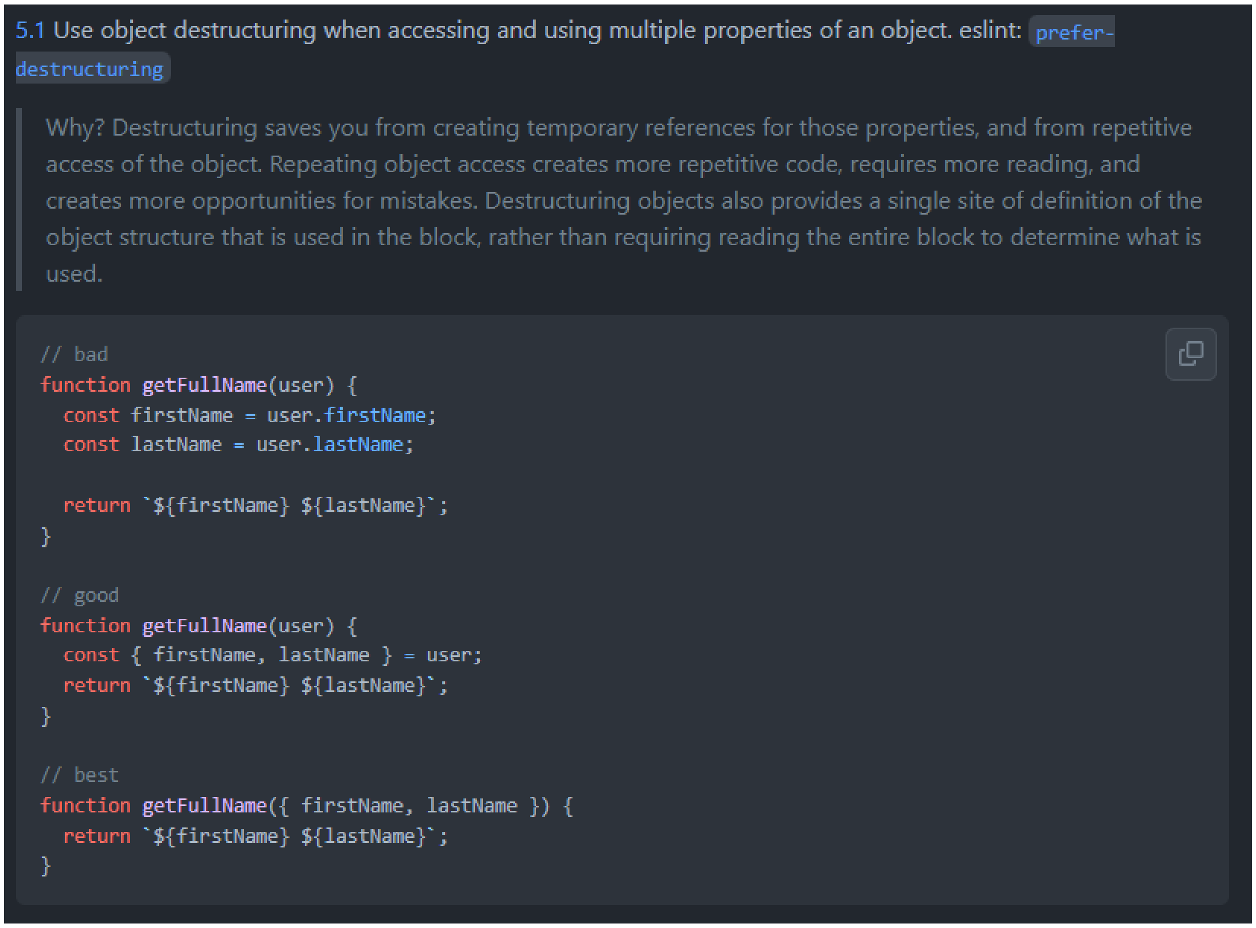
DWA_04.3 Knowledge Check_DWA4

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



5.1 Use object destructuring when accessing and using multiple properties of an object. eslint: `prefer-destructuring`

> Why? Destructuring saves you from creating temporary references for those properties, and from repetitive access of the object. Repeating object access creates more repetitive code, requires more reading, and creates more opportunities for mistakes. Destructuring objects also provides a single site of definition of the object structure that is used in the block, rather than requiring reading the entire block to determine what is used.

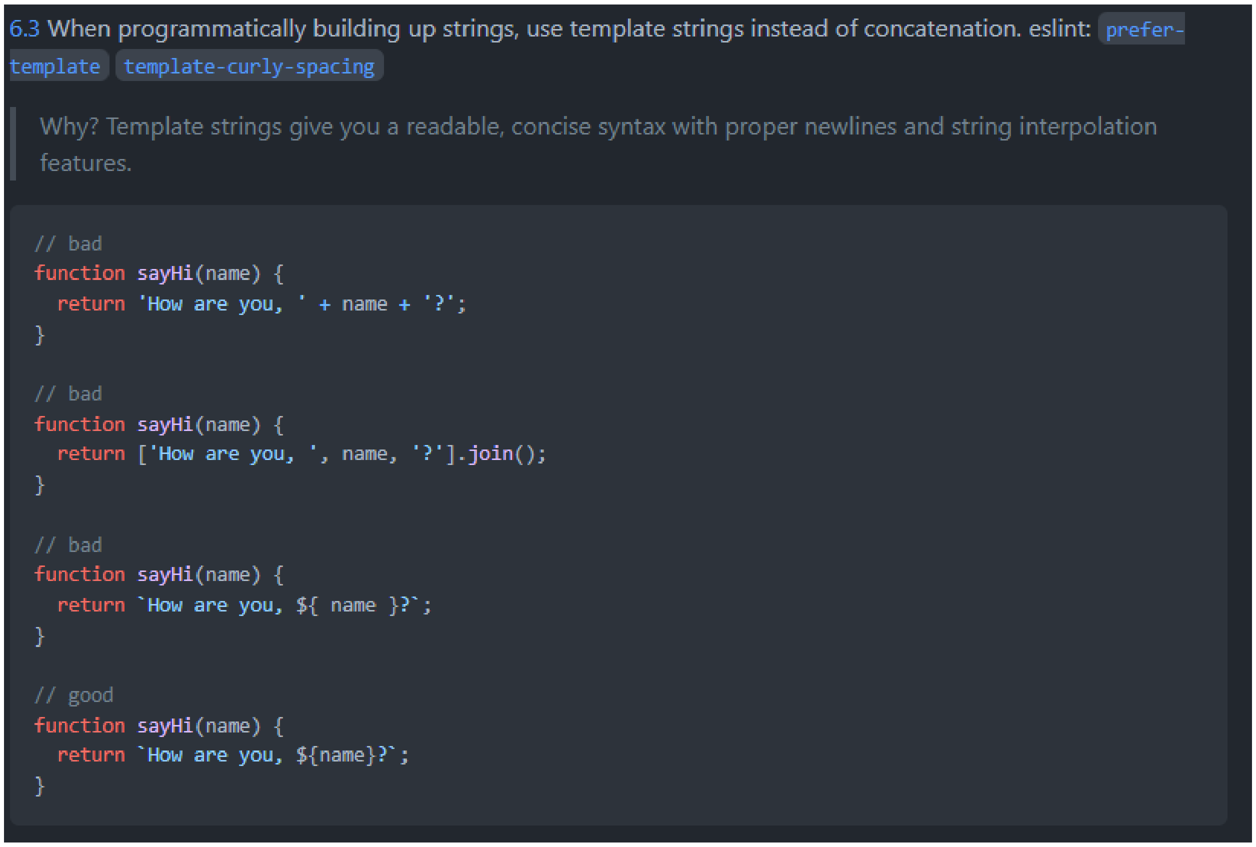
```
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}

// good
function getFullName(user) {
  const { firstName, lastName } = user;
  return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

I find the use of destructuring assignment in general to be very concise and readable. It saves having to write out multiple lines to assign each variable. It also clearly indicates all properties of the object that are being accessed in one place, so you know exactly what the code that follows will be using.


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.

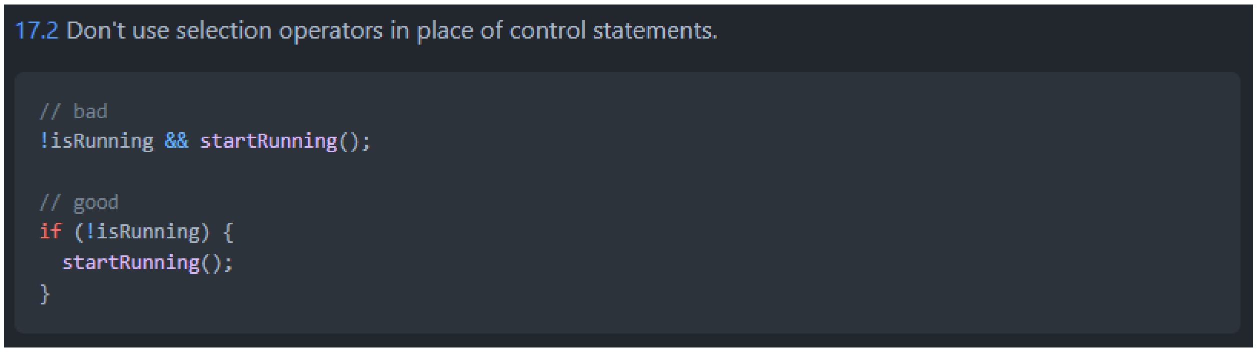
```
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// 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}?`;
}
```

I have found template strings to be incredibly useful. String concatenation often looks sprawled and messy with so many characters interrupting the flow of the text, while template strings are concise, clear, and can be easily split across multiple lines without breaking the string. The ability to interpolate variables and computed values within the string itself also really helps with visualizing the final result and prevents errors with punctuation and spacing.


17.2 Don't use selection operators in place of control statements.

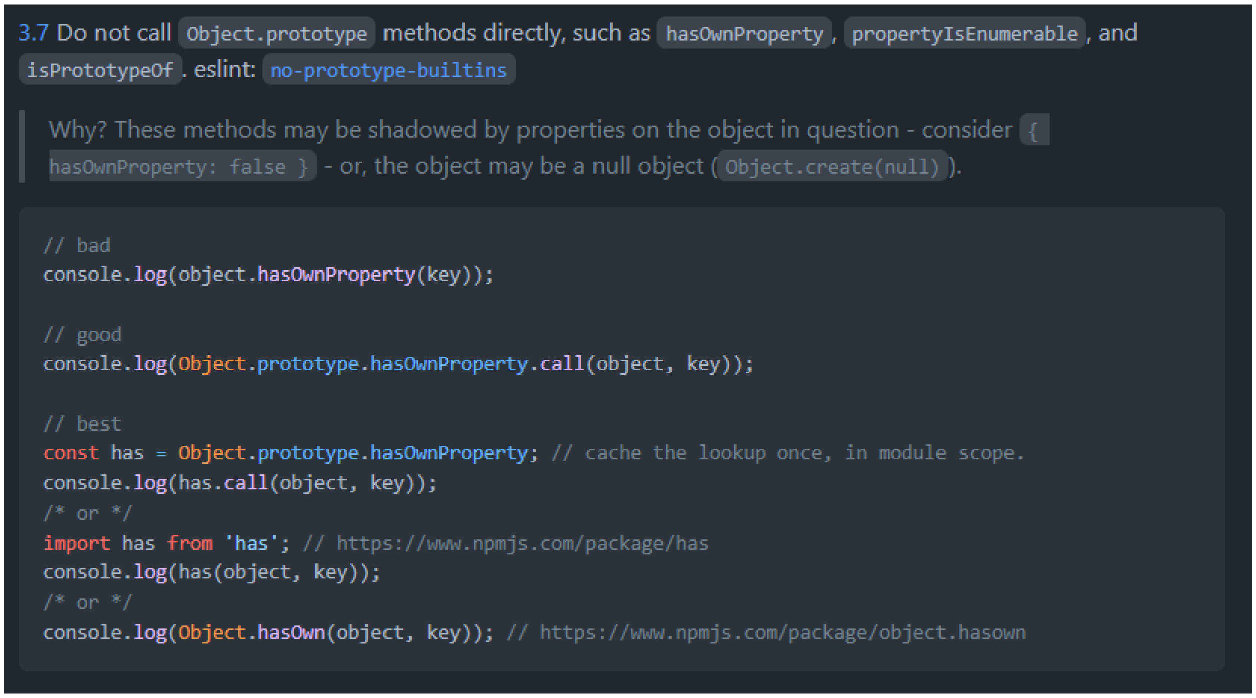
```
// bad
!isRunning && startRunning();

// good
if (!isRunning) {
  startRunning();
}
```

I find this rule very helpful because when you read the code in natural language (i.e. saying “AND” or “OR”) using these operators creates confusion about what the code is actually doing. It is saying “this AND that”, instead of “if this, then do that”. The language doesn’t correspond with the actual function of the code in this context.

---

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



3.7 Do not call `Object.prototype` methods directly, such as `hasOwnProperty`, `propertyIsEnumerable`, and `isPrototypeOf`. eslint: `no-prototype-builtins`

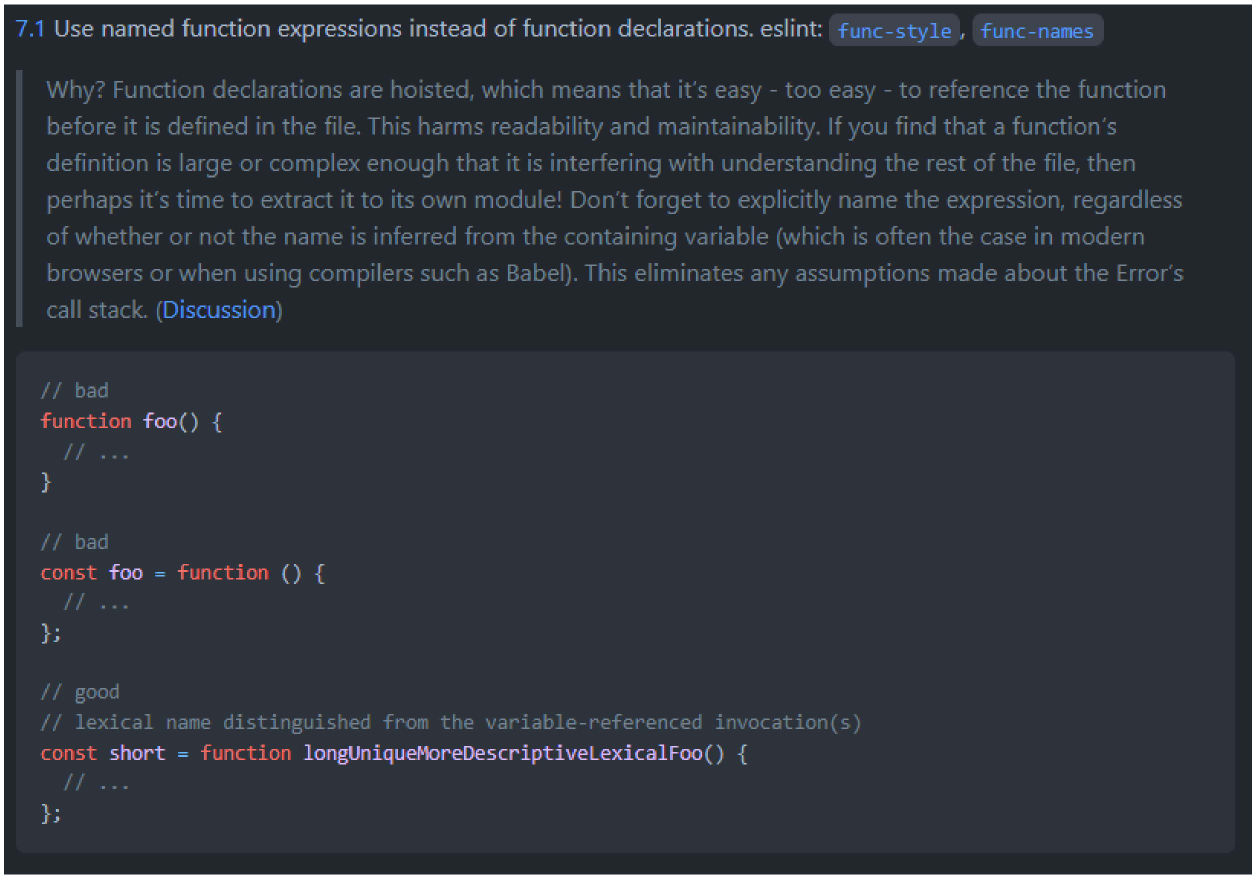
> Why? These methods may be shadowed by properties on the object in question - consider `{ hasOwnProperty: false }` - or, the object may be a null object (`Object.create(null)`).

```
// bad
console.log(object.hasOwnProperty(key));

// good
console.log(Object.prototype.hasOwnProperty.call(object, key));

// best
const has = Object.prototype.hasOwnProperty; // cache the lookup once, in module scope.
console.log(has.call(object, key));
/* or */
import has from 'has'; // https://www.npmjs.com/package/has
console.log(has(object, key));
/* or */
console.log(Object.hasOwn(object, key)); // https://www.npmjs.com/package/object.hasown
```

I find this confusing because I don’t understand what “Object.prototype methods” are and how they differ from regular methods. This makes it hard to see why they shouldn’t be accessed directly - the syntax seems less readable and also needs to use call() method.


7.1 Use named function expressions instead of function declarations. eslint: `func-style`, `func-names`

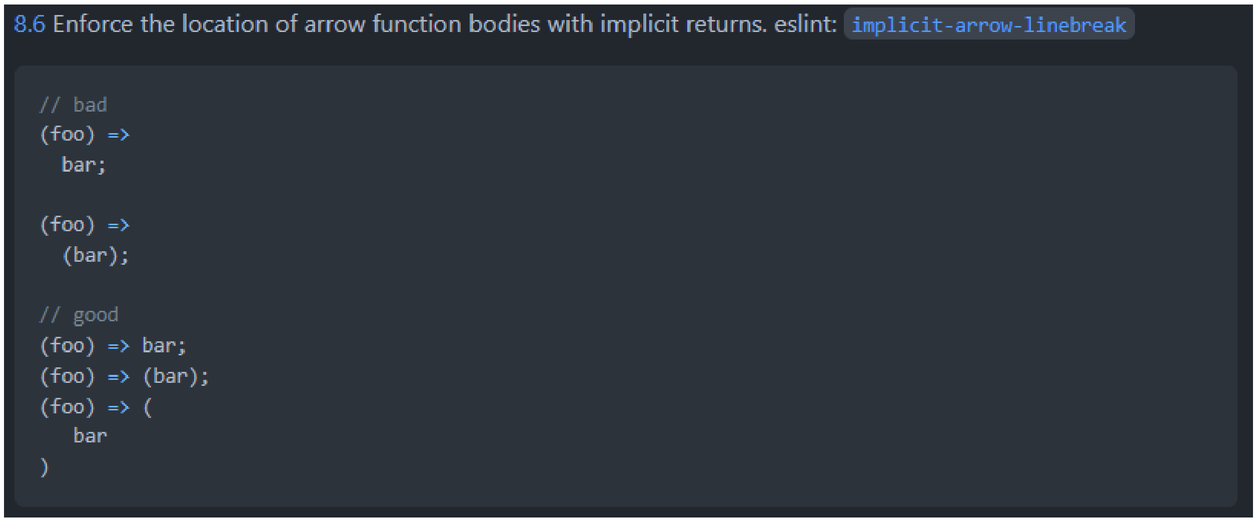
> Why? Function declarations are hoisted, which means that it's easy - too easy - to reference the function before it is defined in the file. This harms readability and maintainability. If you find that a function's definition is large or complex enough that it is interfering with understanding the rest of the file, then perhaps it's time to extract it to its own module! Don't forget to explicitly name the expression, regardless of whether or not the name is inferred from the containing variable (which is often the case in modern browsers or when using compilers such as Babel). This eliminates any assumptions made about the Error's call stack. (Discussion)

```
// bad
function foo() {
  // ...
}

// bad
const foo = function () {
  // ...
};

// good
// lexical name distinguished from the variable-referenced invocation(s)
const short = function longUniqueMoreDescriptiveLexicalFoo() {
  // ...
};
```

I understand the reason for avoiding hoisted function declarations, but I do not understand why functions must always be named. If named functions are preferred, should we be avoiding arrow functions which are anonymous? This conflicts with advice in lectures to use arrow functions as far as possible.


8.6 Enforce the location of arrow function bodies with implicit returns. eslint: `implicit-arrow-linebreak`

```
// bad
(foo) =>
  bar;

(foo) =>
  (bar);

// good
(foo) => bar;
(foo) => (bar);
(foo) => (
   bar
)
```

I find this rule confusing as I am not sure what “enforce the location” means and I don’t really see any functional difference between the good and bad code examples. I don’t see the pattern for what makes the good code examples more correct than the bad ones.
