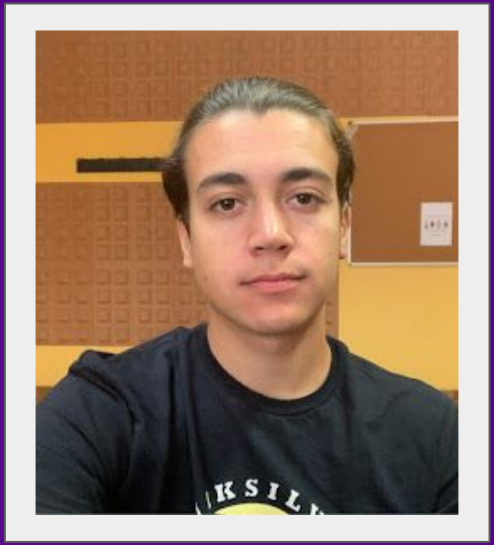
Universidad
de La Laguna

Coding style

05/02/2024

Our Team



Tomás de Armas Déniz

alu0101491776@ull.edu.es

Raúl Álvarez Pérez

alu0101471136@ull.edu.es

Index

Google style for JavaScript and Typescript

Source file

Source file basics:

- File names must be all lowercase
- File names just include '_' or '-' but no additional punctuation
- Source files are encoded in UTF-8

(file-name-example.js)

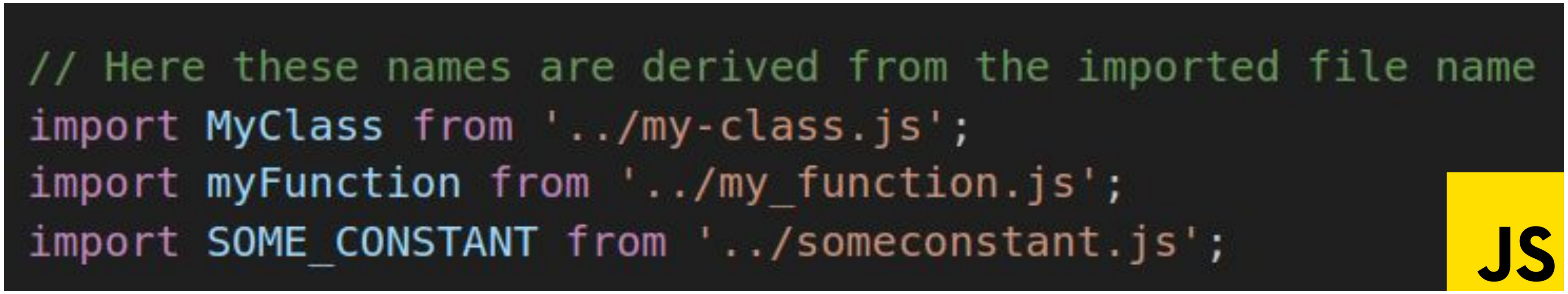
Source file structure:

1. License or copyright information, if present
2. @fileoverview JSDoc, if present
3. ES import statements, if an ES module
4. The file's implementation

ES modules: Imports

Import statements

- Use the import statement to import ES modules.
- In the import path we can't omit the file extension. (f.e. .js)
- Module import names follow the same rules as the type imported.
- Do not import the same file multiple times



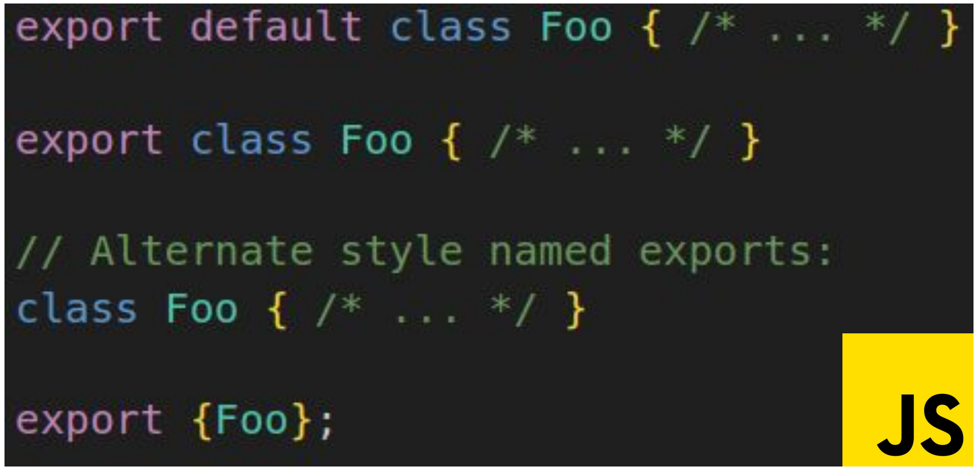
```js
// Here these names are derived from the imported file name
import MyClass from '../my-class.js';
import myFunction from '../my_function.js';
import SOME_CONSTANT from '../someconstant.js';
```

Example (es-modules.js)

ES modules: Exports

Export statements

- When exporting we don't use default exports.
- Do not export classes or objects with static methods or properties.
- DON'T create cycles between ES modules, even though it's allowed. (this stands for imports too)



```
export default class Foo { /* ... */ }

export class Foo { /* ... */ }

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

export {Foo};
```

✗

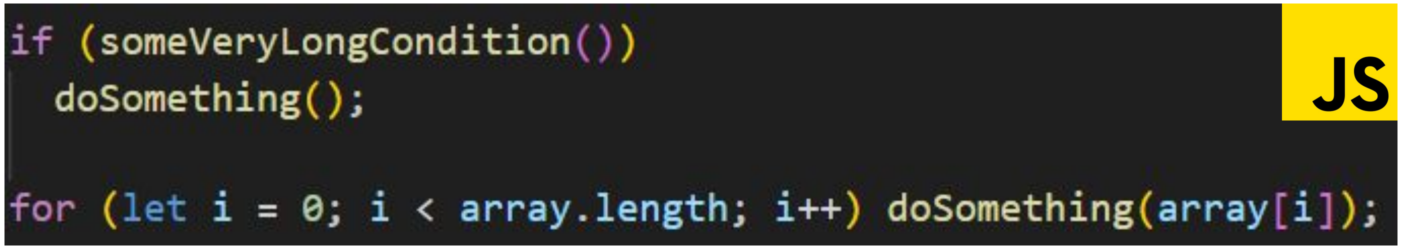
✓

Example (es-modules.js)

Formatting: Braces

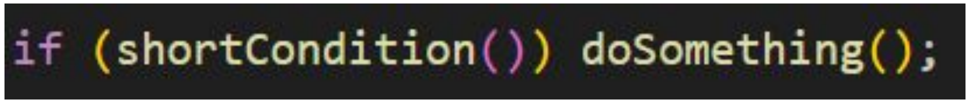
EACH TIME A NEW BLOCK IS OPENED, THE INDENT INCREASE BY 2

All control structures must have braces, except if an if statement can fit entirely on a single line.



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

for (let i = 0; i < array.length; i++) doSomething(array[i]);
```

❌



```
if (shortCondition()) doSomething();
```

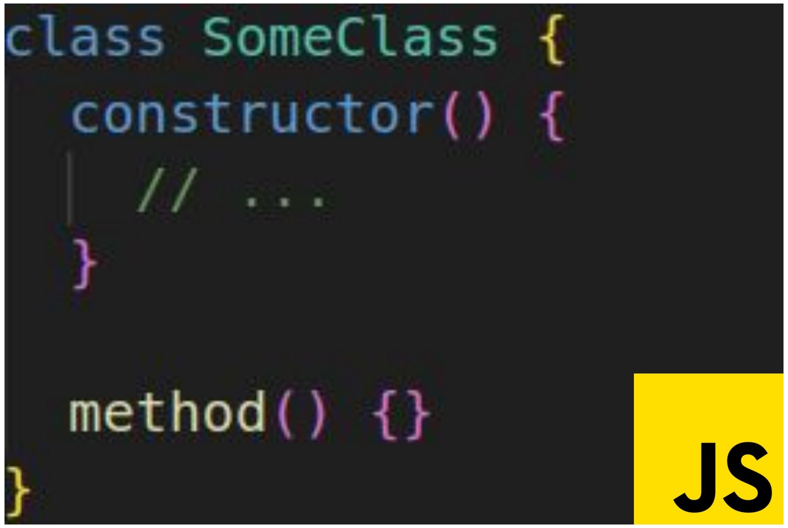
✓

Example (braces-control-structure.js)

Formatting: Braces

When there's an empty block, brace may be closed immediately

No semicolons after methods, or after the closing brace of a class declaration



```
class SomeClass {
  constructor() {
    // ...
  }

  method() {}
}
```

Example (braces-control-structure.js)

Formatting: Statements

One statement per line, each statement is followed by a line-break

SEMICOLONS ARE REQUIRED, every statement must be terminated with a semicolon

*** The column limit in JavaScript is 80 characters ***

When line-wrapping, each line after the first is indented at least +4 from the original line.

Formatting: Vertical whitespaces

A single blank line appears:

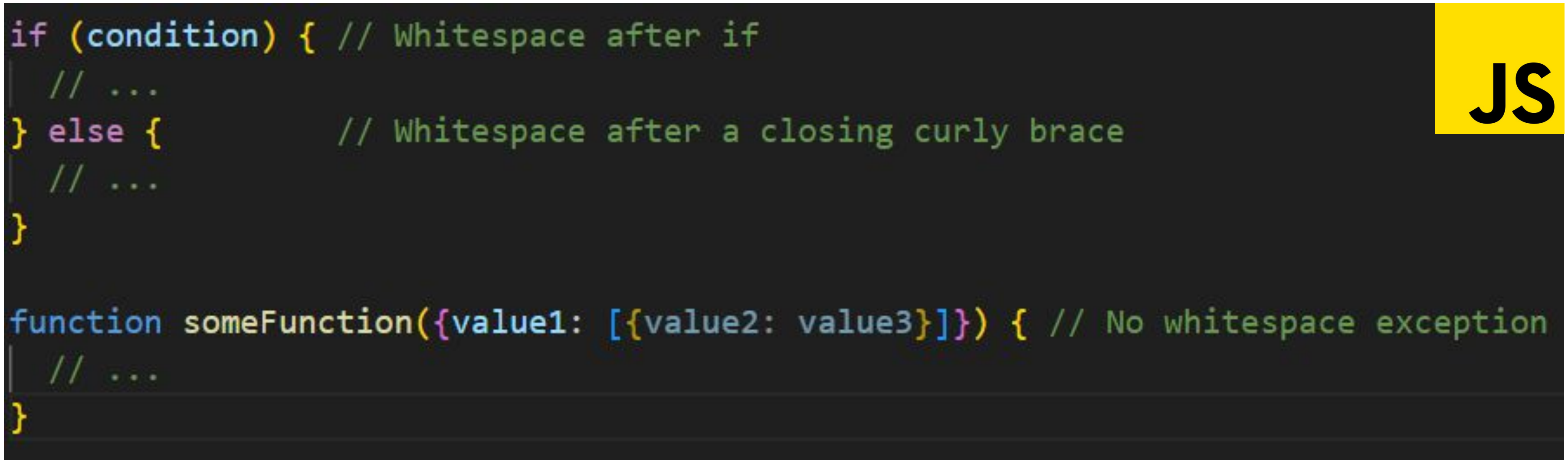
1. **Between consecutive** methods in a class or object literal
2. Within method bodies, to create logical groupings of statements.
3. Optionally **before** the first or **after** the last method in a class or object literal
4. As required by other sections of the Google JavaScript Style Guide

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

Multiple consecutive blank lines are permitted, but never required.
*****WE WAN’T TO SEE CODE NOT WHITESPACES*****

Formatting: Horizontal whitespaces

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



```
if (condition) { // Whitespace after if
  // ...
} else {         // Whitespace after a closing curly brace
  // ...
}

function someFunction({value1: [{value2: value3}]}) { // No whitespace exception
  // ...
}
```

JS

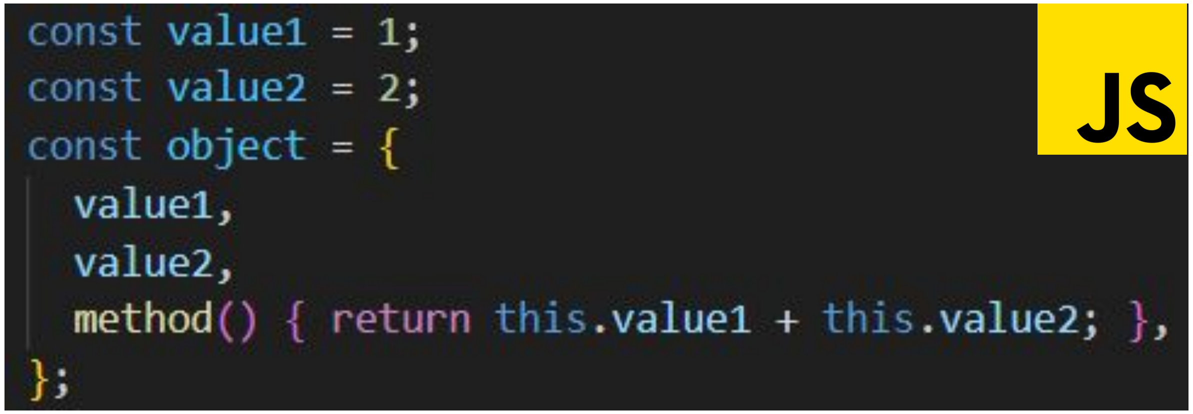
Example (horizontal-whitespaces.js)

Features: Local variables

JUST USE const **AND** let

Only **one** variable per declaration

The declaration must be as near as possible to the point they're first used



```
const value1 = 1;
const value2 = 2;
const object = {
  value1,
  value2,
  method() { return this.value1 + this.value2; },
};
```

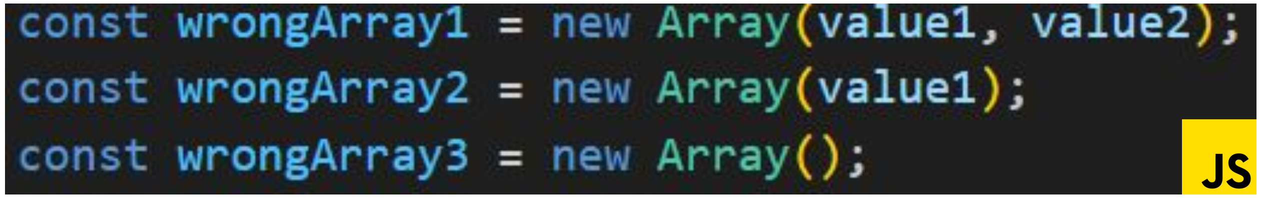
Example (shorthand-properties.js)

Features: Array literals

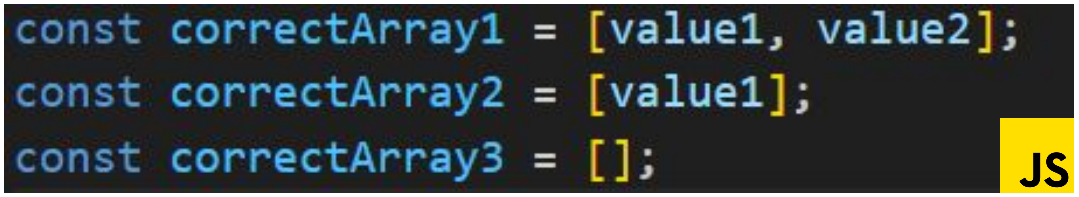
Is permitted the use of trailing commas

Do **not** use the variadic **Array constructor**

Do **not** use non-numeric properties on an array



```js
const wrongArray1 = new Array(value1, value2);
const wrongArray2 = new Array(value1);
const wrongArray3 = new Array();
```



```js
const correctArray1 = [value1, value2];
const correctArray2 = [value1];
const correctArray3 = [];
```

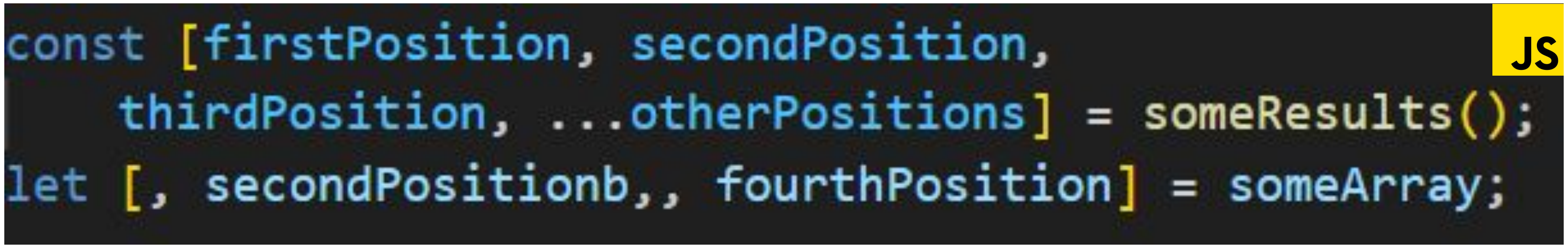
Example (array-constructor.js)

Features: Array literals

Destructuring

Destructuring in JavaScript, is similar to unpacking values from arrays, or properties from objects, into distinct variables.

When destructuring we must add a final “rest” element with no space between the ... and the variable name. Whether we’re not using an element we should omit it.



```
const [firstPosition, secondPosition,
    thirdPosition, ...otherPositions] = someResults();
let [, secondPositionb,, fourthPosition] = someArray;
```

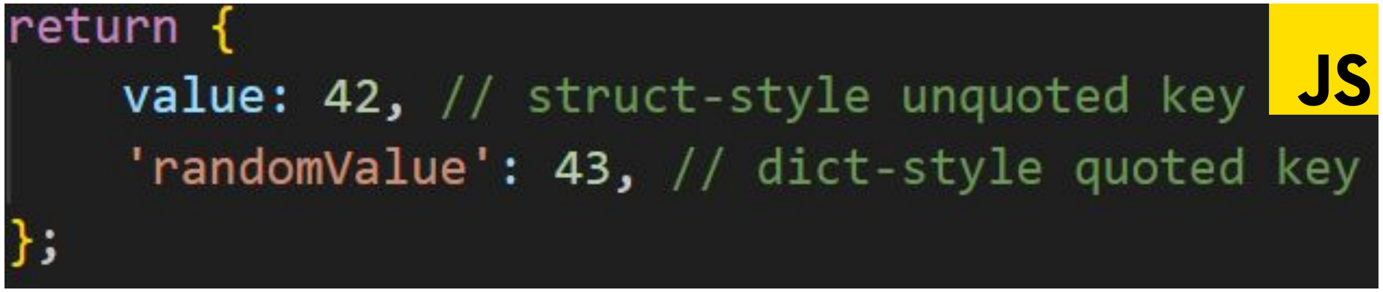
Example (destructuring.js)

Features: Object literals

Disallowed to use the constructor (new Object( )).

Methods can be defined on object literals using method shorthand, basically short declarations.

Do **not** mix quoted and unquoted keys.



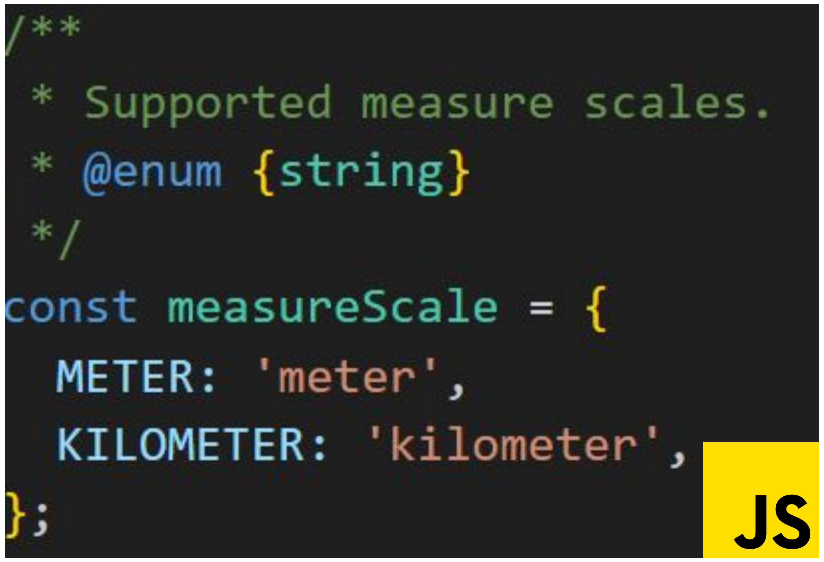
```
return {
    value: 42, // struct-style unquoted key
    'randomValue': 43, // dict-style quoted key
};
```

Example (quoted-unquoted.js)

Features: Object literals

Enums

Additional properties may not be added to an enum after it is defined. Enums must be constant and all enum values must be deeply immutable



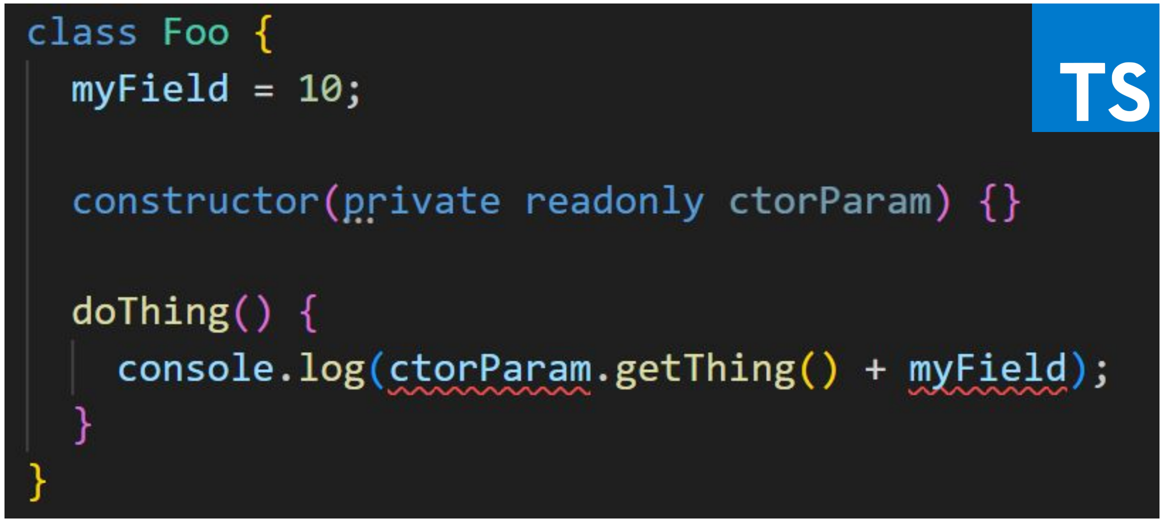
```
/**
 * Supported measure scales.
 * @enum {string}
 */
const measureScale = {
  METER: 'meter',
  KILOMETER: 'kilometer',
};
```

Example (enums.js)

Features: Classes: Constructor

Constructor should be separated from surrounding code both above and below by a single blank line.

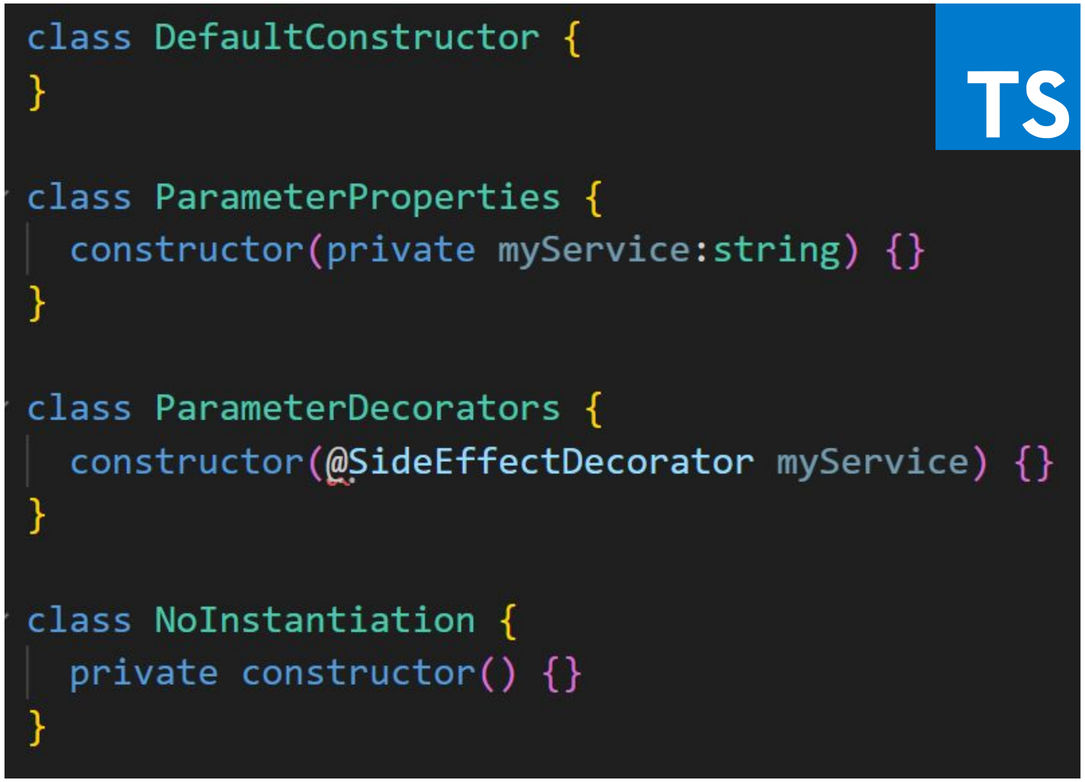
It is unnecessary to provide an empty constructor or one that simply delegates into its parent class. ES2015 provides a default class constructor.



```
class Foo {
  myField = 10;

  constructor(private readonly ctorParam) {}

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

Features: Classes: Constructor

However constructors with parameter properties, visibility modifiers or parameter decorators should not be omitted even if the body of the constructor is empty.

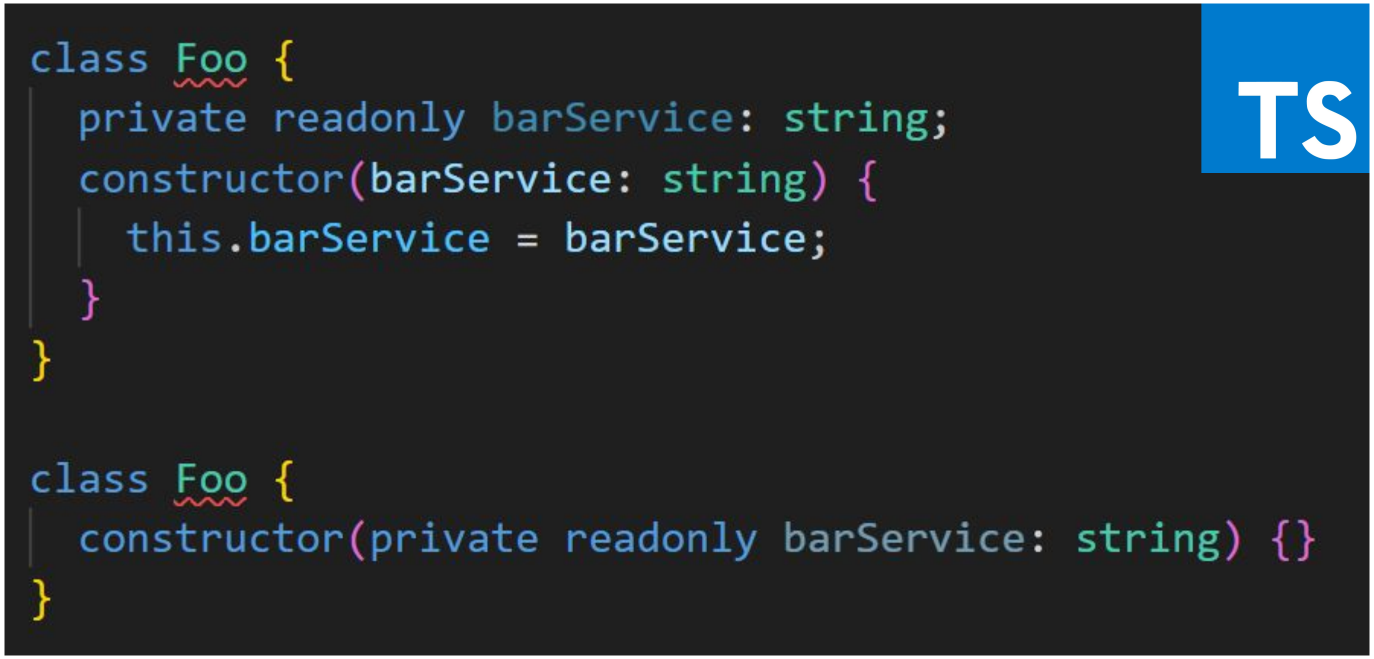


```
class DefaultConstructor {
}

class ParameterProperties {
  constructor(private myService:string) {}
}

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

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

Features: Classes: Properties

Mark properties that are never reassigned outside of the constructor with the **readonly** modifier.

You can skip an obvious initializer using parameter property.

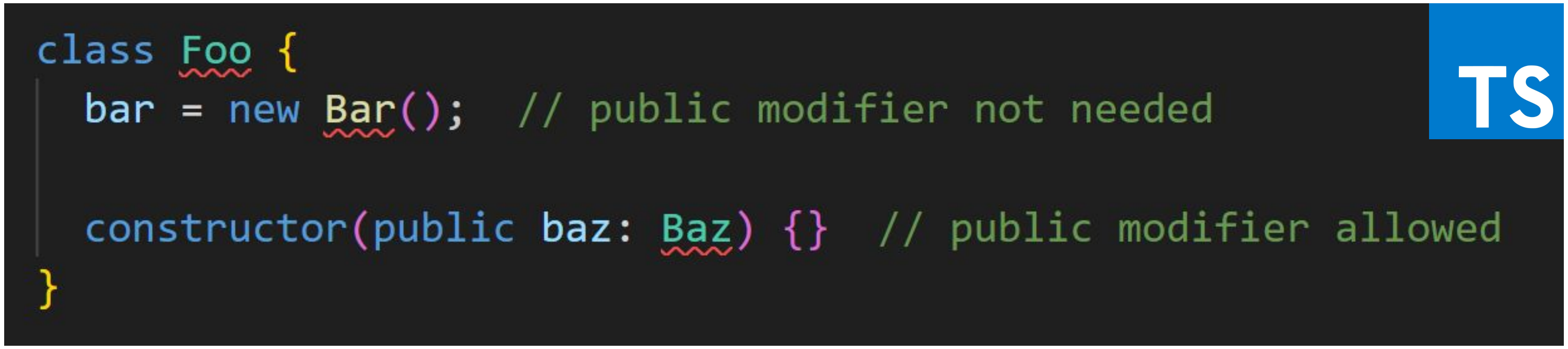


```
class Foo {
  private readonly barService: string;
  constructor(barService: string) {
    this.barService = barService;
  }
}

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

Features: Classes: Properties

Visibility: (**private**, **public**, **protected**)

- Properties used outside of class lexical scope must not use **private** visibility.
- TypeScript symbols are **public by default**. Use public only for non-readonly public parameters properties in constructors.



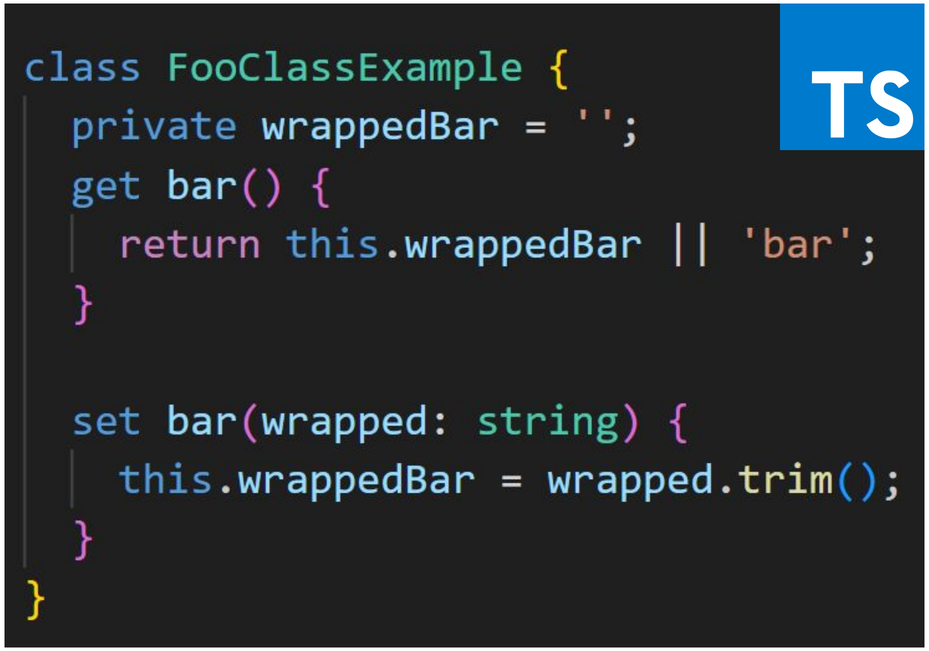
```
class Foo {
  bar = new Bar();  // public modifier not needed

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

Features: Classes: Getter and Setter

A getter must be a pure function (consistent and non side effects).

When an accessor is used to hide a class property, the property may be prefixed or suffixed with **internal** or **wrapped**.

*** Google Style Guide says that assessors must be non-trivial, but this contradicts the principles of OOP. ***



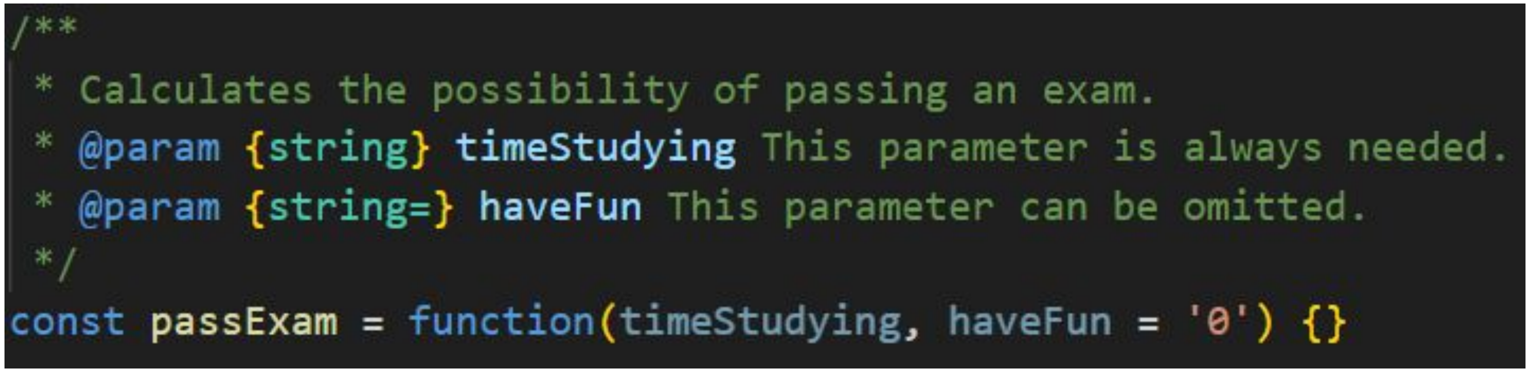
```
class FooClassExample {
  private wrappedBar = '';
  get bar() {
    return this.wrappedBar || 'bar';
  }

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

Features: Functions

When we declare a nested function this should be assigned to a local const

All **optional parameters** must have default values, in abstract and interface methods **default parameter** values are omitted

Prefer **arrow functions** over the function keyword, particularly for nested functions



```
/**
 * Calculates the possibility of passing an exam.
 * @param {string} timeStudying This parameter is always needed.
 * @param {string=} haveFun This parameter can be omitted.
 */
const passExam = function(timeStudying, haveFun = '0') {}
```

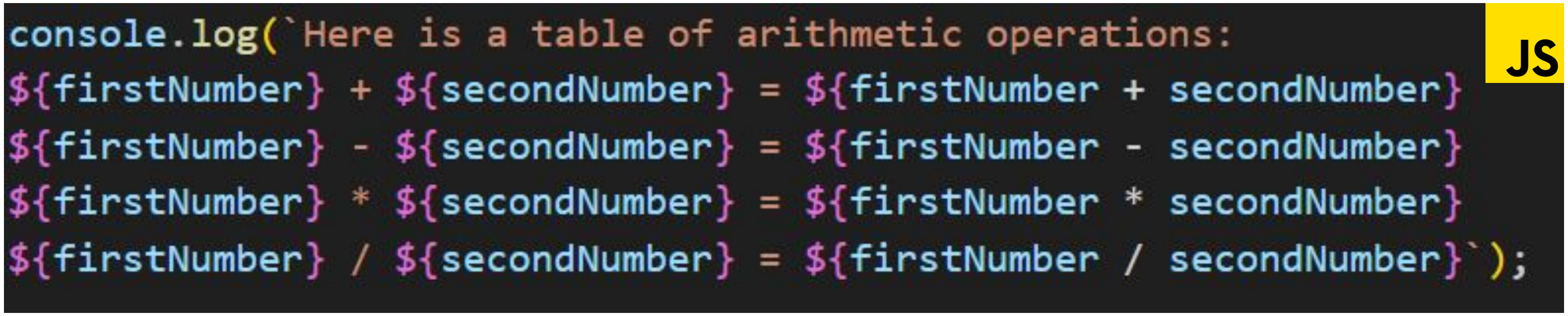
Example (optional-parameters.js)

Features: String literals

We use **single quotes ' '** rather than double quotes " ".

Use template literals (delimited with `) over complex string concatenation, particularly if multiple string literals are involved

Do not use backslash to separate new lines in either ordinary or template string literals



```
console.log(`Here is a table of arithmetic operations:
${firstNumber} + ${secondNumber} = ${firstNumber + secondNumber}
${firstNumber} - ${secondNumber} = ${firstNumber - secondNumber}
${firstNumber} * ${secondNumber} = ${firstNumber * secondNumber}
${firstNumber} / ${secondNumber} = ${firstNumber / secondNumber}`);
```

Example (template-literals.js)

Features: Number literals and this

Numbers

Numbers may be specified in decimal, hex, octal or binary. Use exactly **0x**, **0o**, and **0b** prefixes respectively, with lowercase letters

This

Only use this in class constructors and methods, in arrow functions defined within class constructors and methods.

Never use this to refer to the global object

Features: Control structures

For loops

for-in loops may only be used on dict-style objects, and should not be used to iterate over an array.

Prefer *for-of* over *for-in* when possible.

Exceptions

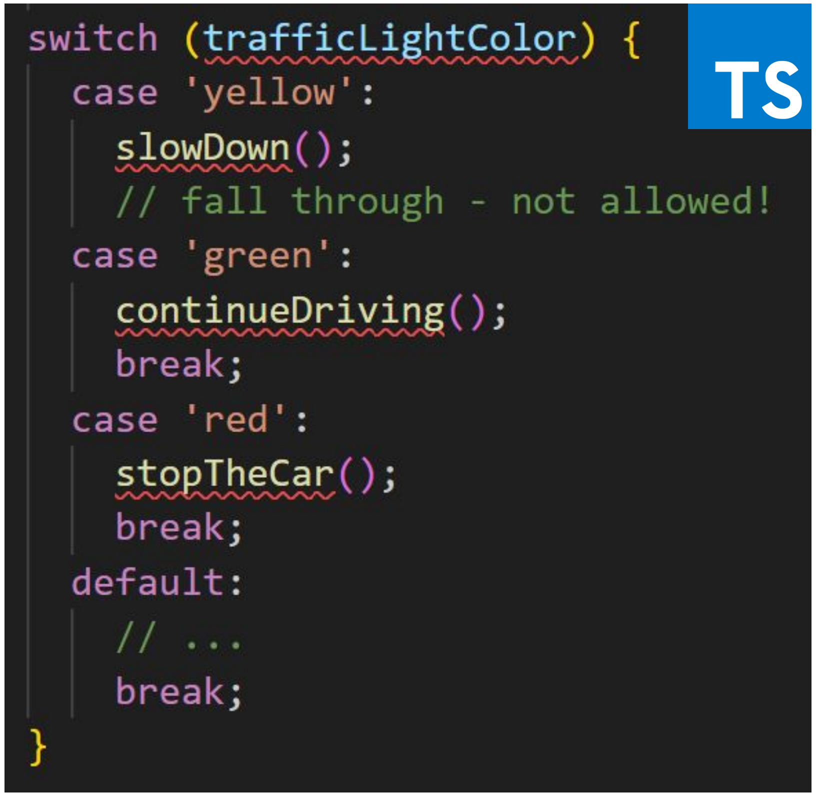
Always throw Errors or subclasses of Error: never throw string literals or other objects.

Always use new when constructing an Error. Catch blocks shouldn’t be empty.

Features: Switch statement

Whenever we want to move to the next case in a switch block we must annotate with // fall through. *default* statement is obligatory

Fall Through allowed in JavaScript and not allowed in Typescript

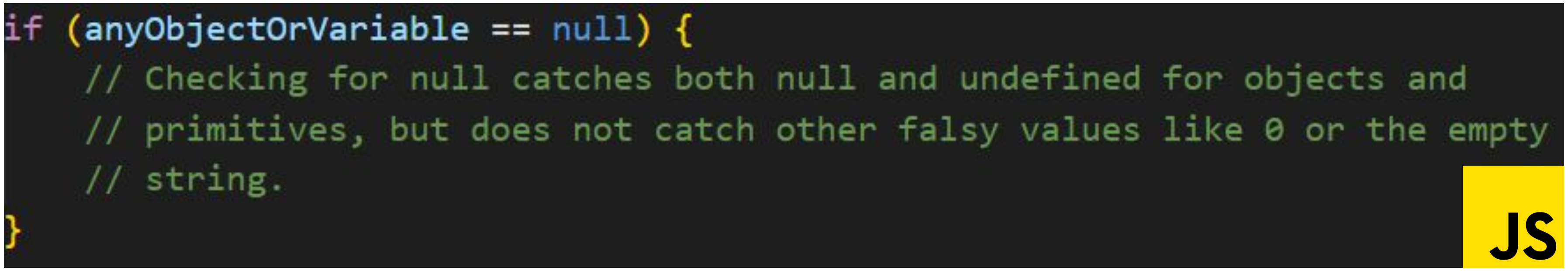


TS

```
switch (trafficLightColor) {
  case 'yellow':
    slowDown();
    // fall through - not allowed!
  case 'green':
    continueDriving();
    break;
  case 'red':
    stopTheCar();
    break;
  default:
    // ...
    break;
}
```

Features: Equality checks

Always use identity operators (===/!==) except when catching both **null and undefined values.**



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

Example (equality-exception.js)

Disallowed features

1. *with* keyword makes your code harder to understand
2. *eval* or the Function(...string) constructor. These features are dangerous and simply do not work in CSP environments.
3. Modifying built-in objects
4. Omit parentheses when invoking a constructor
5. Never use new on the primitive object wrappers

Naming: Introduction

Identifiers use only ASCII letters and digits (except '_' and '$').

Do not worry about saving horizontal space, make your code immediately understandable by a new reader.

Do not use abbreviations that are ambiguous.

The time you spend thinking about a good identifier is **not** time wasted.

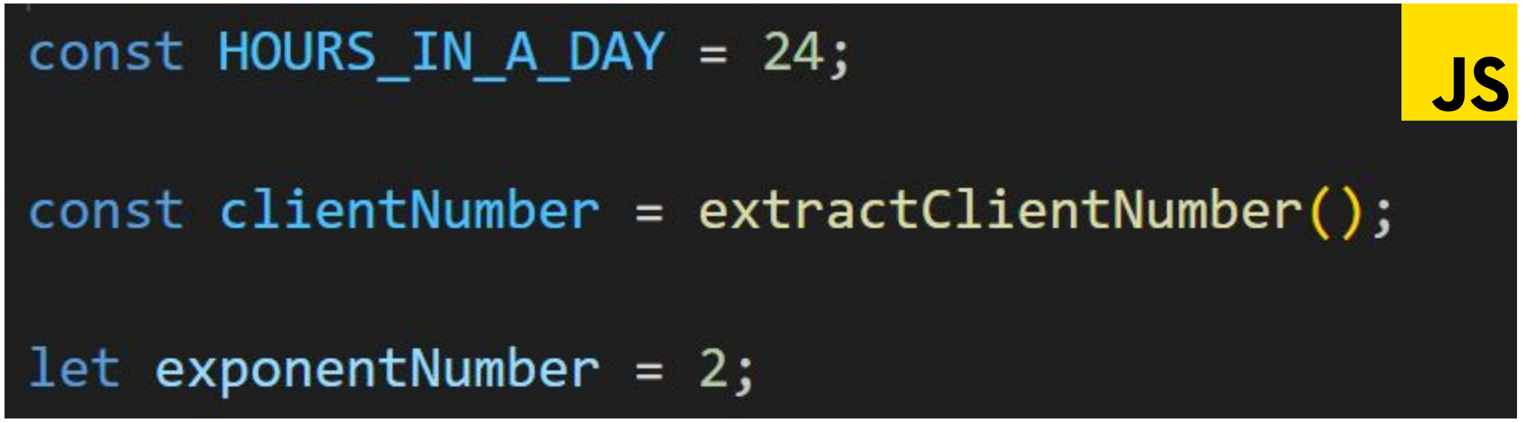
Good examples	Disallowed
errorCount customerId referrerUrl	n // Meaningless nCompCons // Ambiguous abbreviation. kSecondsPerDay // Don't use Hungarian notation

Naming: Variables and constants

Use **lowerCamelCase**.

A "real constant" in JavaScript is one that is initialized using const and its value is not modified during execution time.

For "real constant" use **CONSTANT_CASE**.

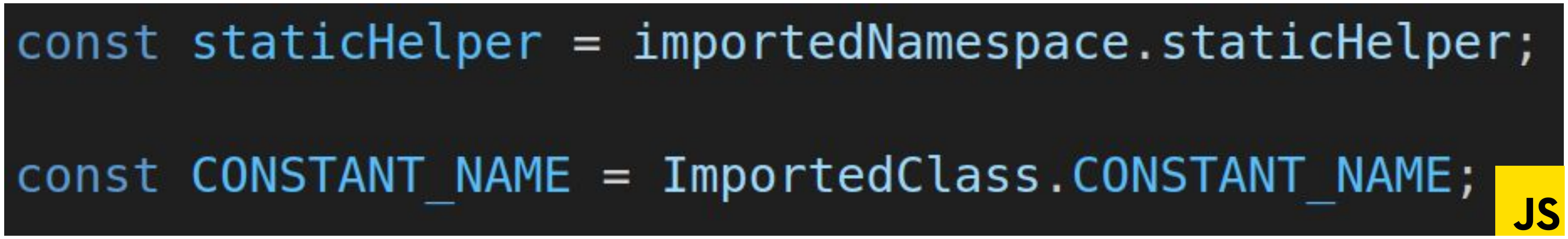


```
const HOURS_IN_A_DAY = 24;

const clientNumber = extractClientNumber();

let exponentNumber = 2;
```

Naming: Local aliases

A local alias is a reference to an object in a specific scope

- A local alias must be const
- Should be used whenever they improve readability
- Try to maintain the last part of the aliased name



```
const staticHelper = importedNamespace.staticHelper;

const CONSTANT_NAME = ImportedClass.CONSTANT_NAME;
```

Naming: Summary

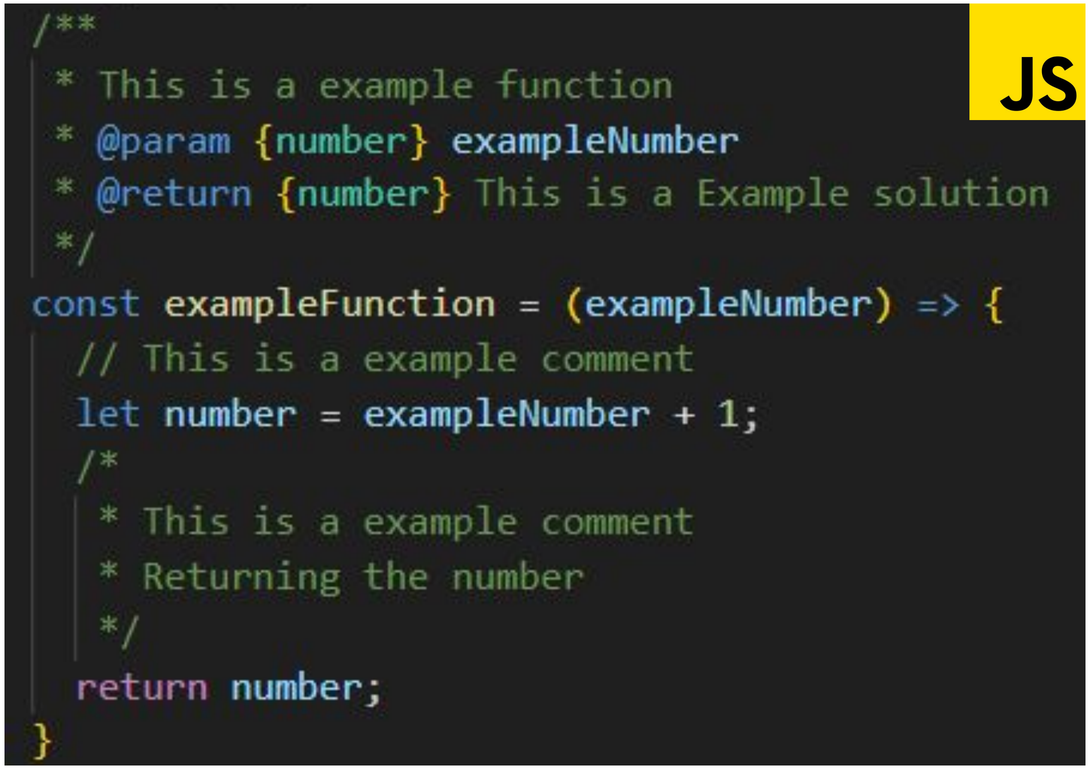
lowerCamelCase	packages functions and methods parameters local variables and local constants
UpperCamelCase	classes, interfaces and records enums names type parameters
CONSTANT_CASE	module-local (global scope) constant items within a enum template parameters names static readonly property of a class

Comments

Use /* ... */ or // ... to add additional information to your code.

JSDoc standard allows the inclusion of special comments in the source code (/** ... */).

DO NOT FORGET THE SPACES



```
/**
 * This is a example function
 * @param {number} exampleNumber
 * @return {number} This is a Example solution
 */
const exampleFunction = (exampleNumber) => {
  // This is a example comment
  let number = exampleNumber + 1;
  /*
   * This is a example comment
   * Returning the number
   */
  return number;
}
```

JSDoc

JSDoc is a standard for commenting and documenting JavaScript code

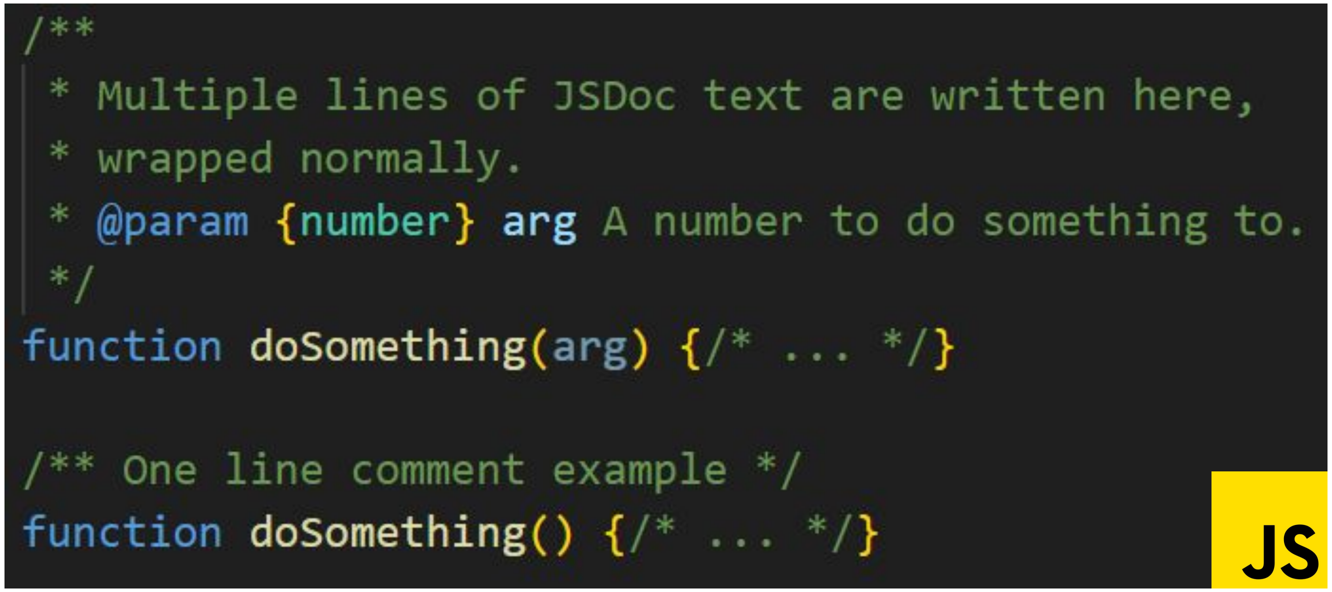
More info: JSDoc

JSDoc: Basic Concepts

JSDoc comments use the '/** … */' structure.

In a multiple-lines comments, you may start one line below.

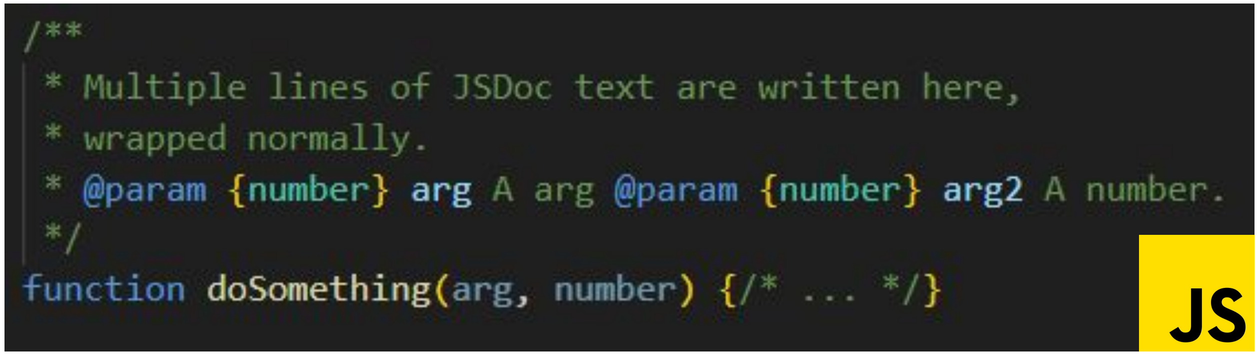
JSDoc is written in Markdown.



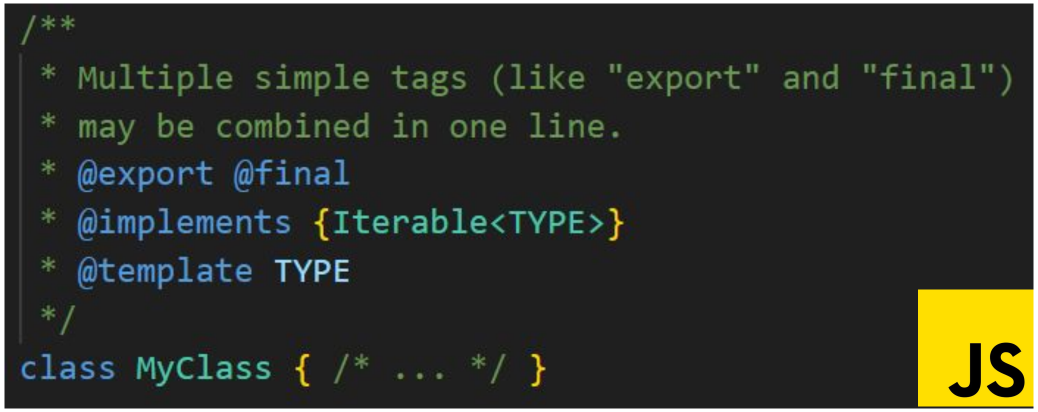
```
/**
 * Multiple lines of JSDoc text are written here,
 * wrapped normally.
 * @param {number} arg A number to do something to.
 */
function doSomething(arg) {/* ... */}

/** One line comment example */
function doSomething() {/* ... */}
```

JSDoc: Basic Concepts

Tags that require any additional data (**@param**, **@return**...) must occupy their own line, with the tag at the beginning of the line.



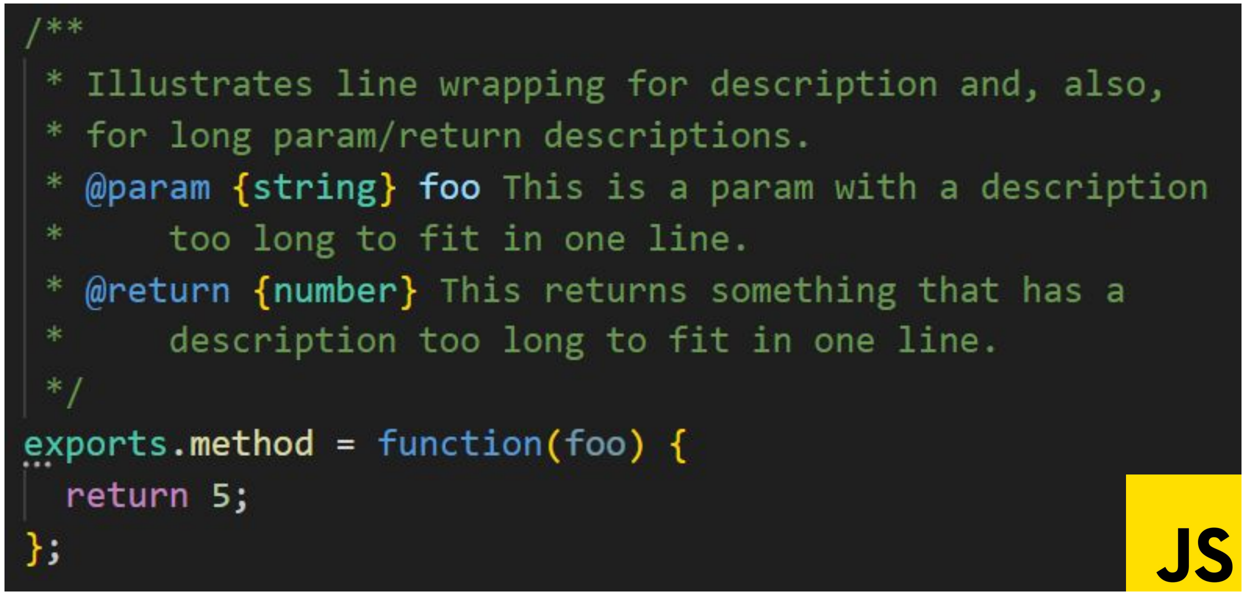
```
/**
 * Multiple lines of JSDoc text are written here,
 * wrapped normally.
 * @param {number} arg A arg @param {number} arg2 A number.
 */
function doSomething(arg, number) {/* ... */}
```



```
/**
 * Multiple simple tags (like "export" and "final")
 * may be combined in one line.
 * @export @final
 * @implements {Iterable<TYPE>}
 * @template TYPE
 */
class MyClass { /* ... */ }
```

JSDoc: Basic Concepts

In the description, **line-wrapped** are not indented and in the block tags are indented four spaces.



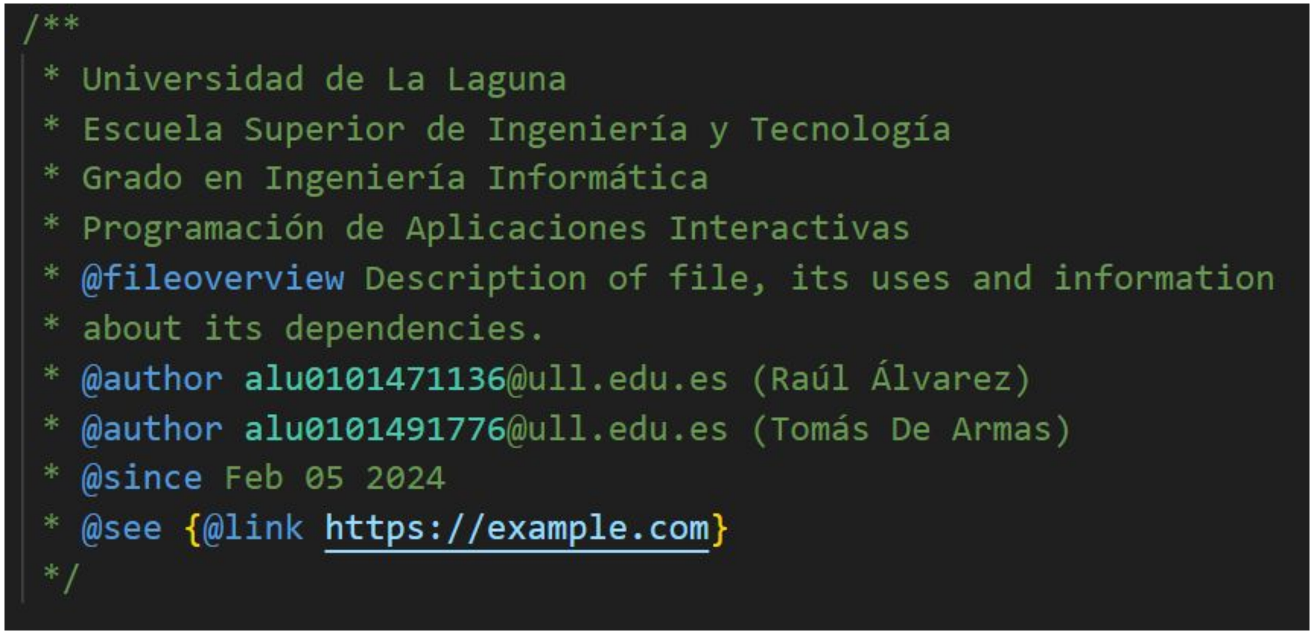
```
/**
 * Illustrates line wrapping for description and, also,
 * for long param/return descriptions.
 * @param {string} foo This is a param with a description
 *     too long to fit in one line.
 * @return {number} This returns something that has a
 *     description too long to fit in one line.
 */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

JSDoc: Top/file-level comment

It has to provide a description of the file's contents and any dependencies or compatibility information.

Use **@fileoverview** whenever a file consists of more than a single class definition. Do not use **@author** without **@fileoverview**.

Wrapped lines are not indented.



```
/**
 * Universidad de La Laguna
 * Escuela Superior de Ingeniería y Tecnología
 * Grado en Ingeniería Informática
 * Programación de Aplicaciones Interactivas
 * @fileoverview Description of file, its uses and information
 * about its dependencies.
 * @author alu0101471136@ull.edu.es (Raúl Álvarez)
 * @author alu0101491776@ull.edu.es (Tomás De Armas)
 * @since Feb 05 2024
 * @see {@link https://example.com}
 */
```

JSDoc: Class comments

The class description should provide the reader with enough information to know how and when to use the class

- **@implements** indicates that implements an interface
- **@extends** is used to indicate that a class or constructor inherits from another class or constructor
- **@record** for record type and **@interface** for interfaces

See: Class comments

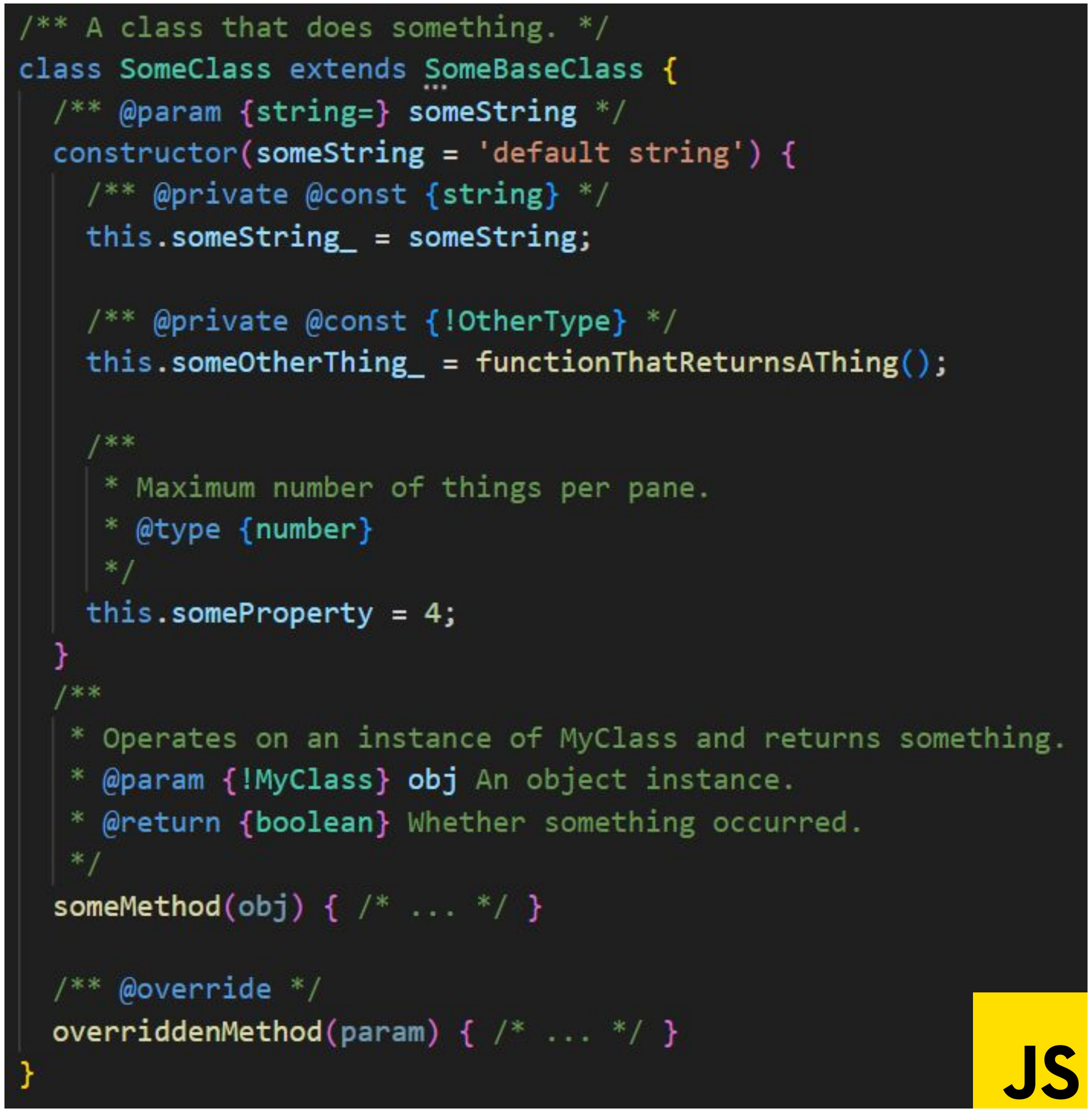
JSDoc: Methods and properties comments

Constructors do not need a description, private properties either, but only if name and type provide enough documentation.

Overridden methods inherit all JSDoc annotations from the super class method. Use **@override** and the rest they should be omitted. However, if any type is redefined, it must be specified.

Method descriptions begin with a verb phrase.
Do not use “This method ....”.

JSDoc: Method and properties comments



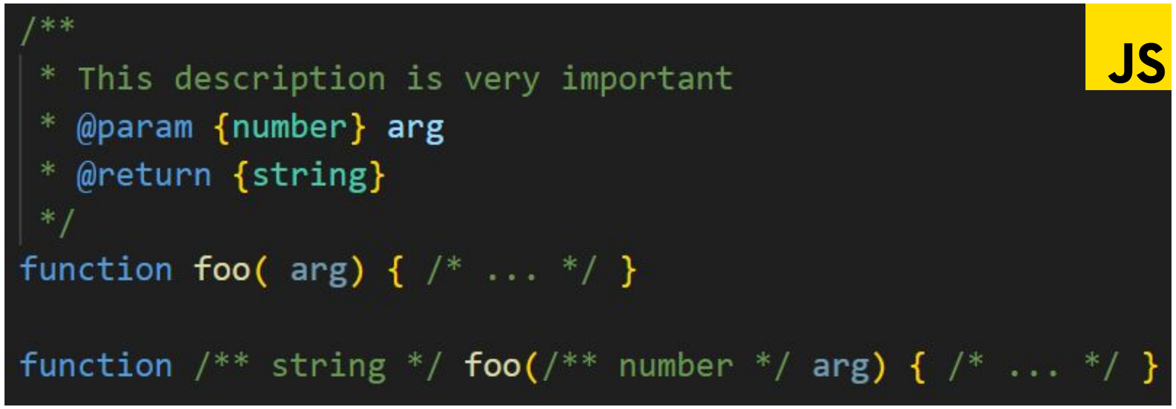
```
/** A class that does something. */
class SomeClass extends SomeBaseClass {
  /** @param {string=} someString */
  constructor(someString = 'default string') {
    /** @private @const {string} */
    this.someString_ = someString;

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

    /**
     * Maximum number of things per pane.
     * @type {number}
     */
    this.someProperty = 4;
  }
  /**
   * Operates on an instance of MyClass and returns something.
   * @param {!MyClass} obj An object instance.
   * @return {boolean} Whether something occurred.
   */
  someMethod(obj) { /* ... */ }

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

JSDoc: Functions comments

@params and **@return** must be documented.

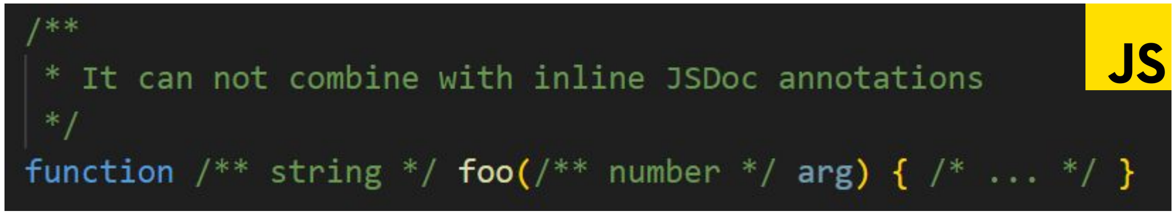
Function, parameter, and return descriptions (but not types) may be omitted if they are obvious. (apply to methods too)

if you omitted the description, you may use inline JSDocs



```js
/**
 * This description is very important
 * @param {number} arg
 * @return {string}
 */
function foo( arg) { /* ... */ }

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

✓



```js
/**
 * It can not combine with inline JSDoc annotations
 */
function /** string */ foo(/** number */ arg) { /* ... */ }
```

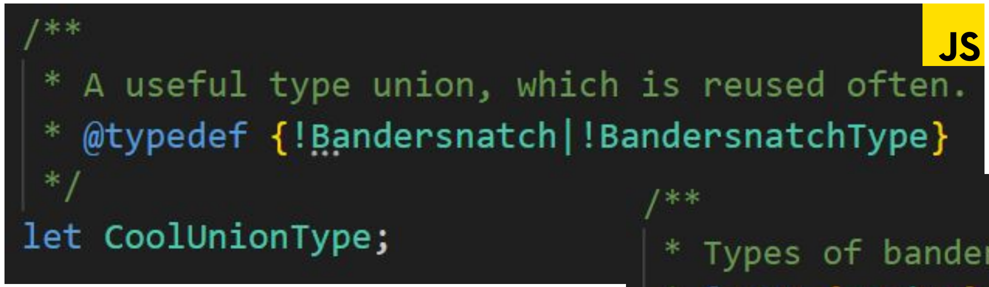
✗

JSDoc: Enum and typedef comments

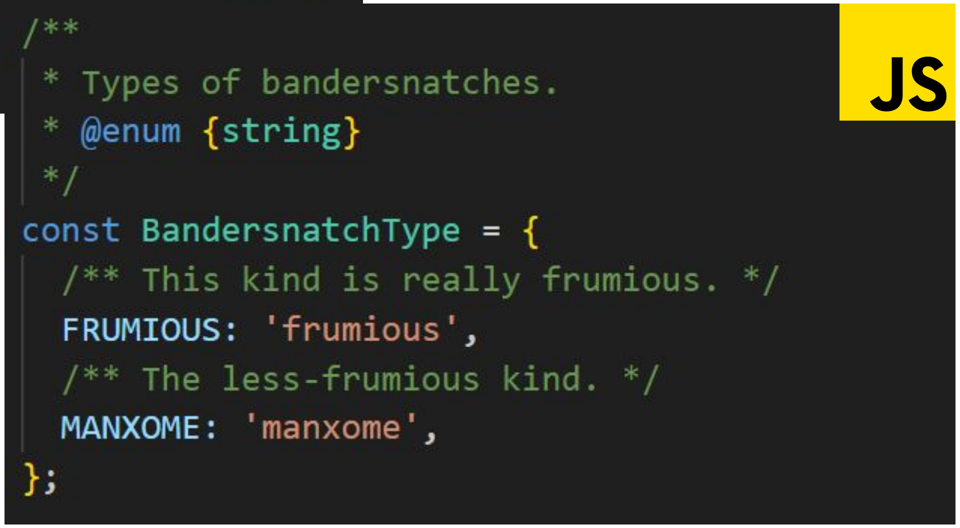
@enum tag is used to document enums

Individual enum items may be documented with a JSDoc comment on the preceding line

@typedef tag is used to document typedefs



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



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

JSDoc: Type annotations

Type annotations must be used with these tags: **@param**, **@return**, **@this**, and **@type**.

Primitives (string, number, boolean, symbol, undefined, null) are in lowerCamelCase

Reference types, refer to a class, enum, array..., are in UpperCamelCase.

Always specify template parameters.

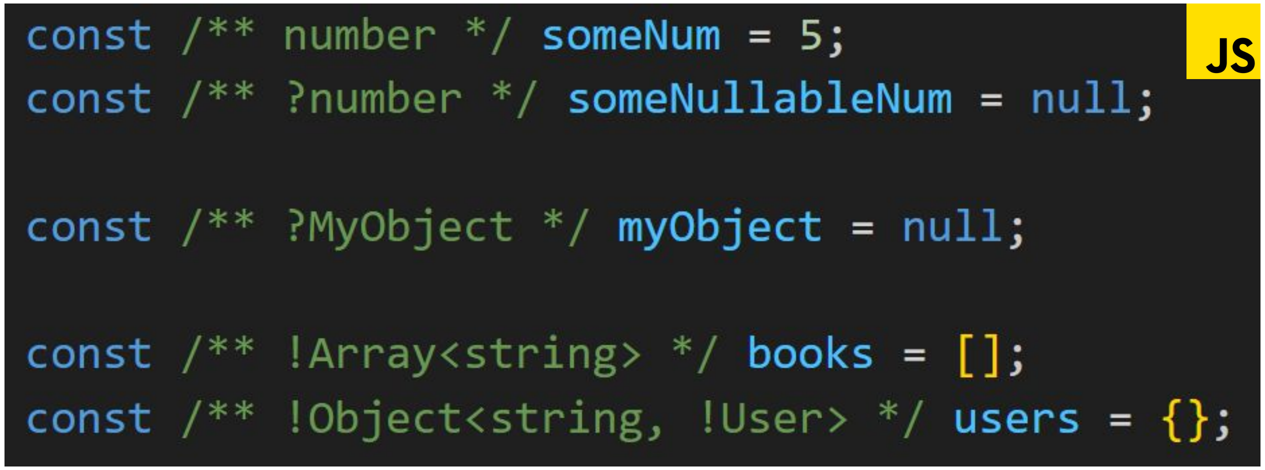
JSDoc: Type annotations: Nullability

Use ‘!’ for non-null and ‘?’ for nullable.

They must precede the type

Primitives are non-nullable by default. ‘!’ is redundant.

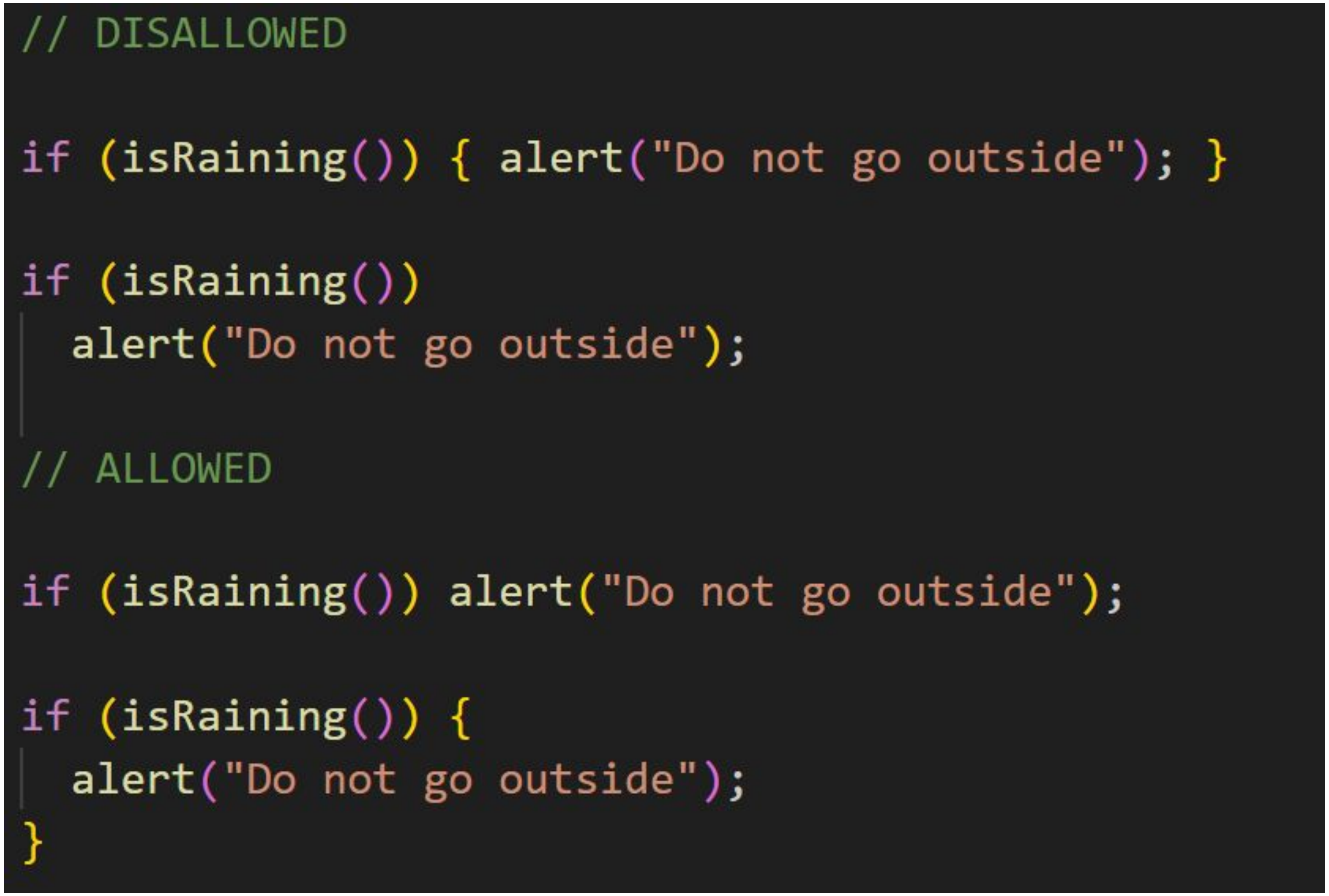
Reference types always need an explicit ‘?’ or ‘!’ to prevent ambiguity.



```
const /** number */ someNum = 5;
const /** ?number */ someNullableNum = null;

const /** ?MyObject */ myObject = null;

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

Good Practices

GP: If statement



```
// DISALLOWED

if (isRaining()) { alert("Do not go outside"); }

if (isRaining())
  alert("Do not go outside");


// ALLOWED

if (isRaining()) alert("Do not go outside");

if (isRaining()) {
  alert("Do not go outside");
}
```

GP: Functions

Try to avoid nesting code too many levels deep.

Make sure that you create functions that fulfill one job at a time.

Do not use global variables, they can be overwritten by other scripts.

We can call a function with fewer parameters than it has, the missing ones set to ‘undefined’ by default. It is a good habit to assign default values to arguments.

GP: Variables

Try to initialize variables when you declare them.

Try to declare Arrays and Objects with const.

Beware of automatic type conversions.

Use the ‘===’ comparison, make the comparison of values without change the type of the variables.

ESLint

ESLint: Enhancing JavaScript Code Quality.

More info: [ESLint]

ESLint: Tutorial

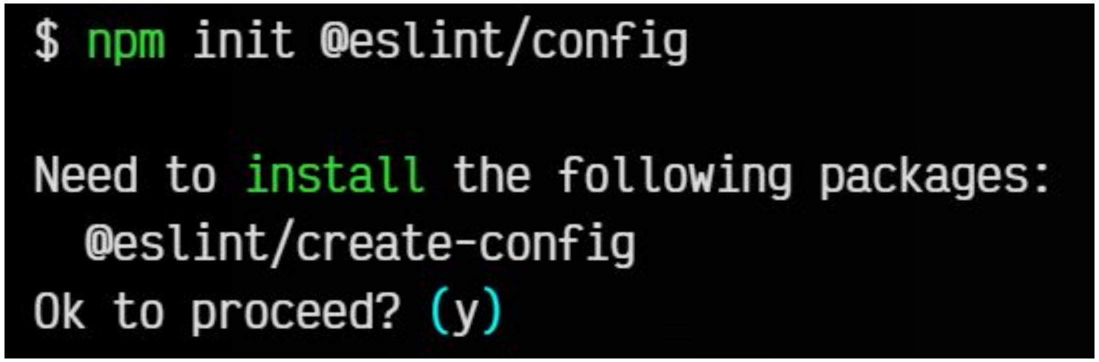
What is ESLint?

ESLint is a powerful linter designed to elevate the quality and style of your JavaScript code. As a linter, it plays a crucial role in ensuring that your code adheres to specific rules and standards.

ESLint: Tutorial

How to install ESLint?

ESLint requires at least Node 12.22+, but we recommend you to update Node to the latest version of Node. When we are ready to go, we can write in the terminal the following command:



```
$ npm init @eslint/config

Need to install the following packages:
  @eslint/create-config
Ok to proceed? (y)
```

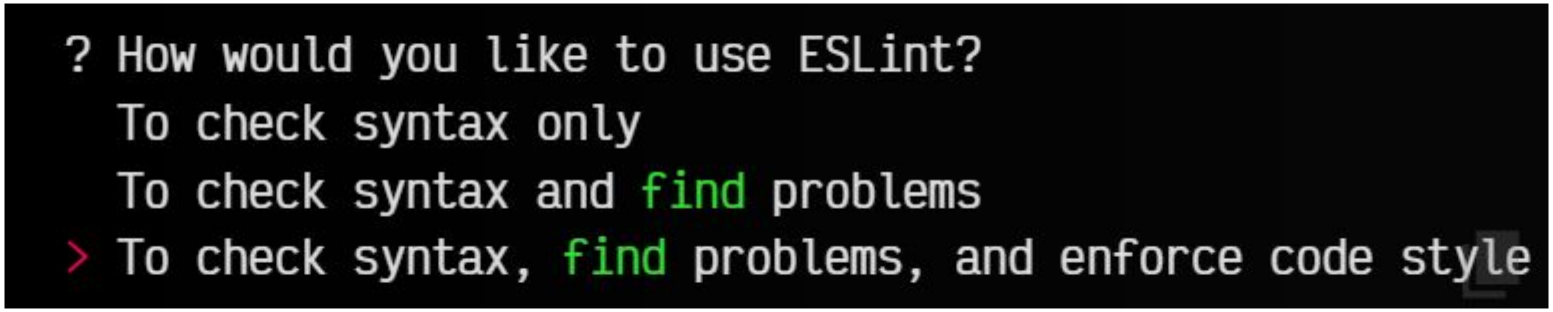
Note: We're covering how to set up ESLint as a terminal tool. However, it might be much more convenient (and even common) to configure it in the code editor we use.

ESLint: Tutorial

Initial configuration

When the executable starts to configure ESLint in our project, it will ask us some questions about the project to review.

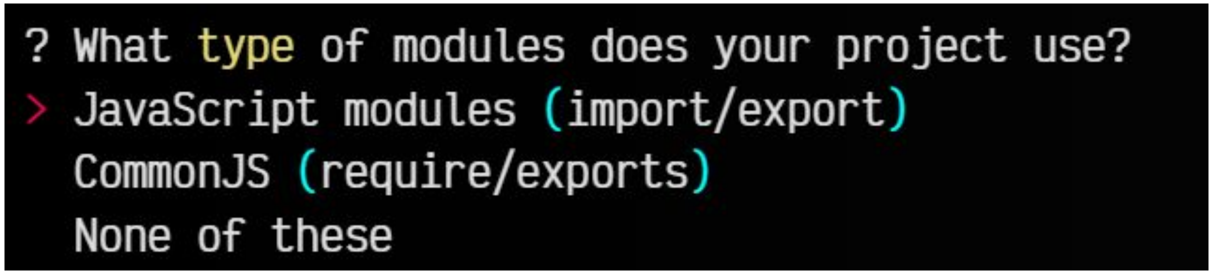
Mode, We can select which mode will use the linter.



```
? How would you like to use ESLint?
  To check syntax only
  To check syntax and find problems
> To check syntax, find problems, and enforce code style
```

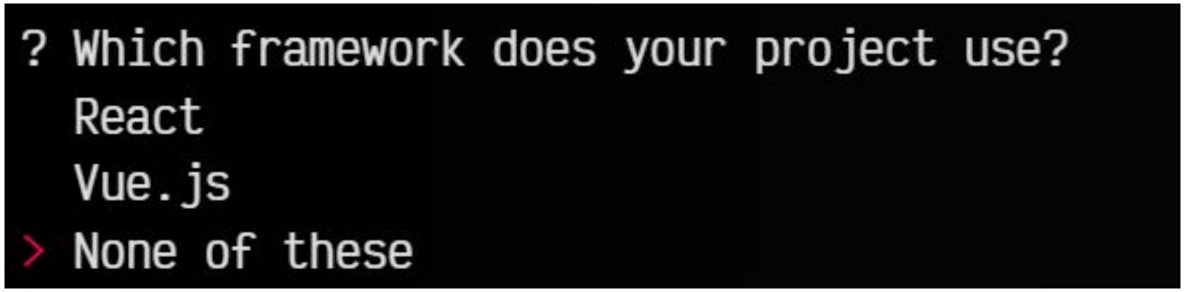
ESLint: Tutorial

JavaScript modules, We can decide the way it import/export your code. (ES-modules vs CommonJS)



```
? What type of modules does your project use?
> JavaScript modules (import/export)
  CommonJS (require/exports)
  None of these
```

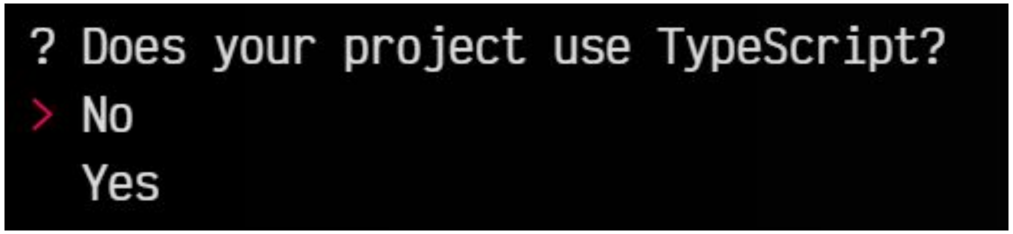
Framework JavaScript, We can use two different frameworks



```
? Which framework does your project use?
  React
  Vue.js
> None of these
```

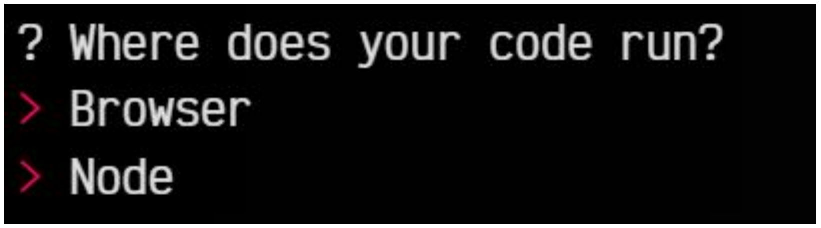
ESLint: Tutorial

TypeScript support, it supports TypeScript too



```
? Does your project use TypeScript?
> No
  Yes
```

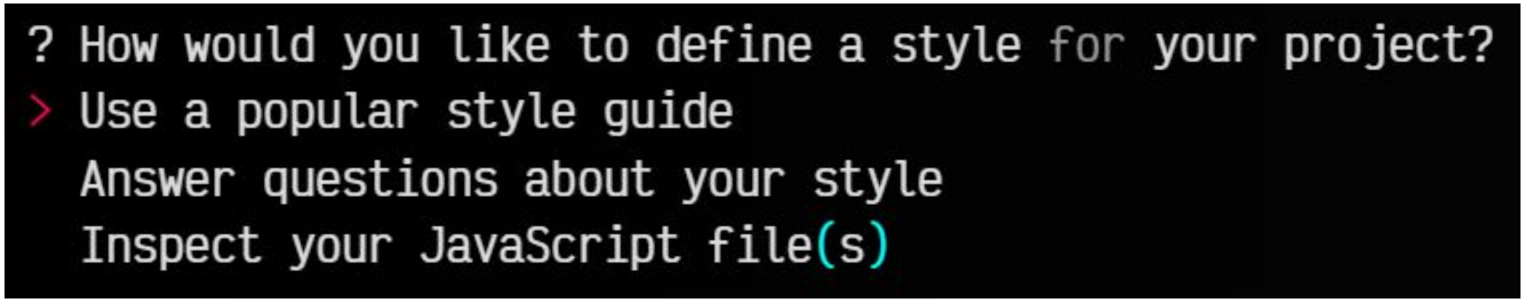
Where your code runs, here we can select both options



```
? Where does your code run?
> Browser
> Node
```

ESLint: Tutorial

Style guide, provides the option to define a style for your project

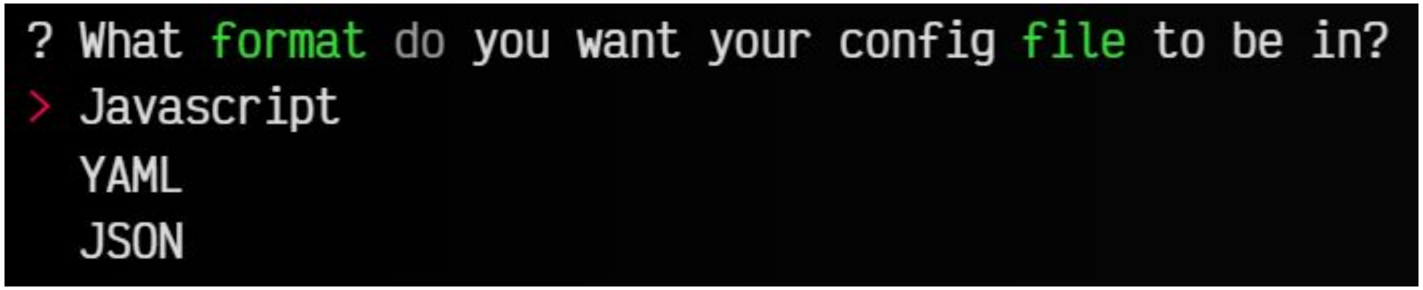


```
? How would you like to define a style for your project?
> Use a popular style guide
  Answer questions about your style
  Inspect your JavaScript file(s)
```

this is the option why we are recommending this tool, we can use a popular style guide (as google style) with ESLint. There are plenty of style guides, link to all the ESLint configuration made by the community

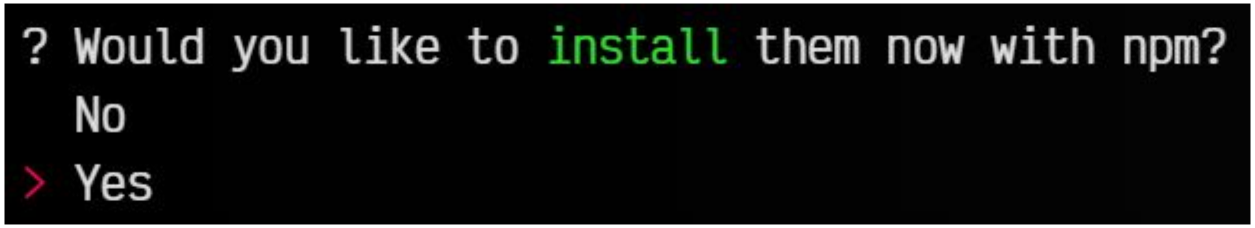
ESLint: Tutorial

Format for config file, The most used are .js and .json



```
? What format do you want your config file to be in?
> Javascript
  YAML
  JSON
```

The final question is if you want to install every npm packet to finalize the installation.



```
? Would you like to install them now with npm?
  No
> Yes
```

ESLint: VSCode

ESLint can be used in your favorite editor as a **plugin** or **extension**. These are some of the **editors** or **built tools** that can use ESLint: Visual Studio Code, IntelliJ IDEA, Sublime Text 3, Gulp, Webpack, RollUp, Vim.

Bibliography

Information used for all the presentation - Coding style

Google Style (JS) - Google JavaScript Style Guide

Google Style (TS) - Google TypeScript Style Guide

ESLint (Tutorial) - ESLint: Linter Javascript

ESLint - https://eslint.org/

npm - https://www.npmjs.com/

Good practices [1] - JavaScript best practices - W3C Wiki

Good practices [2] - Deepsource

Clean Code: A Handbook of Agile Software Craftsmanship - O'reilly

Universidad
de La Laguna