Guidelines for writing JavaScript code examples

The following guidelines cover writing JavaScript example code for MDN Web Docs. This article is a list of rules for writing concise examples that will be understandable by as many people as possible.

General guidelines for JavaScript code examples

This section explains the general guidelines to keep in mind while writing JavaScript code examples. The later sections will cover more specific details.

Choosing a format

Opinions on correct indentation, whitespace, and line lengths have always been controversial. Discussions on these topics are a distraction from creating and maintaining content.

On MDN Web Docs, we use Prettier as a code formatter to keep the code style consistent (and to avoid off-topic discussions). You can consult our configuration file to learn about the current rules, and read the Prettier documentation.

Prettier formats all the code and keeps the style consistent. Nevertheless, there are a few additional rules that you need to follow.

Use modern JavaScript features when supported

You can use new features once every major browser — Chrome, Edge, Firefox, and Safari — supports them (a.k.a. Baseline).

This rule does not apply to the JavaScript feature being documented on the page (which is dictated instead by the criteria for inclusion). For example, you can document non-standard or experimental features and write complete examples demonstrating their behavior, but you should refrain from using these features in the demos for other unrelated features, such as a web API.

Arrays

Array creation

For creating arrays, use literals and not constructors.

Create arrays like this:

JS

```js
const visitedCities = [];
```

✓

Don't do this while creating arrays:

JS

```js
const visitedCities = new Array(length);
```

✗

Item addition

When adding items to an array, use `push()` and not direct assignment. Consider the following array:

JS

```js
const pets = [];
```

Add items to the array like this:

JS

```js
pets.push("cat");
```

✓

Don't add items to the array like this:

JS

```js
pets[pets.length] = "cat";
```

✗

Asynchronous methods

Writing asynchronous code improves performance and should be used when possible. In particular, you can use:

- Promises
- `async` / `await`

When both techniques are possible, we prefer using the simpler `async` / `await` syntax. Unfortunately, you can't use `await` at the top level unless you are in an ECMAScript module. CommonJS modules used by Node.js are not ES modules. If your example is intended to be used everywhere, avoid top-level `await`.

Comments

Comments are critical to writing good code examples. They clarify the intent of the code and help developers understand it. Pay special attention to them.

- If the purpose or logic of the code isn't obvious, add a comment with your intention, as shown below:

  JS

  ```js
  let total = 0;

  // Calculate the sum of the four first elements of arr
  for (let i = 0; i < 4; i++) {
    total += arr[i];
  }
  ```

  On the other hand, restating the code in prose is not a good use of comments:

  JS

  ```js
  let total = 0;

  // For loop from 1 to 4
  for (let i = 0; i < 4; i++) {
    // Add value to the total
    total += arr[i];
  }
  ```

- Comments are also not necessary when functions have explicit names that describe what they're doing. Write:

JS

```js
closeConnection();
```

✓

Don't write:

JS

```js
closeConnection(); // Closing the connection
```

✗

Use single-line comments

Single-line comments are marked with `//`, as opposed to block comments enclosed between `/* … */`.

In general, use single-line comments to comment code. Writers must mark each line of the comment with `//`, so that it's easier to notice commented-out code visually. In addition, this convention allows to comment out sections of code using `/* … */` while debugging.

- Leave a space between the slashes and the comment. Start with a capital letter, like a sentence, but don't end the comment with a period.

JS

```js
// This is a well-written single-line comment
```

✓

- If a comment doesn't start immediately after a new indentation level, add an empty line and then add the comment. It will create a code block, making it obvious what the comment refers to. Also, put your comments on separate lines preceding the code they are referring to. This is shown in the following example:

JS

```
function checkout(goodsPrice, shipmentPrice, taxes) {
  // Calculate the total price
  const total = goodsPrice + shipmentPrice + taxes;

  // Create and append a new paragraph to the document
  const para = document.createElement("p");
  para.textContent = `Total price is ${total}`;
  document.body.appendChild(para);
}
```

Output of logs

- In code intended to run in a production environment, you rarely need to comment when you log some data. In code examples, we often use `console.log()`, `console.error()`, or similar functions to output important values. To help the reader understand what will happen without running the code, you can put a comment *after* the function with the log that will be produced. Write:

  JS

  ```
  function exampleFunc(fruitBasket) {
    console.log(fruitBasket); // ['banana', 'mango', 'orange']
  }
  ```

  Don't write:

  JS

  ```
  function exampleFunc(fruitBasket) {
    // Logs: ['banana', 'mango', 'orange']
    console.log(fruitBasket);
  }
  ```

- In case the line becomes too long, put the comment *after* the function, like this:

JS

```
function exampleFunc(fruitBasket) {
  console.log(fruitBasket);
  // ['banana', 'mango', 'orange', 'apple', 'pear', 'durian', 'lemon']
}
```

✓

Multi-line comments

Short comments are usually better, so try to keep them in one line of 60–80 characters. If this is not possible, use `//` at the beginning of each line:

JS

```
// This is an example of a multi-line comment.
// The imaginary function that follows has some unusual
// limitations that I want to call out.
// Limitation 1
// Limitation 2
```

✓

Don't use `/* … */`:

JS

```
/* This is an example of a multi-line comment.
  The imaginary function that follows has some unusual
  limitations that I want to call out.
  Limitation 1
  Limitation 2 */
```

✗

Use comments to mark ellipsis

Skipping redundant code using ellipses (…) is necessary to keep examples short. Still, writers should do it thoughtfully as developers frequently copy & paste examples into their code, and all of our code samples should be valid JavaScript.

In JavaScript, you should put the ellipses (`…`) in a comment. When possible, indicate what action somebody reusing this snippet is expected to add.

Using a comment for the ellipses (…) is more explicit, preventing errors when a developer copies and pastes a sample code. Write:

JS

```
function exampleFunc() {
  // Add your code here
  // …
}
```

Don't use ellipses (…) like this:

JS

```
function exampleFunc() {
  …
}
```

Comment out parameters

When writing code, you usually omit parameters you don't need. But in some code examples, you want to demonstrate that you didn't use some possible parameters.

To do so, use `/* … */` in the parameter list. This is an exception to the rule to only use single-line comments (`//`).

JS

```
array.forEach((value /* , index, array */) => {
  // …
});
```

Functions

Function names

For function names, use camel case, starting with a lowercase character. Use concise, human-readable, and semantic names where appropriate.

The following is a correct example of a function name:

JS

```js
function sayHello() {
  console.log("Hello!");
}
```

✓

Don't use function names like these:

JS

```js
function SayHello() {
  console.log("Hello!");
}

function doIt() {
  console.log("Hello!");
}
```

✗

Function declarations

- Where possible, use the function declaration over function expressions to define functions.

  Here is the recommended way to declare a function:

  JS

  ```js
  function sum(a, b) {
    return a + b;
  }
  ```

  ✓

  This is not a good way to define a function:

  JS

  ```js
  let sum = function (a, b) {
    return a + b;
  };
  ```

  ✗

- When using anonymous functions as a callback (a function passed to another method invocation), if you do not need to access `this`, use an arrow function to make the code shorter and cleaner.

Here is the recommended way:

JS

```js
const array = [1, 2, 3, 4];
const sum = array.reduce((a, b) => a + b);
```

✓

Instead of this:

JS

```js
const array = [1, 2, 3, 4];
const sum = array.reduce(function (a, b) {
  return a + b;
});
```

✗

- Consider avoiding using arrow function to assign a function to an identifier. In particular, don't use arrow functions for methods. Use function declarations with the keyword `function`:

JS

```js
function x() {
  // …
}
```

✓

Don't do:

JS

```js
const x = () => {
  // …
};
```

✗

- When using arrow functions, use implicit return (also known as *expression body*) when possible:

JS

```js
arr.map((e) => e.id);
```

✓

And not:

JS

```js
arr.map((e) => {
  return e.id;
});
```

Loops and conditional statements

Loop initialization

When loops are required, choose the appropriate one from `for(;;)`, `for...of`, `while`, etc.

- When iterating through all collection elements, avoid using the classical `for (;;)` loop; prefer `for...of` or `forEach()`. Note that if you are using a collection that is not an `Array`, you have to check that `for...of` is actually supported (it requires the variable to be iterable), or that the `forEach()` method is actually present.

  Use `for...of`:

  JS

  ```js
  const dogs = ["Rex", "Lassie"];
  for (const dog of dogs) {
    console.log(dog);
  }
  ```

  Or `forEach()`:

  JS

  ```js
  const dogs = ["Rex", "Lassie"];
  dogs.forEach((dog) => {
    console.log(dog);
  });
  ```

  Do not use `for (;;)` — not only do you have to add an extra index, `i`, but you also have to track the length of the array. This can be error-prone for beginners.

JS

```js
const dogs = ["Rex", "Lassie"];
for (let i = 0; i < dogs.length; i++) {
  console.log(dogs[i]);
}
```

- Make sure that you define the initializer properly by using the `const` keyword for `for...of` or `let` for the other loops. Don't omit it. These are correct examples:

JS

```js
const cats = ["Athena", "Luna"];
for (const cat of cats) {
  console.log(cat);
}

for (let i = 0; i < 4; i++) {
  result += arr[i];
}
```

The example below does not follow the recommended guidelines for the initialization (it implicitly creates a global variable and will fail in strict mode):

JS

```js
const cats = ["Athena", "Luna"];
for (i of cats) {
  console.log(i);
}
```

- When you need to access both the value and the index, you can use `.forEach()` instead of `for (;;)`. Write:

JS

```js
const gerbils = ["Zoé", "Chloé"];
gerbils.forEach((gerbil, i) => {
  console.log(`Gerbil #${i}: ${gerbil}`);
});
```

Do not write:

JS

```js
const gerbils = ["Zoé", "Chloé"];
for (let i = 0; i < gerbils.length; i++) {
  console.log(`Gerbil #${i}: ${gerbils[i]}`);
}
```

Warning: Never use `for...in` with arrays and strings.

Note: Consider not using a `for` loop at all. If you are using an `Array` (or a `String` for some operations), consider using more semantic iteration methods instead, like `map()`, `every()`, `findIndex()`, `find()`, `includes()`, and many more.

Control statements

There is one notable case to keep in mind for the `if...else` control statements. If the `if` statement ends with a `return`, do not add an `else` statement.

Continue right after the `if` statement. Write:

JS

```js
if (test) {
  // Perform something if test is true
  // …
  return;
}

// Perform something if test is false
// …
```

Do not write:

JS

```
if (test) {
  // Perform something if test is true
  // …
  return;
} else {
  // Perform something if test is false
  // …
}
```

Use braces with control flow statements and loops

While control flow statements like `if`, `for`, and `while` don't require the use of braces when the content is made of one single statement, you should always use braces. Write:

JS

```
for (const car of storedCars) {
  car.paint("red");
}
```

Don't write:

JS

```
for (const car of storedCars) car.paint("red");
```

This prevent forgetting to add the braces when adding more statements.

Switch statements

Switch statements can be a little tricky.

- Don't add a `break` statement after a `return` statement in a specific case. Instead, write `return` statements like this:

JS

```
switch (species) {
  case "chicken":
    return farm.shed;
  case "horse":
    return corral.entry;
  default:
    return "";
}
```

If you add a `break` statement, it will be unreachable. Do not write:

JS

```
switch (species) {
  case "chicken":
    return farm.shed;
    break;
  case "horse":
    return corral.entry;
    break;
  default:
    return "";
}
```

- Use `default` as the last case, and don't end it with a `break` statement. If you need to do it differently, add a comment explaining why.
- Remember that when you declare a local variable for a case, you need to use braces to define a scope:

JS

```
switch (fruits) {
  case "Orange": {
    const slice = fruit.slice();
    eat(slice);
    break;
  }
  case "Apple": {
    const core = fruit.extractCore();
    recycle(core);
    break;
  }
}
```

Error handling

- If certain states of your program throw uncaught errors, they will halt execution and potentially reduce the usefulness of the example. You should, therefore, catch errors using a `try...catch` block, as shown below:

JS

```
try {
  console.log(getResult());
} catch (e) {
  console.error(e);
}
```

- When you don't need the parameter of the `catch` statement, omit it:

JS

```
try {
  console.log(getResult());
} catch {
  console.error("An error happened!");
}
```

Note: Keep in mind that only *recoverable* errors should be caught and handled. All non-recoverable errors should be let through and bubble up the call stack.

Objects

Object names

- When defining a class, use *PascalCase* (starting with a capital letter) for the class name and *camelCase* (starting with a lowercase letter) for the object property and method names.
- When defining an object instance, either a literal or via a constructor, use *camelCase*, starting with lower-case character, for the instance name. For example:

JS

```js
const hanSolo = new Person("Han Solo", 25, "he/him");

const luke = {
  name: "Luke Skywalker",
  age: 25,
  pronouns: "he/him",
};
```

Object creation

For creating general objects (i.e., when classes are not involved), use literals and not constructors.

For example, do this:

JS

```js
const object = {};
```

Don't create a general object like this:

JS

```js
const object = new Object();
```

Object classes

- Use ES class syntax for objects, not old-style constructors.

  For example, this is the recommended way:

JS

```js
class Person {
  constructor(name, age, pronouns) {
    this.name = name;
    this.age = age;
    this.pronouns = pronouns;
  }

  greeting() {
    console.log(`Hi! I'm ${this.name}`);
  }
}
```

- Use `extends` for inheritance:

JS

```js
class Teacher extends Person {
  // …
}
```

Methods

To define methods, use the method definition syntax:

JS

```js
const obj = {
  foo() {
    // …
  },
  bar() {
    // …
  },
};
```

Instead of:

JS

```js
const obj = {
  foo: function () {
    // …
  },
  bar: function () {
    // …
  },
};
```

Object properties

- The `Object.prototype.hasOwnProperty()` method has been deprecated in favor of `Object.hasOwn()`.
- When possible, use the shorthand avoiding the duplication of the property identifier. Write:

  JS

  ```js
  function createObject(name, age) {
    return { name, age };
  }
  ```

  Don't write:

  JS

  ```js
  function createObject(name, age) {
    return { name: name, age: age };
  }
  ```

Operators

This section lists our recommendations of which operators to use and when.

Conditional operators

When you want to store to a variable a literal value depending on a condition, use a conditional (ternary) operator instead of an `if...else` statement. This rule also applies when returning a value. Write:

JS

```js
const x = condition ? 1 : 2;
```

✓

Do not write:

JS

```js
let x;
if (condition) {
  x = 1;
} else {
  x = 2;
}
```

✗

The conditional operator is helpful when creating strings to log information. In such cases, using a regular `if...else` statement leads to long blocks of code for a side operation like logging, obfuscating the central point of the example.

Strict equality operator

Prefer the strict equality (triple equals) and inequality operators over the loose equality (double equals) and inequality operators.

Use the strict equality and inequality operators like this:

JS

```js
name === "Shilpa";
age !== 25;
```

✓

Don't use the loose equality and inequality operators, as shown below:

JS

```js
name == "Shilpa";
age != 25;
```

✗

If you need to use `==` or `!=`, remember that `== null` is the only acceptable case. As TypeScript will fail on all other cases, we don't want to have them in our example code. Consider adding a comment to explain why you need it.

Shortcuts for boolean tests

Prefer shortcuts for boolean tests. For example, use `if (x)` and `if (!x)`, not `if (x === true)` and `if (x === false)`, unless different kinds of truthy or falsy values are handled differently.

Strings

String literals can be enclosed within single quotes, as in `'A string'`, or within double quotes, as in `"A string"`. Don't worry about which one to use; Prettier keeps it consistent.

Template literals

For inserting values into strings, use template literals.

- Here is an example of the recommended way to use template literals. Their use prevents a lot of spacing errors.

  JS

  ```js
  const name = "Shilpa";
  console.log(`Hi! I'm ${name}!`);
  ```
  ✓

  Don't concatenate strings like this:

  JS

  ```js
  const name = "Shilpa";
  console.log("Hi! I'm" + name + "!"); // Hi! I'mShilpa!
  ```
  ✗

- Don't overuse template literals. If there are no substitutions, use a normal string literal instead.

Variables

Variable names

Good variable names are essential to understanding code.

- Use short identifiers, and avoid non-common abbreviations. Good variable names are usually between 3 to 10-character long, but as a hint only. For example, `accelerometer` is more descriptive than abbreviating to `acclmtr` for the sake of character length.

- Try to use real-world relevant examples where each variable has clear semantics. Only fall back to placeholder names like `foo` and `bar` when the example is simple and contrived.
- Do not use the Hungarian notation naming convention. Do not prefix the variable name with its type. For example, write `bought = car.buyer !== null` rather than `bBought = oCar.sBuyer != null` or `name = "Maria Sanchez"` instead of `sName = "Maria Sanchez"`.
- For collections, avoid adding the type such as list, array, queue in the name. Use the content name in the plural form. For example, for an array of cars, use `cars` and not `carArray` or `carList`. There may be exceptions, like when you want to show the abstract form of a feature without the context of a particular application.
- For primitive values, use *camelCase*, starting with a lowercase character. Do not use `_`. Use concise, human-readable, and semantic names where appropriate. For example, use `currencyName` rather than `currency_name`.
- Avoid using articles and possessives. For example, use `car` instead of `myCar` or `aCar`. There may be exceptions, like when describing a feature in general without a practical context.
- Use variable names as shown here:

  JS

  ```js
  const playerScore = 0;
  const speed = distance / time;
  ```

  Don't name variables like this:

  JS

  ```js
  const thisIsaveryLONGVariableThatRecordsPlayerscore345654 = 0;
  const s = d / t;
  ```

Note: The only place where it's allowed not to use human-readable, semantic names is where a very commonly recognized convention exists, such as using `i` and `j` for loop iterators.

Variable declarations

When declaring variables and constants, use the `let` and `const` keywords, not `var`. The following examples show what's recommended and what's not on MDN Web Docs:

- If a variable will not be reassigned, prefer `const`, like so:

JS

```
const name = "Shilpa";
console.log(name);
```

✓

- If you'll change the value of a variable, use `let` as shown below:

JS

```
let age = 40;
age++;
console.log("Happy birthday!");
```

✓

- The example below uses `let` where it should be `const`. The code will work, but we want to avoid this usage in MDN Web Docs code examples.

JS

```
let name = "Shilpa";
console.log(name);
```

✗

- The example below uses `const` for a variable that gets reassigned. The reassignment will throw an error.

JS

```
const age = 40;
age++;
console.log("Happy birthday!");
```

✗

- The example below uses `var`, polluting the global scope:

JS

```
var age = 40;
var name = "Shilpa";
```

✗

- Declare one variable per line, like so:

JS

```js
let var1;
let var2;
let var3 = "Apapou";
let var4 = var3;
```

Do not declare multiple variables in one line, separating them with commas or using chain declaration. Avoid declaring variables like this:

JS

```js
let var1, var2;
let var3 = var4 = "Apapou"; // var4 is implicitly created as a global variable; fails in strict mode
```

Type coercion

Avoid implicit type coercions. In particular, avoid `+val` to force a value to a number and `"" + val` to force it to a string. Use `Number()` and `String()`, without `new`, instead. Write:

JS

```js
class Person {
  #name;
  #birthYear;

  constructor(name, year) {
    this.#name = String(name);
    this.#birthYear = Number(year);
  }
}
```

Don't write:

JS

```js
class Person {
  #name;
  #birthYear;

  constructor(name, year) {
    this.#name = "" + name;
    this.#birthYear = +year;
  }
}
```

Web APIs to avoid

In addition to these JavaScript language features, we recommend a few guidelines related to Web APIs to keep in mind.

Avoid browser prefixes

If all major browsers (Chrome, Edge, Firefox, and Safari) support a feature, don't prefix the feature. Write:

JS

```js
const context = new AudioContext();
```

Avoid the added complexity of prefixes. Don't write:

JS

```js
const AudioContext = window.AudioContext || window.webkitAudioContext;
const context = new AudioContext();
```

The same rule applies to CSS prefixes.

Avoid deprecated APIs

When a method, a property, or a whole interface is deprecated, do not use it (outside its documentation). Instead, use the modern API.

Here is a non-exhaustive list of Web APIs to avoid and what to replace them with:

- Use `fetch()` instead of XHR (`XMLHttpRequest`).
- Use `AudioWorklet` instead of `ScriptProcessorNode`, in the Web Audio API.

Use safe and reliable APIs

- Do not use `Element.innerHTML` to insert purely textual content into an element; use `Node.textContent` instead. The property `innerHTML` leads to security problems if a developer doesn't control the parameter. The more we as writers avoid using it, the fewer security flaws are created when a developer copies and pastes our code.

  The example below demonstrates the use of `textContent`.

  JS

  ```js
  const text = "Hello to all you good people";
  const para = document.createElement("p");
  para.textContent = text;
  ```

  ✓

  Don't use `innerHTML` to insert *pure text* into DOM nodes.

  JS

  ```js
  const text = "Hello to all you good people";
  const para = document.createElement("p");
  para.innerHTML = text;
  ```

  ✗

- The `alert()` function is unreliable. It doesn't work in live examples on MDN Web Docs that are inside an `<iframe>`. Moreover, it is modal to the whole window, which is annoying. In static code examples, use `console.log()` or `console.error()`. In live examples, avoid `console.log()` and `console.error()` because they are not displayed. Use a dedicated UI element.

Use the appropriate log method

- When logging a message, use `console.log()`.
- When logging an error, use `console.error()`.

See also

JavaScript language reference - browse through our JavaScript reference pages to check out some good, concise, meaningful JavaScript snippets.

mdn_

Your blueprint for a better internet.