1. Comments

a. Block Style Comments

i. 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. */
```

2. Language features

a. **One variable per declaration**

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

b. **Declared when needed, initialized as soon as possible**

i. 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.'

c. **5.10 Equality Checks**

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

ii. **Exceptions Where Coercion is Desirable (This is a rule)**

1. Catching both `null` and `undefined` values:

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

d. **With**

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

3. Statements

a. Braces are used around all statements, even single statements, when they are part of a control structure.

b. A return statement with a value should not use parentheses unless they make the return value more obvious in some way

c. A for statement should have the following form:
for (initialization. condition. update) {
statements.
}
d. A while statement should have the following form:
while (condition) {
statements.
}
e. A do-while statement should have the following form:
do {
statements.
} while (condition).

4. Whitespace

a. Blank lines improve readability by setting off sections of code that are logically related. One blank line should always be used in the following circumstances:
• Between functions
• Between the local variables in a function and its first statement
• Before a block or single-line comment
• Between logical sections inside a function to improve readability
b. The expressions in a for statement should be separated by blank spaces.
Example: for (expr1. expr2. expr3)

5. Naming

a. Rules common to all identifiers

i. 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.
ii. 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.
iii. `errorCount // No abbreviation.`
iv. `dnsConnectionIndex // Most people know what "DNS" stands for.`
v. `referrerUrl // Ditto for "URL".`

vi. `customerId // "Id" is both ubiquitous and unlikely to be misunderstood.`

vii. Disallowed:

viii. `n // Meaningless.`

ix. `nErr // Ambiguous abbreviation.`

x. `nCompConns // Ambiguous abbreviation.`

xi. `wgcConnections // Only your group knows what this stands for.`

xii. `pcReader // Lots of things can be abbreviated "pc".`

xiii. `cstmrId // Deletes internal letters.`

xiv. `kSecondsPerDay // Do not use Hungarian notation.`

b. **Package names**

i. Package names are all `lowerCamelCase`. For example, `my.exampleCode.deepSpace`, but not `my.examplecode.deepspace` or `my.example_code.deep_space`.

c. **Class names**

i. Class, interface, record, and typedef names are written in `UpperCamelCase`. Unexported classes are simply locals: they are not marked `@private` and therefore are not named with a trailing underscore.

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

d. **Method names**

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

ii. 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.

iii. 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.

e. **Enum names**

i. 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`.

f. **Non-constant field names**

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

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

g. **Parameter names**

i. Parameter names are written in `lowerCamelCase`. Note that this applies even if the parameter expects a constructor.
ii. One-character parameter names should not be used in public methods.
iii. **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).

h. **Local variable names**

i. 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. # JSDoc

a. ## Top/file-level comments

i. **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.**

ii. **Example:**

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

7. Policies

a. ## 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.

b. **Local style rules**

   i. **Teams and projects may adopt additional style rules beyond those in this document, but must accept that cleanup changes may not abide by these additional rules, and must not block such 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.**

c. **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. General

a. Avoid global variables
b. Always put braces on the right of block definitions
c. Never leave dangling commas in lists and object definitions
d. Never use "eval()"
e. Never leave debugger statements in code.
f. Never use hyphens in variable names
g. Use standard logging with useful details
h. JavaScript files should be labeled as .js