Google TypeScript Style Guide

Table of Contents

Syntax
- Identifiers
- File encoding: UTF-8
- Comments & Documentation

Language Rules
- Visibility
- Constructors
- Class Members
- Primitive Types & Wrapper Classes
- Array constructor
- Type coercion
- Variables
- Exceptions
- Iterating objects
- Iterating containers
- Using the spread operator
- Control flow statements & blocks
- Switch Statements
- Equality Checks
- Function Declarations
- Function Expressions
- Automatic Semicolon Insertion
- @ts-ignore
- Type and Non-nullability Assertions
- Member property declarations
- Enums
- Debugger statements
- Decorators

Source Organization
- Modules
- Exports
- Imports
- Organize By Feature

Type System
- Type Inference
- Null vs Undefined
- Structural Types vs Nominal Types
- Interfaces vs Type Aliases
- Array<T> Type
- Indexable ({[key: string]: number}) Type
- Mapped & Conditional Types
- any Type
- Tuple Types
- Wrapper types
- Return type only generics

Consistency
- Goals

TypeScript style guide

This is the external guide that's based on the internal Google version but has been adjusted for the broader audience. There is no automatic deployment process for this version as it's pushed on-demand by volunteers.
It contains both rules and best practices. Choose those that work best for your team.

This Style Guide uses RFC 2119 terminology when using the phrases *must*, *must not*, *should*, *should not*, and *may*. All examples given are non-normative and serve only to illustrate the normative language of the style guide.

Syntax

Identifiers

Identifiers must use only ASCII letters, digits, underscores (for constants and structured test method names), and the '\(' sign. Thus each valid identifier name is matched by the regular expression `[\)\w]+`.

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

- **Abbreviations**: Treat abbreviations like acronyms in names as whole words, i.e. use `loadHttpUrl`, not ~~`loadHTTPURL`~~, unless required by a platform name (e.g. `XMLHttpRequest`).
- **Dollar sign**: Identifiers *should not* generally use `$`, except when aligning with naming conventions for third party frameworks. See below for more on using `$` with `Observable` values.
- **Type parameters**: Type parameters, like in `Array<T>`, may use a single upper case character (`T`) or `UpperCamelCase`.

- **Test names**: Test method names in Closure `testSuite`s and similar xUnit-style test frameworks may be structured with `_` separators, e.g. `testX_whenY_doesZ()`.

- `_` **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
```

- **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
- **Constants**: `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;
    }
  }
  ```

  If a value can be instantiated more than once over the lifetime of the program, or if users mutate it in any way, it must use `lowerCamelCase`.

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

-

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.

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

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

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.

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.

File encoding: UTF-8

For 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;
```

Comments & Documentation

JSDoc vs 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.

JSDoc rules follow the JavaScript style

In general, follow the JavaScript style guide's rules for JSDoc, sections 7.1 - 7.5. The remainder of this section describes exceptions to those rules.

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.

Omit comments that are redundant with TypeScript

For example, do not declare types in `@param` or `@return` blocks, do not write `@implements`, `@enum`, `@private` etc. on code that uses the `implements`, `enum`, `private` etc. keywords.

Do not use `@override`

Do not use `@override` in TypeScript source code.

`@override` is not enforced by the compiler, which is surprising and leads to annotations and implementation going out of sync. Including it purely for documentation purposes is confusing.

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) {
  // ...
}
```

Parameter property comments

A parameter property is when a class declares a field and a constructor parameter in a single declaration, by marking a parameter in the constructor. E.g. `constructor(private readonly foo: Foo)`, declares that the class has a `foo` field.

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;
  }
}
```

Comments when calling a function

If needed, document parameters at call sites inline using block comments. Also consider named parameters using object literals and destructuring. The exact formatting and placement of the comment is not prescribed.

```
// Inline block comments for parameters that'd be hard to understand:
new Percolator().brew(/* amountLitres= */ 5);
// Also consider using named arguments and destructuring parameters (in brew's declaration):
new Percolator().brew({amountLitres: 5});
```

```
/** An ancient {@link CoffeeBrewer} */
export class Percolator implements CoffeeBrewer {
  /**
   * Brews coffee.
```

```
    * @param amountLitres The amount to brew. Must fit the pot size!
    */
  brew(amountLitres: number) {
    // This implementation creates terrible coffee, but whatever.
    // TODO(b/12345): Improve percolator brewing.
  }
}
```

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 {}
```

Language Rules

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's a property which defaults t
}
```

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

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

See also export visibility below.

Constructors

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

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

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

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, 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() {}
}
```

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[] = [];
}
```

Properties used outside of class lexical scope

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

Prefer `public` visibility for these properties, however `protected` visibility can also be used as needed. For example, Angular and Polymer template properties should use `public`, but AngularJS should use `protected`.

TypeScript code must not 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 may 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 (Accessors)

Getters and setters for class members may be used. The getter method must be a pure function (i.e., result is consistent and has no side effects). 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;
  }
}
```

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;
  }
}
```

Primitive Types & Wrapper Classes

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

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

```
const s = 'hello';
const b = false;
const n = 5;
```

Array constructor

TypeScript code must 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);
```

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}`;
```

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. These cases may require special handling.

```
const aNumber = Number('123');
if (isNaN(aNumber)) throw new Error(...);  // Handle NaN if the string might not contain a number
```

```
assertFinite(aNumber, ...);               // Optional: if NaN cannot happen because it was validated
```

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 must also 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 must parse using a radix *must* check that its input is a number 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);
```

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) {...}
```

Code may use explicit comparisons:

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

Variables

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 causes bugs.
```

Variables must not be used before their declaration.

Exceptions

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.');
```

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
}
```

Iterating containers

Do not use `for (... in ...)` to iterate over arrays. It will counterintuitively give the array's indices (as strings!), not values:

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

Use `for (... of someArr)` or vanilla `for` loops with indices to iterate over arrays.

```
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 form.
  const x = someArr[i];
  // ...
}
for (const [i, x] of someArr.entries()) {
  // Alternative version of the above.
}
```

Do not use `Array.prototype.forEach`, `Set.prototype.forEach`, and `Map.prototype.forEach`. They make code harder to debug and defeat some useful compiler checks (e.g. reachability).

```
someArr.forEach((item, index) => {
  someFn(item, index);
});
```

Why?
Consider this code:

```
let x: string|null = 'abc';
myArray.forEach(() => { x.charAt(0); });
```

You can recognize that this code is fine: `x` isn't null and it doesn't change before it is accessed. But the compiler cannot know that this `.forEach()` call doesn't hang on to the closure that was passed in and call it at some later point, maybe after `x` was set to null, so it flags this code as an error. The equivalent for-of loop is fine.

See the error and non-error in the playground

In practice, variations of this limitation of control flow analysis show up in more complex codepaths where it is more surprising.

Using the spread operator

Using the spread operator `[...foo]; {...bar}` is a convenient shorthand for copying arrays and objects. When using the spread operator on objects, 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 the spread operator, the value being spread must match what is being created. That is, when creating an object, only objects may be used with the spread operator; when creating an array, only spread iterables. Primitives, including `null` and `undefined`, may never be spread.

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

Control flow statements & blocks

Control flow statements spanning multiple lines always use blocks for the containing code.

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

```
if (x)
  x.doFoo();
for (let i = 0; i < x; i++)
  doSomethingWithALongMethodName(i);
```

The exception is that `if` statements fitting on one line may elide the block.

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

Switch Statements

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

```
switch (x) {
  case Y:
```

```
    doSomethingElse();
    break;
  default:
    // nothing to do.
}
```

Non-empty statement groups (`case ...`) may 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.
}
```

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.
}
```

Function Declarations

Use `function foo() { ... }` to declare named functions, including functions in nested scopes, e.g. within another function.

Use function declarations instead of assigning a function expression into a local variable (~~`const x = function() {...};`~~). TypeScript already disallows rebinding functions, so preventing overwriting a function declaration by using `const` is unnecessary.

Exception: Use arrow functions assigned to variables instead of function declarations if the function accesses the outer scope's `this` .

```
function foo() { ... }
```

```
// Given the above declaration, this won't compile:
foo = () => 3;  // ERROR: Invalid left-hand side of assignment expression.

// So declarations like this are unnecessary.
const foo = function() { ... }
```

Note the difference between function declarations (`function foo() {}`) discussed here, and function expressions (~~`doSomethingWith(function() {});`~~) discussed below.

Top level arrow functions *may* be used to explicitly declare that a function implements an interface.

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

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

Function Expressions

Use arrow functions in expressions

Always use arrow functions instead of pre-ES6 function expressions defined with the `function` keyword.

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

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

Function expressions (defined with the `function` keyword) may only be used if code has to dynamically rebind the `this` pointer, but code *should not* rebind the `this` pointer in general. Code in regular functions (as opposed to arrow functions and methods) *should not* access `this`.

Expression bodies vs block bodies

Use arrow functions with expressions or blocks as their body as appropriate.

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

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

  function payMoney(amount: number) {
    // function declarations are fine, but don't access `this` in them.
  }
}
```

Only use an expression body if the return value of the function is actually used.

```
// BAD: use a block ({ ... }) if the return value of the function is not used.
myPromise.then(v => console.log(v));
```

```
// 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);
});
```

Rebinding `this`

Function expressions 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.
document.body.onclick = clickHandler;
```

```
// 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);
```

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 callback
    // 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;
  }
}
```

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 must be uninstalled, 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 properties.
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.listener will
    // keep a reference to `this` because it's bound, causing a memory leak.
    window.removeEventListener('onbeforeunload', this.listener);
  }
  // An arrow function stored in a property is bound to `this` automatically.
  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 uninstalling.
class Component {
  onAttached() {
    // This creates a temporary reference that we won't be able to uninstall
    window.addEventListener('onbeforeunload', this.listener.bind(this));
  }
  onDetached() {
    // This bind creates a different reference, so this line does nothing.
    window.removeEventListener('onbeforeunload', this.listener.bind(this));
  }
  private listener() {
    confirm('Do you want to exit the page?');
  }
}
```

Automatic Semicolon Insertion

Do not rely on Automatic Semicolon Insertion (ASI). Explicitly terminate 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).

@ts-ignore

Do not use `@ts-ignore`. It superficially seems 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.

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 Assertions 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;
```

```
const x = (z as Foo).length;
```

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.
  };
}
```

Member property declarations

Interface and class declarations must use the `;` character to separate individual member declarations:

```
interface Foo {
  memberA: string;
  memberB: number;
}
```

Interfaces specifically must not use the `,` character to separate fields, for symmetry with class declarations:

```
interface Foo {
  memberA: string,
  memberB: number,
}
```

Inline object type declarations must use the comma as a separator:

```
type SomeTypeAlias = {
  memberA: string,
  memberB: number,
};

let someProperty: {memberC: string, memberD: number};
```

Optimization compatibility for property access

Code must not mix quoted property access with dotted property access:

```
// Bad: code must use either non-quoted or quoted access for any property
// consistently across the entire application:
console.log(x['someField']);
console.log(x.someField);
```

Code must not rely on disabling renaming, but must rather declare all properties that are external to the application to prevent renaming:

Prefer for code to account for a possible property-renaming optimization, and declare all properties that are external to the application to prevent renaming:

```
// Good: declaring an interface
declare interface ServerInfoJson {
  appVersion: string;
  user: UserJson;
}
const data = JSON.parse(serverResponse) as ServerInfoJson;
console.log(data.appVersion); // Type safe & renaming safe!
```

Optimization compatibility for module object imports

When importing a module object, directly access properties on the module object rather than passing it around. This ensures that modules can be analyzed and optimized. Treating module imports as namespaces is fine.

```
import {method1, method2} from 'utils';
class A {
  readonly utils = {method1, method2};
}
```

```
import * as utils from 'utils';
class A {
  readonly utils = utils;
}
```

Exception

This optimization compatibility rule applies to all web apps. It does not apply to code that only runs server side (e.g. in NodeJS for a test runner). It is still strongly encouraged to always declare all types and avoid mixing quoted and unquoted property access, for code hygiene.

Enums

Always use `enum` and not `const enum`. 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.

Debugger statements

Debugger statements must not be included in production code.

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

```
}
```

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 line...

  @Input()
  myOtherField: string;  // ... or wrap.
}
```

Source Organization

Modules

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.

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

```
import {Symbol1} from 'google3/path/from/root';
import {Symbol2} from '../parent/file';
import {Symbol3} from './sibling';
```

Namespaces vs Modules

TypeScript supports two methods to organize code: *namespaces* and *modules*, but namespaces are disallowed. google3 code must use TypeScript *modules* (which are ECMAScript 6 modules). 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.

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
}
```

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.

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 change after a second.
// In TS, if foo is re-exported by a second file, importers will not see the value change.
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();
```

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; }
```

Imports

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

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

Module versus destructuring imports

Both module and destructuring imports have advantages depending on the situation.

Despite the `*`, a module import is not comparable to a "wildcard" import as seen in other languages. Instead, module imports give a name to the entire module and each symbol reference mentions the module, which can make code more readable and gives autocompletion on all symbols in a module. They also require less import churn (all symbols are available), fewer name collisions, and allow terser names in the module that's imported. Module imports are particularly useful when using many different symbols from large APIs.

Destructuring imports give local names for each imported symbol. They allow terser code when using the imported symbol, which is particularly useful for very commonly used symbols, such as Jasmine's `describe` and `it`.

```
// Bad: overlong import statement of needlessly namespaced names.
import {TableViewItem, TableViewHeader, TableViewRow, TableViewModel,
  TableViewRenderer} from './tableview';
let item: TableViewItem = ...;
```

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

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

// All tests will use the same three functions repeatedly.
// When importing only a few symbols that are used very frequently, also
// consider importing the symbols directly (see below).
testing.describe('foo', () => {
  testing.it('bar', () => {
    testing.expect(...);
    testing.expect(...);
  });
});
```

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

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

Renaming imports

Code *should* fix name collisions by using a module 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`.

Import & export type

Do not use `import type ... from` or `export type ... from`.

> Note: this does not apply to exporting type definitions, i.e. `export type Foo = ...;`.

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

Instead, just use regular imports:

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

TypeScript tooling automatically distinguishes symbols used as types vs symbols used as values and only generates runtime loads for the latter.

Why?
TypeScript tooling automatically handles the distinction and does not insert runtime loads for type references. This gives a better developer UX: toggling back and forth between `import type` and `import` is bothersome. At the same time, `import type` gives no guarantees: your code might still have a hard dependency on some import through a different transitive path.

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

`export type` might 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.

Organize By Feature

Organize packages by feature, not by type. For example, an online shop *should* have packages named `products`, `checkout`, `backend`, not ~~`views`~~, ~~`models`~~, ~~`controllers`~~.

Type System

Type Inference

Code may rely on type inference as implemented by the TypeScript compiler for all type expressions (variables, fields, return types, etc). The google3 compiler flags reject code that does not have a type annotation and cannot be inferred, so all code is guaranteed to be typed (but might use the `any` type explicitly).

```
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 readability
```

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

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

For more complex expressions, type annotations can help with readability of the program. Whether an annotation is required is decided by the code reviewer.

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.

Null vs Undefined

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

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.

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 { ... };
}
```

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

class CoffeeService {
  getLatte(): CoffeeResponse {
    return assert(fetchResponse(), 'Coffee maker is broken, file a ticket');
  };
}
```

Optionals vs `|undefined` type

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 = '';
}
```

Structural Types vs Nominal 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.

Use structural typing where appropriate in your code. Outside of test code, use interfaces to define structural types, not classes. In test code it can be useful to have mock implementations structurally match the code under test without introducing an extra interface.

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',
}
```

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 the
 * TypeScript compiler errors here, which may be very far from where 'cat' is
 * defined.
 */
makeSound(cat);

/**
 * Horse has a structural type and the type error shows here rather than the
 * 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);
```

Interfaces vs Type 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 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.”

`Array<T>` Type

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

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

This also applies for `readonly T[]` vs `ReadonlyArray<T>`.

```
const a: string[];
const b: readonly string[];
const c: ns.MyObj[];
const d: Array<string|number>;
const e: ReadonlyArray<string|number>;
```

```
const f: Array<string>;            // the syntax sugar is shorter
const g: ReadonlyArray<string>;
const h: {n: number, s: string}[]; // the braces/parens make it harder to read
const i: (string|number)[];
const j: readonly (string|number)[];
```

Indexable (`{[key: string]: number}`) Type

In JavaScript, it's common to use an object as an associative array (aka “map”, “hash”, or “dict”):

```
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.

Mapped & 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 shoeSize.
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.

`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

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 represent
// they don't care what type the user is operating on (but note "Return type
// 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>.
}
```

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 cannot
// 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

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 BookService;
// Shopping cart is not used in this test
// tslint:disable-next-line:no-any
const component = new MyComponent(mockBookService, /* unused ShoppingCart */ null as any);
```

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:
const {host, port} = splitHostPort(userAddress);
```

Wrapper types

There are a few types related to JavaScript primitives that should never 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`).

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.

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.

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.**

   Examples:

   - The `any` type is easy to misuse (is that variable *really* both a number and callable as a function?), so we have recommendations for how to use it.
   - TypeScript `namespace` causes trouble for Closure optimization.
   - Periods within filenames make them ugly/confusing to import from JavaScript.
   - Static functions in classes optimize confusingly, while often file-level functions accomplish the same goal.
   - Users unaware of the `private` keyword will attempt to obfuscate their function names with underscores.
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.

   We should usually match JavaScript style as well, because people often write both languages together.

   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 Closure compilation 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, or google3-wide library changes, 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.