JavaScript

General Style and Coding Conventions

June 30, 2022

Revision: 1.00

4050 Esplanade Way

Tallahassee, Florida 32399-0950

Ron DeSantis, Governor

J. Todd Inman, Secretary

Table of Contents

****

1. Revision History

Date	Revision	Author	Document Changes
6/30/2022	1.00	Stephen Mitchell	Initial Draft
Date			
Date			

Table 1: Revision History

2. Approvals

Role	Name	Title	Signature	Date
Project Sponsor:	Raghib Quresh	Utility System/Engineering Specialist		Date
Project Manager:	Rhonda Ballew	Service Delivery		Date
System Engineer:	Nelson Blocker	Service Delivery		Date

Table 2: Approvals

3. Points of Contact

Name	Title	Email	Contact
Rhonda Ballew	Service Delivery	rhonda.ballew@dms.fl.gov	(850) 922-7483
Nelson Blocker	Service Delivery	nelson.blocker@dms.fl.gov	(850) 921-6734
Raghib Quresh	System/Engineering Specialist	raghib.qureshi@dms.fl.gov	(850) 591-0260
DMS SUNCOM Helpdesk			(888) 478-6266

Table 3: Points of Contact

4. Related Documents

Related Code Standard Documents
C — General Style and Coding Conventions
CSS — General Style and Coding Conventions
HTML — General Style and Coding Conventions
PHP — General Style and Coding
PL/SQL — General Style and Coding Conventions
SQL — General Style and Coding Conventions
Typescript — General Style and Coding Conventions

Table 4: Related Documents

5. Introduction

A developer's method of coding will vary from developer to developer. Organizations must establish some house rules in order to keep their entire software engineering staff on the same page.

It is estimated that Google employs tens of thousands of software engineers. Together, they make tens of thousands of submissions of code every day, creating billions of lines of code. Without a coding style guide, individual preferences make the codebases difficult -- if not impossible — to understand and manage efficiently.

Google's extensive programming style guides are therefore not a surprise, which is why the company has crafted a collection of them. It is a set of rules and guidelines that are meant to ensure consistency between projects and programmers for the use of a particular programming language.

Even though the code might be written only once, it is read many times over the course of its lifetime.

Style guides for programming are a collection of rules and guidelines that describe how coding should be done. It is rules that explicitly constitutes good and bad programming.

The purpose of this coding style guide is to give programmers among other things, information on such topics as acceptable naming conventions, how to store source files, and how to format their code.

There are many benefits to code consistency, but the most important one is readability. Having consistent and readable code across an organization has the following benefits for engineers:

- focus on the value of the work rather than the minutiae of how it is coded;
- an easy understanding of any codebase by using familiar interfaces and formats;
- simplify the scaling and implementation of automation; and
- facilitate change, such as transferring products ownership, onboarding new personnel and codes handoffs.

There are some things that might not affect the functionality of software, such as naming conventions, indent spacing, and ordered import, but it is important that they are consistent and that they are understood by all engineering teams. When we decide on one [style], we've dropped out of the endless cycle of debate and can simply move on.

As a result of an extensive review of the literature on JavaScript coding and style conventions, the following conclusion was drawn; Google and Airbnb, tech giants, have two of the most prevailing style guides concerning JavaScript out there. No matter how you feel about their particular rules, it is still important to keep stylistic consistency in mind when writing any sort of code.

Rather than re-inventing a wheel that has been developed time after time, demonstrated by the references cited in this manuscript, the majority of this standard is based on the venerable Google JavaScript Style Guide for its foundation. For the adoption of Google Style, it should be noted that the rules and code examples in this document have been directly imported from Google's style guide for this JavaScript coding standard.

6. JavaScript

This type of programming style guide covers a wide range of issues, ranging from the aesthetic aspects of formatting to other types of conventions or coding standards, similar to the style guides for other programming languages. The purpose of this document, however, is to focus mainly on the hard and fast rules that we apply universally and to avoid giving advice that isn't clearly enforceable (by humans or tools) in any way.

6.1. Terminology Notes

Unless otherwise noted in this document:

Whenever the term **comment** is used, it refers to implementation comments. Rather than 'documentation comments', we refer to both human-readable text and machine-readable annotations as "JSDoc"within `/** ... */`.

In this style guide, **must**, **must not**, **should**, **should not**, and **may** are defined using RFC 2119 terminology. The terms **prefer** and **avoid** are equivalent to **should** and **should not**, respectively. Statements that are imperative or declarative are prescriptive and correspond to **must**.

Throughout the document, there will be other terminology notes.

6.2. Guide Notes

This document contains non-normative example code. In other words, despite the examples being in several acceptable styles, they may not represent all stylish ways to represent code. An example should not be construed as a rule enforcing the formatting choices made in it.

7. Source File Basics

7.1. File Name

The name of the file must be all lowercase and may contain underscores (_) or dashes (-), but no additional punctuation should be used. Follow the file name convention that DMS/DivTel projects use. The extension of the filename must be `.js`.

7.2. File Encoding: UTF-8

UTF-8 encoding is used for source files.

7.3. Special Characters

7.3.1. Whitespace Characters

The ASCII horizontal space character (0x20) is the only whitespace character in a source file besides the line terminator sequence. In string literals, this implies that all other whitespace characters are escaped, and Tab characters are **not** used to indent.

7.3.2. Special Escape Sequences

When a character has a special escape sequence (`/'`, `/"`, `//`, `/b`, `/f`, `/n`, `/r`, `/t`, `/v`), that sequence is used rather than the corresponding numeric escape (e.g., `/x0a`, `/u000a`, or `/u[a]`) The legacy octal escape sequences are never used.

7.3.3. Non-ASCII Characters

In the case of the remaining non-ASCII characters, either the actual Unicode character (e.g., `∞`) or its equivalent escape (e.g., `/u221e`) will be used, depending only on which makes the code easier to read and understand.

> **Tip:**
>
> The use of an explanatory comment can be very useful in Unicode escape cases, and sometimes even when Unicode characters are used.

Best:

```
/* Best: perfectly clear even without a comment. */
const units = 'µs';

/* Allowed: but unnecessary as µ is a printable character. */
const units = '\u03bcs'; // 'µs'

/* Good: use escapes for non-printable characters with a comment for clarity. */
return '\ufeff' + content;  // Prepend a byte order mark.
```

Figure 1: Explanatory Comment - Correct

Poor:

```
/* Poor: the reader has no idea what character this is. */
const units = '\u03bcs';
```

Figure 2: Explanatory Comment - Incorrect

> **Tip:**
>
> In no case should you reduce the readability of your code in fear that some programs cannot handle non-ASCII characters. In such a case, the programs are broken and need to be fixed.

▲ Table of Contents

8. Source File Structure

New source files should either be a `goog.module` file (a file that contains a call to `goog.module`) or an ECMAScript (ES) file (uses import and export statements). The files contain the following information, in the following order:

- If present, license or copyright information
- `@fileoverview` JSDoc, if present
- The `goog.module` statement, if there is a `goog.module` file
- ES `import` statements, if an ES module
- `goog.require` and `goog.requireType` statements
- The file's implementation

Each section of a file is separated by exactly one blank line, except for the file's implementation, which may be preceded by one or two blank lines.

8.1. License or copyright information

If present, files containing license or copyright information belong here.

8.2. @fileoverview JSDoc, if present

For formatting rules, see Top/file-level comments.

8.3. goog.module STATEMENT

`goog.module` files must declare exactly one `goog.module` name per line: declarations of `goog.module` must not be wrapped; therefore, the 80-column limit does not apply.

A namespace is defined by the entire argument to `goog.module`. It is the name of the package (the fragment of the directory structure that contains the code), and, optionally, the main `class`/`enum`/`interface` that it defines concatenated to the end.

Example:

```
1.goog.module('search.urlHistory.UrlHistoryService');
2.
```

Figure 3: goog.module Statement

8.3.1. HIERARCHY

A module's namespace cannot be named as a direct child of another module's namespace.

Not Allowed:

```
1.goog.module('foo.bar');   // 'foo.bar.qux' would be fine, though
2.goog.module('foo.bar.baz');
3.
```

Figure 4: HIerarchy

The directory structure mirrors the namespace structure so that deeper-nested children are subdirectories of higher-level parent directories.

> **Note:**
> As child namespaces exist in the same directory, the owners of "parent" namespace groups are necessarily aware of all child namespaces.

8.3.2. GOOG.MODULE.DECLARELEGACYNAMESPACE

Following the `goog.module statement`, you can optionally call `goog.module.declareLegacyNamespace`();. When possible, avoid using `goog.module.declareLegacyNamespace()`.

+ Example:

```
1.goog.module('my.test.helpers');
2.goog.module.declareLegacyNamespace();
3.goog.setTestOnly();
4.
```

Figure 5: goog.module.declareLegacyNamespace

The `goog.module.declareLegacyNamespace` is intended to ease the transition from traditional object hierarchy-based namespaces, but there are some naming restrictions. It is necessary to create the name of the child module after the parent namespace, so this name must not be a child or parent of any other `goog.module` (e.g., `goog.module('parent');` and `goog.module('parent.child');` both cannot exist safely, nor can `goog.module('parent');` and `goog.module('parent.child.grandchild');`).

8.3.3. GOOG.MODULE EXPORTS

The exports object is used to export classes, enums, functions, constants, and other symbols. In addition to being defined directly on the exports object, symbols may also be declared locally and exported separately. A symbol is only exported if it is intended for use outside the module. Module-local symbols that are not exported are neither declared `@private` nor end with an underscore. For exported and module-local symbols, there is no prescribed order.

Examples:

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

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

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

/** @return {number} */
function exportedFunction() {
  return moduleLocalFunction() * 2;
}

exports = {exportedArray, exportedFunction};

```

Figure 6: goog.module Exports

```
/** @const {number} */
exports.CONSTANT_ONE = 1;

/** @const {string} */
exports.CONSTANT_TWO = 'Another constant';

```

Figure 7: goog.module Exports

As the compiler already treats the exports object as a constant, do not annotate it as `@const`.

⊘ **Not Allowed**:

```
/** @const */
exports = {exportedFunction};

```

Figure 8: goog.module Exports

8.4. ES MODULES

8.4.1. IMPORTS

The 80-column limit does not apply to import statements because they must not be line wrapped.

8.4.1.1. IMPORT PATHS

For ES module files to import other ES module files, they must use the import statement. Do not `goog.require` a different ES module.

```
1.import './sideeffects.js';
2.
3.import * as goog from '../closure/goog/goog.js';
4.import * as parent from '../parent.js';
5.
6.import {name} from './sibling.js';
7.
```

Figure 9: ES Module

8.4.1.1.1. FILE EXTENSIONS IN IMPORT PATHS

Import paths must always include the `.js` file extension.

⊘ **Disallowed:**

```
1.import '../directory/file';
2.
```

Figure 10: Import Paths - Disallowed

✓ **Correct:**

```
1.import '../directory/file.js';
2.
```

Figure 11: Import Paths - Correct

8.4.1.2. IMPORTING THE SAME FILE MULTIPLE TIMES

Do not import the same file more than once. Otherwise, a file's aggregate imports can be difficult to determine.

```
1.  // Imports have the same path, but since it doesn't align it can be hard to see.
2.  import {short} from './long/path/to/a/file.js';
3.  import {aLongNameThatBreaksAlignment} from './long/path/to/a/file.js';
4.
```

Figure 12: Importing The Same File Multiple Times

8.4.1.3. NAMING IMPORTS

8.4.1.4. NAMING MODULE IMPORTS

Names of modules imported (import * as name) are lowerCamelCase names derived from the file name imported.

```
1. import * as fileOne from '../file-one.js';
2. import * as fileTwo from '../file_two.js';
3. import * as fileThree from '../filethree.js';
4.
```

Figure 13: Naming Module Imports

```
1.  import * as libString from './lib/string.js';
2.  import * as math from './math/math.js';
3.  import * as vectorMath from './vector/math.js';
4.
```

Figure 14: Naming Module Imports

8.4.1.4.1. NAMING DEFAULT IMPORTS

A default import name is determined by the imported file name and follows the rules in Rules by identifier type.

```
1.  import MyClass from '../my-class.js';
2.  import myFunction from '../my_function.js';
3.  import SOME_CONSTANT from '../someconstant.js';
4.
```

Figure 15: Naming Default Imports

> **Note**:
>
> This should not occur in general since default exports are prohibited in this style guide (refer to Named vs. default exports). Modules that do not conform to this style guide will be imported using default imports.

8.4.1.4.2. NAMING NAMED IMPORTS

Symbols imported via named imports (`import {name}`) should generally retain their names. Avoid aliasing imports (`import {SomeThing as SomeOtherThing}`). It is preferable to fix name collisions by using a module import (`import *`) or by renaming the exports themselves.

```
1.  import * as bigAnimals from './biganimals.js';
2.  import * as domesticatedAnimals from './domesticatedanimals.js';
3.
4.  new bigAnimals.Cat();
5.  new domesticatedAnimals.Cat();
6.
```

Figure 16: Naming Named Imports

For renaming named imports, use the imported module's file name or path in the alias.

```
1.  import {Cat as BigCat} from './biganimals.js';
2.  import {Cat as DomesticatedCat} from './domesticatedanimals.js';
3.
4.  new BigCat();
5.  new DomesticatedCat();
6.
```

Figure 17: Naming Named Imports

8.4.2. EXPORTS

Only symbols that are intended for use outside of the module are exported. In non-exported module-local symbols, `@private` is not declared, and their names do not end in an underscore. For exported and module-local symbols, there is no prescribed order.

8.4.2.1. NAMED VS. DEFAULT EXPORTS

All code should use named exports. An export keyword can be applied to a declaration, or export [name]; can be used.

Do not use default exports. Importing modules must give these values a name, which can lead to inconsistencies in the naming of modules.

👎 **Bad:**

```
1.  // Do not use default exports:
2.  export default class Foo { ... } // BAD!
3.
```

Figure 18: Named vs. Default Exports - Incorrect

👍 **Good:**

```
1.  // Use named exports:
2.  export class Foo { ... }
3.
```

Figure 19: Named vs. Default Exports - Correct

```
1.  // Alternate style named exports:
2.  class Foo { ... }
3.
4.  export {Foo};
5.
```

Figure 20: Named vs. Default Exports - Correct

8.4.2.2. EXPORTING STATIC CONTAINER CLASSES AND OBJECTS

For the sake of namespacing, do not export container classes or objects that have static methods or properties.

👎 **Bad:**

```
// container.js
// Bad: Container is an exported class that has only static methods and fields.
export class Container {
  /** @return {number} */
  static bar() {
    return 1;
  }
}

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

```

Figure 21: Exporting Static Container Classes and Objects - Incorrect

Export constants and functions instead:

👍 **Good:**

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

export const /** number */ FOO = 1;
```

Figure 22: Exporting Static Container Classes and Objects - Correct

8.4.2.3. MUTABILITY OF EXPORTS

The exported variables must not be mutated outside of the module initialization process.

If mutation is needed, there are alternatives, including exporting a constant reference to an object with mutable fields or exporting accessor functions for mutable data.

Bad:

```
// Bad: both foo and mutateFoo are exported and mutated.
export let /** number */ foo = 0;

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

/**
 * @param {function(number): number} newMutateFoo
 */
export function setMutateFoo(newMutateFoo) {
  // Exported classes and functions can be mutated!
  mutateFoo = () => {
    foo = newMutateFoo(foo);
  };
}

```

Figure 23: Mutability Of Exports - Incorrect

Good:

```
// Good: Rather than export the mutable variables foo and mutateFoo directly,
// instead make them module scoped and export a getter for foo and a wrapper for
// mutateFooFunc.
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;
}

```

Figure 24: Mutability Of Exports - Correct

8.4.2.4. EXPORT FROM

There is an exception to the 80-column limit for `export from` statements, since they cannot be line wrapped. This applies to both `export from` flavors.

```
1.  export {specificName} from './other.js';
2.  export * from './another.js';
3.
```

Figure 25: Export From

8.4.3. CIRCULAR DEPENDENCIES IN ES MODULES

In spite of the fact that the *ECMAScript* specification permits this, do not create cycles between ES modules.

> **NOTE:**
>
> The import and export statements can both be used to create cycles.

```
1.// a.js
2.import './b.js';
```

Figure 26: Circular Dependencies In Es Modules

```
1.// b.js
2.import './a.js';
3.
4.
5.// `export from` can cause circular dependencies too!
6.export {x} from './c.js';
```

Figure 27: Circular Dependencies In Es Modules

```
1.// c.js
2.import './b.js';
3.
4.export let x;
5.
```

Figure 28: Circular Dependencies In Es Modules

8.4.4. Interoperating with Closure

8.4.4.1. Referencing goog

Import Closure's `goog.js` to reference the Closure `goog` namespace.

```
1.  import * as goog from '../closure/goog/goog.js';
2.
3.  const name = goog.require('a.name');
4.
5.  export const CONSTANT = name.compute();
6.
```

Figure 29: Referencing goog

ES modules can use only a subset of properties exported by `goog.js`.

8.4.4.2. goog .require in ES modules

In ES modules, `goog.require` works as it does in `goog.module` files. You can require any symbol of the Closure namespace (i.e., symbols created by `goog.provide` or `goog.module`) and `goog.require` returns the value.

```
1.  import * as goog from '../closure/goog/goog.js';
2.  import * as anEsModule from './anEsModule.js';
3.
4.  const GoogPromise = goog.require('goog.Promise');
5.  const myNamespace = goog.require('my.namespace');
6.
```

Figure 30: goog.require in ES modules

8.4.4.3. Declaring Closure Module IDs in ES modules

Within ES modules, `goog.declareModuleId` can be used to declare a module ID similar to that of `goog.module`. Therefore, this module ID can be `goog.require` d, `goog.module.get` d, `goog.forwardDeclare` 'd, etc. It would be as if `goog.module.declareLegacyNamespace` was not called. The module ID is not created as a JavaScript symbol that is available globally.

When `goog.require` (or `goog.module.get`) is used for a module ID from `goog.declareModuleId`, it will always return the module object (as if it were `import *` 'd). Therefore, the argument to `goog.declareModuleId` should always end with `lowerCamelCaseName`.

> **Note**:
>
> Calling `goog.module.declareLegacyNamespace` in an ES module is an error. It can only be called from `goog.module` files. An ES module cannot be directly associated with a legacy namespace.

In cases where named exports are used, `goog.declareModuleId` should only be used to upgrade Closure files to ES modules in place.

```
1.  import * as goog from '../closure/goog.js';
2.
3.  goog.declareModuleId('my.esm');
4.
5.  export class Class {};
6.
```

Figure 31: goog.declareModuleId

8.5. goog.setTestOnly

An optional call to `goog.setTestOnly()` may follow the `goog.module` statement in a `goog.module file`.

You may optionally call `goog.setTestOnly()` after an import statement in an ES module.

8.6. goog.require AND goog.requireType STATEMENTS

`goog.require` and `goog.requireType` are used to import files. You can use the names imported by `goog.require` statement both in code and in type annotations. Those imported by a `goog.requireType` can only be used in type annotations.

The statements `goog.require` and `goog.requireType` form a contiguous block without empty lines. A single empty line separates this block from the `goog.module` declaration. In `goog.require` or `goog.requireType`, the entire argument is a namespace defined by a `goog.module` in a separate file. There may be no other instances of `goog.require` and `goog.requireType` statements in the file.

`goog.require` and `goog.requireType` are assigned to a single constant alias, or they are destructured into several constant aliases. The only way to refer to dependencies in type annotations or code is through these aliases. The use of fully qualified namespaces is restricted to arguments to `goog.require` and `goog.requireType`.

> **Exception**:
> Type annotations and code must use the fully qualified names of types, variables, and functions declared in externs files.

Aliases must correspond to the final dot-separated component of the namespace of the imported module.

> **Exception**:
>
> In certain cases, to form a longer alias, additional components of the namespace can be used. The resulting alias must preserve the original identifier's casing so that its type is still correctly identified. If longer aliases improve readability, they can be used to disambiguate otherwise identical aliases. It is also necessary to use a longer alias to avoid masking native types such as `Element`, `Event`, `Error`, `Map`, and `Promise` (for a more complete list, see Standard Built-in Objects and Web APIs at MDN). When renaming destructured aliases, a space must follow the colon as required in Horizontal whitespace.

For the same namespace, a file cannot contain both a `goog.require` and a `goog.requireType` statement. An imported name should be imported by a single `goog.require` statement if it is used both in code and in type annotations.

Whenever a module is imported for its side effects, the call must be `goog.require` (not `goog.requireType`), and assignment can be omitted. There needs to be a comment explaining why this is necessary and to suppress a compiler warning.

Sorting the lines is done according to the following rules: All requires with a name on the left-hand side come first, sorted alphabetically by those names. Next, destructuring requires are sorted by their names on the left. Finally, any standalone require calls (generally for modules imported just for their effects).

> **Tip**:
>
> This order does not need to be memorized and enforced manually. You can rely on your IDE to report requires that are incorrectly sorted.

In the case of aliases or module names that exceed the 80-column limit, it **must not** be wrapped. The 80-column limit does not apply to require lines.

Example:

```
// Standard alias style.
const MyClass = goog.require('some.package.MyClass');
const MyType = goog.requireType('some.package.MyType');
// Namespace-based alias used to disambiguate.
const NsMyClass = goog.require('other.ns.MyClass');
// Namespace-based alias used to prevent masking native type.
const RendererElement = goog.require('web.renderer.Element');
// Out of sequence namespace-based aliases used to improve readability.
// Also, require lines longer than 80 columns must not be wrapped.
const SomeDataStructureModel =
    goog.requireType('identical.package.identifiers.models.SomeDataStructure');
const SomeDataStructureProto =
    goog.require('proto.identical.package.identifiers.SomeDataStructure');
// Standard alias style.
const asserts = goog.require('goog.asserts');
// Namespace-based alias used to disambiguate.
const testingAsserts = goog.require('goog.testing.asserts');
// Standard destructuring into aliases.
const {clear, clone} = goog.require('goog.array');
const {Rgb} = goog.require('goog.color');
// Namespace-based destructuring into aliases in order to disambiguate.
const {SomeType: FooSomeType} = goog.requireType('foo.types');
const {clear: objectClear, clone: objectClone} = goog.require('goog.object');
// goog.require without an alias in order to trigger side effects.
/** @suppress {extraRequire} Initializes MyFramework. */
goog.require('my.framework.initialization');

```

Figure 32: goog.require and goog.requireType statements

Discouraged:

```
// If necessary to disambiguate, prefer PackageClass over SomeClass as it is
// closer to the format of the module name.
const SomeClass = goog.require('some.package.Class');

```

Figure 33: goog.require and goog.requireType statements - Discouraged

⊘ **Not Allowed:**

```
// Extra terms must come from the namespace.
const MyClassForBizzing = goog.require('some.package.MyClass');
// Alias must include the entire final namespace component.
const MyClass = goog.require('some.package.MyClassForBizzing');
// Alias must not mask native type (should be `const JspbMap` here).
const Map = goog.require('jspb.Map');
// Don't break goog.require lines over 80 columns.
const SomeDataStructure =
    goog.require('proto.identical.package.identifiers.SomeDataStructure');
// Alias must be based on the namespace.
const randomName = goog.require('something.else');
// Missing a space after the colon.
const {Foo:FooProto} = goog.require('some.package.proto.Foo');
// goog.requireType without an alias.
goog.requireType('some.package.with.a.Type');


/**
 * @param {!some.unimported.Dependency} param All external types used in JSDoc
 *     annotations must be goog.require'd, unless declared in externs.
 */
function someFunction(param) {
  // goog.require lines must be at the top level before any other code.
  const alias = goog.require('my.long.name.alias');
  // ...
}

```

Figure 34: goog.require and goog.requireType statements – Not allowed

8.7. The File's Implementation

After all dependency information has been declared (separated by at least one blank line), the actual implementation follows.

This may include any module-local declarations (constants, variables, classes, functions, etc.), as well as any exported symbols.

▲ Table of Contents

9. FORMATTING

Terminology Note:

The body of a class, function, method, or brace-delimited block of code is called a block-like construct. Note that, by

Array literals and Object literals, arrays and object literals may optionally be treated as if it were a block-like construct.

Tip:

Use `clang-format`. Several efforts have been made by the JavaScript community to ensure clang-format handles JavaScript files correctly. Several popular editors are integrated with `clang-format`.

9.1. BRACES

9.1.1. BRACES ARE USED FOR ALL CONTROL STRUCTURES

Not Allowed:

```
1.if (someVeryLongCondition())
2.  doSomething();
3.
4.for (let i = 0; i < foo.length; i++) bar(foo[i]);
5.
```

Figure 35: Braces – Not allowed

> **Exception**:
>
> When it improves readability, simple `if` statements that can fit entirely on one line without wrapping (and don't have an `else`) can be kept on a single line without braces. Braces and newlines can only be omitted in this case

```
1.if (shortCondition()) foo();
2.
```

Figure 36: Braces omitted

9.1.2. NONEMPTY BLOCKS: K&R STYLE

For blocks with no empty space and block-like constructs, braces follow the Kernighan and Ritchie style (Egyptian brackets):

- Before the opening brace, there is no line break.
- A line break follows the opening brace.
- Before the closing brace, there is a line break.
- Whenever a closing brace ends a statement, a function or class statement, or a class method, there should be a line break. Line breaks are not allowed after braces if they are followed by `else`, `catch`, `while`, or by a `comma`, `semicolon`, or right-parenthesis.

Example:

```
1.class InnerClass {
2.  constructor() {}
3.
4.  /** @param {number} foo */
5.  method(foo) {
6.    if (condition(foo)) {
7.      try {
8.        // Note: this might fail.
9.        something();
10.       } catch (err) {
11.         recover();
12.       }
13.     }
14.   }
15. }
16.
```

Figure 37: Nonempty blocks: K&R style

9.1.3. EMPTY BLOCKS: MAY BE CONCISE

If an empty block or block-like construct is opened without a character, space, or line break in between (i.e., `{ }`), the block or block-like construct can be closed immediately after opening unless it is a part of a multi-block statement (one that directly contains multiple blocks: `if`/`else` or `try`/`catch`/`finally`).

+ Example:

```
1.function doNothing() {}
2.
```

Figure 38: Empty blocks - Example

⊘ Not Allowed:

```
1.if (condition) {
2.  // …
3.} else if (otherCondition) {} else {
4.  // …
5.}
6.
7.try {
8.  // …
9.} catch (e) {}
10.
```

Figure 39: Empty blocks – Not allowed

9.2. BLOCK INDENTATION: +2 SPACES

The indent increases by two spaces each time a new block or block-like construct is opened. As soon as the block ends, the indent returns to its previous indent level. Both code and comments have the same indented level throughout the block. (See the example in Nonempty blocks: K&R style

9.2.1. ARRAY LITERALS: OPTIONALLY 'BLOCK-LIKE'

It is optional to format array literals as if they were "block-like constructs." In this case, all of the following are valid (not an exhaustive list):

```
1.const a = [
2.  0,
3.  1,
4.  2,
5.];
6.
7.const b =
8.    [0, 1, 2];
9.
```

Figure 40: Array literals

```
1.const c = [0, 1, 2];
2.
3.someMethod(foo, [
4.  0, 1, 2,
5.], bar);
6.
```

Figure 41: Array literals

It is acceptable to use other combinations, especially when emphasizing semantic grouping, but should not be used solely to reduce the vertical size of larger arrays.

9.2.2. Object literals: optionally 'block-like'

You can optionally format any object literal as if it were a "block-like construct." The same examples apply as . As an example (and not an exhaustive list), the following are all valid:

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

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

```

Figure 42: Array literals: 'Block-Like'.

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

someMethod(foo, {
  a: 0, b: 1,
}, bar);

```

Figure 43: Array literals: 'Block-Like'

9.2.3. Class literals

Indentation of class literals (irrespective of whether they are declarations or expressions) is done as blocks. Semicolons must not be added after methods or after the closing brace of a class declaration (statements—such as assignments—that contain class expressions are still terminated with a semicolon). If the class does not extend a templatized type, use the extends keyword. But not the `@extends` JSDoc annotation.

+ Example:

```
class Foo {
  constructor() {
    /** @type {number} */
    this.x = 42;
  }

  /** @return {number} */
  method() {
    return this.x;
  }
}
Foo.Empty = class {};

```

Figure 44: Class Literals

```
/** @extends {Foo<string>} */
foo.Bar = class extends Foo {
  /** @override */
  method() {
    return super.method() / 2;
  }
};

/** @interface */
class Frobnicator {
  /** @param {string} message */
  frobnicate(message) {}
}

```

Figure 45: Class Literals

9.2.4. FUNCTION EXPRESSIONS

When an anonymous function is declared in the list of arguments for a function call, the body of the function is indented two spaces beyond the preceding indentation depth.

+ Example:

```
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();
      }
    });

```

Figure 46: Anonymous Function

9.2.5. SWITCH STATEMENTS

Switch blocks are indented +2 as with any other block.

Following a switch label, a newline appears, and the indentation level increases by 2, just as if a block were being opened. Lexical scoping may require an explicit block. As if a block had been closed, the following switch label returns to the previous indentation level.

Between a `break` and the following `case`, a blank line is optional.

+ Example:

```
switch (animal) {
  case Animal.BANDERSNATCH:
    handleBandersnatch();
    break;

  case Animal.JABBERWOCK:
    handleJabberwock();
    break;

  default:
    throw new Error('Unknown animal');
}
```

Figure 47: Switch Statements

9.3. STATEMENTS

9.3.1. ONE STATEMENT PER LINE

A line break follows each statement.

9.3.2. SEMICOLONS ARE REQUIRED

Semicolons must be used to terminate every statement. It is forbidden to use automatic semicolon insertion.

9.4. COLUMN LIMIT: 80

The maximum number of characters per column in JavaScript code is 80. All lines exceeding this limit, except as noted below, must be line-wrapped, as explained in Line-wrapping.

Exceptions:

1. `goog.module`, `goog.require` and `goog.requireType` statements (see goog.module Statement and goog.require and goog.requireType Statements).
2. ES module `import` and `export` from statements (see Imports and export from).
3. There are some lines where it is impossible or would hinder discovery to obey the column limit. Among them are:
    a. The source code contains a long URL that can be clicked.
    b. Shell command that can be copied and pasted.
    c. A long string literal that may require copying or searching for in its entirety (e.g., a long file path).

9.5. LINE-WRAPPING

> **Terminology Note**:
> During line wrapping, a chunk of code is split into multiple lines to conform to the column limit (80), where the chunk could otherwise fit on one line legally.

Line-wrapping cannot be defined in a comprehensive, deterministic way in every situation. There are frequently several valid ways to line-wrap the same code.

> **Note**:
> In most cases, line-wrapping is used to avoid exceeding the column limit. However, at the author's discretion, even code that would fit within the column limit can be line-wrapped.

> **Tip**:
> By extracting a method or local variable, the problem can be solved without the need to wrap the lines.

9.5.1. WHERE TO BREAK

As a general rule, line-wrapping should break at a higher syntactic level when possible.

✓ **Preferred:**

```
1.currentEstimate =
2.    calc(currentEstimate + x * currentEstimate) /
3.        2.0;
4.
```

Figure 48: Line-Wrapping - Preferred

Discouraged:

```
1.currentEstimate = calc(currentEstimate + x *
2.    currentEstimate) / 2.0;
3.
```

Figure 49: Line-Wrapping - Discouraged

According to the preceding example, the syntactic levels are as follows: assignment, `division`, `function call`, `parameters`, `number constant`.

Operators are wrapped as follows:

1. The break comes after the symbol when a line is broken at an operator.

   > **Note:**
   > that this is not the same practice used in Google style for Java.)

   a. In the case of the dot (`.`), which is not a valid operator, this does not apply.

2. Whenever a method or constructor name follows an open parenthesis (`(`), it stays attached to it.

3. It is necessary to keep the comma (`,`) attached to the token that precedes it.

   > **Note:**
   > The primary goal for line wrapping is to have clear code, not necessarily code that fits in the smallest number of lines.

9.5.2. Indent continuation lines at least +4 spaces

When line wrapping, every line after the first (each continuation line) is indented +4 from the original line unless it falls under the rules of Block indentation: +2 spaces.

Indentation may be varied beyond +4 when there are multiple continuation lines. Generally, at a deeper syntactic level, continuation lines are indented by larger multiples of 4, and two lines use the same indentation level when and only when they begin with syntactically parallel elements.

Horizontal alignment: discouraged addresses the horizontal alignment and discourages the practice of aligning certain tokens with previous lines by using a variable number of spaces.

9.6. Whitespace

9.6.1. Vertical whitespace

A single blank line appears:

1. Between consecutive methods in a class or object literal
   a. **Exception**: In an object literal, a blank line between two consecutive property definitions (with no other code between them) is optional. In order to group fields logically, blank lines are used as needed.
2. Use sparingly within method bodies in order to create logical groupings of statements. A function body cannot begin or end with a blank line.
3. It can be placed before or after the first or last method in a class or object literal (neither encouraged nor discouraged).
4. In accordance with other sections of this document (e.g., `goog.require` and `goog.requireType` Statements).

The use of multiple consecutive blank lines is permitted, but it is not required (nor encouraged).

9.6.2. HORIZONTAL WHITESPACE

Whitespace is used in three broad categories depending on its location: preceding (at the beginning of a line), trailing (the end of a line), and internal. Indentation (leading whitespace) is dealt with elsewhere in this document. It is forbidden to use trailing whitespace.

Beyond where required by language or other style rules, and apart from literals, comments, and JSDoc, only one internal ASCII space appears in the following places.

1. Separating any reserved word (such as `if`, `for`, or `catch`) with the exception of `function` and `super` from an open parenthesis (`(`) that follows it on that line.
2. Separating any reserved word (such as `else` or `catch`) from a closing curly brace (`}`) that precedes it on that line.
3. Before any open curly brace (`{`), with two exceptions:
   a. Before an object literal, that is the first argument of a function or the first element in an array literal (e.g., `foo({a: [{c: d}]})`).
   b. In a template expansion, as it is forbidden by the language (e.g., valid: `` `ab${1 + 2}cd` ``, invalid: `` `xy$ {3}z` ``).
4. On both sides of any binary or ternary operator.
5. After a comma (`,`) or semicolon (`;`).

> **Note:**
> Spaces are never allowed before these characters.

6. In an object literal, after the colon (`:`).
7. On both sides of the double slash (`//`) that begins an end-of-line comment. Here, multiple spaces are allowed but not required.
8. After an open-block comment character and on both sides of close characters (e.g., for short-form type declarations, casts, and parameter name comments: `this.foo = /** @type {number} */ (bar);` or `function(/** string */ foo) {`; or `baz(/* buzz= */ true))`.

9.6.3. HORIZONTAL ALIGNMENT: DISCOURAGED

Terminology Note:

By using horizontal alignment in your code, you can make certain tokens appear directly below certain other tokens on previous lines by adding a variable number of additional spaces.

The practice is allowed but is generally discouraged. In places where it was already used, it isn't even necessary to maintain horizontal alignment.

A sample without alignment is shown below, followed by a sample with alignment. It is permissible to do both, but the latter is discouraged:

```
{
  tiny: 42, // this is great
  longer: 435, // this too
};

{
  tiny:   42,  // permitted, but future edits
  longer: 435, // may leave it unaligned
};
```

Figure 50

Tip:

Although alignment aids readability, it can create maintenance problems in the future. Imagine a future change that only affects one line. It is possible that the previously appealing formatting will be distorted by this change, and that is allowed. Coders (or perhaps you) are more likely to be prompted to adjust whitespace on nearby lines as well, potentially a cascading series of reformatting may occur as a result. There is now a blast radius around that one-line change. As a result, pointless busywork may result at worst, however, version history information is still corrupted at best, increases merge conflicts and it slows down reviewers.

9.6.4. FUNCTION ARGUMENTS

The function name should appear on the same line as all function arguments. The arguments must be line-wrapped in a readable way if doing so would exceed the 80-column limit. Consider wrapping as close to 80 as possible or put each argument on its own line for better readability. For indentation, there should be four spaces. Parenthesis alignment is allowed but discouraged. Argument wrapping patterns include the following:

```
// Arguments start on a new line, indented four spaces. Preferred when the
// arguments don't fit on the same line with the function name (or the keyword
// "function") but fit entirely on the second line. Works with very long
// function names, survives renaming without reindenting, low on space.
doSomething(
    descriptiveArgumentOne, descriptiveArgumentTwo, descriptiveArgumentThree) {
  // …
}

// If the argument list is longer, wrap at 80. Uses less vertical space,
// but violates the rectangle rule and is thus not recommended.
doSomething(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
  // …
}

// Four-space, one argument per line.  Works with long function names,
// survives renaming, and emphasizes each argument.
doSomething(
    veryDescriptiveArgumentNumberOne,
    veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy,
    artichokeDescriptorAdapterIterator) {
  // …
}
```

Figure 51: Function Arguments

9.7. GROUPING PARENTHESES: RECOMMENDED

The optional grouping parentheses should only be omitted when the author and reviewer agree that there is no reasonable likelihood that the code will be misinterpreted without them or that they would make the code easier to read. The reader should not be assumed to know the entire operator precedence table.

Avoid using unnecessary parentheses around the entire expression following `delete`, `typeof`, `void`, `return`, `throw`, `case`, `in`, `of`, or `yield`.

For type casts, parentheses are required: `/** @type {!Foo} */ (foo)`.

9.8. COMMENTS

The purpose of this section is to address implementation comments. JSDoc is addressed separately in 7 JSDoc.

9.8.1. BLOCK COMMENT STYLE

Comments in blocks are indented at the same level as the surrounding code. The style may be in `/* … */` or `//`. A multi-line `/* ... */` comment must start with a `*` aligned with the `*` on the previous line so that the comment is obvious without any additional context.

```
/*
 * This is
 * okay.
 */

// And so
// is this.

/* This is fine, too. */
```

Figure 52: Block Comment Style

Asterisks or other characters are not used to enclose comments.

For implementation comments, do not use JSDoc (`/**... */`).

9.8.2. Parameter Name Comments

When the value and method name do not sufficiently convey the meaning, and refactoring the method to be clearer is not possible, "*parameter name*" comments should be used. Their referred format appears before the value with `=:`

```
1.someFunction(obviousParam, /* shouldRender= */ true, /* name= */ 'hello');
2.
```

Figure 53: Parameter Name Comments

If you want to keep them consistent with surrounding code, you can put them after the value without `=:`

```
1.someFunction(obviousParam, true /* shouldRender */, 'hello' /* name */);
2.
```

Figure 54: Parameter Name Comments

▲ Table of Contents

10. LANGUAGE FEATURES

Many of the features of JavaScript are dubious (or even dangerous). A description of which features may or may not be used, along with additional restrictions, is provided in this section.

10.1. LOCAL VARIABLE DECLARATIONS

10.1.1. USE CONST AND LET

Use either `const` or `let` to declare local variables. Generally, `const` should be used by default unless reassigning a variable is necessary. There should be no use of the `var` keyword.

10.1.2. ONE VARIABLE PER DECLARATION

Whenever a local variable declaration is used, it declares only one variable: for example, `let a = 1, b = 2;` is not used.

10.1.3. DECLARED WHEN NEEDED, INITIALIZED AS SOON AS POSSIBLE

Normally, local variables are not declared at the start of the block or block-like construct in which they are contained. To minimize their scope, local variables are declared close to the point where they are first used (within reason).

10.1.4. DECLARE TYPES AS NEEDED

A JSDoc type annotation may either appear in the line above the declaration or else inline before the variable name in the case there is no other JSDoc present.

Example:

```
1.const /** !Array<number> */ data = [];
2.
3./**
4. * Some description.
5. * @type {!Array<number>}
6. */
7.const data = [];
8.
```

Figure 55: JSDoc type Annotation

Inline annotations and JSDoc styles cannot be mixed: the compiler will only process the first JsDoc.

```
1./** Some description. */
2.const /** !Array<number> */ data = [];
3.
```

Figure 56: JSDoc type Annotation - Inline

Tip:

Compilers can infer a templatized type but not its parameters in many cases. In particular, this occurs when no values of the template parameter type are included in the initializing literal or constructor call (e.g., empty arrays, objects, `map` s, or `set` s), or when the variable is modified in a closure. It is particularly important to provide type annotations for local variables in these cases since otherwise the compiler will infer that the template parameter is unknown.

10.2. ARRAY LITERALS

10.2.1. USE TRAILING COMMAS

When there is a line break between the final element and the closing bracket, include a trailing comma.

Example:

```
const values = [
  'first value',
  'second value',
];
```

Figure 57: Trailing Commas

10.2.2. DO NOT USE THE VARIADIC ARRAY CONSTRUCTOR

Adding or removing arguments can result in errors in the constructor. Use a literal instead.

Not Allowed:

```
const a1 = new Array(x1, x2, x3);
const a2 = new Array(x1, x2);
const a3 = new Array(x1);
const a4 = new Array();
```

Figure 58: Variadic Array Constructor

All works above as expected except for the third case: if `x1` is a whole number, then `a3` is an array of size `x1` where all elements are `undefined`. If `x1` is any other number, then an exception will be thrown, and if it is anything else, then it will be a single-element array.

Instead, write:

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

Figure 59: Variadic Array Constructor

In some cases, it may be appropriate to explicitly allocate a new array of a given length using `new Array(length)`.

10.2.3. NON-NUMERIC PROPERTIES

An array should not have any non-numeric properties (other than `length`). Instead, use a `Map` (or `Object`).

10.2.4. DESTRUCTURING

Destructuring can be accomplished by using array literals on the left-hand side of an assignment (such as when unpacking multiple values from a single array or iterable). A final rest element may be included (with no space between the `...` and the variable name). You should omit elements that are unused.

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

Figure 60: Destructuring

Function parameters can also be destructed (parameter names are required but ignored). When a destructured array parameter is optional, specify `[]` as the default value, and provide default values on the left hand side:

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

Figure 61: Destructed Function Parameters

⊘ **Not Allowed**:

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

Figure 62: Destructed Function Parameters – Not Allowed

Tip:

In cases where multiple values need to be destructed into a function's parameter or return, use object destructuring rather than array destructuring, as it allows naming the individual elements and specifying a different type for each.

10.2.5. SPREAD OPERATOR

Array literals can be used to flatten elements from other iterables by using the spread operator (`...`). It is preferable to use the spread operator rather than awkward constructs with `Array.prototype.` After the `....`, there is no space.

+ Example:

```
1.[...foo]   // preferred over Array.prototype.slice.call(foo)
2.[...foo, ...bar]   // preferred over foo.concat(bar)
3.
```

Figure 63: Spread Operator

10.3. OBJECT LITERALS

10.3.1. USE TRAILING COMMAS

If there is a line break between the final property and the closing brace, use a trailing comma.

10.3.2. DO NOT USE THE OBJECT CONSTRUCTOR

Even though `Object` does not have the same problems as `Array`, For consistency, it is. Instead, use an object literal (`{}` or `{a: 0, b: 1, c: 2}`).

10.3.3. DO NOT MIX QUOTED AND UNQUOTED KEYS

An object literal can represent either a struct (with unquoted keys and/or symbols) or a dict (with quoted and/or computed keys). Do not mix quoted and unquoted key types in a single object literal.

⊘ Not Allowed:

```
1.{
2.  width: 42, // struct-style unquoted key
3.  'maxWidth': 43, // dict-style quoted key
4.}
5.
```

Figure 64: Quoted And Unquoted Keys – Not Allowed

Similarly, it extends to passing the property name to functions, like `hasOwnProperty`. Due to the fact that the compiler cannot rename/obfuscate string literals, as doing so will break compiled code.

⊘ **Not Allowed**:

```
/** @type {{width: number, maxWidth: (number|undefined)}} */
const o = {width: 42};
if (o.hasOwnProperty('maxWidth')) {
  ...
}
```

Figure 65: Quoted And Unquoted Keys – Not Allowed

This is best implemented as:

```
/** @type {{width: number, maxWidth: (number|undefined)}} */
const o = {width: 42};
if (o.maxWidth != null) {
  ...
}
```

Figure 66: Quoted And Unquoted Keys

10.3.4. COMPUTED PROPERTY NAMES

`{['key' + foo()]: 42}` is a valid computed property name, which is considered a dict-style (quoted) key (i.e., it cannot be mixed with a non-quoted key) unless the computed property is a symbol (e.g., `[Symbol.iterator]`). It is also possible to use enum values for computed keys, but they should not be mixed with non-enum keys in the same literal.

10.3.5. METHOD SHORTHAND

On object literals, methods can be defined using the method shorthand `([method() [... ]])` instead of a colon followed by a `function` or `arrow` function literal.

Example:

```
return {
  stuff: 'candy',
  method() {
    return this.stuff;  // Returns 'candy'
  },
};
```

Figure 67: Method Shorthand

Note:

In a method shorthand or function, `this` refers to the object literal itself, whereas in an arrow function, `this` refers to the scope outside the object literal.

Example:

```
class {
  getObjectLiteral() {
    this.stuff = 'fruit';
    return {
      stuff: 'candy',
      method: () => this.stuff,  // Returns 'fruit'
    };
  }
}
```

Figure 68: Method Shorthand

10.3.6. SHORTHAND PROPERTIES

Shorthand properties are allowed on object literals.

Example:

```
1.const foo = 1;
2.const bar = 2;
3.const obj = {
4.  foo,
5.  bar,
6.  method() { return this.foo + this.bar; },
7.};
8.assertEquals(3, obj.method());
9.
```

Figure 69: Shorthand Properties

10.3.7. DESTRUCTURING

To unpack multiple values from a single object, object destructuring patterns can be used on the left side of an assignment.

You can also use destructured objects as function parameters but keep them as simple as possible: just one level of unquoted properties. In parameter destructuring, deeper nesting and computed properties cannot be used. In the left-hand side of the destructured parameter, specify any default values (`{str = 'some default'} = {}`, rather than `{str} = {str: 'some default'}`), and if a destructured object is itself optional, it must default to `{}`. Any name may be given to the JSDoc for the destructured parameter (the name is unused but is required by the compiler).

Example:

```
/**
 * @param {string} ordinary
 * @param {{num: (number|undefined), str: (string|undefined)}=} param1
 *     num: The number of times to do something.
 *     str: A string to do stuff to.
 */
function destructured(ordinary, {num, str = 'some default'} = {})
assertEquals(3, obj.method());

```

Figure 70: Destructuring

Not Allowed:

```
/** @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: number, b: string}=} param1 */
function nontrivialDefault({a, b} = {a: 2, b: 4}) {};

```

Figure 71: Destructuring – Not Allowed

Similarly, destructuring may be used for `goog.require` statements and, in this case, must not be wrapped: a complete statement occupies one line, regardless of its length (see goog.require and goog.requireType Statements).

10.3.8. Enums

The `@enum` annotation is used to define enumerations in object literals. Enums may not have additional properties added after they have been defined. The values of all enums must be deeply immutable, and all enums must be constant.

```
/**
 * Supported temperature scales.
 * @enum {string}
 */
const TemperatureScale = {
  CELSIUS: 'celsius',
  FAHRENHEIT: 'fahrenheit',
};

/**
 * An enum with two options.
 * @enum {number}
 */
const Option = {
  /** The option used shall have been the first. */
  FIRST_OPTION: 1,
  /** The second among two options. */
  SECOND_OPTION: 2,
};

```

Figure 72: Enums

10.4. Classes

10.4.1. Constructors

There is no requirement for optional constructors. `Super()` must be called before subclass constructors can set fields or otherwise access `this`. Interfaces should declare non-method properties in the constructor.

10.4.2. Fields

Fields of a concrete object (all properties other than methods) should be specified in its constructor. Add `@const` annotations to fields that are never reassigned (these need not be deeply immutable). All non-public fields should be annotated with the appropriate visibility identifier (`@private`, `@protected`, `@package`), and all `@private` fields' names should end with an underscore. Fields are never set on a concrete class' `prototype`.

Example:

```
1.class Foo {
2.  constructor() {
3.    /** @private @const {!Bar} */
4.    this.bar_ = computeBar();
5.
6.    /** @protected @const {!Baz} */
7.    this.baz = computeBaz();
8.  }
9.}
10.
```

Figure 73: Field Annotations

Tip:

Adding or removing properties from an instance should never occur after the constructor has finished, Since it significantly hinders VM optimization. In order to prevent later changes to the shape, fields that will be initialized later should be explicitly set to `undefined` in the constructor. By adding `@struct` to an object, undeclared properties won't be added or accessed.

10.4.3. COMPUTED PROPERTIES

A computed property may only be used in classes if it is a symbol. A dict-style property (that is, a property that contains quoted or computed non-symbol keys, as defined in Do not mix quoted and unquoted keys) is not allowed. Classes that are logically iterable should be defined with a `[Symbol.iterator]` method. The use of `Symbol` should be limited beyond this.

Tip:

If you use any other built-in symbols (e.g., `Symbol.isConcatSpreadable`), be careful because they aren't polyfilled by the compiler and therefore won't work with older browsers.

10.4.4. STATIC METHODS

Prefer module-local functions over static methods where it does not interfere with readability.

It is only appropriate to call static methods on the base class itself. There should be no static methods called on variables containing dynamic instances, which may either be constructors or subclass constructors (and must be defined with `@nocollapse` if it is done), and they cannot be called directly on a subclass that does not define the method itself.

⊘ **Not Allowed:**

```
class Base { /** @nocollapse */ static foo() {} }
class Sub extends Base {}
function callFoo(cls) { cls.foo(); }  // discouraged: don't call static methods dynamically
Sub.foo();  // Disallowed: don't call static methods on subclasses that don't define it themselves

```

Figure 74: Static Methods – Not Allowed

10.4.5. OLD-STYLE CLASS DECLARATIONS

It is preferable to use ES6 classes, but there may be situations where this is not possible. For example:

1. Suppose there are existing subclasses or subclasses to be created, including frameworks that create subclasses that cannot be immediately converted to use *ES6* class syntax. It would be necessary to modify all downstream subclasses which do not use ES6 class syntax if such a class were to use ES6 syntax.
2. In frameworks requiring a known `this` value before calling the superclass constructor, since constructors with ES6 super classes don't have access to the instance `this` value until the call to `super` returns.

The style guide still applies to this code in all other respects: `let`, `const`, default parameters, `rest`, and `arrow` functions should all be used when they are appropriate.

A class-like definition can be defined with `goog.defineClass`, similar to *ES6* class syntax:

```
let C = goog.defineClass(S, {
  /**
   * @param {string} value
   */
  constructor(value) {
    S.call(this, 2);
    /** @const */
    this.prop = value;
  },

  /**
   * @param {string} param
   * @return {number}
   */
  method(param) {
    return 0;
  },
});

```

Figure 75: Class-like Definition w/ goog.defineClass

For new code, `goog.defineClass` is preferred; however, the more traditional syntax is also acceptable.

```
/**
 * @constructor @extends {S}
 * @param {string} value
 */
function C(value) {
  S.call(this, 2);
  /** @const */
  this.prop = value;
}
goog.inherits(C, S);

/**
 * @param {string} param
 * @return {number}
 */
C.prototype.method = function(param) {
  return 0;
};

```

Figure 76: Class Definition – Traditional

The constructor should define per-instance properties after calling the constructor of the super class if there is a super class. The constructor prototype should define methods.

Hierarchies for constructor prototypes are more difficult to define than they appear at first! Because of this, you should use `goog.inherits` from the Closure Library.

10.4.6. Do not manipulate prototype s directly

As compared to prototype properties, `class` keyword definitions are clearer and more readable. Although ordinary implementation code has no business manipulating these objects, they are still useful for defining classes as outlined in Old-style class declarations. Mixins and modifications to prototypes of builtin objects are explicitly forbidden.

> **Exception**:
>
> The code in frameworks (such as Polymer, or Angular) may need to use `prototype` s, so don't resort to worse workarounds to avoid doing so.

10.4.7. Getters and Setters

Do not use JavaScript getter and setter properties. This is because there is a potential for surprise and difficulty in reasoning, and there is limited support in the compiler. Provide ordinary methods instead.

> **Exception**:
>
> Defining a getter or setter is necessary in certain cases (for example, in data binding frameworks like Angular and Polymer, or to work with external APIs that cannot be adjusted). If getters and setters are defined with the `get` and `set` shorthand method keywords or `Object.defineProperties`, (not `Object.defineProperty`, which interferes with property renaming), they may be used with caution if the getter does not change the state observable state.

⊘ **Not Allowed:**

```
1.class Foo {
2.  get next() { return this.nextId++; }
3.}
4.
```

Figure 77: Getters and Setters – Not Allowed

10.4.8. OVERRIDING TOSTRING

It is possible to override the `toString` method, but it must always succeed and never have visible side effects.

> **Tip**:
>
> It is especially important to be careful when calling other methods from `toString` due to the possibility of infinite loops under exceptional conditions.

10.4.9. INTERFACES

Interfaces can be declared with either `@interface` or `@record`. Classes or object literals can implement interfaces explicitly (using `@implements)` or implicitly (using `@record`).

A non-static method body on an interface must be an empty block. In the constructor of the class, fields must be declared as uninitialized members if they are to be used.

➕ **Example:**

```
1./**
2. * Something that can frobnicate.
3. * @record
4. */
5.class Frobnicator {
6.  constructor() {
7.    /** @type {number} The number of attempts before giving up. */
8.    this.attempts;
9.  }
10.
11.  /**
12.   * Performs the frobnication according to the given strategy.
13.   * @param {!FrobnicationStrategy} strategy
14.   */
15.  frobnicate(strategy) {}
16.}
17.
```

Figure 78: Interfaces

5.4.10 Abstract Classes

When it is appropriate, abstract classes can be used. All abstract classes and methods must be annotated with the `@abstract` annotation. Do not use `goog.abstractMethod`. See abstract classes and methods.

10.5. Functions

10.5.1. Top-level functions

In addition to defining top-level functions directly on the exports object, top-level functions may also be declared locally and optionally exported. See goog.module Exports for more information about exports.

Examples:

```
/** @param {string} str */
exports.processString = (str) => {
  // Process the string.
};

```

Figure 79: Top-Level Functions

```
/** @param {string} str */
const processString = (str) => {
  // Process the string.
};

exports = {processString};
```

Figure 80: Top-Level Functions

10.5.2. Nested functions and closures

It is possible for functions to contain nested function definitions. A name can be given to the function if it is useful to do so, and it should be assigned to a local `const`.

10.5.3. ARROW FUNCTIONS

In nested functions, arrow functions can be used to provide a concise function syntax and simplify scoping `this` for nested functions. There is a strong preference for arrow functions over the `function` keyword, especially when it comes to nested functions (but see Method shorthand).

The arrow functions are preferred over other `this` scoping approaches such as the `f.bind(this)` and `goog.bind(f, this)` functions, as well `as const self = this`. It is particularly useful to use arrow functions to call into callbacks, since they permit explicit specification of which parameters are to be passed to the callback, as opposed to binding, which blindly passes along all parameters.

There are zero or more parameters associated with the left-hand side of the arrow. In the case of only a single non-destructured parameter, parentheses around the parameter are optional. The use of parentheses allows the specification of inline parameter types within the parentheses (see Method and function comments).

> **Tip**:
>
> Parentheses should always be used, even when an arrow function has one parameter, to avoid situations where the addition of parameters can cause problems, By failing to add parentheses, the parseable code may no longer perform as intended.

On the right-hand side of the arrow, there is the body of the function. The body of a statement is, by default, a block of statements (either zero or more statements enclosed in curly braces). It is also possible for the body to be a single expression that is implicitly returned in one of two cases: either when the program logic requires returning a value or when the void operator precedes a single function or method call (by using `void` it ensures that `undefined` is returned, and prevents leakage of values, and communicates intent). A single expression is generally preferred if it improves the statement's readability (e.g., for short or simple expressions).

➕ Examples:

```
/**
 * Arrow functions can be documented just like normal functions.
 * @param {number} numParam A number to add.
 * @param {string} strParam Another number to add that happens to be a string.
 * @return {number} The sum of the two parameters.
 */
const moduleLocalFunc = (numParam, strParam) => numParam + Number(strParam);

// Uses the single expression syntax with `void` because the program logic does
// not require returning a value.
getValue((result) => void alert(`Got ${result}`));

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

    // For inline callbacks, you can use inline typing for parameters.
    // Uses a block statement because the value of the single expression should
    // not be returned and the expression is not a single function call.
    getNullableValue((/** ?number */ result) => {
      this.cachedValue_ = result == null ? 0 : result;
    });
  }
}

```

Figure 81: Arrow Functions

⊘ Not Allowed:

```
/**
 * A function with no params and no returned value.
 * This single expression body usage is illegal because the program logic does
 * not require returning a value and we're missing the `void` operator.
 */
const moduleLocalFunc = () => anotherFunction();

```

Figure 82: Arrow Functions – Not Allowed

10.5.4. GENERATORS

Generators provide a variety of useful abstractions that can be used whenever they are needed.

You should attach the `*` to the function keyword when you define a generator function, and you should separate it with a space from the name of the function when you define a generator function. Delegating yields is achieved by attaching a `*` to the `yield` keyword.

Example:

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

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

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

Figure 83

10.5.5. PARAMETER AND RETURN TYPES

As a general rule, function parameters and return types should be documented using 7 JSDoc annotations. For more information, see Method and function comments.

10.5.5.1. DEFAULT PARAMETERS

It is possible to specify Parameters that are optional by using the equals operator in the parameter list. Optional parameters must include spaces on both sides of the equals operator, they must have the same name as required parameters (i.e., not prefixed with `opt_`), use the `=` suffix in their JSDoc type, they must come after required parameters, and they must not use initializers that produce observable side effects. Whenever a parameter is optional for a concrete function, that parameter must have a default value, even if that value is `undefined`. Abstract and interface methods are different from concrete functions in that they do not require default parameter values.

➕ Example:

```
1.  /**
2.   * @param {string} required This parameter is always needed.
3.   * @param {string=} optional This parameter can be omitted.
4.   * @param {!Node=} node Another optional parameter.
5.   */
6.  function maybeDoSomething(required, optional = '', node = undefined) {}
7.
8.  /** @interface */
9.  class MyInterface {
10.   /**
11.    * Interface and abstract methods must omit default parameter values.
12.    * @param {string=} optional
13.    */
14.   someMethod(optional) {}
15. }
16.
```

Figure 84: Default Parameters

Default parameters should be used sparingly. Whenever there is more than a handful of optional parameters that do not have a natural order, it is preferable to use destructuring (as in Destructuring) to create APIs that are readable.

Note:

In contrast to Python's default parameters, it is okay to use initializers that return mutable objects (such as `{}` or `[]`). This is due to the fact that the initializer is evaluated every time the default value is used, so a single object cannot be shared across invocations.

Tip:

There is no restriction on the use of arbitrary expressions or function calls as initializers as long as they are kept as simple as possible. Initiators that expose shared mutable state should be avoided, as these can easily introduce unintended coupling between the calls to a function.

10.5.5.2. REST PARAMETERS

Consider using a *rest* parameter instead of accessing the `arguments`. JSDoc specifies rest parameters with a prefix of `...` In the list of parameters, the rest parameter must be the last parameter to be listed. It is important to note that there is no space between the `...` and the parameter name. Do not name the rest parameter `var_args`. Never name a local variable or parameter `arguments` , which confusingly shadows the built-in name.

Example:

```
/**
 * @param {!Array<string>} array This is an ordinary parameter.
 * @param {...number} numbers The remainder of arguments are all numbers.
 */

function variadic(array, ...numbers) {}

```

Figure 85: Rest Parameters

10.5.6. GENERICS

If necessary, declare generic functions and methods with a `@template TYPE` in the JSDoc above the function or method definition.

10.5.7. SPREAD OPERATOR

A function call can be used with the spread operator (`...`). The spread operator is preferable to `Function.prototype.apply` when an array or iterable is unpacked into multiple parameters of a variadic function. There is no space after the `...`.

Example:

```
1.function myFunction(...elements) {}
2.myFunction(...array, ...iterable, ...generator());
3.
```

Figure 86: Spread Operator

10.6. STRING LITERALS

10.6.1. USE SINGLE QUOTES

The ordinary string literal is delimited with a single quote ('), not a double quote ("), the usual way of delimiting strings.

> **Tip**:
> if a string contains a single quote character, consider using a template string to avoid having to escape the quote.

Ordinary string literals may not span multiple lines.

10.6.2. TEMPLATE LITERALS

The use of template literals (delimited with the backtick `) is much more efficient than the concatenation of complex strings, especially when multiple literal strings are involved. A template literal may span more than one line.

It is not necessary for a template literal that spans multiple lines to follow the indentation of the enclosing block, although it can be done if the added whitespace doesn't matter.

+ Example:

```
function arithmetic(a, b) {
  return `Here is a table of arithmetic operations:
${a} + ${b} = ${a + b}
${a} - ${b} = ${a - b}
${a} * ${b} = ${a * b}
${a} / ${b} = ${a / b}`;
}

```

Figure 87: Template Literal

10.6.3. NO LINE CONTINUATIONS

Line continuations (i.e., ending a line inside a string literal with a backslash) are not permitted either in ordinary string literals or in template string literals. The ES5 allows you to do this, but if there is a trailing whitespace after the slash, then it can lead to tricky errors, in addition to being less obvious to the reader.

⊘ Not Allowed:

```
const longString = 'This is a very long string that far exceeds the 80 \
    column limit. It unfortunately contains long stretches of spaces due \
    to how the continued lines are indented.';

```

Figure 88: Line Continuations

Instead, write

```
const longString = 'This is a very long string that far exceeds the 80 ' +
    'column limit. It does not contain long stretches of spaces since ' +
    'the concatenated strings are cleaner.';

```

Figure 89: Line Continuations

10.7. Number Literals

A number can be expressed in decimal, hexadecimal, octal, or binary form. The prefixes `0x`, `0o`, and `0b`, accompanied by lowercase letters, are used for hex, octal, and binary numbers, respectively. There should never be a leading zero unless it is immediately followed by `x`, `o`, or `b`.

10.8. Control Structures

10.8.1. For Loops

With *ES6*, the language now provides three different kinds of `for` loops. When possible, `for-of` loops should be preferred over all other loop types, however.

The use of `for-in` loops is only allowed on dict-style objects (see Do not mix quoted and unquoted keys); thus, they should not be used to iterate over an array. `Object.prototype.hasOwnProperty` should be used in `for-in` loops in order to exclude prototype properties that are not wanted. When possible, prefer `for-of` and `Object.keys` over `for-in`.

10.8.2. Exceptions

The exceptions are an important part of the language and should always be used whenever there are exceptional cases that occur. Throw only `Error` s or subclasses of `Error` : you should never throw string literals or other objects. An `Error` should always be constructed using the `new`.

This treatment extends to `Promise` rejection values as `Promise.reject(obj)` is equivalent to `throw obj;` in async functions.

An exception can be customized to provide additional error information from a function. As such, it is important that they are defined and used wherever the native `Error` type is not sufficient.

Throwing exceptions is preferred over ad-hoc error-handling techniques (such as passing an error container reference type or returning an object with an error property).

10.8.2.1. EMPTY CATCH BLOCKS

There are very few instances when it is correct to do nothing in response to a caught exception. There are times when it is appropriate not to take any action whatsoever in a catch block, and the reason for this is explained in a comment.

```
try {
  return handleNumericResponse(response);
} catch (ok) {
  // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

Figure 90: Catch Block

⊘ Not Allowed:

```
  try {
    shouldFail();
    fail('expected an error');
  } catch (expected) {
  }
```

Figure 91: Catch Block – Not Allowed

> **Tip**:
>
> As opposed to some other languages, patterns like the one mentioned above simply won't work since this will catch the error thrown by `fail`. Use `assertThrows()` instead.

10.8.3. SWITCH STATEMENTS

> **Terminology Note**:
>
> A switch block contains one or more statement groups within its braces. There are one or more switch labels in each statement group (either `case FOO:` or `default:`), followed by one or more statements.

10.8.3.1. FALL-THROUGH: COMMENTED

Each statement group within a switch block either terminates abruptly (with a `break`, `return`, or `throw` n exception) or is marked with a comment indicating that execution will continue into the next statement group. There can be any comment that conveys the idea of fall-through typically `// fall through` is sufficient. As far as the last statement group of the switch block is concerned, this special comment is not required.

+ Example:

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
  // fall through
  case 3:
    handleOneTwoOrThree();
    break;
  default:
    handleLargeNumber(input);
}

```

Figure 92: Fall-Through: Comment

10.8.3.2. THE DEFAULT CASE IS PRESENT

Even if a switch statement does not contain any code, it still includes a default statement group. The `default` statement group must be placed last.

10.9. this

Ensure that `this` is only used in class constructors and methods, in arrow functions defined within class constructors and methods, as well as in functions that have an explicit `@this` declared in the immediately-enclosing function's JSDoc.

`this` should never be used as a way to refer to the global object, the context of an `eval`, the target of an event, or to unnecessarily `call()`ed or `apply()`ed functions.

10.10. Equality Checks

The identity operators (`===`/`!==`) should be used in all cases except where described below.

10.10.1. Exceptions Where Coercion is Desirable

It is a good idea to catch both null and undefined values, as both represent an absence of a value. That means checking for undefined is one of the few times in JavaScript that using `==` is recommended.

Catch both `null` and `undefined` values:

```
1.if (someObjectOrPrimitive == null) {
2.  // Checking for null catches both null and undefined for objects and
3.  // primitives, but does not catch other falsy values like 0 or the empty
4.  // string.
5.}
6.
```

Figure 93: Catch Null and Undefined Values

10.11. Disallowed Features

10.11.1. with

Do not use the `with` keyword as it may be the source of confusing bugs and compatibility issues. It makes your code harder to understand and has been banned in strict mode since ES5.

10.11.2. Dynamic Code Evaluation

`eval()` is evil. Do not use `eval` or the `Function(...string)` constructor (except for code loaders). There are a number of risks associated with these features, and simply do not work in Content Security Policy (CSP) environments.

10.11.3. AUTOMATIC SEMICOLON INSERTION

Although semicolons are optional, statements must always terminate with semicolons (except for function and class declarations, as noted above). See Semicolons are required.

10.11.4. NON-STANDARD FEATURES

The use of non-standard features should be avoided. `WeakMap.clear` is a good example of old features that are no longer supported, new features that are not yet standardized (e.g., the current TC39 working draft), proposals at any stage of development, or proposed standards that have not yet been finalized), or proprietary features that are only implemented in some browsers. Be sure to only use features that are defined in the current ECMA-262 or WHATWG. standards.

> **Note:**
> If a project is written in accordance with a specific API, such as Chrome extensions or Node.js, then those APIs can in turn be used by the project.

The use of non-standard language "extensions" (such as those provided by some external transpilers) is strictly forbidden.

10.11.5. WRAPPER OBJECTS FOR PRIMITIVE TYPES

The primitive object wrappers should never be accessed via `new` (`Boolean`, `Number`, `String`, `Symbol`), nor should they be included in type annotations for primitive objects.

The first problem is that primitive wrapper objects are, in fact, objects. That means `typeof` will return "object" instead of "string", "number", or "boolean". The second problem comes with boolean objects. Every object is truthy, which means an instance of Boolean always resolves to true even when its actual value is false.

⊘ **Not Allowed:**

```
1.const /** Boolean */ x = new Boolean(false);
2.if (x) alert(typeof x);  // alerts 'object' - WAT?
3.
```

Figure 94: Primitive Wrapper Objects

Alternatively, wrappers may be called as functions in order to coerce (which is preferred over using `+` or concatenating the empty string) or creating symbols.

✚ **Example:**

```
1.const /** boolean */ x = Boolean(0);
2.if (!x) alert(typeof x);  // alerts 'boolean', as expected
3.
```

Figure 95: Primitive Wrapper Objects

10.11.6. MODIFYING BUILTIN OBJECTS

You don't control the built-in object's future. To avoid naming collisions, incompatible implementations, and maintenance nightmares, never modify built-in types, either by adding methods to their constructors or to their prototypes. Avoid depending on libraries that do this.

> **Note:**
> Where possible, JSCompiler's runtime library will provide standards-compliant polyfills; nothing else may modify built-in objects.

You should avoid adding symbols to the global object unless it is absolutely necessary to do so (such as if it is required by a third-party API).

10.11.7. OMITTING () WHEN INVOKING A CONSTRUCTOR

A constructor should never be invoked in a `new` statement without the use of parentheses ().

⊘ **Not Allowed:**

```
1.new Foo;
2.
```

Figure 96: Omitting () When Invoking a Constructor – Not Allowed

Use instead:

```
1.new Foo();
2.
```

Figure 97: Omitting () When Invoking a Constructor

It is possible to make subtle mistakes by omitting parentheses. These two lines are not equivalent:

```
1.new Foo().Bar();
2.new Foo.Bar();
3.
```

Figure 98: Omitting () When Invoking a Constructor

▲ Table of Contents

11. NAMING

11.1. RULES COMMON TO ALL IDENTIFIERS

Only ASCII characters and digits are used in identifiers, along with underscores in a small number of cases and very rarely (when required by frameworks like Angular) dollar signs.

The name should be as descriptive as possible, within the limits of what is reasonable. It is not necessary to worry about saving horizontal space in your code since it is much more important to ensure that your code is immediately understandable to a new reader. There should be no use of abbreviations that are ambiguous or unfamiliar to readers outside your project, and you should never abbreviate by deleting letters within a word.

```
1.errorCount          // No abbreviation.
2.dnsConnectionIndex  // Most people know what "DNS" stands for.
3.referrerUrl         // Ditto for "URL".
4.customerId          // "Id" is both ubiquitous and unlikely to be misunderstood.
5.
```

Figure 99: Identifiers

⊘ **Not Allowed:**

```
1.n                   // Meaningless.
2.nErr                // Ambiguous abbreviation.
3.nCompConns          // Ambiguous abbreviation.
4.wgcConnections      // Only your group knows what this stands for.
5.pcReader            // Lots of things can be abbreviated "pc".
6.cstmrId             // Deletes internal letters.
7.kSecondsPerDay      // Do not use Hungarian notation.
8.
```

Figure 100: Identifiers – Not Allowed

11.2. RULES BY IDENTIFIER TYPE

11.2.1. PACKAGE NAMES

The names of the packages are all written in `lowerCamelCase`. For example, `my.exampleCode.rocketSurgery`, but not `my.examplecode.rocketsurgery` or `my.example_code.computer_brain`.

11.2.2. Class names

Naming convention rules for classes are pretty similar to functions. We have to use descriptive titles that explain the class's capabilities.

The names of classes, interfaces, records, and typedefs are written in `UpperCamelCase`. Oftentimes, unexported classes are simply locals: they are not marked `@private` and are therefore not named with an underscore at the end of their names.

It is common for type names to be nouns or noun phrases. For example, `Request`, `ImmutableList`, or `VisibilityMode`. It should be noted that interface names can also sometimes be adjectives or adjective phrases (for example, `Readable`).

11.2.3. Method names

The structure of a function and a method is very similar, even though there are some differences between them. So, naming convention rules are the same.

Names of methods are written in `lowerCamelCase`. There should be a trailing underscore at the end of the name of the `@private` method.

It is typical for method names to be verbs or verb phrases. For example, `sendMessage` or `stop_`. The use of getter and setter methods for properties is not required, but if they are used, they should be named `getFoo` (or optionally `isFoo` or `hasFoo` for `booleans`) or `setFoo(value)` for setters.

There is also the possibility of using underscores in the names of JsUnit test methods in order to distinguish the logical components of the name. One typical pattern is `test<MethodUnderTest>_<state>_<expectedOutcome>`, for example `testPop_emptyStack_throws`. In terms of naming test methods, there is no single correct way of naming test methods.

11.2.4. Enum names

In the same way as classes, enum names should be written in `UpperCamelCase` and should generally be singular nouns. In the enum, the individual items are referred to by their names in `CONSTANT_CASE`.

11.2.5. Constant names

A constant name must be written in `CONSTANT_CASE`: it must contain all uppercase letters, with words separated by underscores. Since (`implicitly private`) module locals can be used in place of (private static properties), there is no need for a constant to be named with a trailing underscore.

11.2.5.1. DEFINITION OF "CONSTANT"

It is true that every constant is either a static property of `@const` or a module-local `const` declaration; however, not every static property of `@const` or a module-local `const` s declaration is a constant. Consider whether or not the field truly feels like a deeply immutable constant before selecting constant case. For example, if any of the observable states of that instance are capable of changing over time, then it is almost certainly not a constant. In most cases, simply intending to never modify the object is not enough.

+ Examples:

```
// Constants
const NUMBER = 5;
/** @const */ exports.NAMES = ImmutableList.of('Ed', 'Ann');
/** @enum */ exports.SomeEnum = { ENUM_CONSTANT: 'value' };
 Allowed
// Not constants
let letVariable = 'non-const';
class MyClass { constructor() { /** @const {string} */ this.nonStatic = 'non-static'; } };
/** @type {string} */ MyClass.staticButMutable = 'not @const, can be reassigned';
const /** Set<string> */ mutableCollection = new Set();
const /** ImmutableSet<SomeMutableType> */ mutableElements = ImmutableSet.of(mutable);
const Foo = goog.require('my.Foo');  // mirrors imported name
const logger = log.getLogger('loggers.are.not.immutable');

```

Figure 101: Constants

The names of constants are usually nouns or noun phrases.

11.2.5.2. LOCAL ALIASES

Whenever possible, local aliases should be used in place of fully-qualified names whenever they improve readability. Follow the same rules as `goog.requires` (see goog.require and goog.requireType Statements), maintaining the last part of the aliased name. Also, aliases can be used within functions. Aliases must be `const`.

+ Examples:

```
const staticHelper = importedNamespace.staticHelper;
const CONSTANT_NAME = ImportedClass.CONSTANT_NAME;
const {assert, assertInstanceof} = asserts;

```

Figure 102: Local Aliases

11.2.6. NON-CONSTANT FIELD NAMES

The names of non-constant fields (static or otherwise) are written in lowercase, with a trailing underscore for fields that are private.

It is typically the case that these names are nouns or noun phrases. For example, `computedValues` or `index_`.

11.2.7. PARAMETER NAMES

In the case of parameter names, they are written in `lowerCamelCase`. Regardless of whether the parameter expects a constructor or not, this rule still applies.

It is not recommended to use one-character parameter names in public methods.

> **Exception**:
>
> The parameter name may begin with a `$` when required by a third-party framework. This exception does not apply to any other identifiers (e.g., local variables or properties).

11.2.8. LOCAL VARIABLE NAMES

JavaScript variable names are case-sensitive. The names of local variables are written in `lowerCamelCase`, except for module-local (top-level) constants, as described above. Constants in function scopes are presented in `lowerCamelCase`. It should be noted that `lowerCamelCase` is used even if the variable has a constructor.

11.2.9. TEMPLATE PARAMETER NAMES

The name of the template parameter should be a concise, one-word, or one-letter identifier, and it should be all capital letters, such as `TYPE` or `THIS`.

11.2.10. MODULE-LOCAL NAMES

Non-exported module-local names are implicitly private. Neither are they marked `@private`, nor do they end in an underscore. It is important to note that this rule applies to classes, functions, variables, constants, enums, and other module-local identifiers.

11.3. CAMEL CASE: DEFINED

It is often the case that there is more than one acceptable method of converting an English phrase into camel case, for example, when acronyms or unusual constructs such as IPv6 or iOS are present. As a means of improving predictability, this style standard specifies the following (almost) deterministic scheme.

Take a look at the prose form of the name:

1. Remove any apostrophes and convert the phrase to ASCII. For example, Marchée's algorithm might become Marchee algorithm.
2. Then divide this result into words, splitting on any spaces and any punctuation marks that are remaining (typically hyphens).
    a. **Recommended**: in the event that there is a word that already appears in camel case in common usage, break it down into its constituent parts (e.g., AdWords becomes ad words).

> **Note**:
>
> There is no real camel case to a word such as iOS; it defies convention, so this recommendation is not applicable.

3. The next step is to lowercase everything (including acronyms), then to uppercase only the first character of:
    a. … each word, to yield upper camel case, or
    b. … each word except the first, to yield lower camel case
4. Finally, join all the words into a single identifier

> **Note**:
>
> Almost no consideration is given to the casing of the original words.

Examples:

Prose form	Correct	Incorrect
“XML HTTP request”	XmlHttpRequest	XMLHTTPRequest
“new customer ID”	newCustomerId	newCustomerID
“inner stopwatch”	innerStopwatch	innerStopWatch
“supports IPv6 on iOS?”	supportsIpv6OnIos	supportsIPv6OnIOS
“YouTube importer”	YouTubeImporter	YoutubeImporter*

Table 5: Word Casing

*Acceptable, but not recommended.

Note:

There are some words in the English language that are ambiguously hyphenated: Nonempty and nonempty, for example, are both correct, so the method names `checkNonempty` and `checkNonEmpty` are likewise both correct.

▲ Table of Contents

12. 7 JSDoc

All classes, fields, and methods are documented using JSDoc.

12.1. General Form

The basic format of JSDoc blocks can be seen in the following example:

```
/**
 * Multiple lines of JSDoc text are written here,
 * wrapped normally.
 * @param {number} arg A number to do something to.
 */
function doSomething(arg) { … }
```

Figure 103: JSDoc - Blocks

In the case of this single-line example:

```
/** @const @private {!Foo} A short bit of JSDoc. */
this.foo_ = foo;

```

Figure 104: JSDoc - Single Line

When a single-line comment extends into multiple lines, it must use the multi-line style with `/**` and `*/` on their own lines.

JSDoc comments are used by many tools to extract metadata that can be used for code validation and optimization. As a result, it is important that these comments are well-formed.

12.2. MARKDOWN

JSDoc is written in Markdown, though it may occasionally include HTML when it is necessary.

There are some tools that automatically extract JSDoc (e.g., JsDossier), but these tools will often ignore plain text formatting, so if you did this:

```
/**
 * Computes weight based on three factors:
 *   items sent
 *   items received
 *   last timestamp
 */

```

Figure 105: JSdoc Extraction

it would come out like this:

```
Computes weight based on three factors: items sent items received last timestamp
```

Figure 106: JSdoc Extraction

Instead, write a Markdown list:

```
/**
 * Computes weight based on three factors:
 *
 *  - items sent
 *  - items received
 *  - last timestamp
 */

```

Figure 107: JSdoc - Markdown

12.3. JSDoc TAGS

There is a subset of JSDoc tags that are allowed. See JSDoc tag reference for the complete list. Most tags must occupy their own line, with the tag at the beginning of the line.

⊘ Not Allowed:

```
/**5
 * The "param" tag must occupy its own line and may not be combined.
 * @param {number} left @param {number} right
 */
function add(left, right) { ... }

```

Figure 108: JSDoc Tags

A simple tag that does not require any additional data (such as `@private`, `@const`, `@final`, `@export`) can be combined onto the same line and can be accompanied by an optional type when appropriate.

```
/**
 * Place more complex annotations (like "implements" and "template")
 * on their own lines.  Multiple simple tags (like "export" and "final")
 * may be combined in one line.
 * @export @final
 * @enum {Iterable<TYPE>}
 * @template TYPE
 */
class MyClass {
  /**
   * @param {!ObjType} obj Some object.
   * @param {number=} num An optional number.
   */
  constructor(obj, num = 42) {
    /** @private @const {!Array<!ObjType|number>} */
    this.data_ = [obj, num];
  }
}

```

Figure 109: JSdoc Combined

When combining tags, or in which order to combine them, there is no hard rule, but be consistent when doing so.

In order to gain a general understanding of annotating types in JavaScript, please see Annotating JavaScript for the Closure Compiler and Types in the Closure Type System.

12.4. LINE WRAPPING

Block tags that are wrapped in lines are indented by four spaces. Wrapped description text can be lined-up with the description that appears on the following lines; however, this horizontal alignment should be avoided.

```
/**
 * Illustrates line wrapping for long param/return descriptions.
 * @param {string} foo This is a param with a description too long to fit in
 *     one line.
 * @return {number} This returns something that has a description too long to
 *     fit in one line.
 */
exports.method = function(foo) {
  return 5;
};

```

Figure 110: Line Wrapping

When wrapping a `@desc` or `@fileoverview` description, do not indent the description.

12.5. TOP/FILE-LEVEL COMMENTS

There may be a top-level file overview. In addition to the default Visibility annotations, a copyright notice, author information, and copyright notice are all optional. The use of file overviews is generally recommended whenever a file contains more than one class definition. This top-level comment is intended to provide a general orientation to the contents of the file to readers who are unfamiliar with the code. When present, it may provide a description of the contents of the file as well as any dependencies or compatibility information. There is no indentation on wrapped lines.

+ Example:

```
/**
 * @fileoverview Description of file, its uses and information
 * about its dependencies.
 * @package
 */

```

Figure 111: Top/File-Level Comments

12.6. CLASS COMMENTS

A description of each class, interface, and record is required, as well as any template parameters, interfaces that have been implemented, visibility, or other appropriate tags. In the class description, it is essential that the reader is provided with enough information to enable them to understand what the class is, how it should be used, and any additional considerations that need to be taken in order to correctly use the class. You may omit the textual descriptions on the constructor. When a class is used to declare an `@interface` or to extend a generic class, neither the `@constructor` nor the `@extends` annotations should be used with the class keyword.

```
/**
 * A fancier event target that does cool things.
 * @enum {Iterable<string>}
 */
class MyFancyTarget extends EventTarget {
  /**
   * @param {string} arg1 An argument that makes this more interesting.
   * @param {!Array<number>} arg2 List of numbers to be processed.
   */
   constructor(arg1, arg2) {
     // ...
   }
};

/**
 * Records are also helpful.
 * @extends {Iterator<TYPE>}
 * @record
 * @template TYPE
 */
class Listable {
  /** @return {TYPE} The next item in line to be returned. */
  next() {}
}

```

Figure 112: Class Comments

12.7. ENUM AND TYPEDEF COMMENTS

It is important to document all enums and typedefs on the preceding line with appropriate JSDoc tags (`@typedef` or `@enum`). It is also necessary to provide a description for public enums and typedefs. Individual items of an enum can be documented by means of a JSDoc comment that appears on the preceding line.

```
/**
 * A useful type union, which is reused often.
 * @typedef {!Bandersnatch|!BandersnatchType}
 */
let CoolUnionType;


/**
 * Types of bandersnatches.
  * @enum {string}
  */
const BandersnatchType = {
  /** This kind is really frumious. */
  FRUMIOUS: 'frumious',
  /** The less-frumious kind. */
  MANXOME: 'manxome',
};

```

Figure 113: Enum And Typedef Comments

Using typedefs, you can define short record types or use them as aliases for unions, complex functions, or generic types. If the record type has a large number of fields, typedefs should be avoided since they do not allow you to document individual fields, nor do they allow you to use templates or recursive references. If you have a large record type, you should use `@record`.

12.8. METHOD AND FUNCTION COMMENTS

In the case of methods and named functions, the types of parameters and return values must be documented, with the exception of `@override`s with the same signatures, in which case all types are omitted. The documentation of `this` type should be provided when necessary. The return type may be omitted if the function does not have any non-empty `return` statements.

You may omit the method, parameter, and return descriptions (but not the type descriptions) in case the JSDoc or the signature for the method indicates that these descriptions are not necessary.

In the description of a method, an action verb phrase is used to describe what the method does. Instead of being written in the imperative form, this phrase is written in the third person, as if there is an implied 'This method `...` ' before it.

Methods that override superclass methods must include an `@override` annotation. In the overridden method, all JSDoc annotations inherited from the superclass method (including visibility annotations) should be omitted from the overridden method. It is necessary to specify all `@param` and `@return` annotations explicitly if any type is refined in type annotations.

```
/** A class that does something. */
class SomeClass extends SomeBaseClass {
  /**
   * Operates on an instance of MyClass and returns something.
   * @param {!MyClass} obj An object that for some reason needs detailed
   *     explanation that spans multiple lines.
   * @param {!OtherClass} obviousOtherClass
   * @return {boolean} Whether something occurred.
   */
   someMethod(obj, obviousOtherClass) { ... }

   /** @override */
   overriddenMethod(param) { ... }
}

/**
 * Demonstrates how top-level functions follow the same rules.  This one
 * makes an array.
 * @param {TYPE} arg
 * @return {!Array<TYPE>}
 * @template TYPE
 */
function makeArray(arg) { ... }

```

Figure 114: Method and Function Comments

When you need only to document the param and return types of a function, you can optionally use inline JSDocs in the function's signature. The return and param types are described in these inline JSDocs without tags.

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

```

Figure 115: Method and Function Comments – JSdoc Inline

If you would like to add descriptions or tags to the method, then use a single JSDoc comment above the method. A method that returns a value, for example, should have the `@return` tag.

```
class MyClass {
  /**
   * @param {number} arg
   * @return {string}
   */
  bar(arg) {...}
}

```

Figure 116: Method and Function Comments - JSDoc Tags to a Method

```
// Illegal inline JSDocs.

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

/** Function description. */ bar() {...}

```

Figure 117: Method and Function Comments - Illegal Inline JSDocs

It is generally accepted that annotations in anonymous functions are optional. The following example illustrates how to annotate param and return types when automatic type inference is inadequate or explicit annotation improves the readability:

```
promise.then(
    /** @return {string} */
    (/** !Array<string> */ items) => {
      doSomethingWith(items);
      return items[0];
    });

```

Figure 118: Comments - Param and Return Type Annotations

For function type expressions, see Function type expressions.

12.9. PROPERTY COMMENTS

Property types must be documented. The description of a private property may not be necessary if the name and type of the property provide sufficient documentation in order to understand the code.

Constants that are exported publicly are commented in the same manner as properties.

```
/** My class. */
class MyClass {
  /** @param {string=} someString */
  constructor(someString = 'default string') {
    /** @private @const {string} */
    this.someString_ = someString;

    /** @private @const {!OtherType} */
    this.someOtherThing_ = functionThatReturnsAThing();

    /**
     * Maximum number of things per pane.
     * @type {number}
     */
    this.someProperty = 4;
  }
}

/**
 * The number of times we'll try before giving up.
 * @const {number}
 */
MyClass.RETRY_COUNT = 33;

```

Figure 119: Property Comments

12.10. TYPE ANNOTATIONS

There is a requirement that type annotations can be found on the `@param`, `@return`, `@this`, and `@type tags`, and optionally on `@const`, `@export`, and any visibility tags. The type annotations that accompany JSDoc tags must always be enclosed in braces.

12.10.1. NULLABILITY

As part of the type system, there are modifiers `!` and `?` for nullable and non-null, respectively. These modifiers must precede the type.

There are different requirements for nullability modifiers based on different types, broadly divided into two categories:

1. As a default, type annotations for primitives (`string`, `number`, `boolean`, `symbol`, `undefined`, `null`) and literals (`{function(...): ...}` and `{{foo: string...}}`) are non-nullable. You can make it nullable by using the `?` modifier, but don't include the redundant `!`.
2. A reference type (generally, anything in `UpperCamelCase`, such as `some.namespace.ReferenceType`) refers to a class, enum, record, or typedef that is defined elsewhere. In the case of these types, it is impossible to tell whether they are nullable from the name alone since they may or may not be nullable. In order to prevent ambiguity at the point of use, you should always use explicit `!` and `?` modifiers for these types.

👎 **Bad:**

```
const /** MyObject */ myObject = null; // Non-primitive types must be annotated.
const /** !number */ someNum = 5; // Primitives are non-nullable by default.
const /** number? */ someNullableNum = null; // ? should precede the type.
const /** !{foo: string, bar: number} */ record = ...; // Already non-nullable.
const /** MyTypeDef */ def = ...; // Not sure if MyTypeDef is nullable.

// Not sure if object (nullable), enum (non-nullable, unless otherwise
// specified), or typedef (depends on definition).
const /** SomeCamelCaseName */ n = ...;
```

Figure 120: Nullability Modifiers - Incorrect

👍 **Good:**

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

Figure 121: Nullability Modifiers - Correct

12.10.2. Type Casts

In the event that the compiler is unable to accurately infer the type of an expression, in which case the assertion functions in goog.asserts are unable to remedy the problem, it is possible to tighten the type by adding a type annotation comment and enclosing the expression in parentheses. It is important to note that parentheses are a requirement.

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

Figure 122: Type Casts

12.10.3. Template Parameter Types

Parameters for a template should always be specified. In this way, the compiler is able to do a better job, and for the readers, it is much easier to understand what is happening within the code.

👎 **Bad:**

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

Figure 123: Template Parameter Types - Incorrect

👍 **Good:**

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

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

Figure 124: Template Parameter Types - Correct

In the following case, template parameters should not be used:

- `Object` is used for type hierarchy and not as a map-like structure.

12.10.4. FUNCTION TYPE EXPRESSIONS

> **Terminology Note**:
>
> The term *function type expression* refers to a type annotation for function types containing the keyword function in the annotation (see examples below).

You should not use a function type expression where the function definition is given. The parameter and return types should be specified using `@param` and `@return`, or using inline annotations (see Method and function comments). This includes both anonymous functions and functions that have been defined and assigned to a `const` (where the function jsdoc appears above the whole assignment expression).

There is a need to use function type expressions, for example, inside `@typedef`, `@param` or `@return`. It can also be used for variables or properties of the function type, if such variables or properties are not immediately initialized with the function definition.

```
1.  /** @private {function(string): string} */
2.  this.idGenerator_ = googFunctions.identity;
3.
```

Figure 125: Function Type Expressions

The return type should always be specified explicitly whenever you are using a function type expression. When this is not the case, the default return type is unknown (`?`)resulting in strange and unexpected behavior that is rarely what is actually desired.

👎 **Bad:** - type error, but no warning given:

```
1./** @param {function()} generateNumber */
2.function foo(generateNumber) {
3.  const /** number */ x = generateNumber();  // No compile-time type error here.
4.}
5.
6.foo(() => 'clearly not a number');
7.
```

Figure 126: Function Type Expressions - Incorrect

👍 **Good:**

```
/**
 * @param {function(): *} inputFunction1 Can return any type.
 * @param {function(): undefined} inputFunction2 Definitely doesn't return
 *      anything.
 * NOTE: the return type of `foo` itself is safely implied to be {undefined}.
 */
function foo(inputFunction1, inputFunction2) {...}

```

Figure 127: Function Type Expressions - Correct

12.10.5. WHITESPACE

The type annotation should be formatted with a single space or line break after each comma or colon. It may be necessary to insert additional line breaks in order to improve readability or to avoid exceeding the column limit. It is recommended to choose and indent these breaks in accordance with the guidelines (e.g., Line-wrapping and Block indentation: +2 spaces). Type annotations cannot contain any other whitespace.

👍 **Good:**

```
/** @type {function(string): number} */

/** @type {{foo: number, bar: number}} */

/** @type {number|string} */

/** @type {!Object<string, string>} */

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

/**
 * @type {function(
 *     !SuperDuperReallyReallyLongTypedefThatForcesTheLineBreak,
 *     !OtherVeryLongTypedef): string}
 */

/**
 * @type {!SuperDuperReallyReallyLongTypedefThatForcesTheLineBreak|
 *     !OtherVeryLongTypedef}
 */

```

Figure 128: Whitespace - Correct

👎 **Bad:**

```
1.// Only put a space after the colon
2./** @type {function(string) : number} */
3.
4.// Put spaces after colons and commas
5./** @type {{foo:number,bar:number}} */
6.
7.// No space in union types
8./** @type {number | string} */
9.
```

Figure 129: Whitespace - Incorrect

12.11. VISIBILITY ANNOTATIONS

The visibility annotations (`@private`, `@package`, `@protected`) may be specified in the `@fileoverview` block, or they may be specified on any exported symbol or property. There should be no visibility specified for local variables, either within a function or at the top level of a module. At the end of all `@private` names, there must be an underscore.

▲ Table of Contents

13. Policies

13.1. Issues unspecified by this General Style and Coding Conventions Standard: Be consistent!

In the event that a specific style question isn't resolved definitively by this specification, then try to follow what the other code in the same file is already doing. Consider emulating the rest of the files in the same package if that does not resolve the question.

13.2. Compiler warnings

13.2.1. Use a standard warning set

There should be a preference for projects to use `--warning_level=VERBOSE`.

13.2.2. How to handle a warning

Before you take any action, you need to make sure you understand exactly what the warning is telling you. In the event that you are not sure why a warning is appearing, you should ask for help.

Having understood the warning, you should try the following solutions in order:

1. **First, fix it or work around it.** Try your best to address the warnings explicitly, or alternatively, find a way to accomplish the task that will eliminate the need to confront the warning entirely.
2. **Otherwise, determine if it's a false alarm.** It is possible that the warning is invalid and that the code is actually safe and correct, and if you are convinced this is the case, add a comment to further convince the reader of this fact and apply the `@suppress` annotation.
3. **Otherwise, leave a TODO comment.** This is a last resort. In the event that you do this, make sure not to suppress the warning. It is important to keep the warning visible until it can be properly addressed.

13.2.3. SUPPRESS A WARNING AT THE NARROWEST REASONABLE SCOPE

In suppressing warnings, they are suppressed to the narrowest reasonable scope, usually that of a single local variable or a very small method. Many times, a variable or method is extracted for no other reason than that.

+ Example:

```
1./** @suppress {uselessCode} Unrecognized 'use asm' declaration */
2.function fn() {
3.  'use asm';
4.  return 0;
5.}
6.
```

Figure 130: Suppressed Warning

Even if there are a large number of suppressions in a class, this is still better than blinding the entire class to this type of warning.

13.3. DEPRECATION

Deprecated methods, classes, and interfaces should be marked with the `@deprecated` annotation. A deprecation comment must include simple, clear directions for people to fix their call sites.

13.4. CODE NOT IN THIS GENERAL STYLE AND CODING STANDARD

Your codebase may contain files that do not conform to this style and coding standard. It is possible that some of these may have come from an acquisition, that some may have been written before a position was taken on a particular subject, or that some may have been formatted, named, and styled differently for any other reason.

13.4.1. Reformatting existing code

In order to update the style of existing code, you should follow these guidelines.

1. The existing code does not have to be changed in order to comply with current style guidelines. When it comes to reformatting existing code, there is a trade-off between reducing code churn and improving consistency. It is inevitable that style rules will evolve over time, and these kinds of tweaks, in order to maintain compliance, will create unnecessary churn. However, if significant changes are being made to a file, it is expected that the file will be formatted according to this standard in accordance with Google Style.
2. It is very important to be careful not to let opportunistic style fixes end up confusing the focus of a CL. If you notice that you have been making a lot of style changes that aren't essential to the central focus of a CL, then you should move these changes to a separate CL.

13.4.2. Newly added code: use this standard adopted from Google Style

In the case of brand new files, use this standard as adopted from Google Style, regardless of the style choices of the other files in the same package.

In order to add new code to a file that does not comply with this standard or Google Style, it is recommended that the existing code be reformatted first, subject to the advice in Reformatting existing code.

In the event that this reformatting is not done, then new code should be as consistent as possible with existing code within the same file but must not violate this style guide.

13.5. LOCAL STYLE RULES

In addition to the style rules provided in this document, teams and projects may adopt additional rules, but they must accept that cleanup changes may not abide by those additional rules, and they must not block such cleanup changes simply because they violate any additional rules. There is a need to beware of excessive rules which do not serve any purpose. It is not the intention of the style guide to attempt to define style in every possible situation, and neither should you attempt to do so.

13.6. GENERATED CODE: MOSTLY EXEMPT

It is not necessary for the source code generated by the build process to adhere to the style set forth by this standard. Any generated identifiers that will be referenced from hand-written source code, however, must comply with the naming requirements. Such identifiers are allowed to contain underscores as a special exception, which may alleviate any conflicts with handwritten identifiers.

▲ Table of Contents

14. Appendix A

14.1. JSDoc Tag Reference

There are a number of purposes for which JSDoc can be used in JavaScript. Aside from the fact that it is used to generate documentation, it is also used to control the tooling. Probably the best known of these is the Closure Compiler annotations.

14.1.1. Type Annotations and other Closure Compiler Annotations

The documentation for the JSDoc that is used by the Closure Compiler is covered in Annotating JavaScript for the Closure Compiler and Types in the Closure Type System.

14.1.2. Documentation Annotations

In addition to the JSDoc described in Annotating JavaScript for the Closure Compiler, the following tags are widely supported by various software tools that generate documentation (such as JsDossier) for purely documentation purposes.

There are also other types of JSDoc annotations that may appear in third-party code. There are annotations that appear in the JSDoc Toolkit Tag Reference; however, they are not considered part of valid style according to this standard as adopted from Google Style.

14.1.2.1. @author OR @owner - NOT RECOMMENDED.

Not recommended.

Syntax: `@author username@dms.fl.gov (First Last)`

```
1.  /**
2.   * @fileoverview Utilities for handling textareas.
3.   * @author stephen.mitchell@dms.fl.gov (Stephen Mitchell)
4.   */
5.
```

Figure 131:@author or @owner

It is used to indicate who is the author of a file or the owner of a test and is generally used in the `@fileoverview` comment. The unit test dashboard uses the `@owner` tag in order to determine who is responsible for owning the test.

14.1.2.2. @bug

Syntax: `@bug bugnumber`

```
1.  /** @bug 1234567 */
2.  function testSomething() {
3.    // …
4.  }
5.
6.  /**
7.   * @bug 1234568
8.   * @bug 1234569
9.   */
10. function testTwoBugs() {
11.   // …
12. }
13.
```

Figure 132: @bug

Provides information about what bugs the given test function regression tests.

In order to make searching for regression tests as easy as possible, it is recommended that multiple bugs each have a separate `@bug` line.

14.1.2.3. @code - DEPRECATED. DO NOT USE.

Deprecated. Do not use. Use Markdown backticks instead.

Syntax: `{@code ...}`

Historically, `` `BatchItem` `` was written as `{@code BatchItem}`.

```
1.  /** Processes pending `BatchItem` instances. */
2.  function processBatchItems() {}
3.
```

Figure 133: @code

In JSDoc descriptions, this attribute is used to indicate that a term is a line of code so that it will be formatted appropriately in the generated documentation.

14.1.2.4. @desc

Syntax: `@desc Message description`

```
1.  /** @desc Notifying a user that their account has been created. */
2.  exports.MSG_ACCOUNT_CREATED = goog.getMsg(
3.      'Your account has been successfully created.');
4.
```

Figure 134: @desc

14.1.2.5. @link

Syntax: `{@link ...}`

The purpose of this tag is to generate cross-reference links within the generated documentation.

```
1.  /** Processes pending {@link BatchItem} instances. */
2.  function processBatchItems() {}
3.
```

Figure 135: @link

> **Historical Note**:
>
> In generated documentation, `@link` tags have also been used to create external links. Always use Markdown's link syntax for external links instead:

```
1.  /**
2.   * This class implements a useful subset of the
3.   * [native Event interface](https://dom.spec.whatwg.org/#event).
4.   */
5.  class ApplicationEvent {}
6.
```

Figure 136: @link

14.1.2.6. @see

Syntax: `@see Link`

```
1.  /**
2.   * Adds a single item, recklessly.
3.   * @see #addSafely
4.   * @see goog.Collect
5.   * @see goog.RecklessAdder#add
6.   */
7.
```

Figure 137: @see

Make a reference to a lookup of another class function or method.

14.1.2.7. @supported

Syntax: `@supported Description`

```
./**
 * @fileoverview Event Manager
 * Provides an abstracted interface to the browsers' event systems.
 * @supported IE10+, Chrome, Safari
 */

```

Figure 138: @supported

In a fileoverview, this is used to indicate what browsers are supported by the file.

14.1.3. FRAMEWORK SPECIFIC ANNOTATIONS

There are a number of annotations that are specific to a particular framework.

14.1.3.1. @ngInject FOR ANGULAR 1

14.1.3.2. @polymerBehavior FOR POLYMER

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

14.1.4. Notes about standard Closure Compiler annotations

In the past, the following tags were considered standard; however, they have now been deprecated.

14.1.4.1. @expose - Deprecated. Do not use.

Deprecated. Do not use. Use `@export` and/or `@nocollapse` instead.

14.1.4.2. @inheritDoc - Deprecated. Do not use.

Deprecated. Do not use. Use @override instead.

14.2. Commonly misunderstood style rules

The following is a collection of some lesser-known or commonly misunderstood facts about the Google Style for JavaScript that has been adopted for this standard. (These are all true statements; they are not a compilation of myths or misconceptions.)

- Neither a copyright statement nor an `@author` credit is required to be included in a source file. (Neither is explicitly recommended, either.)
- The order in which a class's members should be arranged does not have any hard and fast rules to follow (Classes).
- It is usually possible to represent empty blocks concisely as `{}`, as described in the subsection (Empty blocks: may be concise).
- In line-wrapping, the prime directive is to break at a higher syntactic level (Where to break).
- The use of non-ASCII characters is permitted in string literals, comments, and JSDoc, and is actually recommended in some cases when using non-ASCII characters makes the code easier to read than using an equivalent Unicode escape (Non-ASCII characters).

14.3. Style-related tools

There are several tools available to support various aspects of Google Style and, as a result, this standard.

14.3.1. Closure Compiler

As part of this program, you will find type checking, efficiency optimizations, and other transformations (such as code lowering from ECMAScript 6 to ECMAScript 5).

14.3.2. clang-format

By the use of this program, JavaScript source code will be reformatted into Google Style, which has been adopted by this standard, as well as following a number of formatting practices that are not required but are often helpful in improving readability. The output produced by `clang-format` is compliant with the style guide.

`clang-format` is not required. The authors are allowed to change the output of the program, and the reviewers are allowed to request such changes; in case of disagreements, these are resolved in the usual way. There is, however, the possibility of subtrees opting into such enforcement locally.

14.3.3. Closure Compiler Linter

By using this program, you will be able to check for a variety of missteps and anti-patterns.

14.3.4. Conformance framework

JS Conformance Framework is a tool that is part of the Closure Compiler that is designed to provide developers with a simple means to specify a set of additional checks that will be run against their code base in addition to the standard checks. For example, a conformance check may prevent access to a certain property, or that a certain function may not be called, or it may find that type information is missing (unknowns).

Most commonly, these rules are used to enforce critical restrictions (such as defining globals that could break the codebase or to enforce security patterns (such as using `eval` or assigning to `innerHTML`), or, more loosely, to improve code quality.

For additional information, see the official documentation for the JS Conformance Framework.

14.4. EXCEPTIONS FOR LEGACY PLATFORMS

14.4.1. OVERVIEW

Here, we are going to discuss some of the exceptions and additional rules that need to be followed when ECMAScript 6 syntax is not available to code authors. It is necessary to make exceptions to the recommended style when it is not possible to use the syntax defined in ECMAScript 6 as outlined here:

- Use of `var` declarations is allowed
- Use of `arguments` is allowed
- Optional parameters without default values are allowed

14.4.2. USE VAR

14.4.2.1. var DECLARATIONS ARE NOT BLOCK-SCOPED

In the case of `var` declarations, they are scoped to the beginning of the enclosing function, script or module, which can lead to unexpected behavior, especially when function closures reference `var` declarations that are inside of loops. The following code gives an example:

```
1.  for (var i = 0; i < 3; ++i) {
2.    var iteration = i;
3.    setTimeout(function() { console.log(iteration); }, i*1000);
4.  }
5.
6.  // logs 2, 2, 2 -- NOT 0, 1, 2
7.  // because `iteration` is function-scoped, not local to the loop.
8.
```

Figure 139: var Declarations - Incorrect

14.4.2.2. DECLARE VARIABLES AS CLOSE AS POSSIBLE TO FIRST USE

Although the scope of a `var` declaration is defined at the beginning of the enclosing function, it is advisable to place the `var` declaration as close to its first use as possible for readability purposes. In any case, do not declare a `var` inside a block if it is referenced outside the block. For example:

```
1.  function sillyFunction() {
2.    var count = 0;
3.    for (var x in y) {
4.      // "count" could be declared here, but don't do that.
5.      count++;
6.    }
7.    console.log(count + ' items in y');
8.  }
9.
```

Figure 140: var Declarations

14.4.2.3. USE @const FOR CONSTANTS VARIABLES

It should be noted that for global declarations where the `const` keyword would normally be used if it were available, you should annotate the `var` declaration with the `@const` keyword instead (this is optional for local variables).

14.4.3. DO NOT USE BLOCK SCOPED FUNCTIONS DECLARATIONS

Do not do this:

```
1.if (x) {
2.  function foo() {}
3.}
4.
```

Figure 141: Block Scoped Functions Declarations - Incorrect

The JavaScript VMs implemented before ECMAScript 6 do support function declarations within blocks, but these declarations are not standardized. There were inconsistencies between the implementations of block-scoped functions and the ECMAScript 6 behavior, which is now standard for block-scoped functions. In the strict mode of ECMAScript 5 and earlier versions, function declarations are allowed only in the root statement list of a script or function and are explicitly not allowed in block scopes.

For consistent behavior, use a `var` initialized with a function expression to define a function within a block:

```
1.if (x) {
2.  var foo = function() {};
3.}
4.
```

Figure 142: Block Scoped Functions Declarations - Correct

14.4.4. Dependency management with goog.provide / goog.require

14.4.4.1. Summary

> **Warning**:
> `goog.provide` dependency management is deprecated. Even in projects using `goog.provide` for older files, all new files should use `goog.module`. These rules apply only to pre-existing `goog.provide` files.

- All the `goog.provides` should be placed first, then all the `goog.requires` should be placed second. `provides` should be separated from `requires` by an empty line.
- Sort the entries alphabetically (uppercase first).
- Make sure that you don't wrap the `goog.provide` and `goog.require` statements. If necessary, you may exceed 80 columns.
- Only provide top-level symbols.

`goog.provide` statements should be grouped together and placed first. All `goog.require` statements should follow. It is important to separate the two lists with an empty line.

The `goog.provide` and `goog.require statements` should be written in a single line, just like the import statements in other languages, regardless of whether they exceed the limit of 80 columns in the line.

It is recommended that the lines be sorted alphabetically, with uppercase letters appearing first:

```
1.  goog.provide('namespace.MyClass');
2.  goog.provide('namespace.helperFoo');
3.
4.  goog.require('an.extremelyLongNamespace.thatSomeoneThought.wouldBeNice.andNowItIsLon
    ger.Than80Columns');
5.  goog.require('goog.dom');
6.  goog.require('goog.dom.TagName');
7.  goog.require('goog.dom.classes');
8.  goog.require('goog.dominoes');
9.
```

Figure 143: goog.provide, goog.require Statements

Whenever a class has members, they should all be defined in the same file. You should only provide top-level class definitions in a file that contains multiple members that are defined on the same class (e.g., enums, inner classes, etc.).

👍 **Do this:**

```
1.  goog.provide('namespace.MyClass');
2.
```

Figure 144: Top-Level Class Definitions - goog.provide Statement - Correct

👎 **Not this:**

```
1.  goog.provide('namespace.MyClass');
2.  goog.provide('namespace.MyClass.CONSTANT');
3.  goog.provide('namespace.MyClass.Enum');
4.  goog.provide('namespace.MyClass.InnerClass');
5.  goog.provide('namespace.MyClass.TypeDef');
6.  goog.provide('namespace.MyClass.staticMethod');
7.
```

Figure 145: Top-Level Class Definitions - goog.provide Statement - Incorrect

Members on namespaces may also be provided:

```
1.  goog.provide('foo.bar');
2.  goog.provide('foo.bar.CONSTANT');
3.  goog.provide('foo.bar.method');
4.
```

Figure 146: Top-Level Class Definitions - Members on Namespaces

14.4.4.2. ALIASING WITH goog.scope

> **Warning**:
> `goog.scope` is deprecated. Even in projects that already use goog.scope, new files shouldn't use it.

Using `goog.scope,` it is possible to shorten references to namespaced symbols in code using `goog.provide / goog.require` dependency management.

Each file may only have one instance of `goog.scope` added to it. It should always be placed in the global scope.

The opening `goog.scope(function() {` invocation must be preceded by one blank line and must be followed by any `goog.provide` statements, `goog.require` statements, or top-level comments. It is essential to close the invocation on the last line of the file. As part of the closing statement of the scope, append `// goog.scop`. There should be two spaces between the comment and the semicolon.

In the same way as with C++ namespaces, do not indent under the `goog.scope` declarations. Instead, continue from the 0 column.

There should only be aliases made for names that will not be re-assigned to another object (e.g., most constructors, enums, and namespaces). (see below for how to alias a constructor). Do not do this:

```
1.  goog.scope(function() {
2.  var Button = goog.ui.Button;
3.
4.  Button = function() { ... };
5.  ...
6.
```

Figure 147: Aliasing with Goog.Scope - Incorrect

It is important that the names be the same as the last property of the global that they are aliasing.

```
1.  goog.provide('my.module.SomeType');
2.
3.  goog.require('goog.dom');
4.  goog.require('goog.ui.Button');
5.
6.  goog.scope(function() {
7.  var Button = goog.ui.Button;
8.  var dom = goog.dom;
9.
10. // Alias new types after the constructor declaration.
11. my.module.SomeType = function() { ... };
12. var SomeType = my.module.SomeType;
13.
14. // Declare methods on the prototype as usual:
15. SomeType.prototype.findButton = function() {
16.   // Button as aliased above.
17.   this.button = new Button(dom.getElement('my-button'));
18. };
19. ...
20. });  // goog.scope
21.
```

Figure 148: Aliasing with Goog.Scope - Correct

14.4.4.3. goog.forwardDeclare

Rather than using `goog.forwardDeclare` to break circular dependencies between files within the same library, it is preferable to use `goog.requireType`. The `goog.requireType` statement is different from the `goog.require` statement in that it allows a namespace to be imported before it is defined.

In legacy code, `goog.forwardDeclare` may still be used to break circular references that span across library boundaries; avoid it.

The statements within `goog.forwardDeclare` must follow the same style rules as those found in `goog.require` and `goog.requireType`. All statements related to `goog.forwardDeclare`, `goog.require`, and `goog.requireType` are sorted alphabetically.

▲ Table of Contents

15. Appendix B

15.1. Tool Chain

Tool Name	Description
Acunetix	Quickly find and fix vulnerabilities that put web applications at risk of attack.
Chrome DevTools	Set of web developer tools built directly into the Google Chrome browser.
Clang-Format	Clang-Format is an alternative code formatting tool that helps maintain a common code style across team members and IDEs.
Google Closure Compiler	The Closure Compiler is a tool for making JavaScript download and run faster. It is a true compiler for JavaScript. Instead of compiling from a source language to machine code, it compiles from JavaScript to better JavaScript.
Closure Library	The Closure Library contains many different tools, from user interface widgets to communication utilities. The different parts of the library share a set of conventions that help organize the code and make it easy to use.
Eclipse	JavaScript Development Tools provide plug-ins that implement an IDE supporting the development of JavaScript applications and JavaScript within web applications.

Tool Name	Description
ESLint	An open-source plugin for JavaScript and JSX. All its rules are pluggable, so developers can create the linting rules they prefer.
Firefox JavaScript Debugger	JavaScript Debugger assists in finding bugs by stepping through JavaScript code and examining or modifying its state.
Jasmine	Test suite. It's a fully automated behavior-based test suite.
Javascript Beautifier	Online Javascript beautifier takes ugly, minified, or obfuscated javascript and makes it clean, well-formatted code.
JavaScript Development Tools (JSDT)	Plug-ins that implement an IDE supporting the development of JavaScript applications and JavaScript within web applications.
Javascript Minifier	Minify javascript code using Online Javascript Minifier to make your javascript code more optimized for websites.
Javascript Obfuscator	Online Javascript Obfuscator makes javascript code harder to read or understand to protect from theft or reuse.
Jest	Testing Framework with a focus on simplicity
js-Beauty	Beautify, unpack or deobfuscate JavaScript and HTML, make JSON/JSONP readable, etc.

Tool Name	Description
JS Conformance Framework	The JS Conformance framework is a tool that is part of the Closure Compiler that allows developers to specify configuration for automatically checking compiled code for conformance to certain requirements, such as forbidding access to a certain property or calls to a certain function.
JSDoc	Documenting JavaScript
JSDossier	JSDossier is a JSDoc parsing tool built on top of the Closure Compiler. JSDossier uses the compiler to parse your code and build a type graph. It then traverses the graph to find types to generate documentation for.
JSHint	A tool that helps to detect errors and potential problems in JavaScript code.
MOCHA	Feature-rich JavaScript test framework
Playwright	End-to-end testing for modern web apps.
Prettier	Code formatter that integrates with most editors and supports JavaScript, HTML, CSS, GraphQL, Markdown, YAML
Standard built-in objects	JavaScript's standard, built-in objects, including their methods and properties.
Swagger	Design and document APIs

Tool Name	Description
The Firefox JavaScript Debugger	Debugger enables you to step through JavaScript code and examine or modify its state to help track down bugs.
npm	An open-source package Manager allows developers to install and publish those packages.
X-Team	31 Essential Javascript Tools for Productive Developers

Table 6: JavaScript Tool Chain

▲ Table of Contents

16. Table of Figures

▲ Table of Contents

17. Table of Tables

▲ Table of Contents

18. References

18.1. Fair Use

The Copyright Law of the United States provides legal protection for intellectual property.

The fair use of a copyrighted work, including reproduction through copies or phonorecords, or by other means specified in Section 106, for criticism, commentary, news reporting, teaching, scholarship, or research, is not an infringement of the copyright.

The preceding paper is derived from research, content, code blocks, examples, and paraphrasing of the references cited and embedded in the Reference section of this document, as defined by the "Fair Use" rule of copyright, 17 U.S. Code, §106A.

▲ Table of Contents

18.2. PUBLICATIONS

Airbnb, A. (2021, July 2). GitHub - airbnb/javascript: JavaScript Style Guide. GitHub.

Retrieved from - https://github.com/airbnb/javascript#destructuring

Anton Rogov, A. R. (2011, September 14). JavaScript Coding Guidelines. Git Hub Gist.

Retrieved from — https://gist.github.com/antonrogov/1216380/21800f463af3d3a98e98083c4bc109e44f981ef4

Artem Zakharchenko, A. Z. (2021, February 25). GitHub - kettanaito/naming-cheatsheet:. GitHub.

Retrieved from — https://github.com/kettanaito/naming-cheatsheet

Comprehensive language-agnostic guidelines on a useful framework for naming Classes, Functions, and Variables. Home of the A/HC/LC pattern.

Bent Cardan (Reqshark), B. (2012, November 24). GitHub - reqshark/js-standards: A JavaScript Style Guide. GitHub.

Retrieved from — https://github.com/reqshark/js-standards

This guide was forked from Felix Geisendörfer's guide and is licensed under the Creative Commons BY-SA 3.0 license. The original guide is located here. You are encouraged to fork this repository and make adjustments according to your preferences

BrewersAssociation. (2018, April 9). GitHub - BrewersAssociation/Javascript-Coding-Standards: A set of rules to govern Javascript, code styles, and other best practices. © 2022 GitHub, Inc.

Retrieved from — https://github.com/BrewersAssociation/Javascript-Coding-Standards#functions

This content is licensed under this Creative Commons Zero license. For details, see our site-policy repository.

DeepSource Corp. © 2022, D. (2021, December 4). JavaScript best practices to improve code quality. JavaScript Best Practices to Improve Code Quality.

Retrieved from — https://deepsource.io/blog/javascript-code-quality-best-practices/

We are a technology company that builds tools for developers. Global engineering teams and open-source projects of all sizes use DeepSource to build software that makes a dent in the universe.

Dries Buytaert, & Drupal Association. (2021, March 11). JavaScript coding standards.

Retrieved from — https://www.drupal.org/docs/develop/standards/javascript/javascript-coding-standards

Drupal's online documentation is © 2000-2022 by the individual contributors, and can the content can be shared and reused under the Creative Commons License, Attribution, ShareAlike 2.0

Google GitHub. (2013). Google JavaScript Style Guide. Google JavaScript Style Guide.

Retrieved from — https://google.github.io/styleguide/jsguide.html#features-control-structures

The document serves as the complete definition of Google's coding standards for source code in the JavaScript programming language.

Gupta, T., & The Startup. (2021, December 10). Node.js Coding Style Guidelines - The Startup. Medium.

Retrieved from — https://medium.com/swlh/node-js-coding-style-guidelines-74a20d00c40b

Ilya Kantor © 2007–2022. (2022, June 7). The Modern JavaScript Tutorial. The Modern JavaScript Tutorial.

Retrieved from — https://javascript.info/

JavaScript.info! is licensed as Creative Commons-BY-NC-SA. So it is possible to republish it while obeying the terms of the license.

Jeremy Bao, J. B. (2013, September 17). LabCodeStyle/Javascript.md at master · bao1018/LabCodeStyle. GitHub.

Retrieved from — https://github.com/bao1018/LabCodeStyle/blob/master/Javascript.md

JS Foundation - js.foundation, OpenJS Foundation, & jQuery Foundation. (2016). JavaScript Style Guide | Contribute to jQuery. JavaScript Style Guide.

Retrieved from — https://contribute.jquery.org/style-guide/js/

jQuery is a JavaScript library designed to simplify HTML DOM tree traversal and manipulation. As of May 2019, jQuery is used by 73% of the 10 million most popular websites. jQuery is free, open-source software using the permissive MIT license.

Kantor, I. (2021, December 12). Coding Style. JavaScript Info.

Retrieved from — https://javascript.info/coding-style

The content of this tutorial is Open Source, and everyone is welcome to contribute.

Luiz Eduardo Bertolini (2021, December 16). GitHub - lebertolini/StandardJS: This repository aims to teach the standard js style; it aims to define how code should be written and organized in JavaScript. GitHub.

Retrieved from — https://github.com/lebertolini/StandardJS

This content is licensed under this Creative Commons Zero license. For details, see our site-policy repository.

Mozilla Foundation Portions Of This Content Are ©1998–2022. (2022). JavaScript | MDN. MDM Web Docs.

Retrieved from — https://developer.mozilla.org/en-US/docs/Web/JavaScript

Content authored by Mozilla is generally made available for public sharing and reuse through open licenses such as Creative Commons

Omkar Nagvekar, O. N. (2016, January 23). GitHub - omkarnagvekar/emberjs-standards: JavaScript Coding Standards - ININ Mumbai. GitHub.

Retrieved from — https://github.com/omkarnagvekar/emberjs-standards

Javascript/ES6 specific rules

Open Knowledge Foundation © 2009–2018. (2022, April 21). JavaScript coding standards — CKAN 2.9.5 documentation. JavaScript Coding Standards — CKAN 2.9.5 Documentation.

Retrieved from — https://docs.ckan.org/en/2.9/contributing/javascript.html

CKAN is a tool for making open data websites. (Think of a content management system like WordPress - but for data, instead of pages and blog posts.) It helps you manage and publish collections of data. It is used by national and local governments, research institutions, and other organizations that collect a lot of data. Licensed under Creative Commons BY 4.0, ShareAlike (Unported) v3.0 license.

OpenJS Foundation Dojo Toolkit Contributors., & Dojo Toolkit Contributors. (2020, June 1). Dojo Style Guide — The Dojo Toolkit - Reference Guide. Dojo Style Guide.

Retrieved from — https://dojotoolkit.org/reference-guide/1.9/developer/styleguide.html

Dojo Toolkit (stylized as dōjō toolkit) is an open-source modular JavaScript library (or, more specifically, JavaScript toolkit) designed to ease the rapid development of cross-platform, JavaScript/Ajax-based applications and websites. Content on this site, including all materials posted by the Open JS Foundation, is licensed under a Creative Commons Attribution 4.0 License.

Ortus-Solutions (2020, July 9). coding-standards/javascript.md at master · Ortus-Solutions/coding-standards. GitHub.

Retrieved from — https://github.com/Ortus-Solutions/coding-standards/blob/master/javascript.md

Refsnes Data Copyright 1999–2022. (2009, August 9). JavaScript Introduction.

Retrieved from — https://www.w3schools.com/js/js_intro.asp

W3Schools is optimized for learning, testing, and training. W3Schools is a freemium educational website for learning to code online.

Rick Waldron, R. W. (2012, July 17). GitHub - rwaldron/idiomatic.js: Principles of Writing Consistent, Idiomatic JavaScript. GitHub.

Retrieved from — https://github.com/rwaldron/idiomatic.js

Licensed under a Creative Commons Attribution 3.0 Unported License. (https://creativecommons.org/licenses/by/3.0/deed.en_US) Based on a work at github.com/rwldrn/idiomatic.js. (https://github.com/rwaldron/idiomatic.js)

Ryan McDermott, R. M. (2021, May 23). GitHub - ryanmcdermott/clean-code-javascript: Clean Code concepts adapted for JavaScript. GitHub.

Retrieved from — https://github.com/ryanmcdermott/clean-code-javascript#solid

Uber. (2022, May 6). guide/style.md at master · uber-go/guide. GitHub.

Retrieved from — https://github.com/uber-go/guide/blob/master/style.md

Uber's open-source software for Go development

Vijay Dev, V. D. (2014, March 20). GitHub - vijay-stayntouch/JS-coding-standards: JavaScript Coding Standards. GitHub.

Retrieved from — https://github.com/vijay-stayntouch/JS-coding-standards

Wirfs-Brock, A., & Terlson, B. (2016). ECMAScript® 2016 Language Specification. ECMA-262 7th Edition.

Retrieved from — https://262.ecma-international.org/7.0/#

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met.

WordPress Developer Resources. (2003). JavaScript Coding Standards | Coding Standards Handbook | WordPress Developer Resources. JavaScript Coding Standards | Coding Standards Handbook.

Retrieved from — https://developer.wordpress.org/coding-standards/wordpress-coding-standards/javascript/

▲ References

18.3. WEBSITES

Research Websites	
@use JSDoc	airbnb/javascript
Brewers Association Coding Standards	CKAN JavaScript Coding Standards
Clean-Code-Javascript	Clean-Code-Javascript
Drupal JavaScript Coding Standards	Google JavaScript Style Guide
Idiomatic.js	JavaScript Best Practices to Improve Code Quality
JavaScript Coding Guidelines	Javascript Coding Standard
Javascript Coding Standards - ININ	JavaScript Info
JavaScript Secrets Every Developer Should Know	JavaScript Standard Style
JavaScript Standard Style	JavaScript Standards
JavaScript The Right Way	jQuery JavaScript Style Guide
Kettanaito/Naming-Cheatsheet	LabCodeStyle
MDM JavaScript Guidelines	node-Style-Guide
Standard built-in objects	
Ortus JavaScript Style Guide	Standard JavaScript

Uber Go Style Guide	What is JavaScript Standard Style
WordPress Javascript Coding Standards	

Table 7: Research Websites

▲ References

18.4. ACKNOWLEDGMENT

I would like to thank the following individuals for their expertise and support throughout the development of this manuscript:

Michael Bennet,	Service Delivery
Nelson Blocker,	Service Delivery
Raghib Qureshi,	Utility System/Engineering Specialist
Rhonda Ballew,	Service Delivery
Sean McGinnis,	Computer Audit Analyst
Todd Chambery,	Service Delivery

Table 8: Acknowledgements