



🏠 → [The JavaScript language](#) → [Code quality](#)

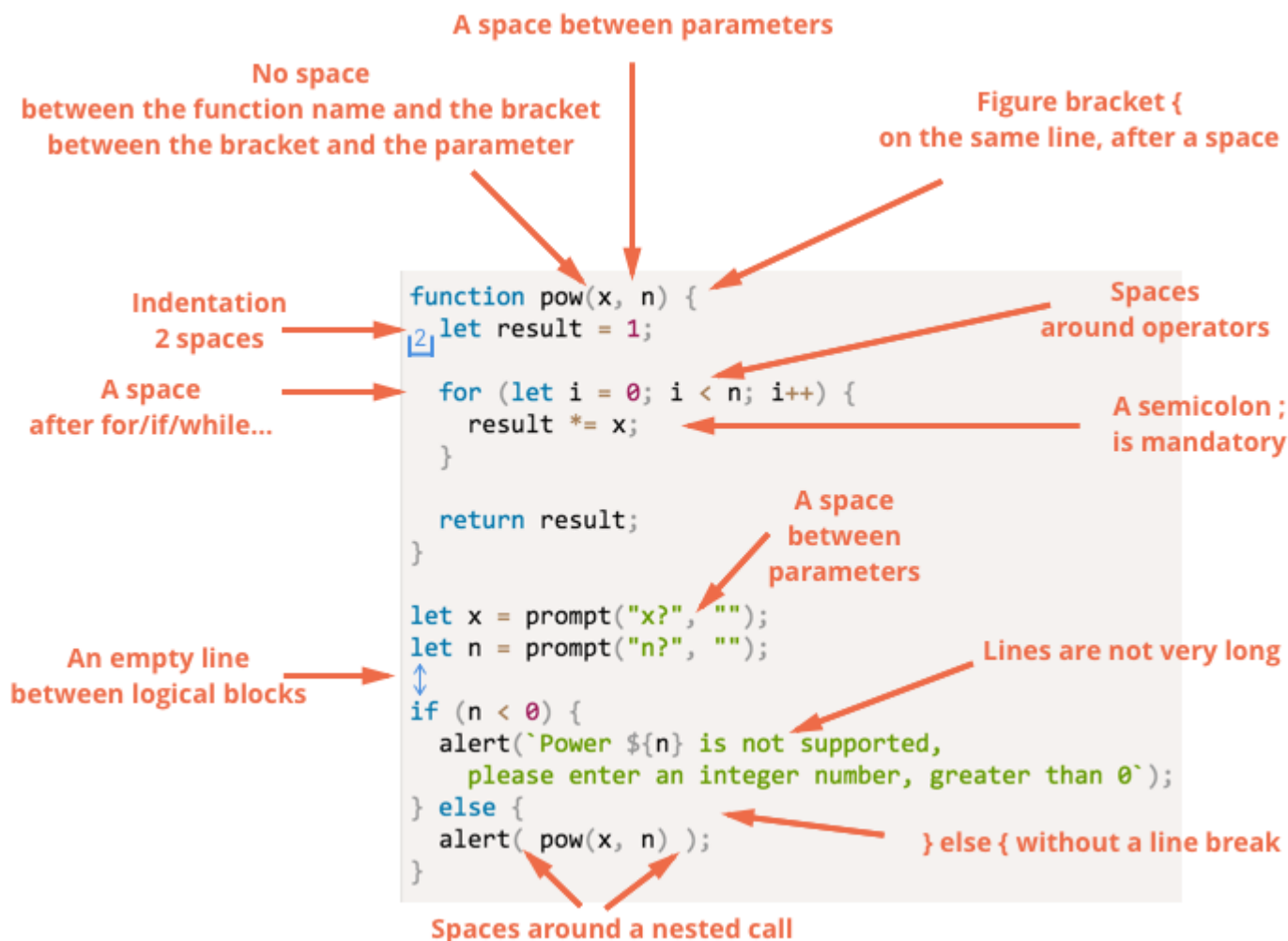
# Coding Style

Our code must be as clean and easy to read as possible.

That is actually the art of programming – to take a complex task and code it in a way that is both correct and human-readable.

## Syntax

Here is a cheatsheet with some suggested rules (see below for more details):



Now let's discuss the rules and reasons for them in detail.

## ⚠ Irony Detected

Nothing is set in stone here. These are style preferences, not religious dogmas.

## Curly Braces

In most JavaScript projects curly braces are written in “Egyptian” style with the opening brace on the same line as the corresponding keyword – not on a new line. There should also be a space before the opening bracket, like this:

```
1  if (condition) {  
2    // do this  
3    // ...and that  
4    // ...and that  
5  }
```

A single-line construct is an important edge case. Should we use brackets at all? If yes, then where?

Here are the annotated variants so you can judge their readability for yourself:



**Beginner sometimes do that. Bad!**  
**Figure brackets are not needed**

```
if (n < 0) {alert(`Power ${n} is not supported`);}
```



**Split to a separate line without brackets. Never do that!**  
**Source of nasty errors.**

```
if (n < 0)  
  alert(`Power ${n} is not supported`);
```



**One line without brackets.**  
**Acceptable if it's short.**

```
if (n < 0) alert(`Power ${n} is not supported`);
```



**The best variant.**

```
if (n < 0) {  
  alert(`Power ${n} is not supported`);  
}
```

In summary:

- For very short code, one line is acceptable. For example: `if (cond) return null`.
- But a separate line for each statement in brackets is usually easier to read.

## Line Length

No one likes to read a long horizontal line of code. It's best practice to split them up and limit the length of your lines.

The maximum line length should be agreed upon at the team-level. It's usually 80 or 120 characters.

## Indents

There are two types of indents:

- **Horizontal indents: 2 or 4 spaces.**

A horizontal indentation is made using either 2 or 4 spaces or the "Tab" symbol. Which one to choose is an old holy war. Spaces are more common nowadays.

One advantage of spaces over tabs is that spaces allow more flexible configurations of indents than the "Tab" symbol.

For instance, we can align the arguments with the opening bracket, like this:

```
1 show(parameters,  
2     aligned, // 5 spaces padding at the left  
3     one,  
4     after,  
5     another  
6 ) {  
7   // ...  
8 }
```

- **Vertical indents: empty lines for splitting code into logical blocks.**

Even a single function can often be divided into logical blocks. In the example below, the initialization of variables, the main loop and returning the result are split vertically:

```
1 function pow(x, n) {  
2   let result = 1;  
3   // <--  
4   for (let i = 0; i < n; i++) {  
5     result *= x;  
6   }  
7   // <--  
8   return result;  
9 }
```

Insert an extra newline where it helps to make the code more readable. There should not be more than nine lines of code without a vertical indentation.

## Semicolons

A semicolon should be present after each statement, even if it could possibly be skipped.

There are languages where a semicolon is truly optional and it is rarely used. In JavaScript, though, there are cases where a line break is not interpreted as a semicolon, leaving the code vulnerable to errors.

As you become more mature as a programmer, you may choose a no-semicolon style like [StandardJS](#). Until then, it's best to use semicolons to avoid possible pitfalls.

## Nesting Levels

Try to avoid nesting code too many levels deep.

Sometimes it's a good idea to use the `"continue"` directive in a loop to avoid extra nesting.

For example, instead of adding a nested `if` conditional like this:

```
1  for (let i = 0; i < 10; i++) {
2    if (cond) {
3      ... // <- one more nesting level
4    }
5  }
```

We can write:

```
1  for (let i = 0; i < 10; i++) {
2    if (!cond) continue;
3    ... // <- no extra nesting level
4  }
```

A similar thing can be done with `if/else` and `return`.

For example, two constructs below are identical.

Option 1:

```
1  function pow(x, n) {
2    if (n < 0) {
3      alert("Negative 'n' not supported");
4    } else {
5      let result = 1;
6
7      for (let i = 0; i < n; i++) {
8        result *= x;
9      }
10
11     return result;
12  }
```

```
13     }  
    }
```

Option 2:

```
1  function pow(x, n) {  
2    if (n < 0) {  
3      alert("Negative 'n' not supported");  
4      return;  
5    }  
6  
7    let result = 1;  
8  
9    for (let i = 0; i < n; i++) {  
10     result *= x;  
11   }  
12  
13   return result;  
14 }
```

The second one is more readable because the “edge case” of  $n < 0$  is handled early on. Once the check is done we can move on to the “main” code flow without the need for additional nesting.

## Function Placement

If you are writing several “helper” functions and the code that uses them, there are three ways to organize the functions.

1. Functions declared above the code that uses them:

```
1  // function declarations  
2  function createElement() {  
3    ...  
4  }  
5  
6  function setHandler(elem) {  
7    ...  
8  }  
9  
10 function walkAround() {  
11   ...  
12 }  
13  
14 // the code which uses them  
15 let elem = createElement();  
16 setHandler(elem);  
17 walkAround();
```

2. Code first, then functions

```
1 // the code which uses the functions
2 let elem = createElement();
3 setHandler(elem);
4 walkAround();
5
6 // --- helper functions ---
7 function createElement() {
8     ...
9 }
10
11 function setHandler(elem) {
12     ...
13 }
14
15 function walkAround() {
16     ...
17 }
```

3. Mixed: a function is declared where it's first used.

Most of time, the second variant is preferred.

That's because when reading code, we first want to know *what it does*. If the code goes first, then it provides that information. Then, maybe we won't need to read the functions at all, especially if their names are descriptive of what they actually do.

## Style Guides

A style guide contains general rules about "how to write" code, e.g. which quotes to use, how many spaces to indent, where to put line breaks, etc. A lot of minor things.

When all members of a team use the same style guide, the code looks uniform, regardless of which team member wrote it.

Of course, a team can always write their own style guide. Most of the time though, there's no need to. There are many existing tried and true options to choose from, so adopting one of these is usually your best bet.

Some popular choices:

- [Google JavaScript Style Guide](#)
- [Airbnb JavaScript Style Guide](#)
- [Idiomatic.JS](#)
- [StandardJS](#)
- (plus many more)

If you're a novice developer, start with the cheatsheet at the beginning of this chapter. Once you've mastered that you can browse other style guides to pick up common principles and decide which one you like best.

## Automated Linters

Linters are tools that can automatically check the style of your code and make suggestions for refactoring.

The great thing about them is that style-checking can also find some bugs, like typos in variable or function names. Because of this feature, installing a linter is recommended even if you don't want to stick to one particular "code style".

Here are the most well-known linting tools:

- [JSLint](#) – one of the first linters.
- [JSHint](#) – more settings than JSLint.
- [ESLint](#) – probably the newest one.

All of them can do the job. The author uses [ESLint](#).

Most linters are integrated with many popular editors: just enable the plugin in the editor and configure the style.

For instance, for ESLint you should do the following:

1. Install [Node.JS](#).
2. Install ESLint with the command `npm install -g eslint` (npm is a JavaScript package installer).
3. Create a config file named `.eslintrc` in the root of your JavaScript project (in the folder that contains all your files).
4. Install/enable the plugin for your editor that integrates with ESLint. The majority of editors have one.

Here's an example of an `.eslintrc` file:

```
1  {
2    "extends": "eslint:recommended",
3    "env": {
4      "browser": true,
5      "node": true,
6      "es6": true
7    },
8    "rules": {
9      "no-console": 0,
10   },
11   "indent": 2
12 }
```

Here the directive "extends" denotes that the configuration is based on the "eslint:recommended" set of settings. After that, we specify our own.

It is also possible to download style rule sets from the web and extend them instead. See <http://eslint.org/docs/user-guide/getting-started> for more details about installation.

Also certain IDEs have built-in linting, which is convenient but not as customizable as ESLint.

## Summary

All syntax rules described in this chapter (and in the style guides referenced) aim to increase the readability of your code, but all of them are debatable.

When we think about writing "better" code, the questions we should ask are, "What makes the code more readable and easier to understand?" and "What can help us avoid errors?" These are the main things to keep in mind when choosing and debating code styles.

Reading popular style guides will allow you to keep up to date with the latest ideas about code style trends and best practices.

## ✓ Tasks

### Bad style

importance: 4

What's wrong with the code style below?

```
1  function pow(x,n)
2  {
3    let result=1;
4    for(let i=0;i<n;i++) {result*=x;}
5    return result;
6  }
7
8  let x=prompt("x?", ''), n=prompt("n?", '')
9  if (n<=0)
10 {
11   alert(`Power ${n} is not supported, please enter an integer number greater than zero`
12 }
13 else
14 {
15   alert(pow(x,n))
16 }
```

Fix it.

solution



Previous lesson

Next lesson





## Comments

- You're welcome to post additions, questions to the articles and answers to them.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox ([plnkr](#), [JSBin](#), [codepen](#)...)
- If you can't understand something in the article – please elaborate.

---

© 2007—2019 Ilya Kantor [contact us](#) [terms of usage](#) [privacy policy](#) [about the project](#) [open source](#)