Google TypeScript Style Guide

Table of Contents

This guide is based on the internal Google TypeScript style guide, but it has been slightly adjusted to remove Google-internal sections. Google's internal environment has different constraints on TypeScript than you might find outside of Google. The advice here is specifically useful for people authoring code they intend to import into Google, but otherwise may not apply in your external environment.

There is no automatic deployment process for this version as it's pushed on-demand by volunteers.

1 Introduction

1.1 Terminology notes

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*.

1.2 Guide notes

All examples given are **non-normative** and serve only to illustrate the normative language of the style guide. 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 encoding: UTF-8

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

2.1.1 Whitespace characters

Aside from the line terminator sequence, the ASCII horizontal space character (0x20) is the only whitespace character that appears anywhere in a source file. This implies that all other whitespace characters in string literals are escaped.

2.1.2 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.

2.1.3 Non-ASCII characters

For the remaining non-ASCII characters, use the actual Unicode character (e.g. `∞`). For non-printable characters, the equivalent hex or Unicode escapes (e.g. `\u221e`) can be used along with an explanatory comment.

```
// Perfectly clear, even without a comment.
const units = 'µs';

// Use escapes for non-printable characters.
const output = '\ufeff' + content;  // byte order mark
```

```
// Hard to read and prone to mistakes, even with the comment.
const units = '\u03bcs'; // Greek letter mu, 's'

// The reader has no idea what this is.
const output = '\ufeff' + content;
```

3 Source file structure

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

1. Copyright information, if present
2. JSDoc with `@fileoverview`, if present
3. Imports, if present
4. The file's implementation

Exactly one blank line separates each section that is present.

3.1 Copyright information

If license or copyright information is necessary in a file, add it in a JSDoc at the top of the file.

3.2 `@fileoverview` JSDoc

A file may have a top-level `@fileoverview` JSDoc. If present, it may provide a description of the file's content, its uses, or information about its dependencies. Wrapped lines are not indented.

Example:

```
/**
 * @fileoverview Description of file. Lorem ipsum dolor sit amet, consec
 * adipiscing elit, sed do eiusmod tempor incididunt.
 */
```

3.3 Imports

There are four variants of import statements in ES6 and TypeScript:

Import type	Example	Use for
module[module_import]	`import * as foo from '...';`	TypeScript imports
named[destructuring_import]	`import {SomeThing} from '...';`	TypeScript imports
default	`import SomeThing from '...';`	Only for other external code that requires them

Import type	Example	Use for
side-effect	`import '...';`	Only to import libraries for their side-effects on load (such as custom elements)

```
// Good: choose between two options as appropriate (see below).
import * as ng from '@angular/core';
import {Foo} from './foo';

// Only when needed: default imports.
import Button from 'Button';

// Sometimes needed to import libraries for their side effects:
import 'jasmine';
import '@polymer/paper-button';
```

3.3.1 Import paths

TypeScript code *must* use paths to import other TypeScript code. Paths *may* be relative, i.e. starting with `.` or `..`, or rooted at the base directory, e.g. `root/path/to/file`.

Code *should* use relative imports (`./foo`) rather than absolute imports `path/to/foo` when referring to files within the same (logical) project as this allows to move the project around without introducing changes in these imports.

Consider limiting the number of parent steps (`../../../`) as those can make module and path structures hard to understand.

```
import {Symbol1} from 'path/from/root';
import {Symbol2} from '../parent/file';
import {Symbol3} from './sibling';
```

3.3.2 Namespace versus named imports

Both namespace and named imports can be used.

Prefer named imports for symbols used frequently in a file or for symbols that have clear names, for example Jasmine's `describe` and `it`. Named imports can be aliased to clearer names as needed with `as`.

Prefer namespace imports when using many different symbols from large APIs. A namespace import, despite using the `*` character, is not comparable to a “wildcard” import as seen in other languages. Instead, namespace imports give a name to all the exports of a module, and each exported symbol from the module becomes a property on the module name. Namespace imports can aid readability for exported symbols that have common names like `Model` or `Controller` without the need to declare aliases.

```
// Bad: overlong import statement of needlessly namespaced names.
import {Item as TableviewItem, Header as TableviewHeader, Row as Tablevi
  Model as TableviewModel, Renderer as TableviewRenderer} from './tablev

let item: TableviewItem|undefined;
```

```
// Better: use the module for namespacing.
import * as tableview from './tableview';

let item: tableview.Item|undefined;
```

```
import * as testing from './testing';

// Bad: The module name does not improve readability.
testing.describe('foo', () => {
  testing.it('bar', () => {
    testing.expect(null).toBeNull();
    testing.expect(undefined).toBeUndefined();
  });
});
```

```
// Better: give local names for these common functions.
import {describe, it, expect} from './testing';

describe('foo', () => {
  it('bar', () => {
    expect(null).toBeNull();
    expect(undefined).toBeUndefined();
  });
});
```

Special case: Apps JSPB protos

Apps JSPB protos must use named imports, even when it leads to long import lines.

This rule exists to aid in build performance and dead code elimination since often `.proto` files contain many `message` s that are not all needed together. By leveraging destructured imports the build system can create finer grained dependencies on Apps JSPB messages while preserving the ergonomics of path based imports.

```
// Good: import the exact set of symbols you need from the proto file.
import {Foo, Bar} from './foo.proto';

function copyFooBar(foo: Foo, bar: Bar) {...}
```

3.3.3 Renaming imports

Code *should* fix name collisions by using a namespace import or renaming the exports themselves. Code *may* rename imports (`import {SomeThing as SomeOtherThing}`) if needed.

Three examples where renaming can be helpful:

1. If it's necessary to avoid collisions with other imported symbols.
2. If the imported symbol name is generated.
3. If importing symbols whose names are unclear by themselves, renaming can improve code clarity. For example, when using RxJS the `from` function might be more readable when renamed to `observableFrom` .

3.4 Exports

Use named exports in all code:

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

Do not use default exports. This ensures that all imports follow a uniform pattern.

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

Why?

Default exports provide no canonical name, which makes central maintenance difficult with relatively little benefit to code owners, including potentially decreased readability:

```
import Foo from './bar';  // Legal.
import Bar from './bar';  // Also legal.
```

Named exports have the benefit of erroring when import statements try to import something that hasn't been declared. In `foo.ts` :

```
const foo = 'blah';
export default foo;
```

And in `bar.ts` :

```
import {fizz} from './foo';
```

Results in `error TS2614: Module '"./foo"' has no exported member 'fizz'.` While `bar.ts` :

```
import fizz from './foo';
```

Results in `fizz === foo`, which is probably unexpected and difficult to debug.

Additionally, default exports encourage people to put everything into one big object to namespace it all together:

```
export default class Foo {
  static SOME_CONSTANT = ...
  static someHelpfulFunction() { ... }
  ...
}
```

With the above pattern, we have file scope, which can be used as a namespace. We also have a perhaps needless second scope (the class `Foo`) that can be ambiguously used as both a type and a value in other files.

Instead, prefer use of file scope for namespacing, as well as named exports:

```
export const SOME_CONSTANT = ...
export function someHelpfulFunction()
export class Foo {
  // only class stuff here
}
```

3.4.1 Export visibility

TypeScript does not support restricting the visibility for exported symbols. Only export symbols that are used outside of the module. Generally minimize the exported API surface of modules.

3.4.2 Mutable exports

Regardless of technical support, mutable exports can create hard to understand and debug code, in particular with re-exports across multiple modules. One way to paraphrase this style point is that `export let` is not allowed.

```
export let foo = 3;
// In pure ES6, foo is mutable and importers will observe the value chan
// In TS, if foo is re-exported by a second file, importers will not see
window.setTimeout(() => {
  foo = 4;
}, 1000 /* ms */);
```

If one needs to support externally accessible and mutable bindings, they *should* instead use explicit getter functions.

```
let foo = 3;
window.setTimeout(() => {
  foo = 4;
```

```
}, 1000 /* ms */);
// Use an explicit getter to access the mutable export.
export function getFoo() { return foo; };
```

For the common pattern of conditionally exporting either of two values, first do the conditional check, then the export. Make sure that all exports are final after the module's body has executed.

```
function pickApi() {
  if (useOtherApi()) return OtherApi;
  return RegularApi;
}
export const SomeApi = pickApi();
```

3.4.3 Container classes

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

```
export class Container {
  static FOO = 1;
  static bar() { return 1; }
}
```

Instead, export individual constants and functions:

```
export const FOO = 1;
export function bar() { return 1; }
```

3.5 Import and export type

3.5.1 Import type

You may use `import type {...}` when you use the imported symbol only as a type. Use regular imports for values:

```
import type {Foo} from './foo';
import {Bar} from './foo';

import {type Foo, Bar} from './foo';
```

Why?

The TypeScript compiler automatically handles the distinction and does not insert runtime loads for type references. So why annotate type imports?

The TypeScript compiler can run in 2 modes:

- In development mode, we typically want quick iteration loops. The compiler transpiles to JavaScript without full type information. This is much faster, but requires `import type` in

certain cases.

- In production mode, we want correctness. The compiler type checks everything and ensures `import type` is used correctly.

> Note: If you need to force a runtime load for side effects, use `import '...';` . See

3.5.2 Export type

Use `export type` when re-exporting a type, e.g.:

```
export type {AnInterface} from './foo';
```

Why?

`export type` is useful to allow type re-exports in file-by-file transpilation. See `isolatedModules` docs.

`export type` might also seem useful to avoid ever exporting a value symbol for an API. However it does not give guarantees, either: downstream code might still import an API through a different path. A better way to split & guarantee type vs value usages of an API is to actually split the symbols into e.g. `UserService` and `AjaxUserService` . This is less error prone and also better communicates intent.

3.5.3 Use modules not namespaces

TypeScript supports two methods to organize code: *namespaces* and *modules*, but namespaces are disallowed. That is, your code *must* refer to code in other files using imports and exports of the form `import {foo} from 'bar';`

Your code *must not* use the `namespace Foo { ... }` construct. `namespace` s *may* only be used when required to interface with external, third party code. To semantically namespace your code, use separate files.

Code *must not* use `require` (as in `import x = require('...');`) for imports. Use ES6 module syntax.

```
// Bad: do not use namespaces:
namespace Rocket {
  function launch() { ... }
}

// Bad: do not use <reference>
/// <reference path="..."/>

// Bad: do not use require()
import x = require('mydep');
```

NB: TypeScript `namespace` s used to be called internal modules and used to use the `module` keyword in the form `module Foo { ... }` . Don't use that either. Always use ES6 imports.

4 Language features

This section delineates which features may or may not be used, and any additional constraints on their use.

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

4.1 Local variable declarations

4.1.1 Use const and let

Always use `const` or `let` to declare variables. Use `const` by default, unless a variable needs to be reassigned. Never use `var` .

```
const foo = otherValue;  // Use if "foo" never changes.
let bar = someValue;     // Use if "bar" is ever assigned into later on.
```

`const` and `let` are block scoped, like variables in most other languages. `var` in JavaScript is function scoped, which can cause difficult to understand bugs. Don't use it.

```
var foo = someValue;     // Don't use - var scoping is complex and cause
```

Variables *must not* be used before their declaration.

4.1.2 One variable per declaration

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

4.2 Array literals

4.2.1 Do not use the `Array` constructor

Do not use the `Array()` constructor, with or without `new` . It has confusing and contradictory usage:

```
const a = new Array(2); // [undefined, undefined]
const b = new Array(2, 3); // [2, 3];
```

Instead, always use bracket notation to initialize arrays, or `from` to initialize an `Array` with a certain size:

```
const a = [2];
const b = [2, 3];

// Equivalent to Array(2):
const c = [];
c.length = 2;

// [0, 0, 0, 0, 0]
Array.from<number>({length: 5}).fill(0);
```

4.2.2 Do not define properties on arrays

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

4.2.3 Using spread syntax

Using spread syntax `[...foo];` is a convenient shorthand for shallow-copying or concatenating iterables.

```
const foo = [
  1,
];

const foo2 = [
  ...foo,
  6,
  7,
];

const foo3 = [
  5,
  ...foo,
];

foo2[1] === 6;
foo3[1] === 1;
```

When using spread syntax, the value being spread *must* match what is being created. When creating an array, only spread iterables. Primitives (including `null` and `undefined`) *must not* be spread.

```
const foo = [7];
const bar = [5, ...(shouldUseFoo && foo)]; // might be undefined

// Creates {0: 'a', 1: 'b', 2: 'c'} but has no length
```

```
const fooStrings = ['a', 'b', 'c'];
const ids = {...fooStrings};
```

```
const foo = shouldUseFoo ? [7] : [];
const bar = [5, ...foo];
const fooStrings = ['a', 'b', 'c'];
const ids = [...fooStrings, 'd', 'e'];
```

4.2.4 Array destructuring

Array literals may be used on the left-hand side of an assignment to perform destructuring (such as when unpacking multiple values from a single array or iterable). A final "rest" element may be included (with no space between the `...` and the variable name). 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. Always specify `[]` as the default value if a destructured array parameter is optional, and provide default values on the left hand side:

```
function destructured([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.

4.3 Object literals

4.3.1 Do not use the `Object` constructor

The `Object` constructor is disallowed. Use an object literal (`{}` or `{a: 0, b: 1, c: 2}`) instead.

4.3.2 Iterating objects

Iterating objects with `for (... in ...)` is error prone. It will include enumerable properties from the prototype chain.

Do not use unfiltered `for (... in ...)` statements:

```
for (const x in someObj) {
  // x could come from some parent prototype!
}
```

Either filter values explicitly with an `if` statement, or use `for (... of Object.keys(...))`.

```
for (const x in someObj) {
  if (!someObj.hasOwnProperty(x)) continue;
  // now x was definitely defined on someObj
}
for (const x of Object.keys(someObj)) { // note: for _of_!
  // now x was definitely defined on someObj
}
for (const [key, value] of Object.entries(someObj)) { // note: for _of_!
  // now key was definitely defined on someObj
}
```

4.3.3 Using spread syntax

Using spread syntax `{...bar}` is a convenient shorthand for creating a shallow copy of an object. When using spread syntax in object initialization, later values replace earlier values at the same key.

```
const foo = {
  num: 1,
};

const foo2 = {
  ...foo,
  num: 5,
};

const foo3 = {
  num: 5,
  ...foo,
}

foo2.num === 5;
foo3.num === 1;
```

When using spread syntax, the value being spread *must* match what is being created. That is, when creating an object, only objects may be spread; arrays and primitives (including `null` and `undefined`) *must not* be spread. Avoid spreading objects that have prototypes other than the Object prototype (e.g. class definitions, class instances, functions) as the behavior is unintuitive (only enumerable non-prototype properties are shallow-copied).

```
const foo = {num: 7};
const bar = {num: 5, ...(shouldUseFoo && foo)}; // might be undefined

// Creates {0: 'a', 1: 'b', 2: 'c'} but has no length
const fooStrings = ['a', 'b', 'c'];
const ids = {...fooStrings};
```

```
const foo = shouldUseFoo ? {num: 7} : {};
const bar = {num: 5, ...foo};
```

4.3.4 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]`).

4.3.5 Object 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 `{}`.

Example:

```
interface Options {
  /** The number of times to do something. */
  num?: number;

  /** A string to do stuff to. */
  str?: string;
}

function destructured({num, str = 'default'}: Options = {}) {}
```

Disallowed:

```
function nestedTooDeeply({x: {num, str}}: {x: Options}) {}
function nontrivialDefault({num, str}: Options = {num: 42, str: 'default
```

4.4 Classes

4.4.1 Class declarations

Class declarations *must not* be terminated with semicolons:

```
class Foo {
}
```

```
class Foo {
}; // Unnecessary semicolon
```

In contrast, statements that contain class expressions *must* be terminated with a semicolon:

```
export const Baz = class extends Bar {
  method(): number {
    return this.x;
  }
}; // Semicolon here as this is a statement, not a declaration
```

```
exports const Baz = class extends Bar {
  method(): number {
    return this.x;
  }
}
```

It is neither encouraged nor discouraged to have blank lines separating class declaration braces from other class content:

```
// No spaces around braces - fine.
class Baz {
  method(): number {
    return this.x;
  }
}

// A single space around both braces - also fine.
class Foo {

  method(): number {
    return this.x;
  }

}
```

4.4.2 Class method declarations

Class method declarations *must not* use a semicolon to separate individual method declarations:

```
class Foo {
  doThing() {
```

```
    console.log("A");
  }
}
```

```
class Foo {
  doThing() {
    console.log("A");
  }; // <-- unnecessary
}
```

Method declarations should be separated from surrounding code by a single blank line:

```
class Foo {
  doThing() {
    console.log("A");
  }

  getOtherThing(): number {
    return 4;
  }
}
```

```
class Foo {
  doThing() {
    console.log("A");
  }
  getOtherThing(): number {
    return 4;
  }
}
```

Overriding toString

The `toString` method may be overridden, but must always succeed and never have visible side effects.

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

4.4.3 Static methods

Avoid private static methods

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

Do not rely on dynamic dispatch

Code *should not* rely on dynamic dispatch of static methods. Static methods *should* only be called on the base class itself (which defines it directly). 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.

Disallowed:

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

// Discouraged: don't call static methods dynamically
function callFoo(cls: typeof Base) {
  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 MyClass {
  static foo() {
    return this.staticField;
  }
}
MyClass.staticField = 1;
```

Avoid static `this` references

Code *must not* use `this` in a static context.

JavaScript allows accessing static fields through `this`. Different from other languages, static fields are also inherited.

```
class ShoeStore {
  static storage: Storage = ...;

  static isAvailable(s: Shoe) {
    // Bad: do not use `this` in a static method.
    return this.storage.has(s.id);
  }
}

class EmptyShoeStore extends ShoeStore {
```

```
  static storage: Storage = EMPTY_STORE;  // overrides storage from Shoe
}
```

Why?

This code is generally surprising: authors might not expect that static fields can be accessed through the this pointer, and might be surprised to find that they can be overridden - this feature is not commonly used.

This code also encourages an anti-pattern of having substantial static state, which causes problems with testability.

4.4.4 Constructors

Constructor calls *must* use parentheses, even when no arguments are passed:

```
const x = new Foo;
```

```
const x = new Foo();
```

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

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

It is unnecessary to provide an empty constructor or one that simply delegates into its parent class because ES2015 provides a default class constructor if one is not specified. However constructors with parameter properties, visibility modifiers or parameter decorators *should not* be omitted even if the body of the constructor is empty.

```
class UnnecessaryConstructor {
  constructor() {}
}
```

```
class UnnecessaryConstructorOverride extends Base {
    constructor(value: number) {
      super(value);
    }
}
```

```
class DefaultConstructor {
}

class ParameterProperties {
  constructor(private myService) {}
}
```

```
class ParameterDecorators {
  constructor(@SideEffectDecorator myService) {}
}

class NoInstantiation {
  private constructor() {}
}
```

The constructor should be separated from surrounding code both above and below by a single blank line:

```
class Foo {
  myField = 10;

  constructor(private readonly ctorParam) {}

  doThing() {
    console.log(ctorParam.getThing() + myField);
  }
}
```

```
class Foo {
  myField = 10;
  constructor(private readonly ctorParam) {}
  doThing() {
    console.log(ctorParam.getThing() + myField);
  }
}
```

4.4.5 Class members

No #private fields

Do not use private fields (also known as private identifiers):

```
class Clazz {
  #ident = 1;
}
```

Instead, use TypeScript's visibility annotations:

```
class Clazz {
  private ident = 1;
}
```

Why?

Private identifiers cause substantial emit size and performance regressions when down-leveled by TypeScript, and are unsupported before ES2015. They can only be downleveled to ES2015, not lower. At the same time, they do not offer substantial benefits when static type checking is used to enforce visibility.

Use readonly

Mark properties that are never reassigned outside of the constructor with the `readonly` modifier (these need not be deeply immutable).

Parameter properties

Rather than plumbing an obvious initializer through to a class member, use a TypeScript parameter property.

```
class Foo {
  private readonly barService: BarService;

  constructor(barService: BarService) {
    this.barService = barService;
  }
}
```

```
class Foo {
  constructor(private readonly barService: BarService) {}
}
```

If the parameter property needs documentation, use an `@param` JSDoc tag.

Field initializers

If a class member is not a parameter, initialize it where it's declared, which sometimes lets you drop the constructor entirely.

```
class Foo {
  private readonly userList: string[];

  constructor() {
    this.userList = [];
  }
}
```

```
class Foo {
  private readonly userList: string[] = [];
}
```

> 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 classes' "shape". Optional fields

> that may be filled in later should be explicitly initialized to `undefined` to prevent later shape changes.

Properties used outside of class lexical scope

Properties used from outside the lexical scope of their containing class, such as an Angular component's properties used from a template, *must not* use `private` visibility, as they are used outside of the lexical scope of their containing class.

Use either `protected` or `public` as appropriate to the property in question. Angular and AngularJS template properties should use `protected`, but Polymer should use `public`.

TypeScript code *must not* use `obj['foo']` to bypass the visibility of a property.

Why?

When a property is `private`, you are declaring to both automated systems and humans that the property accesses are scoped to the methods of the declaring class, and they will rely on that. For example, a check for unused code will flag a private property that appears to be unused, even if some other file manages to bypass the visibility restriction.

Though it might appear that `obj['foo']` can bypass visibility in the TypeScript compiler, this pattern can be broken by rearranging the build rules, and also violates optimization compatibility.

Getters and setters

Getters and setters, also known as accessors, for class members *may* be used. The getter method *must* be a pure function (i.e., result is consistent and has no side effects: getters *must not* change observable state). They are also useful as a means of restricting the visibility of internal or verbose implementation details (shown below).

```
class Foo {
  constructor(private readonly someService: SomeService) {}

  get someMember(): string {
    return this.someService.someVariable;
  }

  set someMember(newValue: string) {
    this.someService.someVariable = newValue;
  }
}
```

```
class Foo {
  nextId = 0;
  get next() {
    return this.nextId++; // Bad: getter changes observable state
  }
}
```

If an accessor is used to hide a class property, the hidden property *may* be prefixed or suffixed with any whole word, like `internal` or `wrapped`. When using these private properties, access the value through the accessor whenever possible. At least one accessor for a property *must* be non-trivial: do not define "pass-through" accessors only for the purpose of hiding a property. Instead, make the property public (or consider making it `readonly` rather than just defining a getter with no setter).

```
class Foo {
  private wrappedBar = '';
  get bar() {
    return this.wrappedBar || 'bar';
  }

  set bar(wrapped: string) {
    this.wrappedBar = wrapped.trim();
  }
}
```

```
class Bar {
  private barInternal = '';
  // Neither of these accessors have logic, so just make bar public.
  get bar() {
    return this.barInternal;
  }

  set bar(value: string) {
    this.barInternal = value;
  }
}
```

Getters and setters *must not* be defined using `Object.defineProperty`, since this interferes with property renaming.

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) are not allowed (see rationale for not mixing key types. 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.

4.4.6 Visibility

Restricting visibility of properties, methods, and entire types helps with keeping code decoupled.

- Limit symbol visibility as much as possible.

- Consider converting private methods to non-exported functions within the same file but outside of any class, and moving private properties into a separate, non-exported class.
- TypeScript symbols are public by default. Never use the `public` modifier except when declaring non-readonly public parameter properties (in constructors).

```
class Foo {
  public bar = new Bar();  // BAD: public modifier not needed

  constructor(public readonly baz: Baz) {}  // BAD: readonly implies it'
}
```

```
class Foo {
  bar = new Bar();  // GOOD: public modifier not needed

  constructor(public baz: Baz) {}  // public modifier allowed
}
```

See also export visibility.

4.4.7 Disallowed class patterns

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. 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.

4.5 Functions

4.5.1 Terminology

There are many different types of functions, with subtle distinctions between them. This guide uses the following terminology, which aligns with MDN:

- “function declaration”: a declaration (i.e. not an expression) using the `function` keyword
- “function expression”: an expression, typically used in an assignment or passed as a parameter, using the `function` keyword
- “arrow function”: an expression using the `=>` syntax
- “block body”: right hand side of an arrow function with braces
- “concise body”: right hand side of an arrow function without braces

Methods and classes/constructors are not covered in this section.

4.5.2 Prefer function declarations for named functions

Prefer function declarations over arrow functions or function expressions when defining named functions.

```
function foo() {
  return 42;
}
```

```
const foo = () => 42;
```

Arrow functions *may* be used, for example, when an explicit type annotation is required.

```
interface SearchFunction {
  (source: string, subString: string): boolean;
}

const fooSearch: SearchFunction = (source, subString) => { ... };
```

4.5.3 Nested functions

Functions nested within other methods or functions *may* use function declarations or arrow functions, as appropriate. In method bodies in particular, arrow functions are preferred because they have access to the outer `this`.

4.5.4 Do not use function expressions

Do not use function expressions. Use arrow functions instead.

```
bar(() => { this.doSomething(); })
```

```
bar(function() { ... })
```

Exception: Function expressions *may* be used *only if* code has to dynamically rebind `this` (but this is discouraged), or for generator functions (which do not have an arrow syntax).

4.5.5 Arrow function bodies

Use arrow functions with concise bodies (i.e. expressions) or block bodies as appropriate.

```
// Top level functions use function declarations.
function someFunction() {
  // Block bodies are fine:
  const receipts = books.map((b: Book) => {
    const receipt = payMoney(b.price);
    recordTransaction(receipt);
    return receipt;
  });
```

```
  // Concise bodies are fine, too, if the return value is used:
  const longThings = myValues.filter(v => v.length > 1000).map(v => Stri

  function payMoney(amount: number) {
    // function declarations are fine, but must not access `this`.
  }

  // Nested arrow functions may be assigned to a const.
  const computeTax = (amount: number) => amount * 0.12;
```

Only use a concise body if the return value of the function is actually used. The block body makes sure the return type is `void` then and prevents potential side effects.

```
// BAD: use a block body if the return value of the function is not used
myPromise.then(v => console.log(v));
// BAD: this typechecks, but the return value still leaks.
let f: () => void;
f = () => 1;
```

```
// GOOD: return value is unused, use a block body.
myPromise.then(v => {
  console.log(v);
});
// GOOD: code may use blocks for readability.
const transformed = [1, 2, 3].map(v => {
  const intermediate = someComplicatedExpr(v);
  const more = acrossManyLines(intermediate);
  return worthWrapping(more);
});
// GOOD: explicit `void` ensures no leaked return value
myPromise.then(v => void console.log(v));
```

> Tip: The `void` operator can be used to ensure an arrow function with an expression body returns `undefined` when the result is unused.

4.5.6 Rebinding `this`

Function expressions and function declarations *must not* use `this` unless they specifically exist to rebind the `this` pointer. Rebinding `this` can in most cases be avoided by using arrow functions or explicit parameters.

```
function clickHandler() {
  // Bad: what's `this` in this context?
  this.textContent = 'Hello';
}
```

```
// Bad: the `this` pointer reference is implicitly set to document.body.
```

```
// Good: explicitly reference the object from an arrow function.
document.body.onclick = () => { document.body.textContent = 'hello'; };
// Alternatively: take an explicit parameter
const setTextFn = (e: HTMLElement) => { e.textContent = 'hello'; };
document.body.onclick = setTextFn.bind(null, document.body);
```

Prefer arrow functions over other approaches to binding `this`, such as `f.bind(this)`, `goog.bind(f, this)`, or `const self = this`.

4.5.7 Prefer passing arrow functions as callbacks

Callbacks can be invoked with unexpected arguments that can pass a type check but still result in logical errors.

Avoid passing a named callback to a higher-order function, unless you are sure of the stability of both functions' call signatures. Beware, in particular, of less-commonly-used optional parameters.

```
// BAD: Arguments are not explicitly passed, leading to unintended behav
// when the optional `radix` argument gets the array indices 0, 1, and 2
const numbers = ['11', '5', '10'].map(parseInt);
// > [11, NaN, 2];
```

Instead, prefer passing an arrow-function that explicitly forwards parameters to the named callback.

```
// GOOD: Arguments are explicitly passed to the callback
const numbers = ['11', '5', '3'].map((n) => parseInt(n));
// > [11, 5, 3]

// GOOD: Function is locally defined and is designed to be used as a cal
function dayFilter(element: string|null|undefined) {
  return element != null && element.endsWith('day');
}

const days = ['tuesday', undefined, 'juice', 'wednesday'].filter(dayFilt
```

4.5.8 Arrow functions as properties

Classes usually *should not* contain properties initialized to arrow functions. Arrow function properties require the calling function to understand that the callee's `this` is already bound, which increases confusion about what `this` is, and call sites and references using such handlers look broken (i.e. require non-local knowledge to determine that they are correct). Code *should* always use arrow functions to call instance methods

(`const handler = (x) => { this.listener(x); };`), and *should not* obtain or pass references to instance methods (~~`const handler = this.listener; handler(x);`~~).

> Note: in some specific situations, e.g. when binding functions in a template, arrow functions as properties are useful and create much more readable code. Use judgement with this rule. Also, see the `Event Handlers` section below.

```
class DelayHandler {
  constructor() {
    // Problem: `this` is not preserved in the callback. `this` in the c
    // will not be an instance of DelayHandler.
    setTimeout(this.patienceTracker, 5000);
  }
  private patienceTracker() {
    this.waitedPatiently = true;
  }
}
```

```
// Arrow functions usually should not be properties.
class DelayHandler {
  constructor() {
    // Bad: this code looks like it forgot to bind `this`.
    setTimeout(this.patienceTracker, 5000);
  }
  private patienceTracker = () => {
    this.waitedPatiently = true;
  }
}
```

```
// Explicitly manage `this` at call time.
class DelayHandler {
  constructor() {
    // Use anonymous functions if possible.
    setTimeout(() => {
      this.patienceTracker();
    }, 5000);
  }
  private patienceTracker() {
    this.waitedPatiently = true;
  }
}
```

4.5.9 Event handlers

Event handlers *may* use arrow functions when there is no need to uninstall the handler (for example, if the event is emitted by the class itself). If the handler requires uninstallation, arrow function properties are the right approach, because they automatically capture `this` and provide a stable reference to uninstall.

```
// Event handlers may be anonymous functions or arrow function propertie
class Component {
  onAttached() {
    // The event is emitted by this class, no need to uninstall.
    this.addEventListener('click', () => {
      this.listener();
    });
    // this.listener is a stable reference, we can uninstall it later.
    window.addEventListener('onbeforeunload', this.listener);
  }
  onDetached() {
    // The event is emitted by window. If we don't uninstall, this.liste
    // keep a reference to `this` because it's bound, causing a memory l
    window.removeEventListener('onbeforeunload', this.listener);
  }
  // An arrow function stored in a property is bound to `this` automatic
  private listener = () => {
    confirm('Do you want to exit the page?');
  }
}
```

Do not use `bind` in the expression that installs an event handler, because it creates a temporary reference that can't be uninstalled.

```
// Binding listeners creates a temporary reference that prevents uninsta
class Component {
  onAttached() {
    // This creates a temporary reference that we won't be able to unins
    window.addEventListener('onbeforeunload', this.listener.bind(this));
  }
  onDetached() {
    // This bind creates a different reference, so this line does nothin
    window.removeEventListener('onbeforeunload', this.listener.bind(this
  }
  private listener() {
    confirm('Do you want to exit the page?');
  }
}
```

4.5.10 Parameter initializers

Optional function parameters *may* be given a default initializer to use when the argument is omitted. Initializers *must not* have any observable side effects. Initializers *should* be kept as simple as possible.

```
function process(name: string, extraContext: string[] = []) {}
function activate(index = 0) {}
```

```
// BAD: side effect of incrementing the counter
let globalCounter = 0;
function newId(index = globalCounter++) {}

// BAD: exposes shared mutable state, which can introduce unintended cou
// between function calls
class Foo {
  private readonly defaultPaths: string[];
  frobnicate(paths = defaultPaths) {}
}
```

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

4.5.11 Prefer rest and spread when appropriate

Use a *rest* parameter instead of accessing `arguments`. Never name a local variable or parameter `arguments`, which confusingly shadows the built-in name.

```
function variadic(array: string[], ...numbers: number[]) {}
```

Use function spread syntax instead of `Function.prototype.apply`.

4.5.12 Formatting functions

Blank lines at the start or end of the function body are not allowed.

A single blank line *may* be used within function bodies sparingly to create *logical groupings* of statements.

Generators should attach the `*` to the `function` and `yield` keywords, as in `function* foo()` and `yield* iter`, rather than ~~`function *foo()`~~ or ~~`yield *iter`~~.

Parentheses around the left-hand side of a single-argument arrow function are recommended but not required.

Do not put a space after the `...` in rest or spread syntax.

```
function myFunction(...elements: number[]) {}
myFunction(...array, ...iterable, ...generator());
```

4.6 this

Only use `this` in class constructors and methods, functions that have an explicit `this` type declared (e.g. `function func(this: ThisType, ...)`), or in arrow functions defined in a scope where `this` may be used.

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.

```
this.alert('Hello');
```

4.7 Interfaces

4.8 Primitive literals

4.8.1 String literals

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.

No line continuations

Do not use *line continuations* (that is, ending a line inside a string literal with a backslash) in either 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 LONG_STRING = 'This is a very very very very very very very long s
    It inadvertently contains long stretches of spaces due to how the \
    continued lines are indented.';
```

Instead, write

```
const LONG_STRING = 'This is a very very very very very very long string
    'It does not contain long stretches of spaces because it uses ' +
    'concatenated strings.';
const SINGLE_STRING =
    'http://it.is.also/acceptable_to_use_a_single_long_string_when_break
```

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

4.8.2 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`.

4.8.3 Type coercion

TypeScript code *may* use the `String()` and `Boolean()` (note: no `new`!) functions, string template literals, or `!!` to coerce types.

```
const bool = Boolean(false);
const str = String(aNumber);
const bool2 = !!str;
const str2 = `result: ${bool2}`;
```

Values of enum types (including unions of enum types and other types) *must not* be converted to booleans with `Boolean()` or `!!`, and must instead be compared explicitly with comparison operators.

```
enum SupportLevel {
  NONE,
  BASIC,
  ADVANCED,
}

const level: SupportLevel = ...;
let enabled = Boolean(level);

const maybeLevel: SupportLevel|undefined = ...;
enabled = !!maybeLevel;
```

```
enum SupportLevel {
  NONE,
  BASIC,
  ADVANCED,
```

```
}

const level: SupportLevel = ...;
let enabled = level !== SupportLevel.NONE;

const maybeLevel: SupportLevel|undefined = ...;
enabled = level !== undefined && level !== SupportLevel.NONE;
```

Why?

For most purposes, it doesn't matter what number or string value an enum name is mapped to at runtime, because values of enum types are referred to by name in source code. Consequently, engineers are accustomed to not thinking about this, and so situations where it *does* matter are undesirable because they will be surprising. Such is the case with conversion of enums to booleans; in particular, by default, the first declared enum value is falsy (because it is 0) while the others are truthy, which is likely to be unexpected. Readers of code that uses an enum value may not even know whether it's the first declared value or not.

Using string concatenation to cast to string is discouraged, as we check that operands to the plus operator are of matching types.

Code *must* use `Number()` to parse numeric values, and *must* check its return for `NaN` values explicitly, unless failing to parse is impossible from context.

> Note: `Number('')`, `Number(' ')`, and `Number('\t')` would return `0` instead of `NaN`. `Number('Infinity')` and `Number('-Infinity')` would return `Infinity` and `-Infinity` respectively. Additionally, exponential notation such as `Number('1e+309')` and `Number('-1e+309')` can overflow into `Infinity`. These cases may require special handling.

```
const aNumber = Number('123');
if (!isFinite(aNumber)) throw new Error(...);
```

Code *must not* use unary plus (`+`) to coerce strings to numbers. Parsing numbers can fail, has surprising corner cases, and can be a code smell (parsing at the wrong layer). A unary plus is too easy to miss in code reviews given this.

```
const x = +y;
```

Code also *must not* use `parseInt` or `parseFloat` to parse numbers, except for non-base-10 strings (see below). Both of those functions ignore trailing characters in the string, which can shadow error conditions (e.g. parsing `12 dwarves` as `12`).

```
const n = parseInt(someString, 10);  // Error prone,
const f = parseFloat(someString);    // regardless of passing a radix.
```

Code that requires parsing with a radix *must* check that its input contains only appropriate digits for that radix before calling into `parseInt`;

```
if (!/^[a-fA-F0-9]+$/.test(someString)) throw new Error(...);
// Needed to parse hexadecimal.
// tslint:disable-next-line:ban
const n = parseInt(someString, 16);  // Only allowed for radix != 10
```

Use `Number()` followed by `Math.floor` or `Math.trunc` (where available) to parse integer numbers:

```
let f = Number(someString);
if (isNaN(f)) handleError();
f = Math.floor(f);
```

Implicit coercion

Do not use explicit boolean coercions in conditional clauses that have implicit boolean coercion. Those are the conditions in an `if`, `for` and `while` statements.

```
const foo: MyInterface|null = ...;
if (!!foo) {...}
while (!!foo) {...}
```

```
const foo: MyInterface|null = ...;
if (foo) {...}
while (foo) {...}
```

As with explicit conversions, values of enum types (including unions of enum types and other types) *must not* be implicitly coerced to booleans, and must instead be compared explicitly with comparison operators.

```
enum SupportLevel {
  NONE,
  BASIC,
  ADVANCED,
}

const level: SupportLevel = ...;
if (level) {...}

const maybeLevel: SupportLevel|undefined = ...;
if (level) {...}
```

```
enum SupportLevel {
  NONE,
  BASIC,
  ADVANCED,
```

```
}

const level: SupportLevel = ...;
if (level !== SupportLevel.NONE) {...}

const maybeLevel: SupportLevel|undefined = ...;
if (level !== undefined && level !== SupportLevel.NONE) {...}
```

Other types of values may be either implicitly coerced to booleans or compared explicitly with comparison operators:

```
// Explicitly comparing > 0 is OK:
if (arr.length > 0) {...}
// so is relying on boolean coercion:
if (arr.length) {...}
```

4.9 Control structures

4.9.1 Control flow statements and blocks

Control flow statements (`if` , `else` , `for` , `do` , `while` , etc) always use braced blocks for the containing code, even if the body contains only a single statement. The first statement of a non-empty block must begin on its own line.

```
for (let i = 0; i < x; i++) {
  doSomethingWith(i);
}

if (x) {
  doSomethingWithALongMethodNameThatForcesANewLine(x);
}
```

```
if (x)
  doSomethingWithALongMethodNameThatForcesANewLine(x);

for (let i = 0; i < x; i++) doSomethingWith(i);
```

Exception: `if` statements fitting on one line *may* elide the block.

```
if (x) x.doFoo();
```

Assignment in control statements

Prefer to avoid assignment of variables inside control statements. Assignment can be easily mistaken for equality checks inside control statements.

```
if (x = someFunction()) {
  // Assignment easily mistaken with equality check
```

```
  // ...
}
```

```
x = someFunction();
if (x) {
  // ...
}
```

In cases where assignment inside the control statement is preferred, enclose the assignment in additional parenthesis to indicate it is intentional.

```
while ((x = someFunction())) {
  // Double parenthesis shows assignment is intentional
  // ...
}
```

Iterating containers

Prefer `for (... of someArr)` to iterate over arrays. `Array.prototype.forEach` and vanilla `for` loops are also allowed:

```
for (const x of someArr) {
  // x is a value of someArr.
}

for (let i = 0; i < someArr.length; i++) {
  // Explicitly count if the index is needed, otherwise use the for/of f
  const x = someArr[i];
  // ...
}
for (const [i, x] of someArr.entries()) {
  // Alternative version of the above.
}
```

`for - in` loops may only be used on dict-style objects (see below for more info). Do not use `for (... in ...)` to iterate over arrays as it will counterintuitively give the array's indices (as strings!), not values:

```
for (const x in someArray) {
  // x is the index!
}
```

`Object.prototype.hasOwnProperty` should be used in `for - in` loops to exclude unwanted prototype properties. Prefer `for - of` with `Object.keys`, `Object.values`, or `Object.entries` over `for - in` when possible.

```
for (const key in obj) {
  if (!obj.hasOwnProperty(key)) continue;
  doWork(key, obj[key]);
}
for (const key of Object.keys(obj)) {
  doWork(key, obj[key]);
}
for (const value of Object.values(obj)) {
  doWorkValOnly(value);
}
for (const [key, value] of Object.entries(obj)) {
  doWork(key, value);
}
```

4.9.2 Grouping parentheses

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.

Do not use unnecessary parentheses around the entire expression following `delete`, `typeof`, `void`, `return`, `throw`, `case`, `in`, `of`, or `yield`.

4.9.3 Exception handling

Exceptions are an important part of the language and should be used whenever exceptional cases occur.

Custom exceptions provide a great way to convey additional error information from functions. They should be defined and used wherever the native `Error` type is insufficient.

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).

Instantiate errors using `new`

Always use `new Error()` when instantiating exceptions, instead of just calling `Error()`. Both forms create a new `Error` instance, but using `new` is more consistent with how other objects are instantiated.

```
throw new Error('Foo is not a valid bar.');
```

```
throw Error('Foo is not a valid bar.');
```

Only throw errors

JavaScript (and thus TypeScript) allow throwing or rejecting a Promise with arbitrary values. However if the thrown or rejected value is not an `Error`, it does not populate stack trace

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

```
// bad: does not get a stack trace.
throw 'oh noes!';
// For promises
new Promise((resolve, reject) => void reject('oh noes!'));
Promise.reject();
Promise.reject('oh noes!');
```

Instead, only throw (subclasses of) `Error`:

```
// Throw only Errors
throw new Error('oh noes!');
// ... or subtypes of Error.
class MyError extends Error {}
throw new MyError('my oh noes!');
// For promises
new Promise((resolve) => resolve()); // No reject is OK.
new Promise((resolve, reject) => void reject(new Error('oh noes!')));
Promise.reject(new Error('oh noes!'));
```

Catching and rethrowing

When catching errors, code *should* assume that all thrown errors are instances of `Error`.

```
function assertIsError(e: unknown): asserts e is Error {
  if (!(e instanceof Error)) throw new Error("e is not an Error");
}

try {
  doSomething();
} catch (e: unknown) {
  // All thrown errors must be Error subtypes. Do not handle
  // other possible values unless you know they are thrown.
  assertIsError(e);
  displayError(e.message);
  // or rethrow:
  throw e;
}
```

Exception handlers *must not* defensively handle non-`Error` types unless the called API is conclusively known to throw non-`Error`s in violation of the above rule. In that case, a comment should be included to specifically identify where the non-`Error`s originate.

```
try {
  badApiThrowingStrings();
} catch (e: unknown) {
  // Note: bad API throws strings instead of errors.
```

```
  if (typeof e === 'string') { ... }
}
```

Why?

Avoid overly defensive programming. Repeating the same defenses against a problem that will not exist in most code leads to boiler-plate code that is not useful.

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 (e: unknown) {
    // Response is not numeric. Continue to handle as text.
  }
  return handleTextResponse(response);
```

Disallowed:

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

> 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.

4.9.4 Switch statements

All `switch` statements *must* contain a `default` statement group, even if it contains no code. The `default` statement group must be last.

```
switch (x) {
  case Y:
    doSomethingElse();
    break;
  default:
    // nothing to do.
}
```

Within a switch block, each statement group either terminates abruptly with a `break`, a `return` statement, or by throwing an exception. Non-empty statement groups (`case ...`) *must not* fall through (enforced by the compiler):

```
switch (x) {
  case X:
    doSomething();
    // fall through - not allowed!
  case Y:
    // ...
}
```

Empty statement groups are allowed to fall through:

```
switch (x) {
  case X:
  case Y:
    doSomething();
    break;
  default: // nothing to do.
}
```

4.9.5 Equality checks

Always use triple equals (`===`) and not equals (`!==`). The double equality operators cause error prone type coercions that are hard to understand and slower to implement for JavaScript Virtual Machines. See also the JavaScript equality table.

```
if (foo == 'bar' || baz != bam) {
  // Hard to understand behaviour due to type coercion.
}
```

```
if (foo === 'bar' || baz !== bam) {
  // All good here.
}
```

Exception: Comparisons to the literal `null` value *may* use the `==` and `!=` operators to cover both `null` and `undefined` values.

```
if (foo == null) {
  // Will trigger when foo is null or undefined.
}
```

4.9.6 Type and non-nullability assertions

Type assertions (`x as SomeType`) and non-nullability assertions (`y!`) are unsafe. Both only silence the TypeScript compiler, but do not insert any runtime checks to match these assertions, so they can cause your program to crash at runtime.

Because of this, you *should not* use type and non-nullability assertions without an obvious or explicit reason for doing so.

Instead of the following:

```
(x as Foo).foo();

y!.bar();
```

When you want to assert a type or non-nullability the best answer is to explicitly write a runtime check that performs that check.

```
// assuming Foo is a class.
if (x instanceof Foo) {
  x.foo();
}

if (y) {
  y.bar();
}
```

Sometimes due to some local property of your code you can be sure that the assertion form is safe. In those situations, you *should* add clarification to explain why you are ok with the unsafe behavior:

```
// x is a Foo, because ...
(x as Foo).foo();

// y cannot be null, because ...
y!.bar();
```

If the reasoning behind a type or non-nullability assertion is obvious, the comments *may* not be necessary. For example, generated proto code is always nullable, but perhaps it is well-known in the context of the code that certain fields are always provided by the backend. Use your judgement.

Type assertion syntax

Type assertions *must* use the `as` syntax (as opposed to the angle brackets syntax). This enforces parentheses around the assertion when accessing a member.

```
const x = (<Foo>z).length;
const y = <Foo>z.length;
```

```
// z must be Foo because ...
const x = (z as Foo).length;
```

Double assertions

From the TypeScript handbook, TypeScript only allows type assertions which convert to a *more specific* or *less specific* version of a type. Adding a type assertion (`x as Foo`) which does not meet this criteria will give the error: “Conversion of type 'X' to type 'Y' may be a mistake because neither type sufficiently overlaps with the other.”

If you are sure an assertion is safe, you can perform a *double assertion*. This involves casting through `unknown` since it is less specific than all types.

```
// x is a Foo here, because...
(x as unknown as Foo).fooMethod();
```

Use `unknown` (instead of `any` or `{}`) as the intermediate type.

Type assertions and object literals

Use type annotations (`: Foo`) instead of type assertions (`as Foo`) to specify the type of an object literal. This allows detecting refactoring bugs when the fields of an interface change over time.

```
interface Foo {
  bar: number;
  baz?: string;  // was "bam", but later renamed to "baz".
}

const foo = {
  bar: 123,
  bam: 'abc',  // no error!
} as Foo;

function func() {
  return {
    bar: 123,
    bam: 'abc',  // no error!
  } as Foo;
}
```

```
interface Foo {
  bar: number;
  baz?: string;
}

const foo: Foo = {
  bar: 123,
  bam: 'abc',  // complains about "bam" not being defined on Foo.
};

function func(): Foo {
  return {
    bar: 123,
    bam: 'abc',   // complains about "bam" not being defined on Foo.
  };
}
```

4.9.7 Keep try blocks focused

Limit the amount of code inside a try block, if this can be done without hurting readability.

```
try {
  const result = methodThatMayThrow();
  use(result);
} catch (error: unknown) {
  // ...
}
```

```
let result;
try {
  result = methodThatMayThrow();
} catch (error: unknown) {
  // ...
}
use(result);
```

Moving the non-throwable lines out of the try/catch block helps the reader learn which method throws exceptions. Some inline calls that do not throw exceptions could stay inside because they might not be worth the extra complication of a temporary variable.

Exception: There may be performance issues if try blocks are inside a loop. Widening try blocks to cover a whole loop is ok.

4.10 Decorators

Decorators are syntax with an `@` prefix, like `@MyDecorator`.

Do not define new decorators. Only use the decorators defined by frameworks:

- Angular (e.g. `@Component`, `@NgModule`, etc.)
- Polymer (e.g. `@property`)

Why?

We generally want to avoid decorators, because they were an experimental feature that have since diverged from the TC39 proposal and have known bugs that won't be fixed.

When using decorators, the decorator *must* immediately precede the symbol it decorates, with no empty lines between:

```
/** JSDoc comments go before decorators */
@Component({...})  // Note: no empty line after the decorator.
class MyComp {
  @Input() myField: string;  // Decorators on fields may be on the same

  @Input()
```

```
  myOtherField: string;  // ... or wrap.
}
```

4.11 Disallowed features

4.11.1 Wrapper objects for primitive types

TypeScript code *must not* instantiate the wrapper classes for the primitive types `String`, `Boolean`, and `Number`. Wrapper classes have surprising behavior, such as `new Boolean(false)` evaluating to `true`.

```
const s = new String('hello');
const b = new Boolean(false);
const n = new Number(5);
```

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

4.11.2 Automatic Semicolon Insertion

Do not rely on Automatic Semicolon Insertion (ASI). Explicitly end all statements using a semicolon. This prevents bugs due to incorrect semicolon insertions and ensures compatibility with tools with limited ASI support (e.g. clang-format).

4.11.3 Const enums

Code *must not* use `const enum`; use plain `enum` instead.

Why?

TypeScript enums already cannot be mutated; `const enum` is a separate language feature related to optimization that makes the enum invisible to JavaScript users of the module.

4.11.4 Debugger statements

Debugger statements *must not* be included in production code.

```
function debugMe() {
  debugger;
}
```

4.11.5 `with`

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

4.11.6 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 environments using strict Content Security Policies.

4.11.7 Non-standard features

Do not use non-standard ECMAScript or Web Platform features.

This includes:

- Old features that have been marked deprecated or removed entirely from ECMAScript / the Web Platform (see MDN)
- New ECMAScript features that are not yet standardized
  - Avoid using features that are in current TC39 working draft or currently in the proposal process
  - Use only ECMAScript features defined in the current ECMA-262 specification
- Proposed but not-yet-complete web standards:
  - WHATWG proposals that have not completed the proposal process.
- Non-standard language “extensions” (such as those provided by some external transpilers)

Projects targeting specific JavaScript runtimes, such as latest-Chrome-only, Chrome extensions, Node.JS, Electron, can obviously use those APIs. Use caution when considering an API surface that is proprietary and only implemented in some browsers; consider whether there is a common library that can abstract this API surface away for you.

4.11.8 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.

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

5 Naming

5.1 Identifiers

Identifiers *must* use only ASCII letters, digits, underscores (for constants and structured test method names), and (rarely) the '$' sign.

5.1.1 Naming style

TypeScript expresses information in types, so names *should not* be decorated with information that is included in the type. (See also Testing Blog for more about what not to include.)

Some concrete examples of this rule:

- Do not use trailing or leading underscores for private properties or methods.
- Do not use the `opt_` prefix for optional parameters.
  - For accessors, see accessor rules below.
- Do not mark interfaces specially (~~`IMyInterface`~~ or ~~`MyFooInterface`~~) unless it's idiomatic in its environment. When introducing an interface for a class, give it a name that expresses why the interface exists in the first place (e.g. `class TodoItem` and `interface TodoItemStorage` if the interface expresses the format used for storage/serialization in JSON).
- Suffixing `Observable` s with `$` is a common external convention and can help resolve confusion regarding observable values vs concrete values. Judgement on whether this is a useful convention is left up to individual teams, but *should* be consistent within projects.

5.1.2 Descriptive names

Names *must* be descriptive and clear to 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.

- **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.

```
// Good identifiers:
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 identifiers:
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.
customerID          // Incorrect camelcase of "ID".
```

5.1.3 Camel case

Treat abbreviations like acronyms in names as whole words, i.e. use `loadHttpUrl` , not ~~`loadHTTPURL`~~ , unless required by a platform name (e.g. `XMLHttpRequest`).

5.1.4 Dollar sign

Identifiers *should not* generally use `$` , except when required by naming conventions for third party frameworks. See above for more on using `$` with `Observable` values.

5.2 Rules by identifier type

Most identifier names should follow the casing in the table below, based on the identifier's type.

Style	Category
`UpperCamelCase`	class / interface / type / enum / decorator / type parameters / component functions in TSX / JSXElement type parameter
`lowerCamelCase`	variable / parameter / function / method / property / module alias
`CONSTANT_CASE`	global constant values, including enum values. See Constants below.
`#ident`	private identifiers are never used.

5.2.1 Type parameters

Type parameters, like in `Array<T>`, *may* use a single upper case character (`T`) or `UpperCamelCase`.

5.2.2 Test names

Test method names inxUnit-style test frameworks *may* be structured with `_` separators, e.g. `testX_whenY_doesZ()`.

5.2.3 `_` prefix/suffix

Identifiers must not use `_` as a prefix or suffix.

This also means that `_` *must not* be used as an identifier by itself (e.g. to indicate a parameter is unused).

> Tip: If you only need some of the elements from an array (or TypeScript tuple), you can insert extra commas in a destructuring statement to ignore in-between elements:

```
const [a, , b] = [1, 5, 10];  // a <- 1, b <- 10
```

5.2.4 Imports

Module namespace imports are `lowerCamelCase` while files are `snake_case`, which means that imports correctly will not match in casing style, such as

```
import * as fooBar from './foo_bar';
```

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 libraries that currently fall under this exception are:

- jquery, using the `$` prefix
- threejs, using the `THREE` prefix

5.2.5 Constants

Immutable: `CONSTANT_CASE` indicates that a value is *intended* to not be changed, and *may* be used for values that can technically be modified (i.e. values that are not deeply frozen) to indicate to users that they must not be modified.

```
const UNIT_SUFFIXES = {
  'milliseconds': 'ms',
  'seconds': 's',
};
// Even though per the rules of JavaScript UNIT_SUFFIXES is
// mutable, the uppercase shows users to not modify it.
```

A constant can also be a `static readonly` property of a class.

```
class Foo {
  private static readonly MY_SPECIAL_NUMBER = 5;

  bar() {
    return 2 * Foo.MY_SPECIAL_NUMBER;
  }
}
```

Global: Only symbols declared on the module level, static fields of module level classes, and values of module level enums, *may* use `CONST_CASE`. If a value can be instantiated more than once over the lifetime of the program (e.g. a local variable declared within a function, or a static field on a class nested in a function) then it *must* use `lowerCamelCase`.

If a value is an arrow function that implements an interface, then it *may* be declared `lowerCamelCase`.

5.2.6 Aliases

When creating a local-scope alias of an existing symbol, use the format of the existing identifier. The local alias *must* match the existing naming and format of the source. For variables use `const` for your local aliases, and for class fields use the `readonly` attribute.

> Note: If you're creating an alias just to expose it to a template in your framework of choice, remember to also apply the proper access modifiers.

```
const {BrewStateEnum} = SomeType;
const CAPACITY = 5;

class Teapot {
  readonly BrewStateEnum = BrewStateEnum;
  readonly CAPACITY = CAPACITY;
}
```

6 Type system

6.1 Type inference

Code *may* rely on type inference as implemented by the TypeScript compiler for all type expressions (variables, fields, return types, etc).

```
const x = 15;  // Type inferred.
```

Leave out type annotations for trivially inferred types: variables or parameters initialized to a `string`, `number`, `boolean`, `RegExp` literal or `new` expression.

```
const x: boolean = true;  // Bad: 'boolean' here does not aid readabilit
```

```
// Bad: 'Set' is trivially inferred from the initialization
const x: Set<string> = new Set();
```

Explicitly specifying types may be required to prevent generic type parameters from being inferred as `unknown`. For example, initializing generic types with no values (e.g. empty arrays, objects, `Map`s, or `Set`s).

```
const x = new Set<string>();
```

For more complex expressions, type annotations can help with readability of the program:

```
// Hard to reason about the type of 'value' without an annotation.
const value = await rpc.getSomeValue().transform();
```

```
// Can tell the type of 'value' at a glance.
const value: string[] = await rpc.getSomeValue().transform();
```

Whether an annotation is required is decided by the code reviewer.

6.1.1 Return types

Whether to include return type annotations for functions and methods is up to the code author. Reviewers *may* ask for annotations to clarify complex return types that are hard to understand. Projects *may* have a local policy to always require return types, but this is not a general TypeScript style requirement.

There are two benefits to explicitly typing out the implicit return values of functions and methods:

- More precise documentation to benefit readers of the code.

- Surface potential type errors faster in the future if there are code changes that change the return type of the function.

6.2 Undefined and null

TypeScript supports `undefined` and `null` types. Nullable types can be constructed as a union type (`string|null`); similarly with `undefined` . There is no special syntax for unions of `undefined` and `null` .

TypeScript code can use either `undefined` or `null` to denote absence of a value, there is no general guidance to prefer one over the other. Many JavaScript APIs use `undefined` (e.g. `Map.get`), while many DOM and Google APIs use `null` (e.g. `Element.getAttribute`), so the appropriate absent value depends on the context.

6.2.1 Nullable/undefined type aliases

Type aliases *must not* include `|null` or `|undefined` in a union type. Nullable aliases typically indicate that null values are being passed around through too many layers of an application, and this clouds the source of the original issue that resulted in `null` . They also make it unclear when specific values on a class or interface might be absent.

Instead, code *must* only add `|null` or `|undefined` when the alias is actually used. Code *should* deal with null values close to where they arise, using the above techniques.

```
// Bad
type CoffeeResponse = Latte|Americano|undefined;

class CoffeeService {
  getLatte(): CoffeeResponse { ... };
}
```

```
// Better
type CoffeeResponse = Latte|Americano;

class CoffeeService {
  getLatte(): CoffeeResponse|undefined { ... };
}
```

6.2.2 Prefer optional over `|undefined`

In addition, TypeScript supports a special construct for optional parameters and fields, using `?` :

```
interface CoffeeOrder {
  sugarCubes: number;
  milk?: Whole|LowFat|HalfHalf;
}

function pourCoffee(volume?: Milliliter) { ... }
```

Optional parameters implicitly include `|undefined` in their type. However, they are different in that they can be left out when constructing a value or calling a method. For example, `{sugarCubes: 1}` is a valid `CoffeeOrder` because `milk` is optional.

Use optional fields (on interfaces or classes) and parameters rather than a `|undefined` type.

For classes preferably avoid this pattern altogether and initialize as many fields as possible.

```
class MyClass {
  field = '';
}
```

6.3 Use structural types

TypeScript's type system is structural, not nominal. That is, a value matches a type if it has at least all the properties the type requires and the properties' types match, recursively.

When providing a structural-based implementation, explicitly include the type at the declaration of the symbol (this allows more precise type checking and error reporting).

```
const foo: Foo = {
  a: 123,
  b: 'abc',
}
```

```
const badFoo = {
  a: 123,
  b: 'abc',
}
```

Use interfaces to define structural types, not classes

```
interface Foo {
  a: number;
  b: string;
}

const foo: Foo = {
  a: 123,
  b: 'abc',
}
```

```
class Foo {
  readonly a: number;
  readonly b: number;
}

const foo: Foo = {
```

```
  a: 123,
  b: 'abc',
}
```

Why?

The “badFoo” object above relies on type inference. Additional fields could be added to “badFoo” and the type is inferred based on the object itself.

When passing a “badFoo” to a function that takes a “Foo”, the error will be at the function call site, rather than at the object declaration site. This is also useful when changing the surface of an interface across broad codebases.

```
interface Animal {
  sound: string;
  name: string;
}

function makeSound(animal: Animal) {}

/**
 * 'cat' has an inferred type of '{sound: string}'
 */
const cat = {
  sound: 'meow',
};

/**
 * 'cat' does not meet the type contract required for the function, so t
 * TypeScript compiler errors here, which may be very far from where 'ca
 * defined.
 */
makeSound(cat);

/**
 * Horse has a structural type and the type error shows here rather than
 * function call.  'horse' does not meet the type contract of 'Animal'.
 */
const horse: Animal = {
  sound: 'niegh',
};

const dog: Animal = {
  sound: 'bark',
  name: 'MrPickles',
};

makeSound(dog);
makeSound(horse);
```

6.4 Prefer interfaces over type literal aliases

TypeScript supports type aliases for naming a type expression. This can be used to name primitives, unions, tuples, and any other types.

However, when declaring types for objects, use interfaces instead of a type alias for the object literal expression.

```
interface User {
  firstName: string;
  lastName: string;
}
```

```
type User = {
  firstName: string,
  lastName: string,
}
```

Why?

These forms are nearly equivalent, so under the principle of just choosing one out of two forms to prevent variation, we should choose one. Additionally, there are also interesting technical reasons to prefer interface. That page quotes the TypeScript team lead: “Honestly, my take is that it should really just be interfaces for anything that they can model. There is no benefit to type aliases when there are so many issues around display/perf.”

6.5 `Array<T>` Type

For simple types (containing just alphanumeric characters and dot), use the syntax sugar for arrays, `T[]` or `readonly T[]`, rather than the longer form `Array<T>` or `ReadonlyArray<T>`.

For multi-dimensional non-`readonly` arrays of simple types, use the syntax sugar form (`T[][]`, `T[][][]`, and so on) rather than the longer form.

For anything more complex, use the longer form `Array<T>`.

These rules apply at each level of nesting, i.e. a simple `T[]` nested in a more complex type would still be spelled as `T[]`, using the syntax sugar.

```
let a: string[];
let b: readonly string[];
let c: ns.MyObj[];
let d: string[][];
let e: Array<{n: number, s: string}>;
let f: Array<string|number>;
let g: ReadonlyArray<string|number>;
let h: InjectionToken<string[]>;  // Use syntax sugar for nested types.
```

```
let i: ReadonlyArray<string[]>;
let j: Array<readonly string[]>;
```

```
let a: Array<string>;  // The syntax sugar is shorter.
let b: ReadonlyArray<string>;
let c: Array<ns.MyObj>;
let d: Array<string[]>;
let e: {n: number, s: string}[];  // The braces make it harder to read.
let f: (string|number)[];         // Likewise with parens.
let g: readonly (string | number)[];
let h: InjectionToken<Array<string>>;
let i: readonly string[][];
let j: (readonly string[])[];
```

6.6 Indexable types / index signatures (`{[key: string]: T}`)

In JavaScript, it's common to use an object as an associative array (aka “map”, “hash”, or “dict”). Such objects can be typed using an index signature (`[k: string]: T`) in TypeScript:

```
const fileSizes: {[fileName: string]: number} = {};
fileSizes['readme.txt'] = 541;
```

In TypeScript, provide a meaningful label for the key. (The label only exists for documentation; it's unused otherwise.)

```
const users: {[key: string]: number} = ...;
```

```
const users: {[userName: string]: number} = ...;
```

Rather than using one of these, consider using the ES6 `Map` and `Set` types instead. JavaScript objects have surprising undesirable behaviors and the ES6 types more explicitly convey your intent. Also, `Map` s can be keyed by—and `Set` s can contain—types other than `string` .

TypeScript's builtin `Record<Keys, ValueType>` type allows constructing types with a defined set of keys. This is distinct from associative arrays in that the keys are statically known. See advice on that below.

6.7 Mapped and conditional types

TypeScript's mapped types and conditional types allow specifying new types based on other types. TypeScript's standard library includes several type operators based on these (`Record` , `Partial` , `Readonly` etc).

These type system features allow succinctly specifying types and constructing powerful yet type safe abstractions. They come with a number of drawbacks though:

- Compared to explicitly specifying properties and type relations (e.g. using interfaces and extension, see below for an example), type operators require the reader to mentally evaluate the type expression. This can make programs substantially harder to read, in particular combined with type inference and expressions crossing file boundaries.
- Mapped & conditional types' evaluation model, in particular when combined with type inference, is underspecified, not always well understood, and often subject to change in TypeScript compiler versions. Code can “accidentally” compile or seem to give the right results. This increases future support cost of code using type operators.
- Mapped & conditional types are most powerful when deriving types from complex and/or inferred types. On the flip side, this is also when they are most prone to create hard to understand and maintain programs.
- Some language tooling does not work well with these type system features. E.g. your IDE's find references (and thus rename property refactoring) will not find properties in a `Pick<T, Keys>` type, and Code Search won't hyperlink them.
-

The style recommendation is:

- Always use the simplest type construct that can possibly express your code.
- A little bit of repetition or verbosity is often much cheaper than the long term cost of complex type expressions.
- Mapped & conditional types may be used, subject to these considerations.

For example, TypeScript's builtin `Pick<T, Keys>` type allows creating a new type by subsetting another type `T`, but simple interface extension can often be easier to understand.

```
interface User {
  shoeSize: number;
  favoriteIcecream: string;
  favoriteChocolate: string;
}

// FoodPreferences has favoriteIcecream and favoriteChocolate, but not s
type FoodPreferences = Pick<User, 'favoriteIcecream'|'favoriteChocolate'
```

This is equivalent to spelling out the properties on `FoodPreferences`:

```
interface FoodPreferences {
  favoriteIcecream: string;
  favoriteChocolate: string;
}
```

To reduce duplication, `User` could extend `FoodPreferences`, or (possibly better) nest a field for food preferences:

```
interface FoodPreferences { /* as above */ }
interface User extends FoodPreferences {
  shoeSize: number;
  // also includes the preferences.
}
```

Using interfaces here makes the grouping of properties explicit, improves IDE support, allows better optimization, and arguably makes the code easier to understand.

6.8 `any` Type

TypeScript's `any` type is a super and subtype of all other types, and allows dereferencing all properties. As such, `any` is dangerous - it can mask severe programming errors, and its use undermines the value of having static types in the first place.

Consider *not* to use `any`. In circumstances where you want to use `any`, consider one of:

- Provide a more specific type
- Use `unknown`
- Suppress the lint warning and document why

6.8.1 Providing a more specific type

Use interfaces , an inline object type, or a type alias:

```
// Use declared interfaces to represent server-side JSON.
declare interface MyUserJson {
  name: string;
  email: string;
}

// Use type aliases for types that are repetitive to write.
type MyType = number|string;

// Or use inline object types for complex returns.
function getTwoThings(): {something: number, other: string} {
  // ...
  return {something, other};
}

// Use a generic type, where otherwise a library would say `any` to repr
// they don't care what type the user is operating on (but note "Return
// only generics" below).
function nicestElement<T>(items: T[]): T {
  // Find the nicest element in items.
  // Code can also put constraints on T, e.g. <T extends HTMLElement>.
}
```

6.8.2 Using `unknown` over `any`

The `any` type allows assignment into any other type and dereferencing any property off it. Often this behaviour is not necessary or desirable, and code just needs to express that a type is unknown. Use the built-in type `unknown` in that situation — it expresses the concept and is much safer as it does not allow dereferencing arbitrary properties.

```
// Can assign any value (including null or undefined) into this but cann
// use it without narrowing the type or casting.
const val: unknown = value;
```

```
const danger: any = value /* result of an arbitrary expression */;
danger.whoops();  // This access is completely unchecked!
```

To safely use `unknown` values, narrow the type using a type guard

6.8.3 Suppressing `any` lint warnings

Sometimes using `any` is legitimate, for example in tests to construct a mock object. In such cases, add a comment that suppresses the lint warning, and document why it is legitimate.

```
// This test only needs a partial implementation of BookService, and if
// we overlooked something the test will fail in an obvious way.
// This is an intentionally unsafe partial mock
// tslint:disable-next-line:no-any
const mockBookService = ({get() { return mockBook; }} as any) as BookSer
// Shopping cart is not used in this test
// tslint:disable-next-line:no-any
const component = new MyComponent(mockBookService, /* unused ShoppingCar
```

6.9 `{}` Type

The `{}` type, also known as an *empty interface* type, represents a interface with no properties. An empty interface type has no specified properties and therefore any non-nullish value is assignable to it.

```
let player: {};

player = {
  health: 50,
}; // Allowed.

console.log(player.health) // Property 'health' does not exist on type '
```

```
function takeAnything(obj:{}) {

}

takeAnything({});
takeAnything({ a: 1, b: 2 });
```

Google3 code **should not** use `{}` for most use cases. `{}` represents any non-nullish primitive or object type, which is rarely appropriate. Prefer one of the following more-descriptive types:

- `unknown` can hold any value, including `null` or `undefined`, and is generally more appropriate for opaque values.
- `Record<string, T>` is better for dictionary-like objects, and provides better type safety by being explicit about the type `T` of contained values (which may itself be `unknown`).
- `object` excludes primitives as well, leaving only non-nullish functions and objects, but without any other assumptions about what properties may be available.

6.10 Tuple types

If you are tempted to create a Pair type, instead use a tuple type:

```
interface Pair {
  first: string;
  second: string;
}
function splitInHalf(input: string): Pair {
  ...
  return {first: x, second: y};
}
```

```
function splitInHalf(input: string): [string, string] {
  ...
  return [x, y];
}

// Use it like:
const [leftHalf, rightHalf] = splitInHalf('my string');
```

However, often it's clearer to provide meaningful names for the properties.

If declaring an `interface` is too heavyweight, you can use an inline object literal type:

```
function splitHostPort(address: string): {host: string, port: number} {
  ...
}

// Use it like:
const address = splitHostPort(userAddress);
use(address.port);
```

```
// You can also get tuple-like behavior using destructuring:
```

6.11 Wrapper types

There are a few types related to JavaScript primitives that *should not* ever be used:

- `String`, `Boolean`, and `Number` have slightly different meaning from the corresponding primitive types `string`, `boolean`, and `number`. Always use the lowercase version.
- `Object` has similarities to both `{}` and `object`, but is slightly looser. Use `{}` for a type that include everything except `null` and `undefined`, or lowercase `object` to further exclude the other primitive types (the three mentioned above, plus `symbol` and `bigint`).

Further, never invoke the wrapper types as constructors (with `new`).

6.12 Return type only generics

Avoid creating APIs that have return type only generics. When working with existing APIs that have return type only generics always explicitly specify the generics.

7 Toolchain requirements

Google style requires using a number of tools in specific ways, outlined here.

7.1 TypeScript compiler

All TypeScript files must pass type checking using the standard tool chain.

7.1.1 @ts-ignore

Do not use `@ts-ignore` nor the variants `@ts-expect-error` or `@ts-nocheck`.

Why?

They superficially seem to be an easy way to “fix” a compiler error, but in practice, a specific compiler error is often caused by a larger problem that can be fixed more directly.

For example, if you are using `@ts-ignore` to suppress a type error, then it's hard to predict what types the surrounding code will end up seeing. For many type errors, the advice in how to best use `any` is useful.

You may use `@ts-expect-error` in unit tests, though you generally *should not*. `@ts-expect-error` suppresses all errors. It's easy to accidentally over-match and suppress more serious errors. Consider one of:

- When testing APIs that need to deal with unchecked values at runtime, add casts to the expected type or to `any` and add an explanatory comment. This limits error suppression to a single expression.
- Suppress the lint warning and document why, similar to suppressing `any` lint warnings.

7.2 Conformance

Google TypeScript includes several *conformance frameworks*, tsetse and tsec.

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.

Google-style TypeScript must abide by any applicable global or framework-local conformance rules.

8 Comments and documentation

8.0.1 JSDoc versus comments

There are two types of comments, JSDoc (`/** ... */`) and non-JSDoc ordinary comments (`// ...` or `/* ... */`).

- Use `/** JSDoc */` comments for documentation, i.e. comments a user of the code should read.
- Use `// line comments` for implementation comments, i.e. comments that only concern the implementation of the code itself.

JSDoc comments are understood by tools (such as editors and documentation generators), while ordinary comments are only for other humans.

8.0.2 Multi-line comments

Multi-line comments are indented at the same level as the surrounding code. They *must* use multiple single-line comments (`//` -style), not block comment style (`/* */`).

```
// This is
// fine
```

```
/*
 * This should
 * use multiple
 * single-line comments
 */

/* This should use // */
```

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

8.1 JSDoc general form

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

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

or in this single-line example:

```
/** This short jsdoc describes the function. */
function doSomething(arg: number) { … }
```

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.

8.2 Markdown

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

This means that tooling parsing JSDoc will ignore plain text formatting, so if you did this:

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

it will be rendered 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
 */
```

8.3 JSDoc tags

Google style allows a subset of JSDoc tags. Most tags must occupy their own line, with the tag at the beginning of the line.

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

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

8.4 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.

```
/**
 * Illustrates line wrapping for long param/return descriptions.
 * @param foo This is a param with a particularly long description that
 *     doesn't fit on one line.
 * @return This returns something that has a lengthy description too lon
 *     in one line.
 */
exports.method = function(foo) {
  return 5;
};
```

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

8.5 Document all top-level exports of modules

Use `/** JSDoc */` comments to communicate information to the users of your code. Avoid merely restating the property or parameter name. You *should* also document all properties and methods (exported/public or not) whose purpose is not immediately obvious from their name, as judged by your reviewer.

Exception: Symbols that are only exported to be consumed by tooling, such as @NgModule classes, do not require comments.

8.6 Class comments

JSDoc comments for classes 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.

8.7 Method and function comments

Method, parameter, and return descriptions may be omitted if they are obvious from the rest of the method's JSDoc or from the method name and type 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.

8.8 Parameter property comments

A parameter property is a constructor parameter that is prefixed by one of the modifiers `private`, `protected`, `public`, or `readonly`. A parameter property declares both a parameter and an instance property, and implicitly assigns into it. For example, `constructor(private readonly foo: Foo)`, declares that the constructor takes a parameter `foo`, but also declares a private readonly property `foo`, and assigns the parameter into that property before executing the remainder of the constructor.

To document these fields, use JSDoc's `@param` annotation. Editors display the description on constructor calls and property accesses.

```
/** This class demonstrates how parameter properties are documented. */
class ParamProps {
  /**
   * @param percolator The percolator used for brewing.
   * @param beans The beans to brew.
   */
  constructor(
    private readonly percolator: Percolator,
    private readonly beans: CoffeeBean[]) {}
}
```

```
/** This class demonstrates how ordinary fields are documented. */
class OrdinaryClass {
  /** The bean that will be used in the next call to brew(). */
  nextBean: CoffeeBean;

  constructor(initialBean: CoffeeBean) {
```

```
    this.nextBean = initialBean;
  }
}
```

8.9 JSDoc type annotations

JSDoc type annotations are redundant in TypeScript source code. Do not declare types in `@param` or `@return` blocks, do not write `@implements`, `@enum`, `@private`, `@override` etc. on code that uses the `implements`, `enum`, `private`, `override` etc. keywords.

8.10 Make comments that actually add information

For non-exported symbols, sometimes the name and type of the function or parameter is enough. Code will *usually* benefit from more documentation than just variable names though!

- Avoid comments that just restate the parameter name and type, e.g.

```
/** @param fooBarService The Bar service for the Foo application. *
```

- Because of this rule, `@param` and `@return` lines are only required when they add information, and *may* otherwise be omitted.

```
/**
 * POSTs the request to start coffee brewing.
 * @param amountLitres The amount to brew. Must fit the pot size!
 */
brew(amountLitres: number, logger: Logger) {
  // ...
}
```

8.10.1 Comments when calling a function

“Parameter name” comments should be used whenever the method name and parameter value do not sufficiently convey the meaning of the parameter.

Before adding these comments, consider refactoring the method to instead accept an interface and destructure it to greatly improve call-site readability.

“Parameter name” comments go before the parameter value, and include the parameter name and a `=` suffix:

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

Existing code may use a legacy parameter name comment style, which places these comments ~after~ the parameter value and omits the `=`. Continuing to use this style within the file for consistency is acceptable.

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

8.11 Place documentation prior to decorators

When a class, method, or property have both decorators like `@Component` and JsDoc, please make sure to write the JsDoc before the decorator.

- Do not write JsDoc between the Decorator and the decorated statement.

```
@Component({
  selector: 'foo',
  template: 'bar',
})
/** Component that prints "bar". */
export class FooComponent {}
```

- Write the JsDoc block before the Decorator.

```
/** Component that prints "bar". */
@Component({
  selector: 'foo',
  template: 'bar',
})
export class FooComponent {}
```

9 Policies

9.1 Consistency

For any style question that isn't settled definitively by this specification, do what the other code in the same file is already doing (“be consistent”). If that doesn't resolve the question, consider emulating the other files in the same directory.

Brand new files *must* 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 below. 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.

9.1.1 Reformatting existing code

You will occasionally encounter files in the 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.

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.

9.2 Deprecation

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

9.3 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.

9.3.1 Style guide goals

In general, engineers usually know best about what's needed in their code, so if there are multiple options and the choice is situation dependent, we should let decisions be made locally. So the default answer should be "leave it out".

The following points are the exceptions, which are the reasons we have some global rules. Evaluate your style guide proposal against the following:

1. **Code should avoid patterns that are known to cause problems, especially for users new to the language.**

2. **Code across projects should be consistent across irrelevant variations.**

   When there are two options that are equivalent in a superficial way, we should consider choosing one just so we don't divergently evolve for no reason and avoid pointless debates in code reviews.

   Examples:

   - The capitalization style of names.
   - `x as T` syntax vs the equivalent `<T>x` syntax (disallowed).

- `Array<[number, number]>` vs `[number, number][]` .

3. **Code should be maintainable in the long term.**

   Code usually lives longer than the original author works on it, and the TypeScript team must keep all of Google working into the future.

   Examples:

   - We use software to automate changes to code, so code is autoformatted so it's easy for software to meet whitespace rules.
   - We require a single set of compiler flags, so a given TS library can be written assuming a specific set of flags, and users can always safely use a shared library.
   - Code must import the libraries it uses (“strict deps”) so that a refactor in a dependency doesn't change the dependencies of its users.
   - We ask users to write tests. Without tests we cannot have confidence that changes that we make to the language, don't break users.

4. **Code reviewers should be focused on improving the quality of the code, not enforcing arbitrary rules.**

   If it's possible to implement your rule as an automated check that is often a good sign. This also supports principle 3.

   If it really just doesn't matter that much -- if it's an obscure corner of the language or if it avoids a bug that is unlikely to occur -- it's probably worth leaving out.

---

1. Namespace imports are often called 'module imports' ↩

2. named imports are sometimes called 'destructuring imports' because they use similar syntax to destructuring assignments. ↩