Google JavaScript Style Guide

Table of Contents

🔗 1 Introduction

Style if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the hard-and-fast rules that we follow universally, and avoids giving advice that isn't clearly enforceable (whether by human or tool).

1.1 Terminology notes

In this document, unless otherwise clarified:

1. The term *comment* always refers to *implementation* comments. We do not use the phrase "documentation comments", instead using the common term "JSDoc" for both human-readable text and machine-readable annotations within `/** … */`.

2. This Style Guide uses [RFC 2119](#) terminology when using the phrases *must*, *must not*, *should*, *should not*, and *may*. The terms *prefer* and *avoid* correspond to *should* and *should not*, respectively. Imperative and declarative statements are prescriptive and correspond to *must*.

Other "terminology notes" will appear occasionally throughout the document.

1.2 Guide notes

Example code in this document is **non-normative**. That is, while the examples are in Google Style, they may not illustrate the *only* stylish way to represent the code. Optional formatting choices made in examples must not be enforced as rules.

2 Source file basics

2.1 File name

File names must be all lowercase and may include underscores (`_`) or dashes (`-`), but no additional punctuation. Follow the convention that your project uses. Filenames' extension must be `.js`.

2.2 File encoding: UTF-8

Source files are encoded in **UTF-8**.

2.3 Special characters

2.3.1 Whitespace characters

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
TypeScript guide.

1. All other whitespace characters in string literals are escaped, and

2. Tab characters are **not** used for indentation.

🔗 2.3.2 Special escape sequences

For any character that 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}`). Legacy octal escapes are never used.

🔗 2.3.3 Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character (e.g. `∞`) or the equivalent hex or Unicode escape (e.g. `\u221e`) is used, depending only on which makes the code **easier to read and understand**.

> Tip: In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

```
/* 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 cla
return '\ufeff' + content;  // Prepend a byte order mark.
```

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

> Tip: Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that happens, those programs are **broken** and they must be **fixed**.

🔗 3 Source file structure

All new source files should either be a `goog.module` file (a file containing a `goog.module` call) or an ECMAScript (ES) module (uses `import` and `export` statements).

Files consist of the following, **in order**:

  3. `goog.module` statement, if a `goog.module` file

  4. ES `import` statements, if an ES module

  5. `goog.require` and `goog.requireType` statements

  6. The file's implementation

Exactly one blank line separates each section that is present, except the file's implementation, which may be preceded by 1 or 2 blank lines.

3.1 License or copyright information, if present

If license or copyright information belongs in a file, it belongs here.

3.2 `@fileoverview` JSDoc, if present

See 7.5 Top/file-level comments for formatting rules.

3.3 `goog.module` statement

All `goog.module` files must declare exactly one `goog.module` name on a single line: lines containing a `goog.module` declaration must not be wrapped, and are therefore an exception to the 80-column limit.

The entire argument to `goog.module` is what defines a namespace. It is the package name (an identifier that reflects the fragment of the directory structure where the code lives) plus, optionally, the main class/enum/interface that it defines concatenated to the end in `lowerCamelCase`.

Example:

```
goog.module('search.urlHistory.urlHistoryService');
```

3.3.1 Hierarchy

Module namespaces may never be named as a *direct* child of another module's namespace.

Disallowed:

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

The directory hierarchy reflects the namespace hierarchy, so that deeper-nested children are subdirectories of higher-level parent directories. Note that this implies that owners of "parent" namespace groups are necessarily aware of all child namespaces, since they exist in the same directory.

3.3.2 `goog.module.declareLegacyNamespace`

~~goog.module.declareLegacyNamespace()~~ when possible.

Example:

```
goog.module('my.test.helpers');
goog.module.declareLegacyNamespace();
goog.setTestOnly();
```

`goog.module.declareLegacyNamespace` exists to ease the transition from traditional object hierarchy-based namespaces but comes with some naming restrictions. As the child module name must be created after the parent namespace, this name **must not** be a child or parent of any other `goog.module` (for example, `goog.module('parent');` and `goog.module('parent.child');` cannot both exist safely, nor can `goog.module('parent');` and `goog.module('parent.child.grandchild');`).

🔗3.3.3 `goog.module` Exports

Classes, enums, functions, constants, and other symbols are exported using the `exports` object. Exported symbols may be defined directly on the `exports` object, or else declared locally and exported separately. Symbols are only exported if they are meant to be used outside the module. Non-exported module-local symbols are not declared `@private`. There is no prescribed ordering for exported and module-local symbols.

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};
```

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

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

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

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

Do not use default exports as they don't translate easily to ES module semantics.

```
exports = FancyClass;
```

🔗 3.4 ES modules

ES modules are files that use the `import` and `export` keywords.

🔗 3.4.1 Imports

Import statements must not be line wrapped and are therefore an exception to the 80-column limit.

3.4.1.1 Import paths

ES module files must use the `import` statement to import other ES module files. Do not
`goog.require` another ES module.

```
import './sideeffects.js';

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

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

3.4.1.1.1 File extensions in import paths

The `.js` file extension is not optional in import paths and must always be included.

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

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

3.4.1.2 Importing the same file multiple times

Do not import the same file multiple times. This can make it hard to determine the aggregate
imports of a file.

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

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

3.4.1.3.1 Naming module imports

Module import names (`import * as name`) are `lowerCamelCase` names that are derived from the imported file name.

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

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

Some libraries might commonly use a namespace import prefix that violates this naming scheme, but overbearingly common open source use makes the violating style more readable. The only library that currently falls under this exception is threejs, using the `THREE` prefix.

3.4.1.3.2 Naming default imports

Default import names are derived from the imported file name and follow the rules in 6.2 Rules by identifier type.

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

> Note: In general this should not happen as default exports are banned by this style guide, see 3.4.2.1 Named vs default exports. Default imports are only used to import modules that do not conform to this style guide.

3.4.1.3.3 Naming named imports

In general symbols imported via the named import (`import {name}`) should keep the same name. Avoid aliasing imports (`import {SomeThing as SomeOtherThing}`). Prefer fixing name collisions by using a module import (`import *`) or renaming the exports themselves.

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

new bigAnimals.Cat();
new domesticatedAnimals.Cat();
```

If renaming a named import is needed then use components of the imported module's file name or path in the resulting alias.

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

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

🔗3.4.2 Exports

Symbols are only exported if they are meant to be used outside the module. Non-exported module-local symbols are not declared `@private`. There is no prescribed ordering for exported and module-local symbols.

3.4.2.1 Named vs default exports

Use named exports in all code. You can apply the `export` keyword to a declaration, or use the `export {name};` syntax.

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

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

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

```
// Alternate style named exports:
class Foo { ... }

export {Foo};
```

3.4.2.2 Mutability of exports

Exported variables must not be mutated outside of module initialization.

There are alternatives if mutation is needed, including exporting a constant reference to an object that has mutable fields or exporting accessor functions for mutable data.

```
// 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
 */
```

```
    foo = newMutateFoo(foo);
  };
}
```

```
// Good: Rather than export the mutable variables foo and mutateFoo dire
// instead make them module scoped and export a getter for foo and a wra
// 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;
}
```

3.4.2.3 export from

`export from` statements must not be line wrapped and are therefore an exception to the 80-column limit. This applies to both `export from` flavors.

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

🔗 3.4.3 Circular Dependencies in ES modules

Do not create cycles between ES modules, even though the ECMAScript specification allows this.
Note that it is possible to create cycles with both the `import` and `export` statements.

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

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

// `export from` can cause circular dependencies too!
export {x} from './c.js';
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

```
export let x;
```

🔗3.4.4 Interoperating with Closure

3.4.4.1 Referencing goog

To reference the Closure `goog` namespace, import Closure's `goog.js` .

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

const {compute} = goog.require('a.name');

export const CONSTANT = compute();
```

`goog.js` exports only a subset of properties from the global `goog` that can be used in ES modules.

3.4.4.2 goog.require in ES modules

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

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

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

3.4.4.3 Declaring Closure Module IDs in ES modules

`goog.declareModuleId` can be used within ES modules to declare a `goog.module` -like module ID. This means that this module ID can be `goog.require` d, `goog.module.get` d etc. as if it were a `goog.module` that did not call `goog.module.declareLegacyNamespace` . It does not create the module ID as a globally available JavaScript symbol.

A `goog.require` (or `goog.module.get`) for a module ID from `goog.declareModuleId` will always return the module object (as if it was `import *` 'd). As a result, the argument to `goog.declareModuleId` should always end with a `lowerCamelCaseName` .

> Note: It is an error to call `goog.module.declareLegacyNamespace` in an ES module, it can only be called from `goog.module` files. There is no direct way to associate a "legacy" namespace with an ES module.

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
TypeScript guide.

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

goog.declareModuleId('my.esm');

export class Class {};
```

🔗3.5 `goog.setTestOnly`

In a `goog.module` file the `goog.module` statement and, if present,
`goog.module.declareLegacyNamespace()` statement may optionally be followed by a call
to `goog.setTestOnly()`.

In an ES module the `import` statements may optionally be followed by a call to
`goog.setTestOnly()`.

🔗3.6 `goog.require` and `goog.requireType` statements

Imports are done with `goog.require` and `goog.requireType` statements. The names
imported by a `goog.require` statement may be used both in code and in type annotations,
while those imported by a `goog.requireType` may be used in type annotations only.

The `goog.require` and `goog.requireType` statements form a contiguous block with no
empty lines. This block follows the `goog.module` declaration separated by a single empty line.
The entire argument to `goog.require` or `goog.requireType` is a namespace defined by a
`goog.module` in a separate file. `goog.require` and `goog.requireType` statements may
not appear anywhere else in the file.

Each `goog.require` or `goog.requireType` is assigned to a single constant alias, or else
destructured into several constant aliases. These aliases are the only acceptable way to refer to
dependencies in type annotations or code. Fully qualified namespaces must not be used anywhere,
except as an argument to `goog.require` or `goog.requireType`.

Exception: Types, variables, and functions declared in externs files have to use their fully qualified
name in type annotations and code.

When `goog.require` is assigned to a single constant alias, it must match the final dot-
separated component of the imported module's namespace.

Exception: In certain cases, additional components of the namespace can be used to form a longer
alias. The resulting alias must retain the original identifier's casing such that it still correctly identifies
its type. Longer aliases may be used to disambiguate otherwise identical aliases, or if it significantly
improves readability. In addition, a longer alias must be used to prevent 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).

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

A file should not contain both a `goog.require` and a `goog.requireType` statement for the
same namespace. If the imported name is used both in code and in type annotations, it should be
imported by a single `goog.require` statement.

If a module is imported only for its side effects, the call must be a `goog.require` (not a
`goog.requireType`) and assignment may be omitted. A comment is required to explain why
this is needed and suppress a compiler warning.

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

> Tip: There's no need to memorize this order and enforce it manually. You can rely on your IDE
> to report requires that are not sorted correctly.

If a long alias or module name would cause a line to exceed the 80-column limit, it **must not** be
wrapped: require lines are an exception to the 80-column limit.

Example:

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

Discouraged:

```
// when using a `default export`, prefer destructuring into aliases.
const MyClass = goog.require('some.package.MyClass');
const MyType = goog.requireType('some.package.MyType');
```

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

Disallowed:

```
// 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
 *     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');
  // ...
}
```

🔗3.7 The file's implementation

The actual implementation follows after all dependency information is declared (separated by at
least one blank line).

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

🔗4 Formatting

Terminology Note: *block-like construct* refers to the body of a class, function, method, or brace-delimited block of code. Note that, by 5.2 Array literals and 5.3 Object literals, any array or object literal may optionally be treated as if it were a block-like construct.

> Tip: Use `clang-format`. The JavaScript community has invested effort to make sure clang-format "does the right thing" on JavaScript files. `clang-format` has integration with several popular editors.

🔗4.1 Braces

🔗4.1.1 Braces are used for all control structures

Braces are required for all control structures (i.e. `if`, `else`, `for`, `do`, `while`, as well as any others), even if the body contains only a single statement. The first statement of a non-empty block must begin on its own line.

Disallowed:

```
if (someVeryLongCondition())
  doSomething();

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

Exception: A simple if statement that can fit entirely on a single line with no wrapping (and that doesn't have an else) may be kept on a single line with no braces when it improves readability. This is the only case in which a control structure may omit braces and newlines.

```
if (shortCondition()) foo();
```

🔗4.1.2 Nonempty blocks: K&R style

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for *nonempty* blocks and block-like constructs:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace *if* that brace terminates a statement or the body of a function or class statement, or a class method. Specifically, there is *no* line break after the brace if it is followed by `else`, `catch`, `while`, or a comma, semicolon, or right-parenthesis.

```
class InnerClass {
  constructor() {}

  /** @param {number} foo */
  method(foo) {
    if (condition(foo)) {
      try {
        // Note: this might fail.
        something();
      } catch (err) {
        recover();
      }
    }
  }
}
```

4.1.3 Empty blocks: may be concise

An empty block or block-like construct *may* be closed immediately after it is opened, with no characters, space, or line break in between (i.e. `{}`), **unless** it is a part of a *multi-block statement* (one that directly contains multiple blocks: `if` / `else` or `try` / `catch` / `finally`).

Example:

```
function doNothing() {}
```

Disallowed:

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

try {
  // …
} catch (e) {}
```

4.2 Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in [4.1.2 Nonempty blocks: K&R style](#)).

4.2.1 Array literals: optionally "block-like"

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

```js
const a = [
  0,
  1,
  2,
];

const b =
    [0, 1, 2];
```

```js
const c = [0, 1, 2];

someMethod(foo, [
  0, 1, 2,
], bar);
```

Other combinations are allowed, particularly when emphasizing semantic groupings between elements, but should not be used only to reduce the vertical size of larger arrays.

🔗 4.2.2 Object literals: optionally "block-like"

Any object literal may optionally be formatted as if it were a "block-like construct." The same examples apply as [4.2.1 Array literals: optionally block-like](#). For example, the following are all valid (**not** an exhaustive list):

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

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

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

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

🔗 4.2.3 Class literals

Class literals (whether declarations or expressions) are indented as blocks. Do not add semicolons 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). For inheritance, the `extends` keyword is sufficient unless the superclass is templatized. Subclasses of templatized types must explicitly specify the template type in an `@extends` JSDoc annotation, even if it is just passing along the same template name.

```
/** @template T */
class Foo {
  /** @param {T} x */
  constructor(x) {
    /** @type {T} */
    this.x = x;
  }
}

/** @extends {Foo<number>} */
class Bar extends Foo {
  constructor() {
    super(42);
  }
}

exports.Baz = class extends Bar {
  /** @return {number} */
  method() {
    return this.x;
  }
};
```

```
/** @extends {Bar} */ // <-- unnecessary @extends
exports.Baz = class extends Bar {
  /** @return {number} */
  method() {
    return this.x;
  }
};
```

🔗 4.2.4 Function expressions

When declaring an anonymous function in the list of arguments for a function call, the body of the function is indented two spaces more than 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)
```

```
      // of the '.then()' call.
      if (result) {
        result.use();
      }
    });
```

4.2.5 Switch statements

As with any other block, the contents of a switch block are indented +2.

After a switch label, a newline appears, and the indentation level is increased +2, exactly as if a
block were being opened. An explicit block may be used if required by lexical scoping. The following
switch label returns to the previous indentation level, as if a block had been closed.

A blank line is optional between a `break` and the following case.

Example:

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

  case Animal.JABBERWOCK:
    handleJabberwock();
    break;

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

4.3 Statements

4.3.1 One statement per line

Each statement is followed by a line-break.

4.3.2 Semicolons are required

Every statement must be terminated with a semicolon. Relying on automatic semicolon insertion is
forbidden.

4.4 Column limit: 80

JavaScript code has a column limit of 80 characters. Except as noted below, any line that would
exceed this limit must be line-wrapped, as explained in 4.5 Line-wrapping.

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

    [goog.module statement](#) and [3.6 goog.require and goog.requireType statements](#)).

2. ES module `import` and `export from` statements (see [3.4.1 Imports](#) and [3.4.2.3](#) [export from](#)).

3. Lines where obeying the column limit is not possible or would hinder discoverability. Examples include:
   - A long URL which should be clickable in source.
   - A shell command intended to be copied-and-pasted.
   - A long string literal which may need to be copied or searched for wholly (e.g., a long file path).

🔗4.5 Line-wrapping

Terminology Note: *Line wrapping* is breaking a chunk of code into multiple lines to obey column limit, where the chunk could otherwise legally fit in a single line.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

> Note: While the typical reason for line-wrapping is to avoid overflowing the column limit, even code that would in fact fit within the column limit may be line-wrapped at the author's discretion.

> Tip: Extracting a method or local variable may solve the problem without the need to line-wrap.

🔗4.5.1 Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**.

Preferred:

```
currentEstimate =
    calc(currentEstimate + x * currentEstimate) /
        2.0;
```

Discouraged:

```
currentEstimate = calc(currentEstimate + x *
    currentEstimate) / 2.0;
```

In the preceding example, the syntactic levels from highest to lowest are as follows: assignment, division, function call, parameters, number constant.

Operators are wrapped as follows:

  a. This does not apply to the "dot" (**.**), which is not actually an operator.

2. A method or constructor name stays attached to the open parenthesis (**(**) that follows it.

3. A comma (**,**) stays attached to the token that precedes it.

4. A line break is never added between a return and the return value as this would change the meaning of the code.

5. JSDoc annotations with type names break after "{". This is necessary as annotations with optional types (@const, @private, @param, etc) do not scan the next line.

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

🔗4.5.2 Indent continuation lines at least +4 spaces

When line-wrapping, each line after the first (each *continuation line*) is indented at least +4 from the original line, unless it falls under the rules of block indentation.

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

4.6.3 Horizontal alignment: discouraged addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

🔗4.6 Whitespace

🔗4.6.1 Vertical whitespace

A single blank line appears:

1. Between consecutive methods in a class or object literal
  a. Exception: A blank line between two consecutive properties definitions in an object literal (with no other code between them) is optional. Such blank lines are used as needed to create *logical groupings* of fields.
2. Within method bodies, sparingly to create *logical groupings* of statements. Blank lines at the start or end of a function body are not allowed.
3. *Optionally* before the first or after the last method in a class or object literal (neither encouraged nor discouraged).
4. As required by other sections of this document (e.g. 3.6 goog.require and goog.requireType statements).

Multiple consecutive blank lines are permitted, but never required (nor encouraged).

🔗4.6.2 Horizontal whitespace

Use of horizontal whitespace depends on location, and falls into three broad categories: *leading* (at the start of a line), *trailing* (at the end of a line), and *internal*. Leading whitespace (i.e., indentation)

JSDoc, a single internal ASCII space also appears in the following places **only**.

1. Separating any reserved word (such as `if` , `for` , or `catch`) except for `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 that spaces are *never* allowed before these characters.

6. After the colon (`:`) in an object literal.

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)`).

🔗4.6.3 Horizontal alignment: discouraged

Terminology Note: *Horizontal alignment* is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but it is **generally discouraged** by Google Style. It is not even required to *maintain* horizontal alignment in places where it was already used.

Here is an example without alignment, followed by one with alignment. Both are allowed, 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
};
```

Tip: Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing

best it still corrupts version history information, slows down reviewers and exacerbates merge
conflicts.

🔗4.6.4 Function arguments

Prefer to put all function arguments on the same line as the function name. If doing so would
exceed the 80-column limit, the arguments must be line-wrapped in a readable way. To save space,
you may wrap as close to 80 as possible, or put each argument on its own line to enhance
readability. Indentation should be four spaces. Aligning to the parenthesis is allowed, but
discouraged. Below are the most common patterns for argument wrapping:

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

// 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) {
  // …
}
```

🔗4.7 Grouping parentheses: recommended

Optional grouping parentheses are omitted only when the author and reviewer agree that there is
no reasonable chance that the code will be misinterpreted without them, nor would they have made
the code easier to read. It is *not* reasonable to assume that every reader has the entire operator
precedence table memorized.

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

Parentheses are required for type casts: `/** @type {!Foo} */ (foo)`.

🔗 4.8 Comments

This section addresses *implementation comments*. JSDoc is addressed separately in [7 JSDoc](#).

🔗 4.8.1 Block comment style

Block comments are indented at the same level as the surrounding code. They may be in
`/* … */` or `//`-style. For multi-line `/* … */` comments, subsequent lines must start with
`*` aligned with the `*` on the previous line, to make comments obvious with no extra context.

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

// And so
// is this.

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

Comments are not enclosed in boxes drawn with asterisks or other characters.

Do not use JSDoc (`/** … */`) for implementation comments.

🔗 4.8.2 Parameter Name Comments

"Parameter name" comments should be used whenever the value and method name do not
sufficiently convey the meaning, and refactoring the method to be clearer is infeasible . Their
preferred format is before the value with "=":

```
someFunction(obviousParam, /* shouldRender= */ true, /* name= */ 'hello'
```

For consistency with surrounding code you may put them after the value without "=":

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

🔗 5 Language features

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

Language features which are not discussed in this style guide may be used with no
recommendations of their usage.

🔗5.1 Local variable declarations

🔗5.1.1 Use `const` and `let`

Declare all local variables with either `const` or `let` . Use `const` by default, unless a variable
needs to be reassigned. The `var` keyword must not be used.

🔗5.1.2 One variable per declaration

Every local variable declaration declares only one variable: declarations such as
`let a = 1, b = 2;` are not used.

🔗5.1.3 Declared when needed, initialized as soon as possible

Local variables are **not** habitually declared at the start of their containing block or block-like
construct. Instead, local variables are declared close to the point they are first used (within reason),
to minimize their scope, and initialized as soon as possible.

🔗5.1.4 Declare types as needed

JSDoc type annotations may be added either on the line above the declaration, or else inline before
the variable name if no other JSDoc is present.

Example:

```
const /** !Array<number> */ data = [];

/**
 * Some description.
 * @type {!Array<number>}
 */
const data = [];
```

Mixing inline and JSDoc styles is not allowed: the compiler will only process the first JsDoc and the
inline annotations will be lost.

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

> Tip: There are many cases where the compiler can infer a templatized type but not its
> parameters. This is particularly the case when the initializing literal or constructor call does not
> include any values of the template parameter type (e.g., empty arrays, objects, `Map` s, or

as unknown.

🔗5.2 Array literals

🔗5.2.1 Use trailing commas

Include a trailing comma whenever there is a line break between the final element and the closing bracket.

Example:

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

🔗5.2.2 Do not use the variadic `Array` constructor

The constructor is error-prone if arguments are added or removed. Use a literal instead.

Disallowed:

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

This works 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 = [];
```

Explicitly allocating an array of a given length using `new Array(length)` is allowed when appropriate.

🔗5.2.3 Non-numeric properties

Do not define or use non-numeric properties on an array (other than `length`). Use a `Map` (or `Object`) instead.

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). Elements should be omitted if
they are unused.

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

Destructuring may also be used for function parameters (note that a parameter name is required but
ignored). Always specify `[]` as the default value if a destructured array parameter is optional, and
provide default values on the left hand side:

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

Disallowed:

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

> Tip: For (un)packing multiple values into a function's parameter or return, prefer object
> destructuring to array destructuring when possible, as it allows naming the individual elements
> and specifying a different type for each.

5.2.5 Spread operator

Array literals may include the spread operator (`...`) to flatten elements out of one or more other
iterables. The spread operator should be used instead of more awkward constructs with
`Array.prototype` . There is no space after the `...` .

Example:

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

5.3 Object literals

5.3.1 Use trailing commas

Include a trailing comma whenever there is a line break between the final property and the closing
brace.

5.3.2 Do not use the `Object` constructor

While `Object` does not have the same problems as `Array` , it is still disallowed for consistency.
Use an object literal (`{}` or `{a: 0, b: 1, c: 2}`) instead.

quoted and/or computed keys). Do not mix these key types in a single object literal.

Disallowed:

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

This also extends to passing the property name to functions, like `hasOwnProperty` . In particular,
doing so will break in compiled code because the compiler cannot rename/obfuscate the string
literal.

Disallowed:

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

This is best implemented as:

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

🔗 5.3.4 Computed property names

Computed property names (e.g., `{['key' + foo()]: 42}`) are allowed, and are considered
dict-style (quoted) keys (i.e., must not be mixed with non-quoted keys) unless the computed
property is a [symbol](#) (e.g., `[Symbol.iterator]`). Enum values may also be used for computed
keys, but should not be mixed with non-enum keys in the same literal.

🔗 5.3.5 Method shorthand

Methods can be defined on object literals using the method shorthand (`{method() {… }}`) in
place of a colon immediately followed by a `function` or arrow function literal.

Example:

```
return {
  stuff: 'candy',
  method() {
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the [TypeScript guide](.

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

Example:

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

5.3.6 Shorthand properties

Shorthand properties are allowed on object literals.

Example:

```
const foo = 1;
const bar = 2;
const obj = {
  foo,
  bar,
  method() { return this.foo + this.bar; },
};
assertEquals(3, obj.method());
```

5.3.7 Destructuring

Object destructuring patterns may be used on the left-hand side of an assignment to perform destructuring and unpack multiple values from a single object.

Destructured objects may also be used as function parameters, but should be kept as simple as possible: a single level of unquoted shorthand properties. Deeper levels of nesting and computed properties may not be used in parameter destructuring. Specify any default values in the left-hand-side of the destructured parameter (`{str = 'some default'} = {}` , rather than `{str} = {str: 'some default'}`), and if a destructured object is itself optional, it must default to `{}` . The JSDoc for the destructured parameter may be given any name (the name is unused but is required by the compiler).

Example:

```
 * @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'} = {}) {}
```

Disallowed:

```
/** @param {{x: {num: (number|undefined), str: (string|undefined)}}} par
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}) {};
```

Destructuring may also be used for `goog.require` statements, and in this case must not be
wrapped: the entire statement occupies one line, regardless of how long it is (see [3.6 goog.require](#)
[and goog.requireType statements](#)).

🔗 5.3.8 Enums

Enumerations are defined by adding the `@enum` annotation to an object literal. Enums must be
module-local or assigned directly on `exports`, not nested under a type or object.

Additional properties may not be added to an enum after it is defined. Enums must be constant. All
enum values must be either a string literal or a number.

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

/**
 * An enum with two values.
 * @enum {number}
 */
const Value = {
  /** The value used shall have been the first. */
  FIRST_VALUE: 1,
  /** The second among two values. */
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

For string enums, all values must be statically initialized and not computed using arithmetic operators, template literal substitution, function calls or even a variable reference.

```js
const ABSOLUTE_ZERO = '−273°F';

/**
 * Not supported computed values in string enum.
 * @enum {string}
 */
const TemperatureInFahrenheit = {
  MIN_POSSIBLE: ABSOLUTE_ZERO,
  ZERO_FAHRENHEIT: 0 + '°F',
  ONE_FAHRENHEIT: `${Values.FIRST_VALUE}°F`,
  TWO_FAHRENHEIT: Values.SECOND_VALUE + '°F',
  SOME_FAHRENHEIT: getTemperatureInFahrenheit() + '°F',
};
```

> Note: Although TypeScript supports a few more patterns for enum values (e.g `A: 'a'+'b'`, etc), the restriction of only allowing string literals and numbers for enum values is to aid migration to TypeScript. For complex values consider using a const object without `@enum`.

🔗 5.4 Classes

🔗 5.4.1 Constructors

Constructors are optional. Subclass constructors must call `super()` before setting any fields or otherwise accessing `this`. Interfaces should declare non-method properties in the constructor.

🔗 5.4.2 Fields

Define all of a concrete object's fields (i.e. all properties other than methods) in the constructor. Annotate fields that are never reassigned with `@const` (these need not be deeply immutable). Annotate non-public fields with the proper visibility annotation (`@private`, `@protected`, `@package`). `@private` fields' names may optionally end with an underscore. Fields must not be defined within a nested scope in the constructor nor on a concrete class's `prototype`.

Example:

```js
class Foo {
  constructor() {
    /** @private @const {!Bar} */
    this.bar_ = computeBar();

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

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the **TypeScript guide**.

> Tip: Properties should never be added to or removed from an instance after the constructor is finished, since it significantly hinders VMs' ability to optimize. If necessary, fields that are initialized later should be explicitly set to `undefined` in the constructor to prevent later shape changes. Adding `@struct` to an object will check that undeclared properties are not added/accessed. Classes have this added by default.

🔗5.4.3 Computed properties

Computed properties may only be used in classes when the property is a symbol. Dict-style properties (that is, quoted or computed non-symbol keys, as defined in 5.3.3 Do not mix quoted and unquoted keys) are not allowed. A `[Symbol.iterator]` method should be defined for any classes that are logically iterable. Beyond this, `Symbol` should be used sparingly.

> Tip: be careful of using any other built-in symbols (e.g., `Symbol.isConcatSpreadable`) as they are not polyfilled by the compiler and will therefore not work in older browsers.

🔗5.4.4 Static methods

Where it does not interfere with readability, prefer module-local functions over private static methods.

Code should not rely on dynamic dispatch of static methods, because it interferes with Closure Compiler optimizations. Static methods should only be called on the base class itself. Static methods should not be called on variables containing a dynamic instance that may be either the constructor or a subclass constructor (and must be defined with `@nocollapse` if this is done), and must not be called directly on a subclass that doesn't define the method itself. Do not access `this` in static methods.

Disallowed:

```
// Context for the examples below; by itself this code is allowed.
class Base {
  /** @nocollapse */ static foo() {}
}
class Sub extends Base {}

// discouraged: don't call static methods dynamically
function callFoo(cls) { cls.foo(); }

// Disallowed: don't call static methods on subclasses that don't define
Sub.foo();

// Disallowed: don't access this in static methods.
class Clazz {
  static foo() {
    return this.staticField;
```

🔗5.4.5 Old-style class declarations

While ES6 classes are preferred, there are cases where ES6 classes may not be feasible. For example:

1. If there exist or will exist subclasses, including frameworks that create subclasses, that cannot be immediately changed to use ES6 class syntax. If such a class were to use ES6 syntax, all downstream subclasses not using ES6 class syntax would need to be modified.

2. Frameworks that require a known `this` value before calling the superclass constructor, since constructors with ES6 super classes do not have access to the instance `this` value until the call to `super` returns.

In all other ways the style guide still applies to this code: `let`, `const`, default parameters, rest, and arrow functions should all be used when appropriate.

`goog.defineClass` allows for a class-like definition 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;
  },
});
```

Alternatively, while `goog.defineClass` should be preferred for all new code, more traditional syntax is also allowed.

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

```
goog.inherits(C, S);

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

Per-instance properties should be defined in the constructor after the call to the super class
constructor, if there is a super class. Methods should be defined on the prototype of the constructor.

Defining constructor prototype hierarchies correctly is harder than it first appears! For that reason, it
is best to use `goog.inherits` from [the Closure Library](#) .

5.4.6 Do not manipulate `prototype` s directly

The `class` keyword allows clearer and more readable class definitions than defining
`prototype` properties. Ordinary implementation code has no business manipulating these
objects, though they are still useful for defining classes as defined in [5.4.5 Old-style class
declarations](#). Mixins and modifying the prototypes of builtin objects are explicitly forbidden.

Exception: Framework code (such as Polymer, or Angular) may need to use `prototype` s, and
should not resort to even-worse workarounds to avoid doing so.

5.4.7 Getters and Setters

Do not use [JavaScript getter and setter properties](#). They are potentially surprising and difficult to
reason about, and have limited support in the compiler. Provide ordinary methods instead.

Exception: there are situations where defining a getter or setter is unavoidable (e.g. data binding
frameworks such as Angular and Polymer, or for compatibility with external APIs that cannot be
adjusted). In these cases only, getters and setters may be used *with caution*, provided they are
defined with the `get` and `set` shorthand method keywords or
`Object.defineProperties` (not `Object.defineProperty` , which interferes with
property renaming). Getters **must not** change observable state.

Disallowed:

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

5.4.8 Overriding toString

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](.).

> Tip: Beware, in particular, of calling other methods from toString, since exceptional conditions
> could lead to infinite loops.

🔗 5.4.9 Interfaces

Interfaces may be declared with `@interface` or `@record`. Interfaces declared with
`@record` can be explicitly (i.e. via `@implements`) or implicitly implemented by a class or
object literal.

All methods on an interface must be non-static and method bodies must be empty blocks. Fields
must be declared as uninitialized members in the class constructor.

Example:

```
/**
 * Something that can frobnicate.
 * @record
 */
class Frobnicator {
  constructor() {
    /** @type {number} The number of attempts before giving up. */
    this.attempts;
  }

  /**
   * Performs the frobnication according to the given strategy.
   * @param {!FrobnicationStrategy} strategy
   */
  frobnicate(strategy) {}
}
```

🔗 5.4.10 Abstract Classes

Use abstract classes when appropriate. Abstract classes and methods must be annotated with
`@abstract`. Do not use `goog.abstractMethod`. See [abstract classes and methods](.).

🔗 5.4.11 Do not create static container classes

Do not use container classes with only static methods or properties for the sake of namespacing.

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

```
Container.FOO = 1;
```

Instead, export individual constants and functions:

```
/** @return {number} */
exports.bar = () => {
  return 1;
}

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

🔗 5.4.12 Do not define nested namespaces

Do not define a nested type (e.g. class, typedef, enum, interface) on another module-local name.

```
// foo.js
goog.module('my.namespace');

class Foo {...}

Foo.Bar = class {...};

/** @enum {number} */
Foo.Baz = {...};

/** @typedef {{value: number}} */
Foo.Qux;

/** @interface */
Foo.Quuz = class {...}

exports.Foo = Foo;
```

These values should be top-level exports. Choose clear names for these values (e.g. FooConverter
for a Converter that could have been nested on Foo). However, when the module name is
redundant with part of the class name, consider omitting the redundancy: `foo.Foo` and
`foo.Converter` rather than `foo.Foo` and `foo.FooConverter`. Importers can add the
prefix when necessary for clarity (e.g.
`import {Converter as FooConverter} from './foo';`) but cannot easily remove the
redundancy when importing the entire module as a namespace.

```
// foo.js
goog.module('my.namespace');
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the **TypeScript guide.**

```
/** @enum {string} */
let FooBaz = {...};

/** @typedef {{value: number}} */
let FooQux;

/** @interface */
class FooQuuz {...};

export = {
  Foo,
  FooBar,
  FooBaz,
  FooQux,
  FooQuuz,
};
```

🔗5.5 Functions

🔗5.5.1 Top-level functions

Top-level functions may be defined directly on the `exports` object, or else declared locally and optionally exported. See 3.3.3 goog.module Exports for more on exports.

Examples:

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

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

exports = {processString};
```

🔗5.5.2 Nested functions and closures

Functions may contain nested function definitions. If it is useful to give the function a name, it should be assigned to a local `const`.

🔗5.5.3 Arrow functions

~~shorthand~~).

Prefer arrow functions over other `this` scoping approaches such as `f.bind(this)` , `goog.bind(f, this)` , and `const self = this` . Arrow functions are particularly useful for calling into callbacks as they permit explicitly specifying which parameters to pass to the callback whereas binding will blindly pass along all parameters.

The left-hand side of the arrow contains zero or more parameters. Parentheses around the parameters are optional if there is only a single non-destructured parameter. When parentheses are used, inline parameter types may be specified (see [7.8 Method and function comments](#)).

> Tip: Always using parentheses even for single-parameter arrow functions can avoid situations where adding parameters, but forgetting to add parentheses, may result in parseable code which no longer works as intended.

The right-hand side of the arrow contains the body of the function. By default the body is a block statement (zero or more statements surrounded by curly braces). The body may also be an implicitly returned single expression if either: the program logic requires returning a value, or the `void` operator precedes a single function or method call (using `void` ensures `undefined` is returned, prevents leaking values, and communicates intent). The single expression form is preferred if it improves 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 s
 * @return {number} The sum of the two parameters.
 */
const moduleLocalFunc = (numParam, strParam) => numParam + Number(strPar

// Uses the single expression syntax with `void` because the program log
// 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
    // not be returned and the expression is not a single function call.
    getNullableValue((/** ?number */ result) => {
      this.cachedValue_ = result == null ? 0 : result;
    });
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

Disallowed:

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

🔗 5.5.4 Generators

Generators enable a number of useful abstractions and may be used as needed.

When defining generator functions, attach the `*` to the `function` keyword when present, and separate it with a space from the name of the function. When using delegating yields, attach the `*` to the `yield` keyword.

Example:

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

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

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

🔗 5.5.5 Parameter and return types

Function parameters and return types should usually be documented with JSDoc annotations. See [7.8 Method and function comments](#) for more information.

5.5.5.1 Default parameters

Optional parameters are permitted using the equals operator in the parameter list. Optional parameters must include spaces on both sides of the equals operator, be named exactly like required parameters (i.e., not prefixed with `opt_`), use the `=` suffix in their JSDoc type, come

contrast to concrete functions, abstract and interface methods must omit default parameter values.

Example:

```
/**
 * @param {string} required This parameter is always needed.
 * @param {string=} optional This parameter can be omitted.
 * @param {!Node=} node Another optional parameter.
 */
function maybeDoSomething(required, optional = '', node = undefined) {}

/** @interface */
class MyInterface {
  /**
   * Interface and abstract methods must omit default parameter values.
   * @param {string=} optional
   */
  someMethod(optional) {}
}
```

Use default parameters sparingly. Prefer destructuring (as in 5.3.7 Destructuring) to create readable APIs when there are more than a small handful of optional parameters that do not have a natural order.

> Note: Unlike Python's default parameters, it is okay to use initializers that return new mutable objects (such as `{}` or `[]`) because the initializer is evaluated each time the default value is used, so a single object won't be shared across invocations.

> Tip: While arbitrary expressions including function calls may be used as initializers, these should be kept as simple as possible. Avoid initializers that expose shared mutable state, as that can easily introduce unintended coupling between function calls.

5.5.5.2 Rest parameters

Use a *rest* parameter instead of accessing `arguments` . Rest parameters are typed with a `...` prefix in their JSDoc. The rest parameter must be the last parameter in the list. 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
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

🔗5.5.6 Generics

Declare generic functions and methods when necessary with `@template TYPE` in the JSDoc
above the function or method definition.

🔗5.5.7 Spread operator

Function calls may use the spread operator (`...`). Prefer the spread operator 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:

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

🔗5.6 String literals

🔗5.6.1 Use single quotes

Ordinary string literals are delimited with single quotes (`'`), rather than double quotes (`"`).

> 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.

🔗5.6.2 Template literals

Use template literals (delimited with `` ` ``) over complex string concatenation, particularly if multiple
string literals are involved. Template literals may span multiple lines.

If a template literal spans multiple lines, it does not need to follow the indentation of the enclosing
block, though it may if the added whitespace does not 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}`;
}
```

ordinary or template string literals. Even though ES5 allows this, it can lead to tricky errors if any trailing whitespace comes after the slash, and is less obvious to readers.

Disallowed:

```
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.';
```

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.';
```

5.7 Number literals

Numbers may be specified in decimal, hex, octal, or binary. Use exactly `0x` , `0o` , and `0b` prefixes, with lowercase letters, for hex, octal, and binary, respectively. Never include a leading zero unless it is immediately followed by `x` , `o` , or `b` .

5.8 Control structures

5.8.1 For loops

With ES6, the language now has three different kinds of `for` loops. All may be used, though `for` - `of` loops should be preferred when possible.

`for` - `in` loops may only be used on dict-style objects (see [5.3.3 Do not mix quoted and unquoted keys](#)), and should not be used to iterate over an array. `Object.prototype.hasOwnProperty` should be used in `for` - `in` loops to exclude unwanted prototype properties. Prefer `for` - `of` and `Object.keys` over `for` - `in` when possible.

5.8.2 Exceptions

Exceptions are an important part of the language and should be used whenever exceptional cases occur. Always throw `Error` s or subclasses of `Error` : never throw string literals or other objects. Always use `new` when constructing an `Error` .

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

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

Prefer throwing exceptions over ad-hoc error-handling approaches (such as passing an error container reference type, or returning an object with an error property).

5.8.2.1 Empty catch blocks

It is very rarely correct to do nothing in response to a caught exception. When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

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

Disallowed:

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

> Tip: Unlike in some other languages, patterns like the above simply don't work since this will catch the error thrown by `fail` . Use `assertThrows()` instead.

🔗 5.8.3 Switch statements

Terminology Note: Inside the braces of a switch block are one or more statement groups. Each statement group consists of one or more switch labels (either `case FOO:` or `default:`), followed by one or more statements.

5.8.3.1 Fall-through: commented

Within a switch block, each statement group either terminates abruptly (with a `break` , `return` or `throw` n exception), or is marked with a comment to indicate that execution will or might continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically `// fall through`). This special comment is not required in the last statement group of the switch block.

Example:

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
```

```
      break;
    default:
      handleLargeNumber(input);
  }
```

5.8.3.2 The `default` case is present

Each switch statement includes a `default` statement group, even if it contains no code. The `default` statement group must be last.

5.9 this

Only use `this` in class constructors and methods, in arrow functions defined within class constructors and methods, or in functions that have an explicit `@this` declared in the immediately-enclosing function's JSDoc.

Never use `this` to refer to the global object, the context of an `eval`, the target of an event, or unnecessarily `call()` ed or `apply()` ed functions.

5.10 Equality Checks

Use identity operators (`===` / `!==`) except in the cases documented below.

5.10.1 Exceptions Where Coercion is Desirable

Catching both `null` and `undefined` values:

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

5.11 Disallowed features

5.11.1 with

Do not use the `with` keyword. It makes your code harder to understand and has been banned in strict mode since ES5.

5.11.2 Dynamic code evaluation

Do not use `eval` or the `Function(...string)` constructor (except for code loaders). These features are potentially dangerous and simply do not work in CSP environments.

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

above).

🔗5.11.4 Non-standard features

Do not use non-standard features. This includes old features that have been removed (e.g., `WeakMap.clear`), new features that are not yet standardized (e.g., the current TC39 working draft, proposals at any stage, or proposed but not-yet-complete web standards), or proprietary features that are only implemented in some browsers. Use only features defined in the current ECMA-262 or WHATWG standards. (Note that projects writing against specific APIs, such as Chrome extensions or Node.js, can obviously use those APIs). Non-standard language "extensions" (such as those provided by some external transpilers) are forbidden.

🔗5.11.5 Wrapper objects for primitive types

Never use `new` on the primitive object wrappers (`Boolean` , `Number` , `String` , `Symbol`), nor include them in type annotations.

Disallowed:

```
const /** Boolean */ x = new Boolean(false);
if (x) alert(typeof x);  // alerts 'object' – WAT?
```

The wrappers may be called as functions for coercing (which is preferred over using `+` or concatenating the empty string) or creating symbols.

Example:

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

🔗5.11.6 Modifying builtin objects

Never modify builtin types, either by adding methods to their constructors or to their prototypes. Avoid depending on libraries that do this. Note that the JSCompiler's runtime library will provide standards-compliant polyfills where possible; nothing else may modify builtin objects.

Do not add symbols to the global object unless absolutely necessary (e.g. required by a third-party API).

🔗5.11.7 Omitting `()` when invoking a constructor

Never invoke a constructor in a `new` statement without using parentheses `()` .

Disallowed:

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

Use instead:

```
new Foo();
```

Omitting parentheses can lead to subtle mistakes. These two lines are not equivalent:

```
new Foo().Bar();
new Foo.Bar();
```

🔗 6 Naming

🔗 6.1 Rules common to all identifiers

Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores and very rarely (when required by frameworks like Angular) dollar signs.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

```
errorCount          // No abbreviation.
dnsConnectionIndex  // Most people know what "DNS" stands for.
referrerUrl         // Ditto for "URL".
customerId          // "Id" is both ubiquitous and unlikely to be misund
```

Disallowed:

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

Exception: Variables that are in scope for 10 lines or fewer, including arguments that are *not* part of an exported API, *may* use short (e.g. single letter) variable names.

🔗 6.2 Rules by identifier type

🔗 6.2.1 Package names

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the [TypeScript guide](#).

Exception: The package name may conform to TypeScript's path-based pattern. This is typically all lower case with underscores where present in filenames.

6.2.2 Class names

Class, interface, record, and typedef names are written in `UpperCamelCase` . Unexported classes are simply locals: they are not marked `@private` .

Type names are typically nouns or noun phrases. For example, `Request` , `ImmutableView` , or `VisibilityMode` . Additionally, interface names may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

6.2.3 Method names

Method names are written in `lowerCamelCase` . Names for `@private` methods may optionally end with a trailing underscore.

Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop_` . Getter and setter methods for properties are never required, but if they are used they should be named `getFoo` (or optionally `isFoo` or `hasFoo` for booleans), or `setFoo(value)` for setters.

Underscores may also appear in JsUnit test method names to separate logical components of the name. One typical pattern is `test<MethodUnderTest>_<state>_<expectedOutcome>` , for example `testPop_emptyStack_throws` . There is no One Correct Way to name test methods.

6.2.4 Enum names

Enum names are written in `UpperCamelCase` , similar to classes, and should generally be singular nouns. Individual items within the enum are named in `CONSTANT_CASE` .

6.2.5 Constant names

Constant names use `CONSTANT_CASE` : all uppercase letters, with words separated by underscores. There is no reason for a constant to be named with a trailing underscore, since private static properties can be replaced by (implicitly private) module locals.

6.2.5.1 Definition of "constant"

Every constant is a `@const` static property or a module-local `const` declaration, but not all `@const` static properties and module-local `const` s are constants. Before choosing constant case, consider whether the field really feels like a *deeply immutable* constant. For example, if any of that instance's observable state can change, it is almost certainly not a constant. Merely intending to never mutate the object is generally not enough.

Examples:

```
/** @const */ exports.NAMES = goog.debug.freeze(['Ed', 'Ann']);
/** @enum */ exports.SomeEnum = { ENUM_CONSTANT: 'value' };

// 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 /** MyImmutableContainer<SomeMutableType> */ stillMutable =
    new MyImmutableContainer(mutableInner);

const {Foo} = goog.require('my.foo');  // mirrors imported name

const logger = log.getLogger('loggers.are.not.immutable');
```

Constants' names are typically nouns or noun phrases.

6.2.5.2 Local aliases

Local aliases should be used whenever they improve readability over fully-qualified names. Follow
the same rules as `goog.require` s ([3.6 goog.require and goog.requireType statements](#)),
maintaining the last part of the aliased name. Aliases may also be used within functions. Aliases
must be `const` .

Examples:

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

6.2.6 Non-constant field names

Non-constant field names (static or otherwise) are written in `lowerCamelCase` , with an optional
trailing underscore for private fields.

These names are typically nouns or noun phrases. For example, `computedValues` or
`index_` .

6.2.7 Parameter names

One-character parameter names should not be used in public methods.

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

6.2.8 Local variable names

Local variable names are written in `lowerCamelCase` , except for module-local (top-level)
constants, as described above. Constants in function scopes are still named in
`lowerCamelCase` . Note that `lowerCamelCase` is used even if the variable holds a
constructor.

6.2.9 Template parameter names

Template parameter names should be concise, single-word or single-letter identifiers, and must be
all-caps, such as `TYPE` or `THIS` .

6.2.10 Module-local names

Module-local names that are not exported are implicitly private. They are not marked `@private` .
This applies to classes, functions, variables, constants, enums, and other module-local identifiers.

6.3 Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case,
such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve
predictability, Google Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's
   algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically
   hyphens).
   a. Recommended: if any word already has a conventional camel case appearance in
      common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad
      words"). Note that a word such as "iOS" is not really in camel case per se; it defies any
      convention, so this recommendation does not apply.
3. Now lowercase everything (including acronyms), then uppercase only the first character of:
   a. … each word, to yield `UpperCamelCase` , or
   b. … each word except the first, to yield `lowerCamelCase`
4. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded.

Examples of `lowerCamelCase` :

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

"new customer ID"	`newCustomerId`	`newCustomerID`
"inner stopwatch"	`innerStopwatch`	`innerStopWatch`
"supports IPv6 on iOS?"	`supportsIpv6OnIos`	`supportsIPv6OnIOS`
"YouTube importer"	`youTubeImporter`	`youtubeImporter` *

*Acceptable, but not recommended.

For examples of `UpperCamelCase` , uppercase the first letter of each correct `lowerCamelCase` example.

> Note: Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names `checkNonempty` and `checkNonEmpty` are likewise both correct.

🔗 7 JSDoc

JSDoc is used on all classes, fields, and methods.

🔗 7.1 General form

The basic formatting of JSDoc blocks is as seen in this example:

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

or in this single-line example:

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

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

Many tools extract metadata from JSDoc comments to perform code validation and optimization. As such, these comments **must** be well-formed.

🔗 7.2 Markdown

formatting, so if you did this:

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

it would come out like this:

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

Instead, write a Markdown list:

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

⌘7.3 JSDoc tags

Google style allows a subset of JSDoc tags. See 9.1 JSDoc tag reference for the complete list. Most
tags must occupy their own line, with the tag at the beginning of the line.

Disallowed:

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

Simple tags that do not require any additional data (such as `@private` , `@const` , `@final` ,
`@export`) may be combined onto the same line, along with 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
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

```javascript
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];
  }
}
```

There is no hard rule for when to combine tags, or in which order, but be consistent.

For general information about annotating types in JavaScript see [Annotating JavaScript for the Closure Compiler](#) and [Types in the Closure Type System](#).

🔗7.4 Line wrapping

Line-wrapped block tags are indented four spaces. Wrapped description text may be lined up with the description on previous lines, but this horizontal alignment is discouraged.

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

Do not indent when wrapping a `@desc` or `@fileoverview` description.

🔗7.5 Top/file-level comments

A file may have a top-level file overview. A copyright notice, author information, and default [visibility level](#) are optional. File overviews are generally recommended whenever a file consists of more than a single class definition. The top level comment is designed to orient readers unfamiliar with the code to what is in this file. If present, it may provide a description of the file's contents and any dependencies or compatibility information. Wrapped lines are not indented.

Example:

```
+ about its dependencies.
 * @package
 */
```

⌘7.6 Class comments

Classes, interfaces and records must be documented with a description and any template
parameters, implemented interfaces, visibility, or other appropriate tags. The class description
should provide the reader with enough information to know how and when to use the class, as well
as any additional considerations necessary to correctly use the class. Textual descriptions may be
omitted on the constructor. When defining a class `@constructor` and `@extends` annotations
are not used with the `class` keyword unless it extends a generic class. When defining an
`@interface` or a `@record`, the `@extends` annotation is used when defining a subclass
and the `extends` keyword is never used.

```
/**
 * A fancier event target that does cool things.
 * @implements {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() {}
}
```

⌘7.7 Enum and typedef comments

All enums and typedefs must be documented with appropriate JSDoc tags (`@typedef` or
`@enum`) on the preceding line. Public enums and typedefs must also have a description. Individual
enum items may be documented with a JSDoc comment on the preceding line.

```
 * @typedef {.FruitType|.FruitTypeEnum}
 */
let CoolUnionType;


/**
 * Types of fruits.
 * @enum {string}
 */
const FruitTypeEnum = {
  /** This kind is very sour. */
  SOUR: 'sour',
  /** The less-sour kind. */
  SWEET: 'sweet',
};
```

Typedefs are useful for defining short record types, or aliases for unions, complex functions, or generic types. Typedefs should be avoided for record types with many fields, since they do not allow documenting individual fields, nor using templates or recursive references. For large record types, prefer `@record` .

7.8 Method and function comments

In methods and named functions, parameter and return types must be documented, even in the case of same-signature `@override` s. The `this` type should be documented when necessary. Return type may be omitted if the function has no non-empty `return` statements.

Method, parameter, and return descriptions (but not types) may be omitted if they are obvious from the rest of the method's JSDoc or from its signature.

Method descriptions begin with a verb phrase that describes what the method does. This phrase is not an imperative sentence, but instead is written in the third person, as if there is an implied "This method ..." before it.

If a method overrides a superclass method, it must include an `@override` annotation. For overridden methods, all `@param` and `@return` annotations must be specified explicitly even if no type from the superclass method is refined. This is to align with TypeScript.

```
/** 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
  */
```

If you only need to document the param and return types of a function, you may optionally use inline JSDocs in the function's signature. These inline JSDocs specify the return and param types without tags.

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

If you need descriptions or tags, use a single JSDoc comment above the method. For example, methods which return values need a `@return` tag.

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

```
// Illegal inline JSDocs.

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

/** No function description allowed inline here. */ function bar() {...}

function /** Function description is also illegal here. */ baz() {...}
```

In anonymous functions annotations are generally optional. If the automatic type inference is insufficient or explicit annotation improves readability, then annotate param and return types like this:

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

```
(/** .Array<String> */ items) => {
    doSomethingWith(items);
    return items[0];
});
```

For function type expressions, see 7.10.4 Function type expressions.

🔗7.9 Property comments

Property types must be documented. The description may be omitted for private properties, if name and type provide enough documentation for understanding the code.

Publicly exported constants are commented the same way 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;
```

🔗7.10 Type annotations

Type annotations are found on `@param` , `@return` , `@this` , and `@type` tags, and optionally on `@const` , `@export` , and any visibility tags. Type annotations attached to JSDoc tags must always be enclosed in braces.

🔗7.10.1 Nullability

The type system defines modifiers `!` and `?` for non-null and nullable, respectively. These modifiers must precede the type.

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the [TypeScript guide](#).

1. Type annotations for primitives (`string` , `number` , `boolean` , `symbol` , `undefined` , `null`) and literals (`{function(...): ...}` and `{{foo: string...}}`) are always non-nullable by default. Use the `?` modifier to make it nullable, but omit the redundant `!` .

2. Reference types (generally, anything in `UpperCamelCase` , including `some.namespace.ReferenceType`) refer to a class, enum, record, or typedef defined elsewhere. Since these types may or may not be nullable, it is impossible to tell from the name alone whether it is nullable or not. Always use explicit `?` and `!` modifiers for these types to prevent ambiguity at use sites.

Bad:

```
const /** MyObject */ myObject = null; // Non-primitive types must be an
const /** !number */ someNum = 5; // Primitives are non-nullable by defa
const /** number? */ someNullableNum = null; // ? should precede the typ
const /** !{foo: string, bar: number} */ record = ...; // Already non-nu
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 = ...;
```

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 = ...;
```

7.10.2 Type Casts

In cases where the compiler doesn't accurately infer the type of an expression, and the assertion functions in [goog.asserts](#) cannot remedy it, it is possible to tighten the type by adding a type annotation comment and enclosing the expression in parentheses. Note that the parentheses are required.

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

7.10.3 Template Parameter Types

Always specify template parameters. This way compiler can do a better job and it makes it easier for readers to understand what code does.

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](.)

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

Good:

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

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

Cases when template parameters should not be used:

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

🔗7.10.4 Function type expressions

Terminology Note: *function type expression* refers to a type annotation for function types with the keyword `function` in the annotation (see examples below).

Where the function definition is given, do not use a function type expression. Specify parameter and return types with `@param` and `@return`, or with inline annotations (see [7.8 Method and function comments](.)). This includes anonymous functions and functions defined and assigned to a const (where the function jsdoc appears above the whole assignment expression).

Function type expressions are needed, for example, inside `@typedef`, `@param` or `@return`. Use it also for variables or properties of function type, if they are not immediately initialized with the function definition.

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

When using a function type expression, always specify the return type explicitly. Otherwise the default return type is "unknown" (`?`), which leads to strange and unexpected behavior, and is rarely what is actually desired.

Bad - type error, but no warning given:

```
/** @param {function()} generateNumber */
function foo(generateNumber) {
  const /** number */ x = generateNumber();  // No compile-time type err
}
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the [**TypeScript guide**](#)**.**

Good:

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

🔗7.10.5 Whitespace

Within a type annotation, a single space or line break is required after each comma or colon. Additional line breaks may be inserted to improve readability or avoid exceeding the column limit. These breaks should be chosen and indented following the applicable guidelines (e.g. [4.5 Line-wrapping](#) and [4.2 Block indentation: +2 spaces](#)). No other whitespace is allowed in type annotations.

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}
 */
```

Bad:

```
// Only put a space after the colon
/** @type {function(string) : number} */
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

```
// No space in union types
/** @type {number | string} */
```

7.11 Visibility annotations

Visibility annotations (`@private` , `@package` , `@protected`) may be specified in a
`@fileoverview` block, or on any exported symbol or property. Do not specify visibility for local
variables, whether within a function or at the top level of a module. `@private` names may
optionally end with an underscore.

8 Policies

8.1 Issues unspecified by Google Style: Be Consistent!

For any style question that isn't settled definitively by this specification, prefer to do what the other
code in the same file is already doing. If that doesn't resolve the question, consider emulating the
other files in the same package.

8.2 Compiler warnings

8.2.1 Use a standard warning set

As far as possible projects should use `--warning_level=VERBOSE` .

8.2.2 How to handle a warning

Before doing anything, make sure you understand exactly what the warning is telling you. If you're
not positive why a warning is appearing, ask for help .

Once you understand the warning, attempt the following solutions in order:

1. **First, fix it or work around it.** Make a strong attempt to actually address the warning, or
   find another way to accomplish the task that avoids the situation entirely.
2. **Otherwise, determine if it's a false alarm.** If you are convinced that the warning is invalid
   and that the code is actually safe and correct, add a comment to convince the reader of this
   fact and apply the `@suppress` annotation.
3. **Otherwise, leave a TODO comment.** This is a **last resort**. If you do this, **do not suppress**
   the warning. The warning should be visible until it can be taken care of properly.

8.2.3 Suppress a warning at the narrowest reasonable scope

Warnings are suppressed at the narrowest reasonable scope, usually that of a single local variable
or very small method. Often a variable or method is extracted for that reason alone.

```
/** @suppress {uselessCode} unrecognized 'use asm' declaration */
function fn() {
  'use asm';
  return 0;
}
```

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

8.3 Deprecation

Mark deprecated methods, classes or interfaces with `@deprecated` annotations. A deprecation comment must include simple, clear directions for people to fix their call sites.

8.4 Code not in Google Style

You will occasionally encounter files in your codebase that are not in proper Google Style. These may have come from an acquisition, or may have been written before Google Style took a position on some issue, or may be in non-Google Style for any other reason.

8.4.1 Reformatting existing code

When updating the style of existing code, follow these guidelines.

1. It is not required to change all existing code to meet current style guidelines. Reformatting existing code is a trade-off between code churn and consistency. Style rules evolve over time and these kinds of tweaks to maintain compliance would create unnecessary churn. However, if significant changes are being made to a file it is expected that the file will be in Google Style.
2. Be careful not to allow opportunistic style fixes to muddle the focus of a CL. If you find yourself making a lot of style changes that aren't critical to the central focus of a CL, promote those changes to a separate CL.

8.4.2 Newly added code: use Google Style

Brand new files use Google Style, regardless of the style choices of other files in the same package.

When adding new code to a file that is not in Google Style, reformatting the existing code first is recommended, subject to the advice in 8.4.1 Reformatting existing code.

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

8.5 Local style rules

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

cleanup changes due to violating any additional rules. Beware of excessive rules which serve no purpose. The style guide does not seek to define style in every possible scenario and neither should you.

8.6 Generated code: mostly exempt

Source code generated by the build process is not required to be in Google Style. However, any generated identifiers that will be referenced from hand-written source code must follow the naming requirements. As a special exception, such identifiers are allowed to contain underscores, which may help to avoid conflicts with hand-written identifiers.

9 Appendices

9.1 JSDoc tag reference

JSDoc serves multiple purposes in JavaScript. In addition to being used to generate documentation it is also used to control tooling. The best known are the Closure Compiler type annotations.

9.1.1 Type annotations and other Closure Compiler annotations

Documentation for JSDoc used by the Closure Compiler is described in Annotating JavaScript for the Closure Compiler and Types in the Closure Type System.

9.1.2 Documentation annotations

In addition to the JSDoc described in Annotating JavaScript for the Closure Compiler the following tags are common and well supported by various documentation generation tools (such as JsDossier) for purely documentation purposes.

9.1.2.1 `@author` or `@owner` - *Not recommended.*

Not recommended.

Syntax: `@author username@google.com (First Last)`

```
/**
 * @fileoverview Utilities for handling textareas.
 * @author kuth@google.com (Uthur Pendragon)
 */
```

Documents the author of a file or the owner of a test, generally only used in the `@fileoverview` comment. The `@owner` tag is used by the unit test dashboard to determine who owns the test results.

9.1.2.2 `@bug`

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

/**
 * @bug 1234568
 * @bug 1234569
 */
function testTwoBugs() {
  // …
}
```

Indicates what bugs the given test function regression tests.

Multiple bugs should each have their own `@bug` line, to make searching for regression tests as easy as possible.

9.1.2.3 `@code` *- Deprecated. Do not use.*

Deprecated. Do not use. Use Markdown backticks instead.

Syntax: `{@code ...}`

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

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

Indicates that a term in a JSDoc description is code so it may be correctly formatted in generated documentation.

9.1.2.4 `@desc`

Syntax: `@desc Message description`

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

9.1.2.5 `@link`

Syntax: `{@link ...}`

This tag is used to generate cross-reference links within generated documentation.

```
/** Processes pending {@link BatchItem} instances. */
```

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](#).

Historical note: @link tags have also been used to create external links in generated
documentation. For external links, always use Markdown's link syntax instead:

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

9.1.2.6 `@see`

Syntax: `@see Link`

```
/**
 * Adds a single item, recklessly.
 * @see #addSafely
 * @see goog.Collect
 * @see goog.RecklessAdder#add
 */
```

Reference a lookup to another class function or method.

9.1.2.7 `@supported`

Syntax: `@supported Description`

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

Used in a fileoverview to indicate what browsers are supported by the file.

You may also see other types of JSDoc annotations in third-party code. These annotations appear
in the [JSDoc Toolkit Tag Reference](#) but are not considered part of valid Google style.

🔗9.1.3 Framework specific annotations

The following annotations are specific to a particular framework.

9.1.3.1 `@ngInject` **for Angular 1**

9.1.3.2 `@polymerBehavior` **for Polymer**

[https://github.com/google/closure-compiler/wiki/Polymer-Pass](#)

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the TypeScript guide.

9.1.4.1 `@expose` - *Deprecated. Do not use.*

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

9.1.4.2 `@inheritDoc` - *Deprecated. Do not use.*

Deprecated. Do not use. Use `@override` **instead.**

9.2 Commonly misunderstood style rules

Here is a collection of lesser-known or commonly misunderstood facts about Google Style for JavaScript. (The following are true statements; this is not a list of "myths.")

- Neither a copyright statement nor `@author` credit is required in a source file. (Neither is explicitly recommended, either.)
- There is no "hard and fast" rule governing how to order the members of a class (5.4 Classes).
- Empty blocks can usually be represented concisely as `{}`, as detailed in (4.1.3 Empty blocks: may be concise).
- The prime directive of line-wrapping is: prefer to break at a higher syntactic level (4.5.1 Where to break).
- Non-ASCII characters are allowed in string literals, comments and JSDoc, and in fact are recommended when they make the code easier to read than the equivalent Unicode escape would (2.3.3 Non-ASCII characters).

9.3 Style-related tools

The following tools exist to support various aspects of Google Style.

9.3.1 Closure Compiler

This program performs type checking and other checks, optimizations and other transformations (such as lowering code to ECMAScript 5).

9.3.2 `clang-format`

This program reformats JavaScript source code into Google Style, and also follows a number of non-required but frequently readability-enhancing formatting practices. The output produced by `clang-format` is compliant with the style guide.

`clang-format` is not required. Authors are allowed to change its output, and reviewers are allowed to ask for such changes; disputes are worked out in the usual way. However, subtrees may choose to opt in to such enforcement locally.

9.3.3 Closure compiler linter

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the [TypeScript guide](#).

9.3.4 Conformance framework

The JS Conformance Framework is a tool that is part of the Closure Compiler that provides developers a simple means to specify a set of additional checks to be run against their code base above the standard checks. Conformance checks can, for example, forbid access to a certain property, or calls to a certain function, or missing type information (unknowns).

These rules are commonly used to enforce critical restrictions (such as defining globals, which could break the codebase) and 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](#).

9.4 Exceptions for legacy platforms

9.4.1 Overview

This section describes exceptions and additional rules to be followed when modern ECMAScript syntax is not available to the code authors. Exceptions to the recommended style are required when modern ECMAScript syntax is not possible and are outlined here:

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

9.4.2 Use `var`

9.4.2.1 `var` declarations are NOT block-scoped

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

```
for (var i = 0; i < 3; ++i) {
  var iteration = i;
  setTimeout(function() { console.log(iteration); }, i*1000);
}


// logs 2, 2, 2 -- NOT 0, 1, 2
// because `iteration` is function-scoped, not local to the loop.
```

9.4.2.2 Declare variables as close as possible to first use

Even though `var` declarations are scoped to the beginning of the enclosing function, `var` declarations should be as close as possible to their first use, for readability purposes. However, do not put a `var` declaration inside a block if that variable is referenced outside the block. For example:

```
  for (var x in y) {
    // "count" could be declared here, but don't do that.
    count++;
  }
  console.log(count + ' items in y');
}
```

9.4.2.3 Use @const for constants variables

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

9.4.3 Do not use block scoped functions declarations

Do **not** do this:

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

While most JavaScript VMs implemented before ECMAScript 6 support function declarations within
blocks it was not standardized. Implementations were inconsistent with each other and with the
now-standard ECMAScript behavior for block scoped function declaration. The ECMAScript 5
standard and prior only allow for function declarations in the root statement list of a script or function
and explicitly ban them in block scopes in strict mode.

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

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

9.4.4 Dependency management with `goog.provide` / `goog.require`

9.4.4.1 Summary

WARNING: `goog.provide` **dependency management is deprecated.** All new files, even in
projects using `goog.provide` for older files, should use `goog.module` . The following rules
are for pre-existing `goog.provide` files only.

- Place all `goog.provide` s first, `goog.require` s second. Separate provides from
  requires with an empty line.
- Sort the entries alphabetically (uppercase first).
- Don't wrap `goog.provide` and `goog.require` statements. Exceed 80 columns if
  necessary.
- Only provide top-level symbols.

Please note: This guide is no longer being updated. Google recommends migrating to TypeScript, and following the
[TypeScript guide](.).

Similar to import statements in other languages, `goog.provide` and `goog.require`
statements should be written in a single line, even if they exceed the 80 column line length limit.

The lines should be sorted alphabetically, with uppercase letters coming first:

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

goog.require('an.extremelyLongNamespace.thatSomeoneThought.wouldBeNice.a
goog.require('goog.dom');
goog.require('goog.dom.TagName');
goog.require('goog.dom.classes');
goog.require('goog.dominoes');
```

All members defined on a class should be in the same file. Only top-level classes should be
provided in a file that contains multiple members defined on the same class (e.g. enums, inner
classes, etc).

Do this:

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

Not this:

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

Members on namespaces may also be provided:

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

9.4.4.2 Aliasing with `goog.scope`

WARNING: `goog.scope` **is deprecated.** New files should not use `goog.scope` even in
projects with existing `goog.scope` usage.

`goog.scope` may be used to shorten references to namespaced symbols in code using
`goog.provide` / `goog.require` dependency management.

line and follow any `goog.provide` statements, `goog.require` statements, or top-level comments. The invocation must be closed on the last line in the file. Append `// goog.scope` to the closing statement of the scope. Separate the comment from the semicolon by two spaces.

Similar to C++ namespaces, do not indent under `goog.scope` declarations. Instead, continue from the 0 column.

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

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

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

Names must be the same as the last property of the global that they are aliasing.

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

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

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

// Alias new types after the constructor declaration.
my.module.SomeType = function() { ... };
var SomeType = my.module.SomeType;

// Declare methods on the prototype as usual:
SomeType.prototype.findButton = function() {
  // Button as aliased above.
  this.button = new Button(dom.getElement('my-button'));
};
...
});  // goog.scope
```

9.4.4.3 `goog.forwardDeclare`

Prefer to use `goog.requireType` instead of `goog.forwardDeclare` to break circular dependencies between files in the same library. Unlike `goog.require`, a `goog.requireType` statement is allowed to import a namespace before it is defined.

`goog.forwardDeclare` statements must follow the same style rules as `goog.require` and `goog.requireType`. The entire block of `goog.forwardDeclare`, `goog.require` and

library boundaries. This pattern however is poorly supported by build tools and should not be used.
Code should be organized to avoid circular dependencies across libraries (by splitting/merging
libraries).

9.4.4.4 `goog.module.get(name)`

If a `goog.provide` file depends on a `goog.module` file, the `goog.provide` file can not
normally refer to the module's exports via a global name. Instead, in addition to
`goog.require()` ing the module, the `goog.provide` file must fetch the module's export
object by calling `goog.module.get('module.name')` .

> Note: Only calling `goog.module.get('module.name')` does not create a build-time
> dependency of your code on the module. The `goog.require` is needed for the build
> dependency.

9.4.4.5 `goog.module.declareLegacyNamespace()`

WARNING: `goog.module.declareLegacyNamespace` **is only for transitional use.**

`goog.module.declareLegacyNamespace` is only for use while migrating a JavaScript file and
its consumers from `goog.provide` to `goog.module` . Update consumers of your
`goog.module` to use `goog.module` themselves. Remove calls to
`goog.module.declareLegacyNamespace` whenever possible.

If you can't update consumers of a legacy namespace from `goog.provide` to `goog.module`
soon, please wrap the contents of your file in a call to `goog.scope` , use `goog.module.get`
to import the legacy namespace--and then delete the call to
`goog.module.declareLegacyNamespace` in your `goog.module` .

Calling `goog.module.declareLegacyNamespace()` inside a `goog.module(name)` will
declare the module's namespace as a global name just like a `goog.provide()` call does. This
allows a non `goog.module` namespace to access the module's exports without calling
`goog.module.get(name)` .