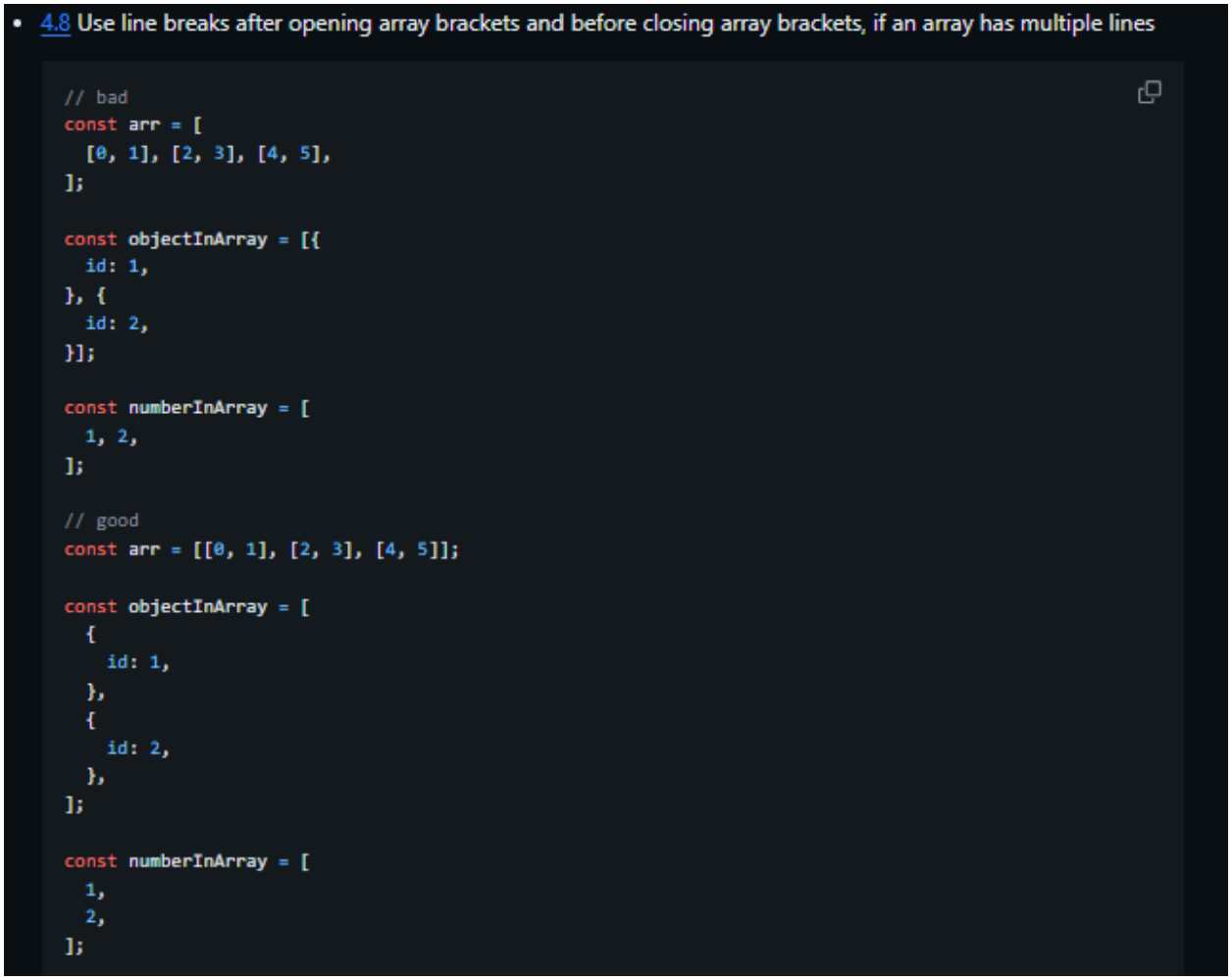
DWA_04.3 Knowledge Check_DWA4

---

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

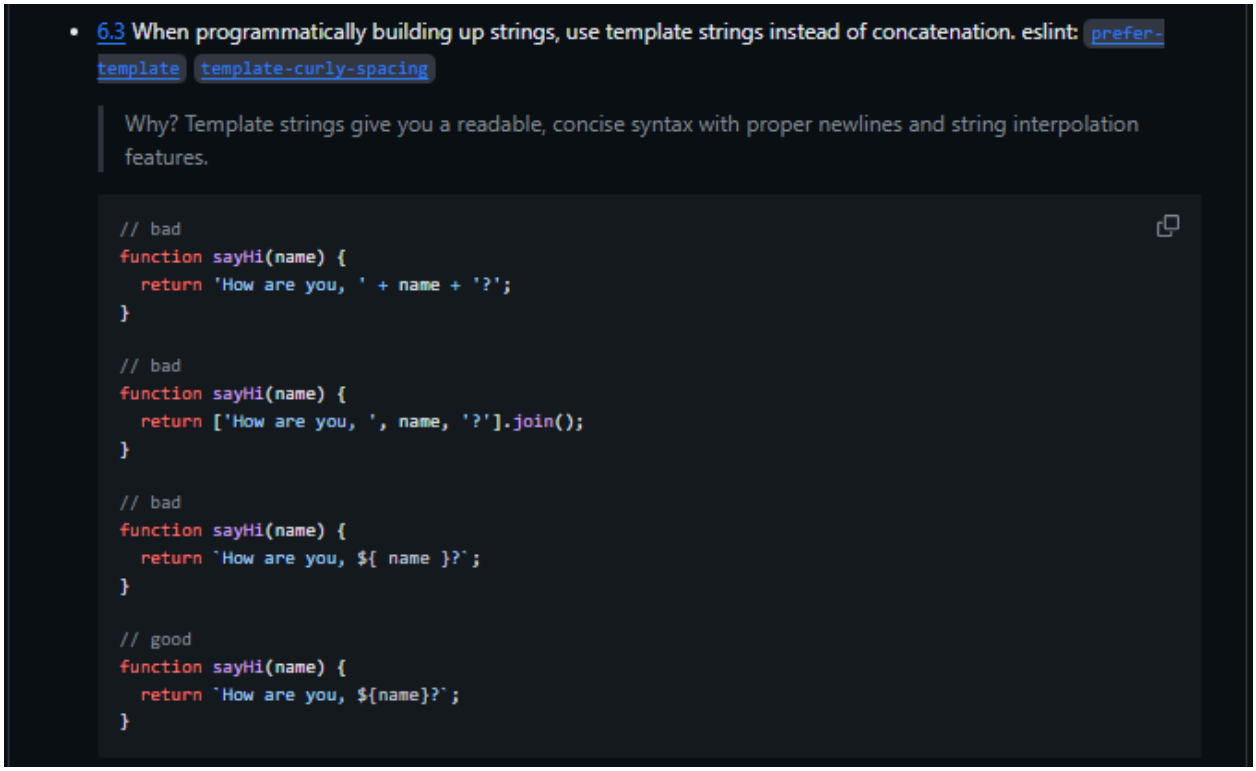
1. Use line breaks after opening array brackets and before closing array brackets, if an array has multiple lines.
   This makes the code easier to read. Adding the breaks helps visually separate the elements in the array. When arrays are extensively nested it becomes difficult to work with them if they are on one line.
   This also helps with the maintenance of an array. Adding, removing or changing elements becomes easier and reduces the chances of syntax errors.



- 4.8 Use line breaks after opening array brackets and before closing array brackets, if an array has multiple lines

```
// bad
const arr = [
  [0, 1], [2, 3], [4, 5],
];

const objectInArray = [{
  id: 1,
}, {
  id: 2,
}];

const numberInArray = [
  1, 2,
];

// good
const arr = [[0, 1], [2, 3], [4, 5]];

const objectInArray = [
  {
    id: 1,
  },
  {
    id: 2,
  },
];

const numberInArray = [
  1,
  2,
];
```

2. When programmatically building up strings, use template strings instead of concatenation.
   Using template literals are much more readable and easier to use. It is much more descriptive. It is also a more modular approach. A variable can be directly included in the string. The code is cleaner and more natural to read. Multiline strings are also a benefit, when dealing with long strings.



- 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}?`;
}
```

3. Never use arguments, opt to use rest syntax '...' instead.
   The rest syntax provides a clearer and concise way to handle function parameters. It explicitly declares that the function is expecting a variable number of arguments and allows you to name arguments in a way that makes the code more readable.
   The 'arguments' object is an array-like object, not a true array. Therefore it lacks many of the array methods. The rest syntax creates a real array, which means array methods can be used on it.

- 7.6 Never use `arguments`, opt to use rest syntax `...` instead. eslint: `prefer-rest-params`

  > Why? `...` is explicit about which arguments you want pulled. Plus, rest arguments are a real Array, and not merely Array-like like `arguments`.

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

---

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

1. A function declaration is not a statement. There is a difference between a function declaration and a function expression, but a function declaration is a type of statement.

- 7.4 **Note:** ECMA-262 defines a `block` as a list of statements. A function declaration is not a statement.

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

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

2. In case the expression spans over multiple lines, wrap it in parentheses for better readability. I found this one confusing, because of the lack of familiarity with it. I think this can lead to misinterpretation of intent. If I would use this it would be different to the style preference, especially when thinking about keeping the code style the same.

- 8.3 In case the expression spans over multiple lines, wrap it in parentheses for better readability.

  > Why? It shows clearly where the function starts and ends.

```
// bad
['get', 'post', 'put'].map((httpMethod) => Object.prototype.hasOwnProperty.call(
    httpMagicObjectWithAVeryLongName,
    httpMethod,
  )
);

// good
['get', 'post', 'put'].map((httpMethod) => (
  Object.prototype.hasOwnProperty.call(
    httpMagicObjectWithAVeryLongName,
    httpMethod,
  )
));
```

3. Do not include JavaScript filename extensions.
   This can be confusing because build tools may have default settings that expect it. It can lead to unexpected behavior.
   It can cause confusion whether import/export refers to a module, file or directory.
   Some tools(IDEs) may expect the extension.

- 10.10 Do not include JavaScript filename extensions eslint: `import/extensions`

  > Why? Including extensions inhibits refactoring, and inappropriately hardcodes implementation details of the module you're importing in every consumer.

```
// bad
import foo from './foo.js';
import bar from './bar.jsx';
import baz from './baz/index.jsx';

// good
import foo from './foo';
import bar from './bar';
import baz from './baz';
```
