

/ \*\*

\* 여기에는 여러 줄의 JSDoc 텍스트가 작성되어 있습니다.

\* 일반적으로 포장됩니다.

\* @param {number} arg 무언가를 할 숫자.

\* / function doSomething ( arg ) { … }

또는이 한 줄 예제에서 :

/ \*\* @const @private {! Foo} JSDoc의 짧은 부분입니다. \* / 이 . foo\_ = foo ;

한 줄 주석이 여러 줄로 넘치면 자체 줄 /\*\*과 함께 여러 줄 스타일을 사용해야합니다 \*/.

많은 도구가 JSDoc 주석에서 메타 데이터를 추출하여 코드 유효성 검사 및 최적화를 수행합니다. 따라서 이러한 주석은 올바른 형식 이어야합니다.

**7.2 마크 다운**

JSDoc은 Markdown으로 작성되지만 필요한 경우 HTML을 포함 할 수 있습니다.

JSDoc (예 : JsDossier ) 을 자동으로 추출하는 도구 는 종종 일반 텍스트 형식을 무시하므로 다음과 같이하면됩니다.

/ \*\*

\* 세 가지 요소를 기반으로 가중치를 계산합니다.

\* 보낸 항목

\*받은 항목

\* 마지막 타임 스탬프

\* /

다음과 같이 나옵니다.

세 가지 요소를 기반으로 무게를 계산합니다. 보낸 항목은 마지막 타임 스탬프를 받았습니다.

대신 Markdown 목록을 작성하십시오.

/ \*\*

\* 세 가지 요소를 기반으로 가중치를 계산합니다.

\*-보낸 항목

\*-받은 항목

\*-마지막 타임 스탬프

\* /

**7.3 JSDoc 태그**

Google 스타일은 JSDoc 태그의 하위 집합을 허용합니다. 전체 목록 은 9.1 JSDoc 태그 참조 를 참조 하십시오. 대부분의 태그는 줄의 시작 부분에 태그가있는 자체 줄을 차지해야합니다.

허용되지 않음 :

/ \*\*

\* "param"태그는 자체 줄을 차지해야하며 결합 할 수 없습니다.

\* @param {number} 왼쪽 @param {number} 오른쪽

\* / 함수 추가 ( 왼쪽 , 오른쪽 ) { ... }

추가 데이터를 필요로하지 않는 간단한 태그 (예는 @private, @const, @final, @export) 때 적절한 옵션 형과 함께 같은 라인에 결합 될 수있다.

/ \*\*

\* 더 복잡한 주석 (예 : "구현"및 "템플릿") 배치

\* 자신의 라인에. 여러 개의 단순 태그 (예 : "내보내기"및 "최종")

\* 한 줄로 결합 할 수 있습니다.

\* @ 내보내기 @final

\* @implements {Iterable <TYPE>}

\* @template TYPE

\* / 클래스 MyClass { / \*\*

\* @param {! ObjType} obj 일부 개체입니다.

\* @param {number =} num 선택적 숫자입니다.

\* / constructor ( obj , num = 42 ) { / \*\* @private @const {! Array <! ObjType | number>} \* / this . data\_ = [ obj , num ]; } }

태그를 결합 할시기 또는 순서에 대한 엄격한 규칙은 없지만 일관성을 유지하십시오.

JavaScript의 주석 유형에 대한 일반적인 정보 는 클로저 컴파일러 및 클로저 유형 시스템의 유형에 대한 JavaScript 주석 달기를 참조하십시오.

**7.4 줄 바꿈**

줄 바꿈 된 블록 태그는 네 개의 공백으로 들여 쓰기됩니다. 줄 바꿈 된 설명 텍스트는 이전 줄의 설명과 함께 정렬 될 수 있지만 이러한 수평 정렬은 권장되지 않습니다.

/ \*\*

\* 긴 매개 변수 / 반환 설명을위한 줄 바꿈을 보여줍니다.

\* @param {string} foo 설명이 너무 길어서 들어갈 수없는 매개 변수입니다.

\* 한 줄.

\* @return {number} 설명이 너무 길어서

\* 한 줄에 맞습니다.

\* /

수출 . method = function ( foo ) { return 5 ; };

@desc또는 @fileoverview설명을 줄 바꿈 할 때 들여 쓰지 마십시오 .

**7.5 최상위 / 파일 수준 주석**

파일에는 최상위 파일 개요가있을 수 있습니다. 저작권 고지, 작성자 정보 및 기본 표시 수준 은 선택 사항입니다. 파일 개요는 일반적으로 파일이 하나 이상의 클래스 정의로 구성 될 때마다 권장됩니다. 최상위 주석은 코드에 익숙하지 않은 독자가이 파일에있는 내용을 알 수 있도록 설계되었습니다. 있는 경우 파일의 내용에 대한 설명과 종속성 또는 호환성 정보를 제공 할 수 있습니다. 줄 바꿈은 들여 쓰기되지 않습니다.

예:

/ \*\*

\* @fileoverview 파일 설명, 용도 및 정보

\* 종속성에 대해.

\* @ 패키지

\* /

7.6 수업 댓글

클래스, 인터페이스 및 레코드는 설명과 템플릿 매개 변수, 구현 된 인터페이스, 가시성 또는 기타 적절한 태그와 함께 문서화되어야합니다. 클래스 설명은 클래스를 올바르게 사용하는 데 필요한 추가 고려 사항뿐만 아니라 클래스 사용 방법과시기를 알 수있는 충분한 정보를 독자에게 제공해야합니다. 생성자에서 텍스트 설명을 생략 할 수 있습니다. @constructor그리고 @extends주석이 사용되지 않습니다 class클래스가 선언을 사용하지 않는 키워드 @interface또는 제네릭 클래스를 확장합니다.

/ \*\*

\* 멋진 일을하는 더 멋진 이벤트 대상.

\* @implements {Iterable <string>}

\* / MyFancyTarget 클래스는 EventTarget { / \*\*을 확장합니다.

\* @param {string} arg1 이것을 더 흥미롭게 만드는 인수.

\* @param {! Array <number>} arg2 처리 할 숫자 목록입니다.

\* / constructor ( arg1 , arg2 ) { // ... } };

/ \*\*

\* 기록도 도움이됩니다.

\* @extends {Iterator <TYPE>}

\* @record

\* @template TYPE

\* / class Listable { / \*\* @return {TYPE} 반환 할 줄의 다음 항목입니다. \* /

다음 () {} }

**7.7 열거형 및 typedef 주석**

모든 열거 형 및 typedef는 선행 줄에 적절한 JSDoc 태그 ( @typedef또는 @enum) 로 문서화되어야합니다 . 공용 열거 형 및 typedef에도 설명이 있어야합니다. 개별 열거 항목은 앞 줄에 JSDoc 주석으로 문서화 할 수 있습니다.

/ \*\*

\* 자주 재사용되는 유용한 유형 공용체.

\* @typedef {! Bandersnatch |! BandersnatchType}

\* / let CoolUnionType ;

/ \*\*

\* 밴더 스내치의 종류.

\* @enum {string}

\* / const BandersnatchType = { / \*\* 이런 종류는 정말 답답합니다. \* /

FRUMIOUS : ' frumious ' , / \*\* 덜 시끄러운 종류. \* /

MANXOME : 'manxome' , };

Typedef는 짧은 레코드 유형을 정의하거나 공용체, 복합 함수 또는 일반 유형의 별칭을 정의하는 데 유용합니다. 개별 필드를 문서화하거나 템플릿 또는 재귀 참조를 사용할 수 없으므로 필드가 많은 레코드 유형에 대해서는 Typedef를 사용하지 않아야합니다. 큰 레코드 유형의 경우 @record.

**7.8 방법 및 기능 설명**

메서드와 명명 된 함수 @override에서 모든 유형이 생략 된 동일 서명 의 경우를 제외하고 매개 변수 및 반환 유형은 문서화되어야합니다 . this유형은 필요한 경우 문서화되어야한다. 함수에 비어 있지 않은 return문 이 없으면 반환 유형을 생략 할 수 있습니다 .

메소드, 매개 변수 및 리턴 설명 (유형이 아님)은 메소드의 나머지 JSDoc 또는 해당 서명에서 분명한 경우 생략 할 수 있습니다.

메서드 설명은 메서드가 수행하는 작업을 설명하는 동사 구문으로 시작됩니다. 이 문구는 명령문이 아니지만 마치 그 앞에이 방법이 암시 된 것처럼 3 인칭으로 작성됩니다 .

메서드가 슈퍼 클래스 메서드를 재정의하는 경우 @override 주석을 포함해야합니다 . 재정의 된 메서드는 슈퍼 클래스 메서드 (가시성 주석 포함)에서 모든 JSDoc 주석을 상속하며 재정의 된 메서드에서 생략해야합니다. 그러나 유형 어노테이션에서 유형이 구체화되면 모든 @param및 @return어노테이션을 명시 적으로 지정해야합니다.

/ \*\* 무언가를하는 클래스. \* / class SomeClass는 SomeBaseClass를 확장합니다 { / \*\*

\* MyClass의 인스턴스에서 작동하고 무언가를 반환합니다.

\* @param {! MyClass} obj 어떤 이유로 상세해야하는 객체

\* 여러 줄에 걸친 설명.

\* @param {! OtherClass} obviousOtherClass

\* @return {boolean} 어떤 일이 발생했는지 여부.

\* /

someMethod ( obj , obviousOtherClass ) { ... }

/ \*\* @override \* /

overriddenMethod ( param ) { ... } }

/ \*\*

\* 최상위 기능이 동일한 규칙을 따르는 방법을 보여줍니다. 이 하나

\* 배열을 만듭니다.

\* @param {TYPE} 인수

\* @return {! Array <TYPE>}

\* @template TYPE

\* / 함수 makeArray ( arg ) { ... }

함수의 매개 변수 및 반환 유형 만 문서화해야하는 경우 선택적으로 함수의 서명에 인라인 JSDocs를 사용할 수 있습니다. 이러한 인라인 JSDocs는 태그없이 return 및 param 유형을 지정합니다.

function / \*\* string \* / foo ( / \*\* number \* / arg ) {...}

설명이나 태그가 필요한 경우 메서드 위에 단일 JSDoc 주석을 사용합니다. 예를 들어 값을 반환하는 메서드에는 @return태그 가 필요합니다 .

class MyClass { / \*\*

\* @param {number} 인수

\* @return {문자열}

\* /

bar ( arg ) {...} }

// 잘못된 인라인 JSDocs.

class MyClass { / \*\* @return {string} \* / foo () {...} }

/ \*\* 기능 설명. \* / 막대 () {...}

익명 함수에서 주석은 일반적으로 선택 사항입니다. 자동 유형 추론이 불충분하거나 명시 적 주석이 가독성을 높이면 param에 주석을 달고 다음과 같이 유형을 반환합니다.

약속 . then ( / \*\* @return {string} \* / ( / \*\*! Array <string> \* / items ) => {

doSomethingWith ( items ); return items [ 0 ]; });

함수 유형 표현식은 7.10.4 함수 유형 표현식을 참조하십시오.

**7.9 부동산 설명**

속성 유형을 문서화해야합니다. 이름과 유형이 코드를 이해하는 데 충분한 문서를 제공하는 경우 개인 속성에 대한 설명을 생략 할 수 있습니다.

공개적으로 내 보낸 상수는 속성과 같은 방식으로 주석 처리됩니다.

/\*\* 내 수업. \* / class MyClass { / \*\* @param {string =} someString \* / constructor ( someString = 'default string' ) { / \*\* @private @const {string} \* / this . someString\_ = someString ;

/ \*\* @private @const {! OtherType} \* / 이 . someOtherThing\_ = functionThatReturnsAThing ();

/ \*\*

\* 창당 최대 사물 수.

\* @type {number}

\* / 이 . someProperty = 4 ; } }

/ \*\*

\* 포기하기 전에 시도 할 횟수.

\* @const {숫자}

\* / MyClass . RETRY\_COUNT = 33 ;

**7.10 유형 주석**

유형 주석에서 찾을 수있는 @param, @return, @this, 및 @type태그, 선택적에 @const, @export, 및 가시성 태그입니다. JSDoc 태그에 첨부 된 유형 주석은 항상 중괄호로 묶어야합니다.

**7.10.1 무효성**

유형 시스템은 각각 수정 자 !와 ?널이 아닌 널을 정의합니다 . 이러한 수정자는 유형 앞에 와야합니다.

Nullability 수정자는 다양한 유형에 대해 서로 다른 요구 사항을 가지며 다음 두 가지 범주로 나뉩니다.

1. 유형 프리미티브에 대한 주석 ( string, number, boolean, symbol, undefined, null)와 리터럴 ( {function(...): ...}과 {{foo: string...}}) 항상 기본적으로 비 - 널 (NULL)입니다. ?수정자를 사용하여 null을 허용하지만 중복 !.

2. 참조 유형 (일반적으로을 UpperCamelCase포함하여의 모든 것 some.namespace.ReferenceType)은 다른 곳에 정의 된 클래스, 열거 형, 레코드 또는 typedef를 참조합니다. 이러한 유형은 null을 허용하거나 허용하지 않을 수 있으므로 이름만으로는 null 허용 여부를 알 수 없습니다. 사용 사이트에서 모호함을 방지하기 위해 이러한 유형에 대해 항상 명시 적 ?및 !수정자를 사용하십시오.

나쁜:

const / \*\* MyObject \* / myObject = null ; // 기본이 아닌 유형에는 주석을 달아야합니다. const / \*\*! 숫자 \* / someNum = 5 ; // 프리미티브는 기본적으로 널이 아닙니다. const / \*\* 숫자? \* / someNullableNum = null ; //? 유형 앞에 와야합니다. const / \*\*! {foo : string, bar : number} \* / record = ...; // 이미 nullable이 아닙니다. const / \*\* MyTypeDef \* / def = ...; // MyTypeDef가 nullable인지 확실하지 않습니다.

// object (nullable), enum (nullable, 달리 지정 되지 않은 경우 //) 또는 typedef (정의에 따라 다름) 인지 확실하지 않습니다 . const / \*\* SomeCamelCaseName \* / n = ...;

좋은:

const / \*\*? MyObject \* / myObject = null ; const / \*\* 숫자 \* / someNum = 5 ; const / \*\*? 숫자 \* / someNullableNum = null ; const / \*\* {foo : string, bar : number} \* / record = ...; const / \*\*! MyTypeDef \* / def = ...; const / \*\*? SomeCamelCaseName \* / n = ...;

**7.10.2 유형 캐스트**

컴파일러가 표현식의 유형을 정확하게 추론하지 못하고 goog.asserts 의 assertion 함수가 이를 수정할 수없는 경우 유형 주석 주석을 추가하고 표현식을 괄호로 묶어 유형을 강화할 수 있습니다. 괄호가 필요합니다.

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

**7.10.3 템플릿 매개 변수 유형**

항상 템플릿 매개 변수를 지정하십시오. 이러한 방식으로 컴파일러는 더 나은 작업을 수행 할 수 있으며 독자가 코드의 기능을 더 쉽게 이해할 수 있습니다.

나쁜:

const / \*\*! Object \* / users = {}; const / \*\*! 배열 \* / books = []; const / \*\*! Promise \* / response = ...;

좋은:

const / \*\*! Object <string ,! User> \* / users = {}; const / \*\*! Array <string> \* / books = []; const / \*\*! Promise <! Response> \* / response = ...;

const / \*\*! Promise <undefined> \* / thisPromiseReturnsNothingButParameterIsStillUseful = ...; const / \*\*! Object <문자열, \*> \* / mapOfEverything = {};

템플릿 매개 변수를 사용하지 않아야하는 경우 :

• Object 맵과 같은 구조가 아닌 유형 계층 구조에 사용됩니다.

**7.10.4 함수 유형 표현식**

용어 참고 : 함수 유형 표현식 은 주석에 키워드 function가있는 함수 유형에 대한 유형 주석을 나타냅니다 (아래 예제 참조).

함수 정의가 제공된 경우 함수 유형 표현식을 사용하지 마십시오. @param및 @return또는 인라인 주석을 사용 하여 매개 변수 및 반환 유형을 지정합니다 ( 7.8 메서드 및 함수 주석 참조 ). 여기에는 const에 정의되고 할당 된 익명 함수 및 함수가 포함됩니다 (jsdoc 함수가 전체 할당 표현식 위에 표시됨).

기능 유형 표현식은 예를 들어, 내부를 들어, 필요 @typedef, @param 또는 @return. 함수 정의로 즉시 초기화되지 않는 경우 함수 유형의 변수 또는 속성에도 사용합니다.

/ \*\* @private {function (string) : string} \* / this . idGenerator\_ = googFunctions . 정체성 ;

함수 유형 표현식을 사용하는 경우 항상 반환 유형을 명시 적으로 지정하십시오. 그렇지 않으면 기본 반환 유형이 알 수 없음 ( ?)으로, 이상하고 예기치 않은 동작이 발생하며 실제로 원하는 것은 거의 없습니다.

불량-유형 오류이지만 경고가 제공되지 않음 :

/ \*\* @param {function ()} generateNumber \* / function foo ( generateNumber ) { const / \*\* number \* / x = generateNumber (); // 여기에는 컴파일 타임 유형 오류가 없습니다. }

foo (() => '분명히 숫자가 아닙니다' );

좋은:

/ \*\*

\* @param {function () : \*} inputFunction1 모든 유형을 반환 할 수 있습니다.

\* @param {function () : undefined} inputFunction2 확실히 반환하지 않음

\* 아무것도.

\* 참고 :`foo` 자체의 반환 유형은 {undefined}로 안전하게 암시됩니다.

\* / 함수 foo ( inputFunction1 , inputFunction2 ) {...}

**7.10.5 공백**

유형 주석 내에서 각 쉼표 또는 콜론 뒤에는 단일 공백 또는 줄 바꿈이 필요합니다. 가독성을 높이거나 열 제한을 초과하지 않도록 추가 줄 바꿈을 삽입 할 수 있습니다. 이러한 구분은 해당 지침 (예 : 4.5 줄 바꿈 및 4.2 블록 들여 쓰기 : +2 공백 )에 따라 선택하고 들여 쓰기해야합니다 . 유형 주석에 다른 공백은 허용되지 않습니다.

좋은:

/ \*\* @type {함수 (문자열) : 숫자} \* /

/ \*\* @type {{foo : 숫자, 막대 : 숫자}} \* /

/ \*\* @ 유형 {숫자 | 문자열} \* /

/ \*\* @type {! Object <문자열, 문자열>} \* /

/ \*\* @type {function (this : Object <string, string>, number) : string} \* /

/ \*\*

\* @type {기능 (

\*! SuperDuperReallyReallyLongTypedefThatForcesTheLineBreak,

\*! OtherVeryLongTypedef) : 문자열}

\* /

/ \*\*

\* @type {! SuperDuperReallyReallyLongTypedefThatForcesTheLineBreak |

\*! OtherVeryLongTypedef}

\* /

나쁜:

// 콜론 뒤에 공백 만 넣습니다. / \*\* @type {function (string) : number} \* /

// 콜론과 쉼표 뒤에 공백을 넣습니다. / \*\* @type {{foo : number, bar : number}} \* /

// 공용체 유형에 공백 없음 / \*\* @type {number | 문자열} \* /

**7.11 가시성 주석**

가시성 주석은 ( @private, @package, @protected)이 지정 될 수있다 @fileoverview블록 또는 반출 심볼 또는 구내. 함수 내에서든 모듈의 최상위 수준에서든 지역 변수에 대한 가시성을 지정하지 마십시오. 모든 @private이름은 밑줄로 끝나야합니다.

**8 정책**

**8.1 Google 스타일에서 지정하지 않은 문제 : 일관성 유지!**

이 사양에 의해 명확하게 해결되지 않은 스타일 질문에 대해서는 동일한 파일의 다른 코드가 이미 수행하고있는 작업을 선호합니다. 그래도 문제가 해결되지 않으면 동일한 패키지의 다른 파일을 에뮬레이션하는 것이 좋습니다.

**8.2 컴파일러 경고**

**8.2.1 표준 경고 세트 사용**

가능한 한 프로젝트에서 --warning\_level=VERBOSE.

**8.2.2 경고 처리 방법**

무엇이든하기 전에 경고가 말하는 내용을 정확히 이해해야합니다. 경고가 나타나는 이유가 확실하지 않은 경우 도움을 요청하십시오.

경고를 이해했으면 다음 해결 방법을 순서대로 시도하십시오.

1. 먼저 문제를 해결하거나 해결하십시오. 실제로 경고를 해결하려고 강력하게 시도하거나 상황을 완전히 피하는 작업을 수행 할 다른 방법을 찾으십시오.

2. 그렇지 않으면 잘못된 경보인지 확인하십시오. 경고가 유효하지 않고 코드가 실제로 안전하고 정확하다고 확신하는 경우 독자에게이 사실을 설득하고 @suppress 주석을 적용하는 주석을 추가하십시오.

3. 그렇지 않으면 TODO 코멘트를 남겨주세요. 이것은 최후의 수단 입니다. 이렇게하는 경우 경고를 표시하지 마십시오. 경고가 제대로 처리 될 때까지 표시되어야합니다.

**8.2.3 합리적인 범위에서 경고 표시 안 함**

경고는 일반적으로 단일 지역 변수 또는 매우 작은 방법의 범위와 같은 가장 좁은 합리적인 범위에서 억제됩니다. 종종 그 이유만으로 변수 나 메서드가 추출됩니다.

예

/ \*\* @suppress {uselessCode} 인식 할 수없는 'use asm'선언 \* / function fn () { 'use asm' ; 반환 0 ; }

클래스에서 많은 수의 억제조차도 전체 클래스가 이러한 유형의 경고를 보지 못하게하는 것보다 낫습니다.

**8.3 지원 중단**

사용되지 않는 메서드, 클래스 또는 인터페이스를 @deprecated주석으로 표시합니다. 지원 중단 설명에는 사람들이 통화 사이트를 수정하도록 간단하고 명확한 지침이 포함되어야합니다.

**8.4 Google 스타일에없는 코드**

코드베이스에서 적절한 Google 스타일이 아닌 파일이 가끔 발생합니다. 이는 인수로 인한 것일 수도 있고 Google Style이 일부 문제에 대한 입장을 취하기 전에 작성되었거나 다른 이유로 Google Style이 아닐 수도 있습니다.

**8.4.1 기존 코드 재 형식화**

기존 코드의 스타일을 업데이트 할 때 다음 지침을 따르십시오.

1. 현재 스타일 지침을 충족하기 위해 모든 기존 코드를 변경할 필요는 없습니다. 기존 코드의 형식을 다시 지정하는 것은 코드 변동과 일관성 사이의 균형입니다. 스타일 규칙은 시간이 지남에 따라 진화하며 규정 준수를 유지하기위한 이러한 종류의 조정은 불필요한 이탈을 유발합니다. 그러나 파일이 크게 변경되는 경우 파일이 Google 스타일로되어있을 것으로 예상됩니다.

2. 기회 주의적 스타일 수정이 CL의 초점을 방해하지 않도록주의하십시오. CL의 중심 초점에 중요하지 않은 스타일 변경을 많이하는 경우 해당 변경 사항을 별도의 CL로 승격하십시오.

**8.4.2 새로 추가 된 코드 : Google 스타일 사용**

새로운 파일은 동일한 패키지에있는 다른 파일의 스타일 선택에 관계없이 Google 스타일을 사용합니다.

Google 스타일이 아닌 파일에 새 코드를 추가하는 경우 8.4.1 기존 코드 다시 형식화의 조언에 따라 먼저 기존 코드 형식을 다시 지정하는 것이 좋습니다 .

이 형식 변경이 완료되지 않은 경우 새 코드는 동일한 파일의 기존 코드와 최대한 일치해야하지만 스타일 가이드를 위반해서는 안됩니다.

**8.5 지역 스타일 규칙**

팀과 프로젝트는이 문서에있는 것 이상의 추가 스타일 규칙을 채택 할 수 있지만 정리 변경이 이러한 추가 규칙을 따르지 않을 수 있으며 추가 규칙을 위반하여 이러한 정리 변경을 차단해서는 안된다는 점을 인정해야합니다. 목적이없는 과도한 규칙에주의하십시오. 스타일 가이드는 가능한 모든 시나리오에서 스타일을 정의하려는 것이 아니며 여러분도 그렇게해서는 안됩니다.

**8.6 생성 된 코드 : 대부분 면제**

빌드 프로세스에서 생성 된 소스 코드는 Google 스타일 일 필요가 없습니다. 그러나 손으로 작성한 소스 코드에서 참조 할 생성 된 식별자는 명명 요구 사항을 따라야합니다. 특별한 예외로, 이러한 식별자에는 밑줄을 포함 할 수 있으므로 손으로 쓴 식별자와의 충돌을 방지하는 데 도움이 될 수 있습니다.

**9 부록**

**9.1 JSDoc 태그 참조**

JSDoc은 JavaScript에서 다양한 용도로 사용됩니다. 문서를 생성하는 데 사용되는 것 외에도 도구를 제어하는데도 사용됩니다. 가장 잘 알려진 것은 Closure Compiler 유형 주석입니다.

**9.1.1 유형 주석 및 기타 클로저 컴파일러 주석**

클로저 컴파일러에서 사용하는 JSDoc에 대한 문서 는 클로저 컴파일러 및 클로저 유형 시스템의 유형에 대한 JavaScript 주석 달기에 설명되어 있습니다.

**9.1.2 문서 주석**

Closure Compiler를위한 Annotating JavaScript에 설명 된 JSDoc 외에도 다음 태그는 순전히 문서화 목적으로 다양한 문서 생성 도구 (예 : JsDossier )에서 일반적이며 잘 지원 됩니다.

타사 코드에서 다른 유형의 JSDoc 주석을 볼 수도 있습니다. 이러한 주석은 JSDoc Toolkit 태그 참조에 표시 되지만 유효한 Google 스타일의 일부로 간주되지 않습니다.

**9.1.2.1 @author또는 @owner- 권장하지 않습니다.**

권장하지 않습니다.

통사론: @author username@google.com (First Last)

/ \*\*

\* @fileoverview 텍스트 영역 처리를위한 유틸리티.

\* @author kuth@google.com (Uthur Pendragon)

\* /

일반적으로 @fileoverview주석 에서만 사용되는 파일 작성자 또는 테스트 소유자를 문서화합니다 . @owner태그는 시험 결과를 누가 소유하고 결정하는 단위 테스트 대시 보드에서 사용됩니다.

**9.1.2.2 @bug**

통사론: @bug bugnumber

/ \*\* @bug 1234567 \* / function testSomething () { //… }

/ \*\*

\* @ 버그 1234568

\* @ 버그 1234569

\* / function testTwoBugs () { //… }

주어진 테스트 함수 회귀가 테스트하는 버그를 나타냅니다.

@bug회귀 테스트를 최대한 쉽게 검색 할 수 있도록 여러 버그에 각각 고유 한 줄 이 있어야 합니다.

**9.1.2.3 @code- 추천하지 않습니다. 사용하지 마세요.**

더 이상 사용되지 않습니다. 사용하지 마세요. 대신 마크 다운 백틱을 사용하세요.

통사론: {@code ...}

역사적 `BatchItem`으로 {@code BatchItem}.

/ \*\* 보류중인`BatchItem` 인스턴스를 처리합니다. \* / function processBatchItems () {}

JSDoc 설명의 용어가 코드이므로 생성 된 문서에서 올바르게 형식화 될 수 있음을 나타냅니다.

**9.1.2.4 @desc**

통사론: @desc Message description

/ \*\* @desc 계정이 생성되었음을 사용자에게 알립니다. \* /

수출 . MSG\_ACCOUNT\_CREATED = goog . getMsg ( '귀하의 계정이 성공적으로 생성되었습니다.' );

**9.1.2.5 @link**

통사론: {@link ...}

이 태그는 생성 된 문서 내에서 상호 참조 링크를 생성하는 데 사용됩니다.

/ \*\* 보류중인 {@link BatchItem} 인스턴스를 처리합니다. \* / function processBatchItems () {}

기록 참고 : @link 태그는 생성 된 문서에서 외부 링크를 만드는데도 사용되었습니다. 외부 링크의 경우 항상 Markdown의 링크 구문을 대신 사용하십시오.

/ \*\*

\*이 클래스는

\* [네이티브 이벤트 인터페이스] (https://dom.spec.whatwg.org/#event).

\* / 클래스 ApplicationEvent {}

**9.1.2.6 @see**

통사론: @see Link

/ \*\*

\* 무모하게 단일 항목을 추가합니다.

\* @ #addSafely 참조

\* @see goog.Collect

\* @see goog.RecklessAdder # add

\* /

다른 클래스 함수 또는 메서드에 대한 조회를 참조합니다.

**9.1.2.7 @supported**

통사론: @supported Description

/ \*\*

\* @fileoverview 이벤트 관리자

\* 브라우저의 이벤트 시스템에 추상화 된 인터페이스를 제공합니다.

\* @supported IE10 +, Chrome, Safari

\* /

파일에서 지원하는 브라우저를 나타 내기 위해 파일 개요에 사용됩니다.

**9.1.3 프레임 워크 특정 주석**

다음 주석은 특정 프레임 워크에 고유합니다.

@ngInjectAngular 1의 경우 9.1.3.1

@polymerBehavior폴리머의 경우 9.1.3.2

<https://github.com/google/closure-compiler/wiki/Polymer-Pass>

**9.1.4 표준 클로저 컴파일러 주석에 대한 참고 사항**

다음 태그는 표준 이었지만 이제는 더 이상 사용되지 않습니다.

**9.1.4.1 @expose- 추천하지 않습니다. 사용하지 마세요.**

더 이상 사용되지 않습니다. 사용하지 마세요. @export및 / 또는 @nocollapse대신 사용하십시오 .

**9.1.4.2 @inheritDoc- 추천하지 않습니다. 사용하지 마세요.**

더 이상 사용되지 않습니다. 사용하지 마세요. @override대신 사용하십시오.

**9.2 일반적으로 오해되는 스타일 규칙**

다음은 JavaScript 용 Google 스타일에 대해 잘 알려지지 않았거나 일반적으로 오해되는 사실의 모음입니다. (다음은 진실한 진술입니다. 이것은 신화 목록이 아닙니다 . )

• @author소스 파일 에는 저작권 진술이나 크레딧이 필요 하지 않습니다 . (둘 다 명시 적으로 권장되지 않습니다.)

• 클래스 ( 5.4 클래스 ) 의 구성원을 정렬하는 방법을 결정하는 엄격 하고 빠른 규칙 은 없습니다 .

• 빈 블록은 일반적으로 {}( 4.1.3 빈 블록 : 간결 할 수 있음 )에 자세히 설명 된대로 간결하게 표현할 수 있습니다 .

• 줄 바꿈의 주요 지침은 더 높은 구문 수준에서 중단하는 것을 선호합니다 ( 4.5.1 Where to break ).

• 비 ASCII 문자는 문자열 리터럴, 주석 및 JSDoc에 허용되며 실제로 동등한 유니 코드 이스케이프 ( 2.3.3 비 ASCII 문자 ) 보다 코드를 더 쉽게 읽을 수있는 경우 권장됩니다 .

**9.3 스타일 관련 도구**

다음 도구는 Google 스타일의 다양한 측면을 지원하기 위해 존재합니다.

**9.3.1 클로저 컴파일러**

이 프로그램은 유형 검사 및 기타 검사, 최적화 및 기타 변환 (예 : ECMAScript 6에서 ECMAScript 5 코드 낮추기)을 수행합니다.

**9.3.2 clang-format**

이 프로그램은 자바 스크립트 소스 코드를 Google 스타일로 형식화하고 필수는 아니지만 자주 가독성을 높이는 형식 지정 관행을 따릅니다. 에서 생성 한 출력 clang-format은 스타일 가이드를 준수합니다.

clang-format필요하지 않습니다. 작성자는 출력을 변경할 수 있으며 검토자는 이러한 변경을 요청할 수 있습니다. 분쟁은 일반적인 방식으로 해결됩니다. 그러나 하위 트리는 로컬에서 이러한 시행을 선택할 수 있습니다.

**9.3.3 클로저 컴파일러 린터**

이 프로그램은 다양한 실수와 안티 패턴을 검사합니다.

**9.3.4 적합성 프레임 워크**

JS Conformance Framework는 개발자에게 표준 검사보다 높은 코드 기반에 대해 실행할 추가 검사 집합을 지정하는 간단한 방법을 제공하는 Closure Compiler의 일부인 도구입니다. 예를 들어 적합성 검사는 특정 속성에 대한 액세스, 특정 함수 호출 또는 유형 정보 누락 (알 수 없음)을 금지 할 수 있습니다.

이러한 규칙은 일반적으로 중요한 제한 (예 : 코드베이스를 손상시킬 수있는 전역 정의) 및 보안 패턴 (예 : 사용 eval또는 할당 innerHTML)을 적용하거나 코드 품질을 개선하기 위해 더 느슨하게 사용됩니다.

추가 정보는 JS Conformance Framework의 공식 문서를 참조하십시오.

**9.4 레거시 플랫폼의 예외**

**9.4.1 개요**

이 섹션에서는 코드 작성자가 최신 ECMAScript 6 구문을 사용할 수없는 경우 따라야 할 예외 및 추가 규칙에 대해 설명합니다. 권장 스타일에 대한 예외는 ECMAScript 6 구문이 불가능할 때 필요하며 여기에 설명되어 있습니다.

• var선언 사용 이 허용됩니다.

• 사용 arguments이 허용됩니다

• 기본값이없는 선택적 매개 변수가 허용됩니다.

**9.4.2 사용 var**

**9.4.2.1 var선언은 블록 범위가 아닙니다.**

var선언은 가장 가까운 둘러싸는 함수, 스크립트 또는 모듈의 시작으로 범위가 지정되며, 이는 특히 var루프 내부의 선언 을 참조하는 함수 클로저에서 예기치 않은 동작을 유발할 수 있습니다 . 다음 코드는 예를 제공합니다.

for ( var i = 0 ; i < 3 ; ++ i ) { var iteration = i ;

setTimeout ( function () { console . log ( iteration ); }, i \* 1000 ); }

// 로그 2, 2, 2 금지 - 0, 1, 2 // iteration` 기능 범위의 루프에 로컬이 아닌`때문이다.

**9.4.2.2 최초 사용에 최대한 가깝게 변수 선언**

var선언의 범위가 둘러싸는 함수의 시작으로 지정 되더라도 선언은 var가독성을 위해 가능한 한 첫 번째 사용과 가까워 야합니다. 그러나 var해당 변수가 블록 외부에서 참조되는 경우 블록 내부 에 선언을 넣지 마십시오 . 예를 들면 :

function sillyFunction () { var count = 0 ; for ( var x in y ) { // "count"는 여기서 선언 할 수 있지만 그렇게하지 마십시오.

카운트 ++; }

콘솔 . log ( count + 'items in y' ); }

**9.4.2.3 상수 변수에 @const 사용**

const키워드가 사용되는 전역 선언의 경우 사용 가능한 경우 var대신 @const를 사용 하여 선언에 주석을 추가합니다 (지역 변수의 경우 선택 사항).

**9.4.3 블록 범위 함수 선언을 사용하지 마십시오**

마 하지 이렇게 :

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

ECMAScript 6 이전에 구현 된 대부분의 JavaScript VM은 블록 내에서 함수 선언을 지원하지만 표준화되지 않았습니다. 구현은 서로 일치하지 않고 블록 범위 함수 선언에 대한 현재 표준 ECMAScript 6 동작과 일치하지 않았습니다. ECMAScript 5 및 이전 버전에서는 스크립트 또는 함수의 루트 문 목록에서만 함수 선언을 허용하고 엄격 모드의 블록 범위에서 명시 적으로 금지합니다.

일관된 동작을 얻으려면 대신 var함수 표현식으로 초기화 된을 사용 하여 블록 내에서 함수를 정의하십시오.

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

**9.4.4 goog.provide/를 사용한 종속성 관리goog.require**

**9.4.4.1 요약**

경고 : goog.provide종속성 관리는 더 이상 사용되지 않습니다. goog.provide이전 파일을 사용 하는 프로젝트에서도 모든 새 파일 은 goog.module. 다음 규칙은 기존 goog.provide파일에만 적용됩니다.

• 모든 goog.provides를 먼저 배치하고 goog.requires를 두 번째로 배치하십시오. 빈 줄로 요구 사항에서 제공을 분리하십시오.

• 항목을 알파벳순으로 정렬합니다 (대문자 먼저).

• 포장 goog.provide및 goog.require진술 하지 마십시오 . 필요한 경우 80 열을 초과하십시오.

• 최상위 기호 만 제공하십시오.

goog.provide진술은 함께 그룹화되고 먼저 배치되어야합니다. 모든 goog.require진술은 따라야합니다. 두 목록은 빈 줄로 구분해야합니다.

다른 언어의 import 문과 유사 goog.provide하며 goog.require문은 80 개 열 줄 길이 제한을 초과하더라도 한 줄로 작성해야합니다.

행은 알파벳순으로 정렬되어야하며 대문자가 먼저옵니다.

goog . 제공 ( 'namespace.MyClass' );

goog . 제공 ( 'namespace.helperFoo' );

goog . require ( 'an.extremelyLongNamespace.thatSomeoneThought.wouldBeNice.andNowItIsLonger.Than80Columns' );

goog . 필요 ( 'goog.dom' );

goog . 필요 ( 'goog.dom.TagName' );

goog . 필요 ( 'goog.dom.classes' );

goog . 필요 ( 'goog.dominoes' );

클래스에 정의 된 모든 멤버는 동일한 파일에 있어야합니다. 동일한 클래스 (예 : 열거 형, 내부 클래스 등)에 정의 된 여러 멤버가 포함 된 파일에는 최상위 클래스 만 제공되어야합니다.

이 작업을 수행:

goog . 제공 ( 'namespace.MyClass' );

이거 말고:

goog . 제공 ( 'namespace.MyClass' );

goog . 제공 ( 'namespace.MyClass.CONSTANT' );

goog . 제공 ( 'namespace.MyClass.Enum' );

goog . 제공 ( 'namespace.MyClass.InnerClass' );

goog . 제공 ( 'namespace.MyClass.TypeDef' );

goog . 제공 ( 'namespace.MyClass.staticMethod' );

네임 스페이스의 멤버도 제공 될 수 있습니다.

goog . 제공 ( 'foo.bar' );

goog . 제공 ( 'foo.bar.CONSTANT' );

goog . 제공 ( 'foo.bar.method' );

**9.4.4.2 별칭 지정 goog.scope**

경고 : goog.scope더 이상 사용되지 않습니다. 새 파일은 goog.scope 기존 goog.scope사용이 있는 프로젝트에서도 사용해서는 안됩니다 .

goog.scopegoog.provide/ goog.require종속성 관리를 사용하여 코드에서 네임 스페이스 기호에 대한 참조를 단축하는 데 사용할 수 있습니다 .

goog.scope파일 당 하나의 호출 만 추가 할 수 있습니다. 항상 전역 범위에 배치하십시오.

열기 goog.scope(function() {호출은 정확히 하나의 빈 행 앞에 와야하며 모든 goog.provide명령문, goog.require명령문 또는 최상위 주석 뒤에 와야합니다 . 호출은 파일의 마지막 줄에서 닫아야합니다. // goog.scope범위의 닫는 문에 추가 하십시오. 세미콜론에서 두 개의 공백으로 주석을 분리하십시오.

C ++ 네임 스페이스와 유사하게 goog.scope선언 아래에서 들여 쓰지 마십시오 . 대신 0 열부터 계속하십시오.

다른 개체 (예 : 대부분의 생성자, 열거 형 및 네임 스페이스)에 다시 할당되지 않는 이름에 대해서만 별칭을 만듭니다. 이렇게하지 마십시오 (생성자의 별칭을 지정하는 방법은 아래 참조).

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

버튼 = function () { ... }; ...

이름은 별칭을 지정하는 전역의 마지막 속성과 동일해야합니다.

goog . 제공 ( 'my.module.SomeType' );

goog . 필요 ( 'goog.dom' );

goog . 필요 ( 'goog.ui.Button' );

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

// 생성자 선언 후 새 유형 별칭.

내 . 모듈 . SomeType = function () { ... }; var SomeType = my . 모듈 . SomeType ;

// 평소와 같이 프로토 타입에 메서드를 선언합니다. SomeType . 프로토 타입 . findButton = function () { // 위에서 별칭이 지정된 버튼. 이 . button = new Button ( dom . getElement ( 'my-button' )); }; ... }); // goog.scope

**9.4.4.3 goog.forwardDeclare**

동일한 라이브러리에있는 파일 간의 순환 종속성을 끊는 goog.requireType대신 사용하는 것이 좋습니다 goog.forwardDeclare. 달리가 goog.require하는 goog.requireType문은 정의하기 전에 네임 스페이스를 가져올 수있다.

goog.forwardDeclare 라이브러리 경계를 가로 지르는 순환 참조를 끊기 위해 레거시 코드에서 여전히 사용될 수 있지만 새로운 코드는이를 방지하기 위해 구조화되어야합니다.

goog.forwardDeclare문은 같은 스타일 규칙을 준수해야 goog.require하고 goog.requireType. goog.forwardDeclare, goog.require및 goog.requireType문의 전체 블록은 알파벳순으로 정렬됩니다.