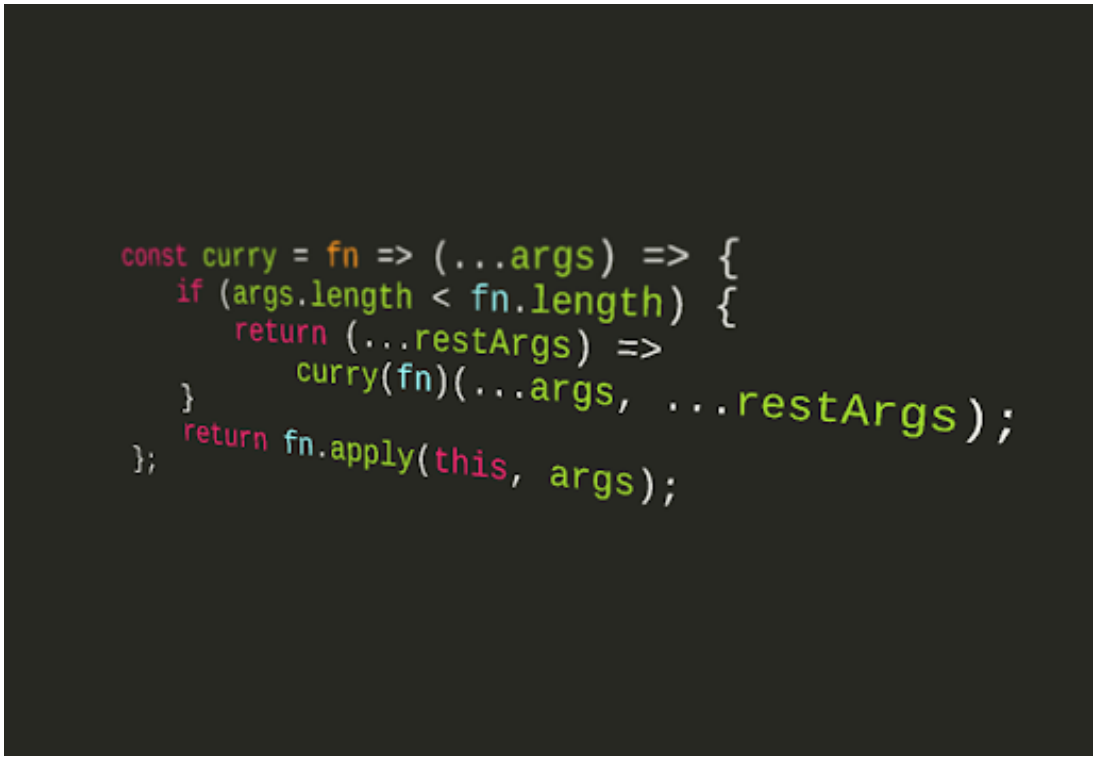
What is point-free programming?

Ferran Buireu Feb 9 · 6 min read ★

Have you ever heard of point-free coding? What is it? When should you use it?



In this article, I will discuss point-free styling, including:

- What point-free programming is
- Practical examples of point-free codes
- The benefits of point-free styling
- How to incorporate point-free styling in your code
- Disadvantages of point-free programming and when not to use it

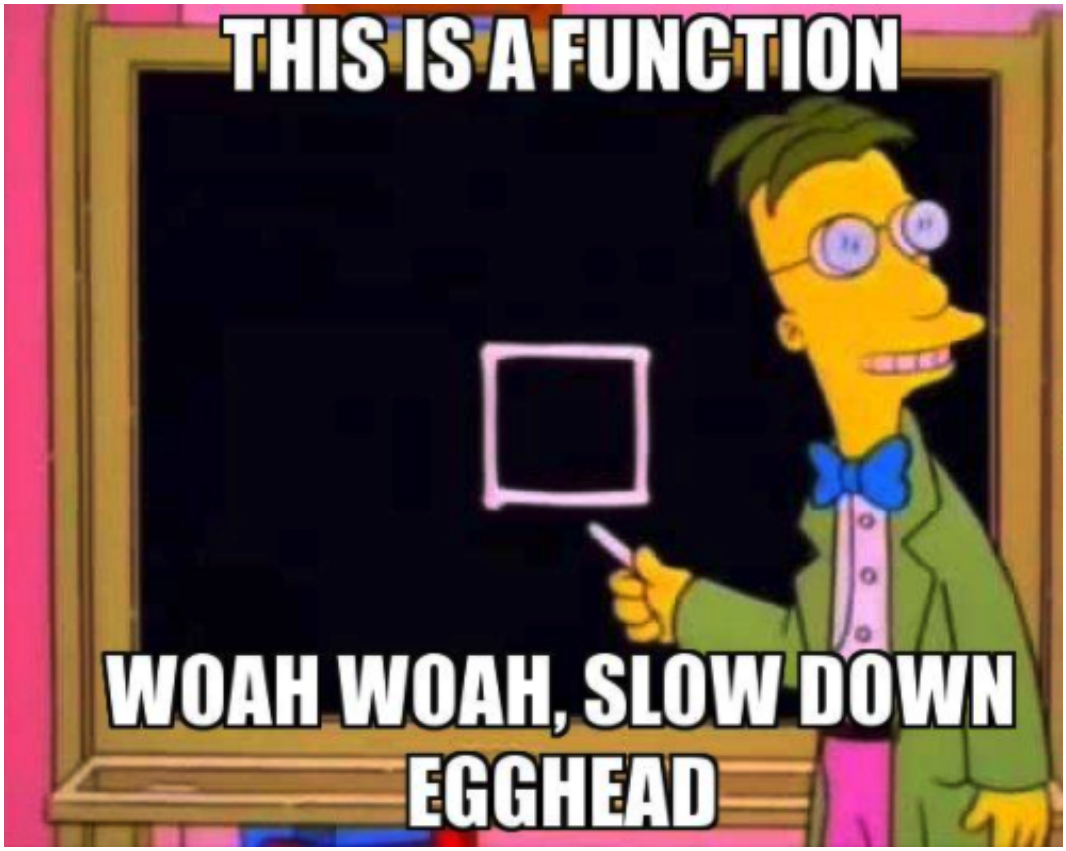
Point-free programming: Definition

Point-free styling (also known as **Tacit programming**), is a programming paradigm in which the **function definitions do not identify or take the arguments** (also known as “points”) on which they operate. Instead, the **definitions are composed**

of other functions themselves, among which are combinators that manipulate the arguments.

In other words, tacit programming functions don't take arguments as we know them, they take other functions and these are the functions, in the end, that take the arguments.

Despite this pattern can be applied in all functional programming languages, like **JavaScript**, **Dart**, **Java**, **Ruby**, **Python**, **Kotlin**, **Haskell**, **C#**, **C++**, etc, I will base this article on JavaScript.



Point-free code: a practical example

As usual, words are but wind. So, here's an example of a

relatively common way of writing a function that returns whether a number is even or not:

```
const isEven = number => number % 2 === 0;

let myNumbers = [1, 2, 3, 4, 5, 6];

console.log(myNumbers.map(myNumber => isEven(myNumber)));

// will output: [false, true, false, true, false, true]
```

Refined into Tacit programming, this should become:

```
const isEven = number => number % 2 === 0;

let myNumbers = [1, 2, 3, 4, 5, 6]

console.log(myNumbers.map(isEven));

// will output: [false, true, false, true, false, true]
```

Wait, what? What's going on?

Note how the point-free function call **does not state the name of the argument nor create a new function**. The reason why this works is that `map` calls its first argument as function, taking the following arguments:

- The item of the array to process: the only mandatory parameter
- The index of that item
- The array itself

The `isEven` function happens to also take the item to be processed as its first (and only) argument and so it works when used point-free in this case.

Benefits of point-free programming

All programs, applications, and code software must follow this basic rule: the **code needs to be easy to read**, **understand** and **maintain**. In other words, it needs to be as simple as it can be.

Programs should be written for people to read, and only incidentally for machines to execute. — Harold Abedson in Structure and Interpretation of Computer Programs

Simplicity

According to this principle, an application that uses redundant arguments may lead to reading and understanding issues.

No variables to name

There are only two hard things in Computer Science: cache

invalidation and naming things — Phil Karlton (Netscape engineer)

Coming up with names for variables and functions can be unexpectedly hard. With point-free styling, you can save yourself some time and effort.

Naming variables explicitly can help you **think about** [illegible] and **improve your code design**.

Given that code is read many more times than it is written, not introducing unnecessary noise in the form of variables may be a very good thing.

Debuggability

One benefit of having all arguments explicitly referred to is **easier debugging**.

You can simply `console.log` parameter values and get an understanding of what is happening in the system in a given line of code. You can easily detect where your code is draining.

Pipe and compose

Let's complicate things a little. `pipe` and `compose` are types of point-free styling that adds extra power to your code by nesting operations.

`compose` is essentially a function that **takes two functions as**

parameters, and runs one after the other:

```
const compose = (f, g) => x => f(g(x));
```

This is the result of `g(x)` that is passed to `f` function.

Applied to something more digestible, it may look like:

```
const add = (a, b) => a + b;
const mult = (a, b) => a * b;

add(2, mult(3, 5));
```

The arithmetic operation above is calculating: `2 + 3 * 5`. As you may know, multiplication has priority over addition. So you start by calculating `3 * 5` and add `2` to the result. This is a form of function composition since it is the result of the multiplication that is passed to the `add` function.

`pipe` is an **intuitive way to read functions in the order they are going to be called** (hence the name). It works exactly as `compose` but applies the functions left-to-right using the `reduceRight` or the `reduce` functions:

```
const pipe = (...fns) => input => fns.reduce((mem, fn) =>
```

```
  fn(mem), input);
```

Given the previous `pipe` we can create three dummy functions to make some simple operations such as:

```
const pipe = (...fns) => input => fns.reduce((mem, fn) =>
fn(mem), input);

const double = x => x * 2;
const addOne = x => x + 1;
const square = x => x * x;
```

At this point, we can proceed as follows:

```
pipe(square, double, addOne)(2)
// same as compose(addOne, double, square)(2)
```

You may notice that this composition of functions is closely related to **Closures and Currying**. Indeed, these types of operations and composing are also forms of currying.

No need to sweat, there are libraries that handle this kind of function composing by themselves, like Rambda (`compose`, and `pipe`) or Lodash (`flowRight` as `compose`, and `flow` as `pipe`). However, it's always useful to know what's happening under the hood to understand the flow, which will allow you to debug, detect and fix potential issues.

Caveats of point-free programming

There are cons or caveats to take into account when overusing point-free styling.

Readability may suffer

One of the obvious downsides to point-free styling is that it **can cause readability issues if taken to the extreme**, for example, by chaining functions and arguments along with an application. It's easy to lose track of the code if you abuse these methods.

If the code is only understandable by the owner, the application is sentenced to death and it will create a handicap in the code because it will be extremely dependant on one person (or a few).

Unexpected `this`

I won't dive much into detail on `this` in JavaScript, but as you may notice, it's easy to lose the context and the scope of functions while chaining and nesting functions with each other.

`this` basically depends on the caller, and, while chaining functions, the caller is the first triggered function so if you are using `this` in a composed function, it will lose the scope.

Unary

While using built-in functions, we may thrive on some **unexpected behaviors**. Given one of the previous snippets:

```
let myStringNumbers= ['1', '2', '3'];

console.log(myStringNumbers.map(number =>
parseInt(number)));

// will output [1, 2, 3]
```

However, when refined to point-free, it results in:

```
let myStringNumbers= ['1', '2', '3'];

console.log(myStringNumbers.map(parseInt));

// will output [1, NaN, NaN]
```

Why's that?

This happens because `map` is passing more than one argument to the `parseInt`. As we've seen before, the `map` loop:

- Passes the current value
- Passes the `index`
- Passes the array itself

While `parseInt` only accepts a string and the radix (the base in mathematical numeral systems). To fix it, it's a common practice when dealing with loops and point-free functions to **use the spread operator along with an auxiliary function** which will allow the function to take only the first argument and ignoring the rest:

```
let myStringNumbers= ['1', '2', '3'];

const unary = fn => (...args) => fn(args[0]);

console.log(myStringNumbers.map(unary(parseInt)));

// will output [1, 2, 3]
```

Conclusion

Point-free programming is a code-style choice and it's **not essential to make your code more functional**. Understanding this concept, however, makes reading and discussing other people's code easier.

As with many things in life (and especially in programming where we like to overkill things), overusing it might introduce more problems than it solves, and you need to trust your intuition (and experience) on **where the balance is**.

Keep it reasonable and ponder whether removing an explicit

variable reference makes the code easier or harder to read. **Use point-free style in moderation** and make your code easier to read and follow.

As much as point-free style is useful in other (functional) languages, its clarity may not be worth the problems it can cause in JavaScript.

I still use it sometimes when the function called is under my control. After experiencing its downsides, however, I will be more careful with it.

There's no evidence that demonstrates higher performance compared to other patterns. However, there is a clear trend in the history of programming towards higher abstractions… and an equally clear history of resistance to this trend.

What do you think? Let me know in the comments below!