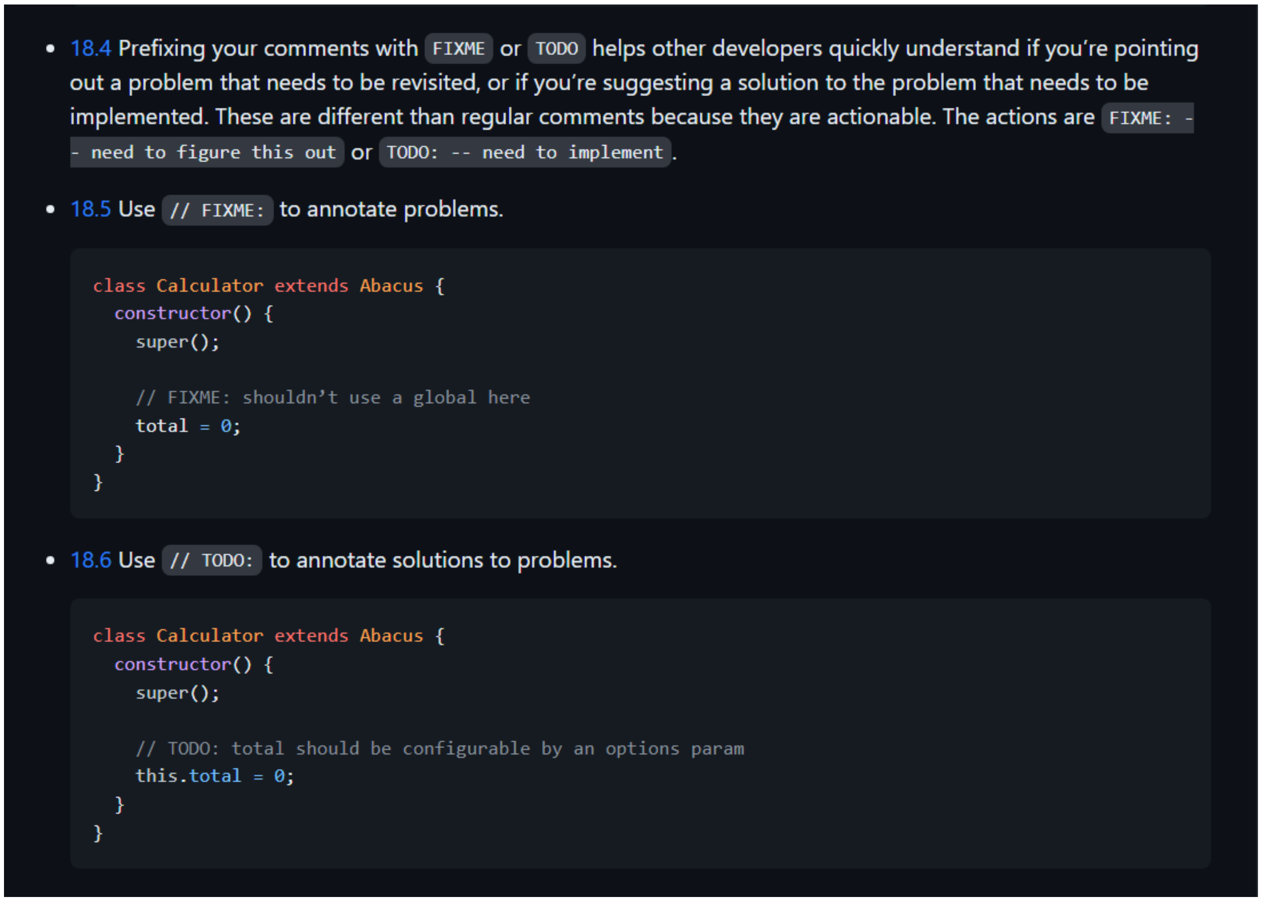
DWA_04.3 Knowledge Check_DWA4

1. Select three rules from the Airbnb Style Guide that you find **useful** and explain why.

   a. Prefixing comments with FIXME and TODO: they give clear prompts/markers to other developers to 'action' the code.



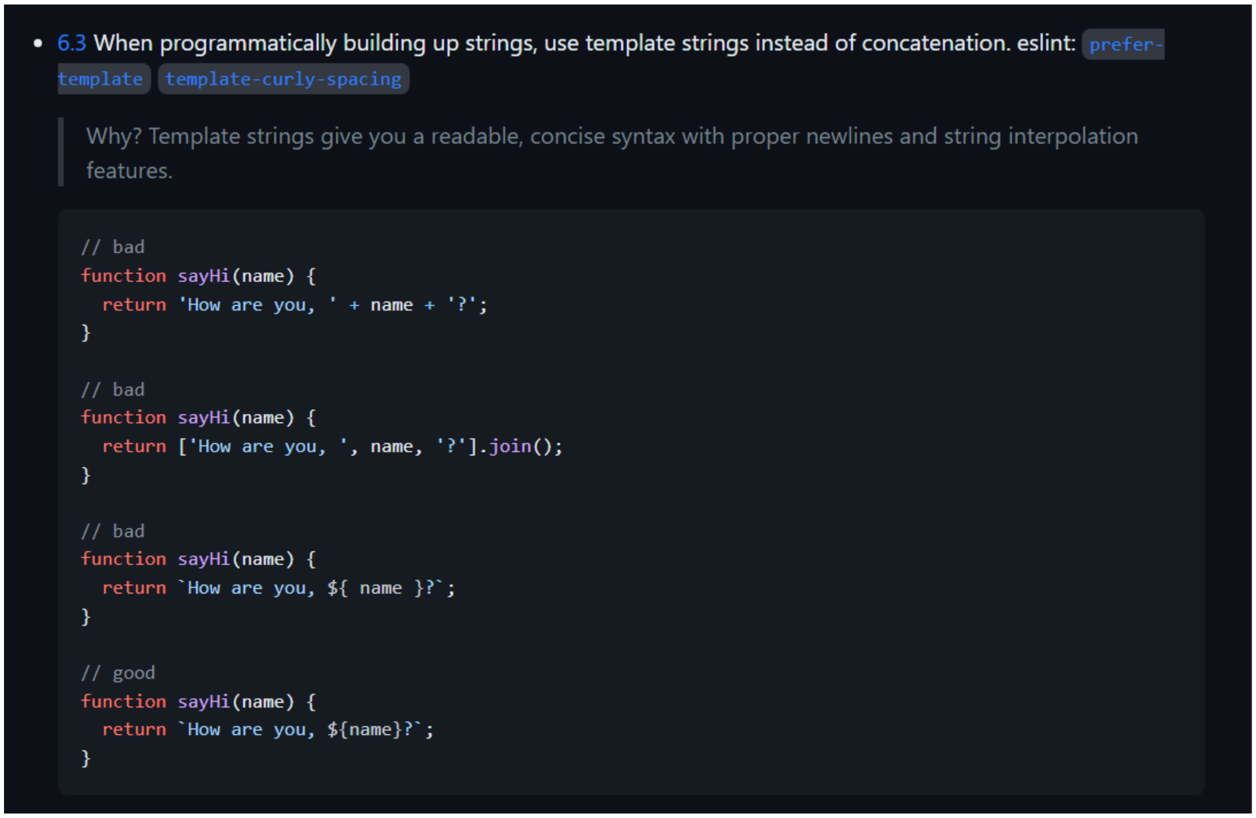
- 18.4 Prefixing your comments with `FIXME` or `TODO` helps other developers quickly understand if you're pointing out a problem that needs to be revisited, or if you're suggesting a solution to the problem that needs to be implemented. These are different than regular comments because they are actionable. The actions are `FIXME: -- need to figure this out` or `TODO: -- need to implement`.
- 18.5 Use `// FIXME:` to annotate problems.

```
class Calculator extends Abacus {
  constructor() {
    super();

    // FIXME: shouldn't use a global here
    total = 0;
  }
}
```

- 18.6 Use `// TODO:` to annotate solutions to problems.

```
class Calculator extends Abacus {
  constructor() {
    super();

    // TODO: total should be configurable by an options param
    this.total = 0;
  }
}
```

b. Use of template literals instead of concatenation: saves time, is more concise, and gives the benefit of using quotes and slashes within the string, as well as newlines without the need to use \n

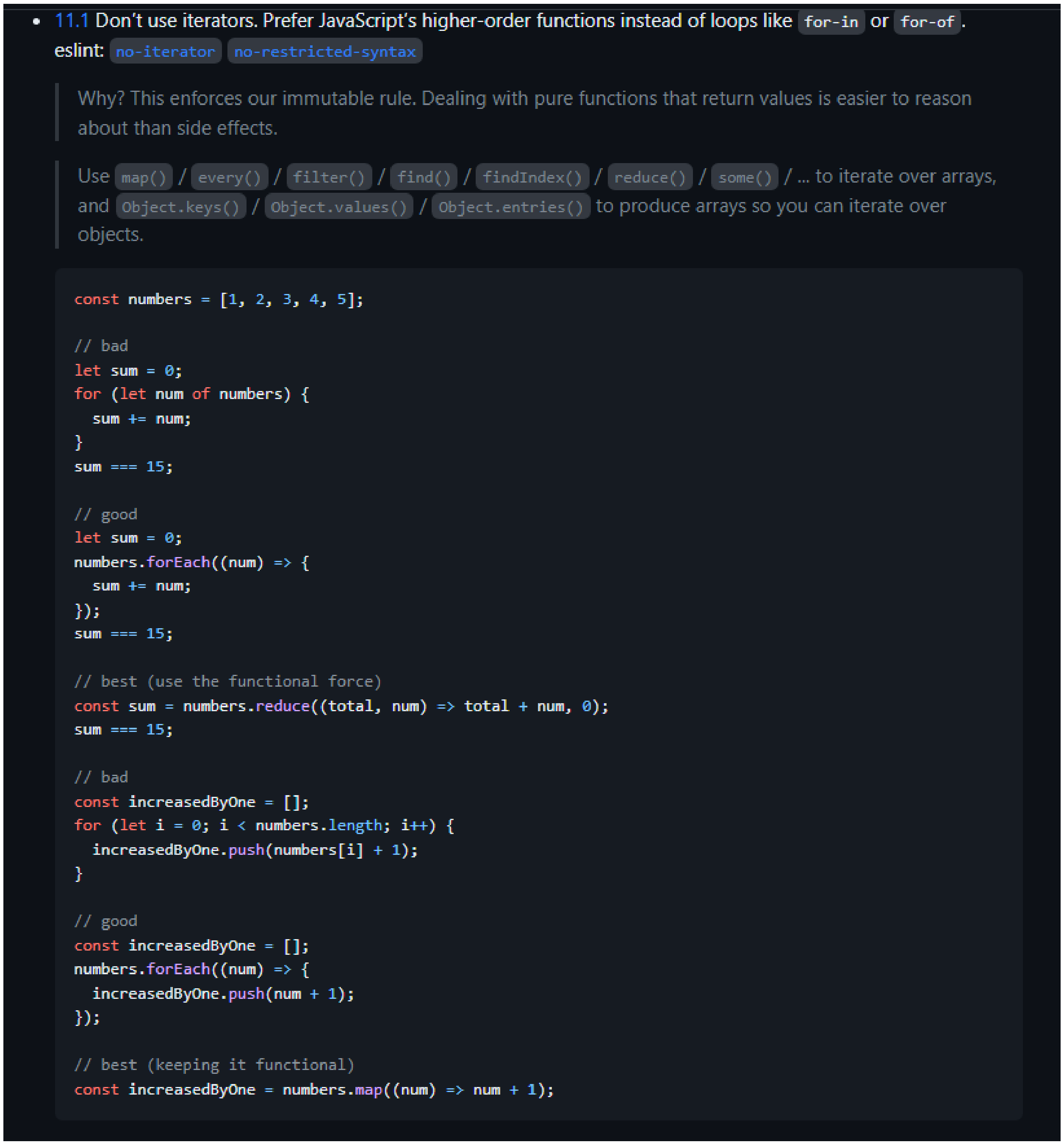


- 6.3 When programmatically building up strings, use template strings instead of concatenation. eslint: `prefer-template` `template-curly-spacing`

> Why? Template strings give you a readable, concise syntax with proper newlines and string interpolation features.

```
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// bad
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// bad
function sayHi(name) {
  return `How are you, ${ name }?`;
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

c. Use JavaScript's higher order functions instead of loops: higher order function are more immutable, thus making the code more predictable. This also improves software performance.



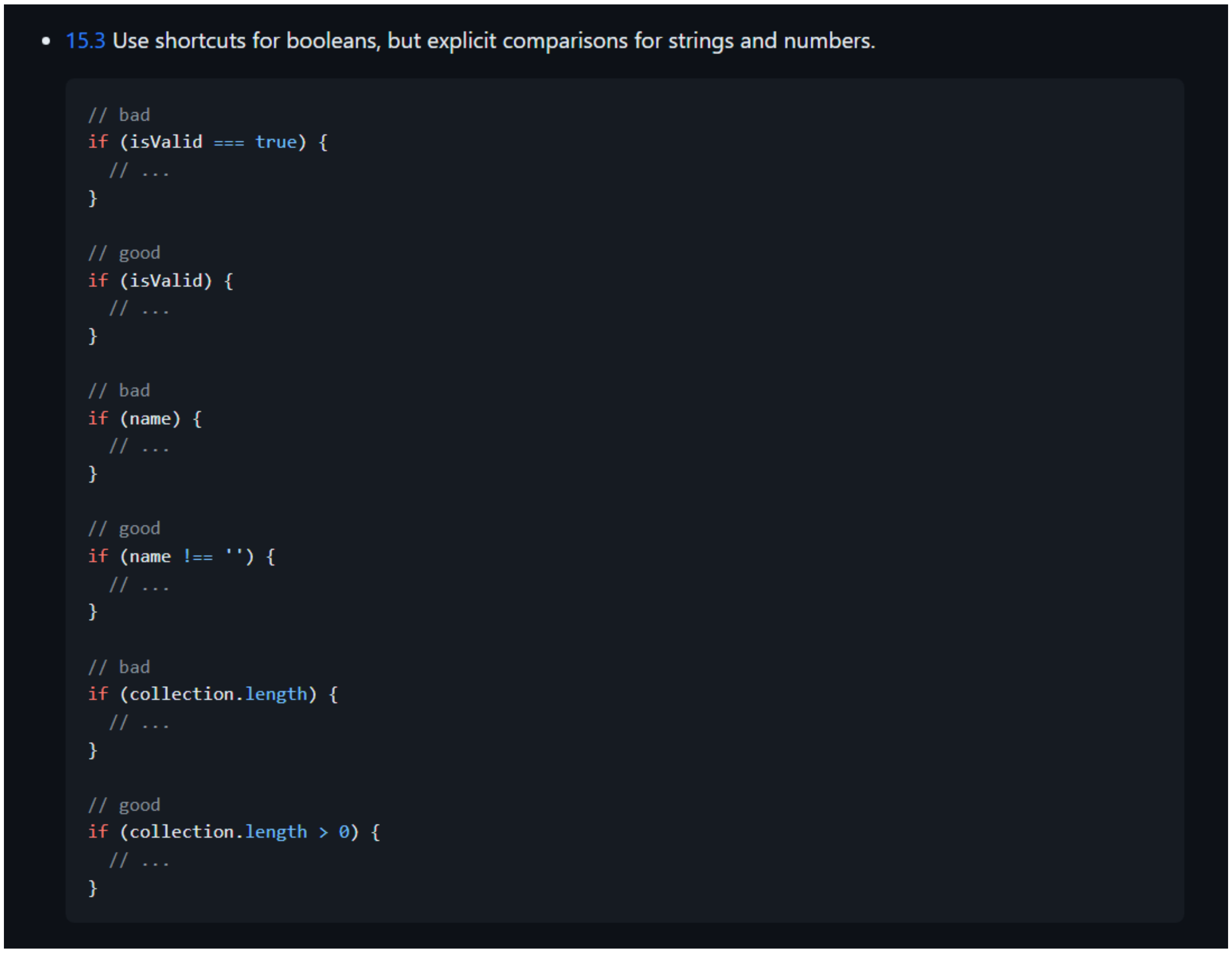
- 11.1 Don't use iterators. Prefer JavaScript's higher-order functions instead of loops like `for-in` or `for-of`. eslint: `no-iterator` `no-restricted-syntax`

> Why? This enforces our immutable rule. Dealing with pure functions that return values is easier to reason about than side effects.

> Use `map()` / `every()` / `filter()` / `find()` / `findIndex()` / `reduce()` / `some()` / ... to iterate over arrays, and `Object.keys()` / `Object.values()` / `Object.entries()` to produce arrays so you can iterate over objects.

```
const numbers = [1, 2, 3, 4, 5];

// bad
let sum = 0;
for (let num of numbers) {
  sum += num;
}
sum === 15;

// good
let sum = 0;
numbers.forEach((num) => {
  sum += num;
});
sum === 15;

// best (use the functional force)
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;

// bad
const increasedByOne = [];
for (let i = 0; i < numbers.length; i++) {
  increasedByOne.push(numbers[i] + 1);
}

// good
const increasedByOne = [];
numbers.forEach((num) => {
  increasedByOne.push(num + 1);
});

// best (keeping it functional)
const increasedByOne = numbers.map((num) => num + 1);
```

d. Shortcuts for booleans, but explicit comparisons for strings and numbers: more concise because the condition is already being evaluated by virtue of being a boolean. For strings and numbers, they cannot be processed like booleans

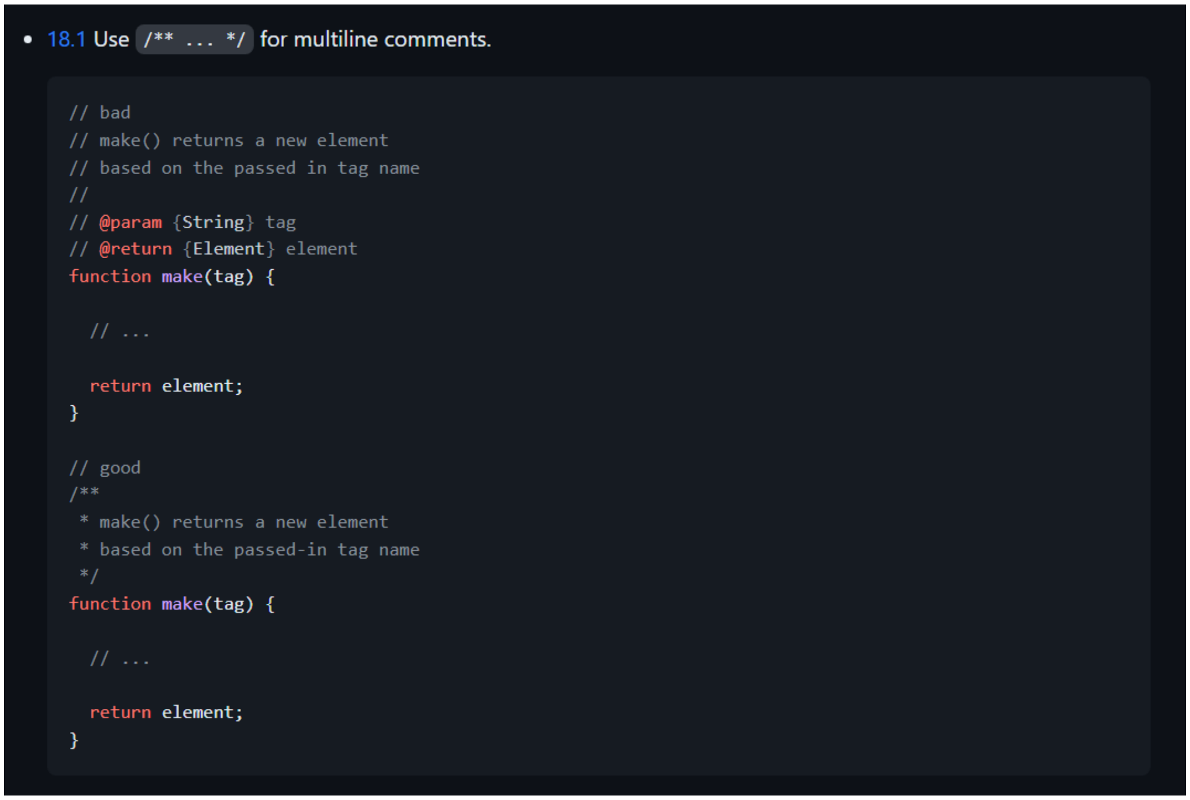


- 15.3 Use shortcuts for booleans, but explicit comparisons for strings and numbers.

```
// bad
if (isValid === true) {
  // ...
}

// good
if (isValid) {
  // ...
}

// bad
if (name) {
  // ...
}

// good
if (name !== '') {
  // ...
}

// bad
if (collection.length) {
  // ...
}

// good
if (collection.length > 0) {
  // ...
}
```

2. Select three rules from the Airbnb Style Guide that you find **confusing** and explain why.

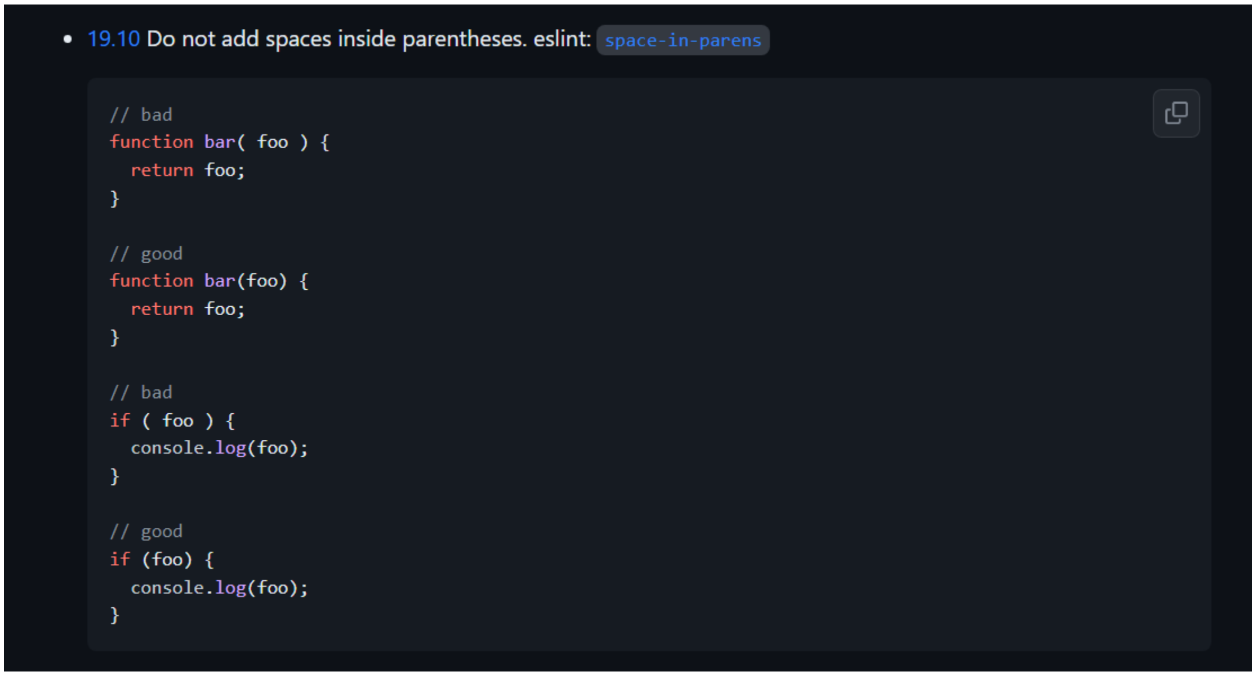
a. /**...*/ for multiline comments is confusing to me because I often reserve /**...*/ for JSDoc, and instead, use /*...*/ for multi-line comments. Seeing /**...*/ makes me expect JSDoc



- 18.1 Use `/** ... */` for multiline comments.

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

  // ...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}
```

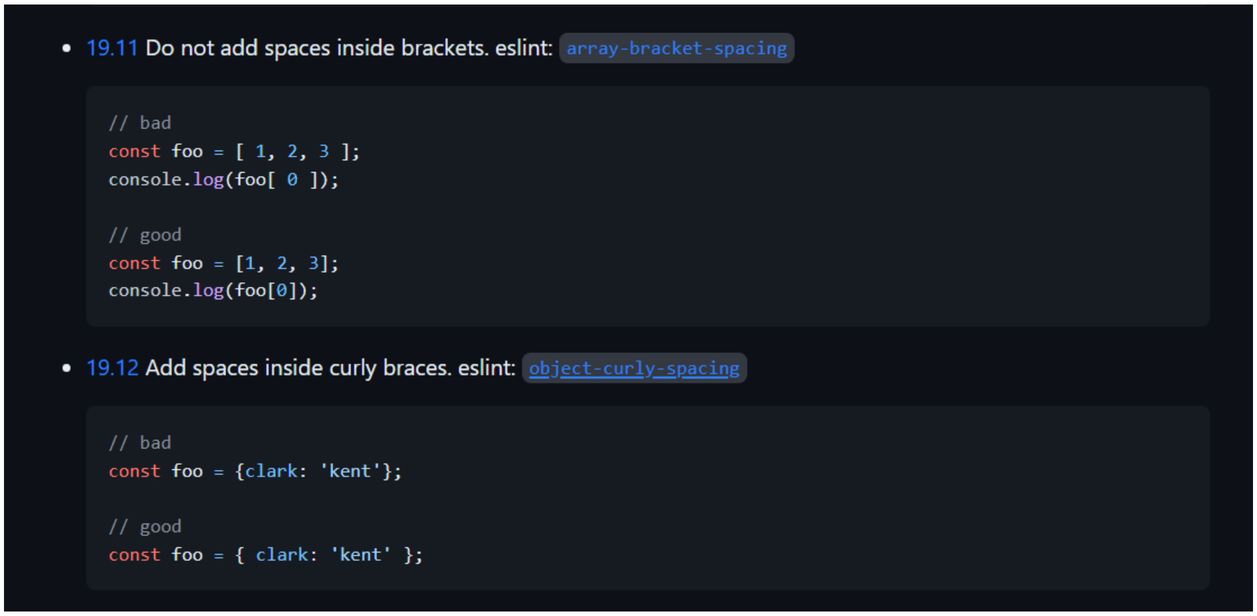
b. Not adding spaces inside parentheses: I find this confusing due to the subsequent guidelines (19.11 and 19.12). One of the reasons it is discouraged is that it creates visual noise, however it is allowed for curly braces.



- 19.10 Do not add spaces inside parentheses. eslint: `space-in-parens`

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

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

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

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

c. Adding spaces is discouraged for square brackets (array brackets), but encouraged for curly brackets (object brackets).



- 19.11 Do not add spaces inside brackets. eslint: `array-bracket-spacing`

```
// bad
const foo = [ 1, 2, 3 ];
console.log(foo[ 0 ]);

// good
const foo = [1, 2, 3];
console.log(foo[0]);
```

- 19.12 Add spaces inside curly braces. eslint: `object-curly-spacing`

```
// bad
const foo = {clark: 'kent'};

// good
const foo = { clark: 'kent' };
```