



JS CLEAN CODE COURSE

APRIL 2022

YONATAN BENEZRA

- DO NOT DISTRIBUTE –

- DO NOT COPY – ALL RIGHTS RESERVED TO THE
AUTHOR –

Variables

Use meaningful and pronounceable variable names.

Bad:

```
const yyyymmddstr = moment().format("YYYY/MM/DD");
```

Good:

```
const currentDate = moment().format("YYYY/MM/DD");
```

Use the same vocabulary for the same type of variable

Bad:

```
getUserInfo();  
getClientData();  
getCustomerRecord();
```

Good:

```
getUser();
```

Use searchable names

We will read more code than we will ever write.

It's important that the code we do write is readable and searchable.

By not naming variables that end up being essential to understanding our program, we hurt our readers. Make your names searchable.

Bad:

```
// What the heck is 86400000 for?  
setTimeout(blastOff, 86400000);
```

Good:



```
// Declare them as capitalized named constants.
```

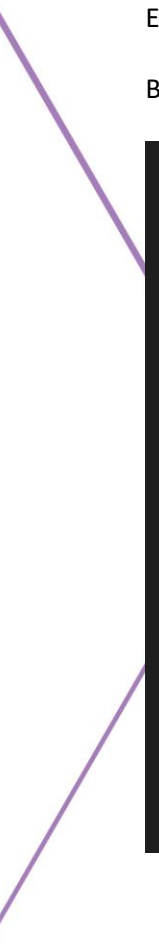
```
const MILLISECONDS_PER_DAY = 60 * 60 * 24 * 1000; //86400000;
```

```
setTimeout(blastOff, MILLISECONDS_PER_DAY);
```

Avoid Mental Mapping

Explicit is better than implicit.

Bad:



```
const locations = ["Austin", "New York", "San Francisco"];
locations.forEach(l => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  // Wait, what is `l` for again?
  dispatch(l);
});
```

Good:



```
const locations = ["Austin", "New York", "San Francisco"];
locations.forEach(location => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  dispatch(location);
});
```

Don't add unneeded context

If your class/object name tells you something, don't repeat that in your variable name.

Bad:

```
const Car = {
  carMake: "Honda",
  carModel: "Accord",
  carColor: "Blue"
};

function paintCar(car, color) {
  car.carColor = color;
}
```

Good:

```
const Car = {  
  make: "Honda",  
  model: "Accord",  
  color: "Blue"  
};  
  
function paintCar(car, color) {  
  car.color = color;  
}
```

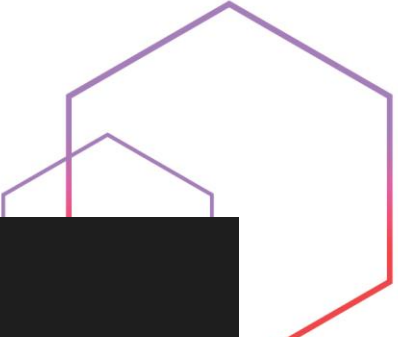
Function arguments

Bad:

```
function createMenu(title, body, buttonText, cancellable) {  
  // ...  
}  
  
createMenu("Foo", "Bar", "Baz", true);
```

Good:

```
function createMenu({ title, body, buttonText, cancellable }) {  
  // ...  
}  
  
createMenu({
```

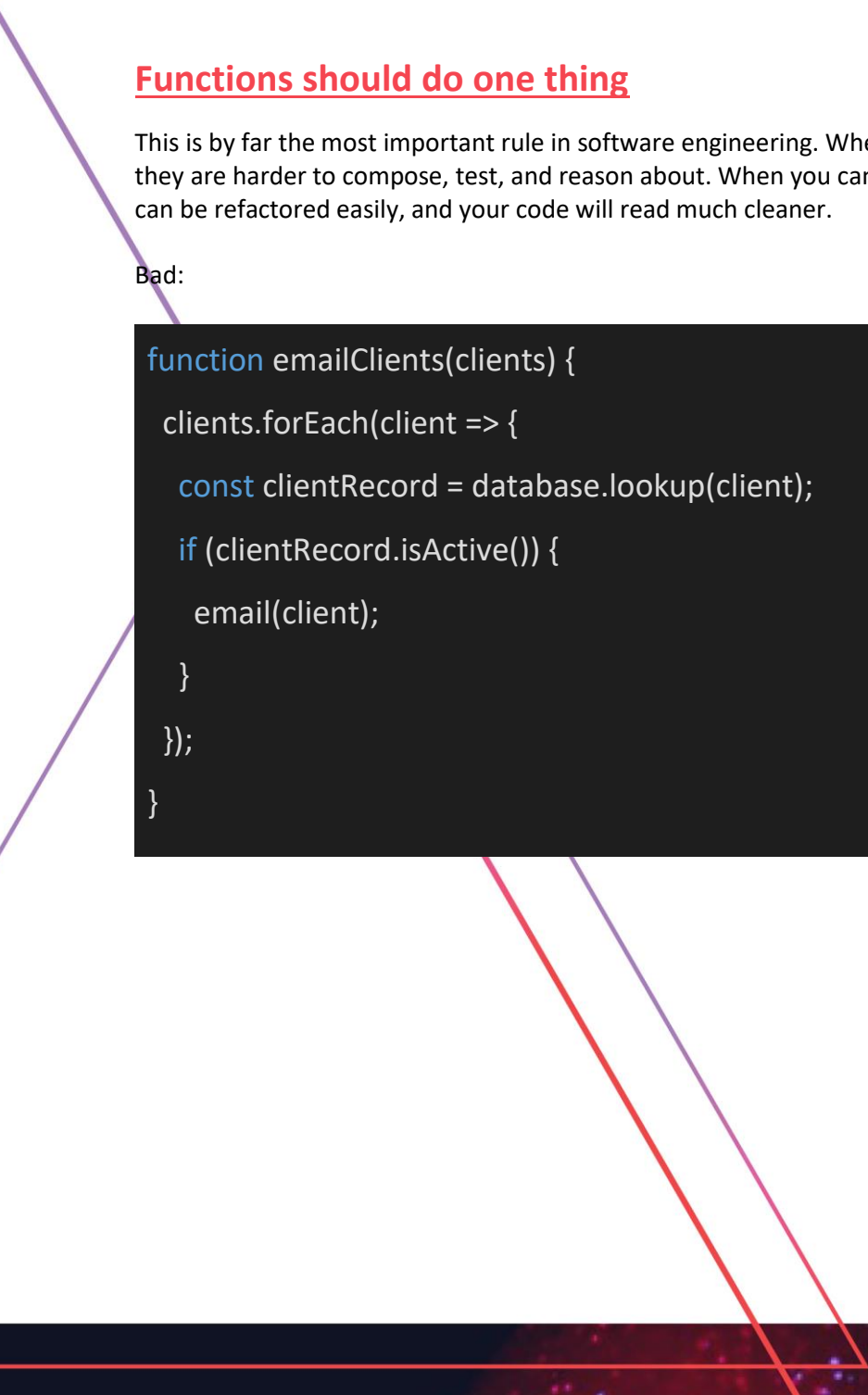


```
title: "Foo",  
body: "Bar",  
buttonText: "Baz",  
cancellable: true  
});
```

Functions should do one thing

This is by far the most important rule in software engineering. When functions do more than one thing, they are harder to compose, test, and reason about. When you can isolate a function to just one action, it can be refactored easily, and your code will read much cleaner.

Bad:



```
function emailClients(clients) {  
  clients.forEach(client => {  
    const clientRecord = database.lookup(client);  
    if (clientRecord.isActive()) {  
      email(client);  
    }  
  });  
}
```

Good:

```
function emailActiveClients(clients) {  
  clients.filter(isActiveClient).forEach(email);  
}  
  
function isActiveClient(client) {  
  const clientRecord = database.lookup(client);  
  return clientRecord.isActive();  
}
```

Function names should say what they do

Bad:

```
function addToDate(date, month) {  
  // ...  
}  
  
const date = new Date();  
  
// It's hard to tell from the function name what is added  
addToDate(date, 1);
```

Good:

```
function addMonthToDate(month, date) {  
  // ...  
}  
  
const date = new Date();  
addMonthToDate(1, date);
```

Avoid Side Effects (part 1)

A function produces a side effect if it does anything other than take a value in and return another value (or values).

A side effect could be writing to a file, modifying some global variable, or accidentally wiring all your money to a stranger.

Now, on occasion, you may need to have side effects in a program.

You might need to write to a file, for example. What you want to do is to centralize where you are doing this. Don't have several functions and classes that write to a particular file. Have one service that does it. One and only one.

The main point is to avoid common pitfalls such as a state of sharing between objects without any structure, using mutable data types that can be written to by anything, and not centralizing where your side effects occur.

Bad:

```
// Global variable referenced by following function.  
// If we had another function that used this name, now it'd be an array and it  
could break it.  
let name = "Ryan McDermott";  
  
function splitIntoFirstAndLastName() {
```



```
name = name.split(" ");  
}  
  
splitIntoFirstAndLastName();  
  
console.log(name); // ['Ryan', 'McDermott'];
```

Good:

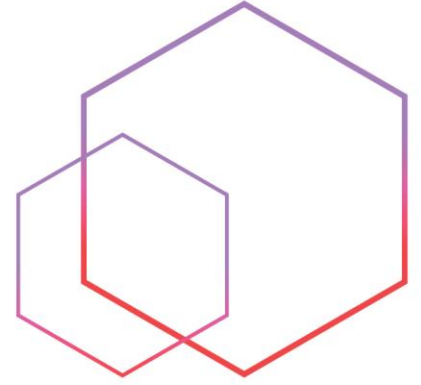
```
function splitIntoFirstAndLastName(name) {  
  return name.split(" ");  
}  
  
const name = "Ryan McDermott";  
const newName = splitIntoFirstAndLastName(name);  
  
console.log(name); // 'Ryan McDermott';  
console.log(newName); // ['Ryan', 'McDermott'];
```

Avoid Side Effects (part 2)

In JavaScript, some values are unchangeable (immutable), and some are changeable (mutable). Objects and arrays are two kinds of mutable values, so it's important to handle them carefully when they're passed as parameters to a function.

A JavaScript function can change an object's properties or alter the contents of an array, which could easily cause bugs elsewhere.

Suppose there's a function that accepts an array parameter representing a shopping cart. If the function makes a change in that shopping cart array - by adding an item to purchase, for example - then any other function that uses that same cart array will be affected by this addition. That may be great, however it could also be bad.



Let's imagine a bad situation:

The user clicks the "Purchase" button which calls a purchase function that spawns a network request and sends the cart array to the server.

Because of a bad network connection, the purchase function has to keep retrying the request.

Now, what if in the meantime, the user accidentally clicks an "Add to Cart" button on an item they don't actually want before the network request begins? If that happens and the network request begins, then that purchase function will send the accidentally added item because the cart array was modified.

A great solution would be for the addItemToCart function to always clone the cart, edit it, and return the clone. This would ensure that functions that are still using the old shopping cart wouldn't be affected by the changes.

Two caveats to mention to this approach:

- There might be cases where you actually want to modify the input object, but when you adopt this programming practice you will find that those cases are pretty rare. Most things can be refactored to have no side effects!
- Cloning big objects can be very expensive in terms of performance. Luckily, this isn't a big issue in practice because there are [great libraries](#) that allow this kind of programming approach to be fast and not as memory intensive as it would be for you to manually clone objects and arrays.

(We will learn about libraries in the React module)

Bad:

```
const addItemToCart = (cart, item) => {  
  cart.push({ item, date: Date.now() });  
};
```

Good:

```
const addItemToCart = (cart, item) => {  
  return [...cart, { item, date: Date.now() }];  
};
```

All the suggestions came from the GitHub branch:

<https://github.com/ryanmcdermott/clean-code-javascript#use-searchable-names>

In my opinion, I covered the most important clean code suggestions, but feel free to go through the GitHub branch and learn everything :)

