JavaScript Coding Styles

When you write JavaScript code in a team setting, either professionally or in open source context, almost every team has their style guide for writing JavaScript. A style guide typically has two types of rules:

- **Formatting and Aesthetic**: This includes indentation, spacing (where to apply white spaces), single quotes vs double quotes, etc. Consistently formatted code is easier to read, and easier to maintain.

- **Best Practices**: This includes how to perform type coercions, how to define variables with hoisting rules, etc. Those rules often have opinionated roots, but they attempt to steer you from the dark alleys and pitfalls of the language by avoiding constructs that cause trouble.

There are several popular style guides for JavaScript; they are mostly similar to each other. We recommend the Airbnb JavaScript Style Guide to our students. Right now, though, you might find it overwhelming to read through the entire style guide; you're only familiar with a small part of the language. For now, we have handpicked relevant rules for the parts of the language you will use in the near future. Stick to these rules and build good habits from the start. Once you've learned about the more advanced features, we'll ask you to read the full style guide.

Finally, we should mention that it's not uncommon to see experienced developers who have their own preferred coding style; sometimes, this style "violates" some of the styles we recommend here. Keep in mind that the most important value of having a style guide is consistency. When you join a project,

you should be flexible enough to adopt the style guide that the team uses. Don't insist on your own rules.

Spacing

Use soft tabs set to two spaces.

```
// bad
function foo() {
····var name;
}

// bad
function bar() {
·var name;
}

// good
function baz() {
··var name;
}
```

Place one space before the leading brace.

```
// bad
function test(){
  console.log('test');
}

// good
```

```
function test() {
  console.log('test');
}
```

Place one space before the opening parenthesis in control statements (`if` , `while` etc.). Place no space between the argument list and the function name in function calls and declarations.

```
// bad
if(isJedi) {
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

// good
function fight() {
  console.log('Swooosh!');
}
```

Set off operators with spaces.

```
// bad
var x=y+5;

// good
var x = y + 5;
```

Do not add spaces inside parentheses.

```
// bad
function bar( foo ) {
  return foo;
}

// good
function bar(foo) {
  return foo;
}

// bad
if ( foo ) {
  console.log(foo);
}

// good
if (foo) {
  console.log(foo);
}
```

Unary special-character operators (e.g., `!`, `++`) must not have spaces between them and their operand.

```
// bad
index ++;

// good
index++;
```

No preceding spaces before `,` and `;`.

```
// bad
func(a ,b) ;

// good
func(a, b);
```

No whitespace at the end of line or on blank lines.

```
// bad
func(a, b);·

// good
func(a, b);
```

The ? and : in a ternary conditional must have space on both sides.

```
// bad
var maybe1 > maybe2?'bar":null;
```

```
// good
var maybe1 > maybe2 ? 'bar" : null;
```

Ternaries should not be nested and should generally be single line expressions.

```
// bad
const foo = maybe1 > maybe2
  ? "bar"
  : value1 > value2 ? "baz" : null;

// better
const maybeNull = value1 > value2 ? 'baz' : null;

const foo = maybe1 > maybe2
  ? 'bar'
  : maybeNull;

// best
const maybeNull = value1 > value2 ? 'baz' : null;

const foo = maybe1 > maybe2 ? 'bar' : maybeNull;
```

Avoid unneeded ternary statements.

```
// bad
const foo = a ? a : b;
const bar = c ? true : false;
const baz = c ? false : true;
```

```
// good
const foo = a || b;
const bar = !!c;
const baz = !c;
```

Blocks

Leave a blank line after blocks and before the next statement.

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo) {
  return bar;
}

return baz;
```

Do not pad your blocks with blank lines.

```
// bad
function bar() {

  console.log(foo);

}
```

```
// also bad
if (baz) {

  console.log(qux);
} else {
  console.log(foo);

}

// good
function bar() {
  console.log(foo);
}

// good
if (baz) {
  console.log(qux);
} else {
  console.log(foo);
}
```

Use braces with all multi-line blocks.

```
// bad
if (test)
  return false;

// good
if (test) return false;

// good
```

```
if (test) {
  return false;
}

// bad
function foo() { return false; }

// good
function bar() {
  return false;
}
```

If you're using multi-line blocks with if and else, put else on the same line as your if block's closing brace.

```
// bad
if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}

// good
if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

Semicolons

Use semicolons after every statement, except for statements ending with blocks

```
// bad
var number
number = 5
number = number + 1

// good
var number;
number = 5;
number = number + 1;

// bad
while (number > 0) {
    number--;
};

// good
while (number > 0) {
    number--;
}
```

Naming Conventions

Use camelCase variable and function names

```
// bad
var Hello = 'hello';
var my_name = 'john';
function call_me() {};

// good
var hello = 'hello';
var myName = 'john';
function callMe() {};
```

Strings

Use single quotes `''` for strings.

```
// bad
var name = "Capt. Janeway";

// good
var name = 'Capt. Janeway';
```

Use explicit coercion

```
var a = 9;

// bad
var string = a + '';

// bad
var string = a.toString();
```

```
// good
var string = String(a);
```

Numbers

Use `Number` for type casting and `parseInt` always with a radix for parsing strings.

```
var inputValue = '4';

// bad
var val = new Number(inputValue);

// bad
var val = +inputValue;

// bad
var val = parseInt(inputValue);

// good
var val = Number(inputValue);

// good
var val = parseInt(inputValue, 10);
```

Boolean

```
var age = 0;

// bad
var hasAge = new Boolean(age);

// good
var hasAge = Boolean(age);

// best
var hasAge = !!age;
```

Functions

Never declare a function in a non-function block (if, while, etc).

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
var test;
if (currentUser) {
  test = function() {
    console.log('Yup.');
  };
}
```

Never name a parameter `arguments`. This takes precedence over the `arguments` object that is given to every function scope.

```
// bad
function nope(name, options, arguments) {
  // ...stuff...
}

// good
function yup(name, options, args) {
  // ...stuff...
}
```