Back

👋 I'm Tyler. I'm a Google Developer Expert and a partner at React Training where we teach React online, in person, and build OSS like React Router.

Imperative vs Declarative Programming

Jul 14, 2016

At this point you've undoubtedly heard about imperative programming vs declarative programming. You might have even searched for what those terms actually mean.

Sadly, you probably encountered a definition similar to this, "You know, imperative programming is like **how** you do something, and declarative programming is more like **what** you do, or something."

That definition makes perfect sense once you actually know the difference between imperative and declarative — but you don't, which is why you asked the question in the first place. It's like trying to answer "What came first, the chicken or the egg?" except everyone seems to think the chicken did, but you don't even like eggs, and you're confused.

Combine this frustration with the bastardization of the actual word "declarative" to basically just mean "good" and all of a sudden your imposter syndrome is tap dancing on your confidence, and you realize you don't even like programming that much.

Don’t worry though, friend. I don’t know what a monad is, so hopefully this post will help you realize that declarative is more than just being “easy to reason about” and “good.”

The hard part about this topic is, as Merrick has observed, “It’s one of those things you have an intuition about but can’t seem to explain.”

I’ve talked with many developers and what seems to help most is a combination of metaphors with actual code examples. So buckle up cause I’m about to #preach.

Let’s go back to the initial definition I made fun of, “Imperative programming is like “How” you do something and declarative programming is more like “What” you do.” There’s actually SOME good information hidden in here. Let’s first see the merit in this definition by taking it out of the context of programming and look at a “real life” example.

You decide that you’ve been spending too much time arguing about “JavaScript Fatigue” ™ and Reactive Functional Programming, and your husband deserves a nice date. You decide to go to Red Lobster since you’ve been listening to a lot of Beyonce lately. 👑🐝. You arrive at Red Lobster, approach the front desk and say…

An imperative approach (HOW): I see that table located under the Gone Fishin’ sign is empty. My husband and I are going to walk over there and sit down.

A declarative approach (WHAT): Table for two, please.

The imperative approach is concerned with HOW you’re actually going to get a seat. You need to list out the steps to be able to show HOW you’re going to get a table.The declarative approach is more concerned with WHAT you want, a table for two.

"Ok." — your brain

More metaphors!

I'm going to ask you a question. I want you to think of both an imperative response and a declarative response.

"I'm by Wal-Mart. How do I get to your house from here?"

An imperative response: Go out of the north exit of the parking lot and take a left. Get on I-15 south until you get to the Bangerter Highway exit. Take a right off the exit like you're going to Ikea. Go straight and take a right at the first light. Continue through the next light then take your next left. My house is #298.

A declarative response: My address is 298 West Immutable Alley, Draper Utah 84020

Regardless of how I get to your house, what really matters is the car I drive. Am I going to drive an *imperative* stick shift car or a *declarative* automatic car. Enough metaphors?

Before we dive into code, it's important to realize that many declarative approaches have some sort of imperative abstraction layer. Look at all of our examples:

The declarative response to the Red Lobster employee is assuming that the Red Lobster employee knows all the imperative steps to get us to the table. Knowing the address assumes you have some sort of GPS that knows the imperative steps of how to get to your house.

An automatic car has some sort of abstraction layer over shifting gears.

That was the realization that really made it click for me, so I'll repeat it: Many (if not all) declarative approaches have some sort of underlying imperative abstraction.

If that sentence makes sense, you're doing great!

Now, we're going to attempt to take the leap from metaphorical happy land to real world code land. To make the leap more graceful, let's look at some programming "languages" that are inherently declarative versus those which are imperative by nature.

Imperative: C, C++, Java
Declarative: SQL, HTML
(Can Be) Mix: JavaScript, C#, Python

Think about your typical SQL or HTML example,

```
SELECT * FROM Users WHERE Country=’Mexico’;
```

```
<article>
  <header>
    <h1>Declarative Programming</h1>
    <p>Sprinkle Declarative in your verbiage to sound smart</p>
  </header>
</article>
```

By glancing at both examples, you have a very clear understanding of what is going on. They're both declarative. They're concerned with WHAT you want done, rather than HOW you want it done.

You're describing what you're trying to achieve, without instructing how to do it. The implementation of selecting all of the users who live in Mexico has been abstracted from you. You're not concerned with how the web browser is parsing your *article* and displaying it to the screen. Your WHAT is *Mexican users* and a *new header* and *paragraph* on your website.

So far so good. Let's dive into more practical JavaScript examples.

I want you to pretend you're now in a technical interview and I'm the interviewer. Open up your console and answer the following questions.

1. Write a function called *double* which takes in an array of numbers and returns a new array after doubling every item in that array. *double([1,2,3]) -> [2,4,6]*

2. Write a function called *add* which takes in an array and returns the result of adding up every item in the array. *add([1,2,3]) -> 6*

3. Using jQuery (or vanilla JavaScript), add a *click* event handler to the element which has an id of *btn*. When clicked, toggle (add or remove) the *highlight* class as well as change the text to *Add Highlight* or *Remove Highlight* depending on the current state of the element.

Let’s look at the most common approaches to these problems, which all happen to also be imperative approaches.

```
function double (arr) {
  let results = []
  for (let i = 0; i < arr.length; i++){
    results.push(arr[i] * 2)
  }
  return results
}
```

2.

```
function add (arr) {
  let result = 0
  for (let i = 0; i < arr.length; i++){
    result += arr[i]
  }
  return result
}
```

3.

```
$("#btn").click(function() {
  $(this).toggleClass("highlight")
  $(this).text() === 'Add Highlight'
    ? $(this).text('Remove Highlight')
    : $(this).text('Add Highlight')
})
```

By examining what all three of these imperative examples have in common, we’ll be able to better identify what actually makes them imperative.

1. The most obvious commonality is that they’re describing **HOW** to do something. In each example we’re either explicitly iterating over an

array or explicitly laying out steps for how to implement the functionality we want.

2. This one might not be as obvious if you’re not used to thinking in the *declarative* or even more specifically *functional* way. In each example we’re mutating some piece of state (If you’re unfamiliar with the term state, it’s basically information about something held in memory — which should sound a lot like variables). In the first two examples we create a variable called results and then we continually modify it. In the third example we don’t have any variables, but we still have state living in the DOM itself — we then modify that state in the DOM.

3. This one is a bit subjective, but to me the code above isn’t very readable. I can’t just glance at the code and understand what’s going on. My brain needs to step through the code just as an interpreter would while also taking into account the context in which the code lives(another negativity of mutable data).

All right, enough 💩ing on the code. Let’s now take a look at some declarative examples. The goal is to fix all the problems from above. So each example needs to describe **WHAT** is happening, can’t mutate state, and should be readable at a glance.

1.

```
function double (arr) {
  return arr.map((item) => item * 2)
}
```

2.

```
function add (arr) {
  return arr.reduce((prev, current) => prev + current, 0)
}
```

3.

```
<Btn
  onToggleHighlight={this.handleToggleHighlight}
  highlight={this.state.highlight}>
    {this.state.buttonText}
</Btn>
```

Much better 🤓

Notice that in the first two example we're leveraging JavaScript's built in *map* and *reduce* methods. This goes back to what we've been talking about over and over in this article that most declarative solutions are an abstraction over some imperative implementation.

In every example we're describing WHAT we want to happen rather than HOW (we don't know HOW map and reduce are implemented, we also don't care). We're not mutating any state. All of the mutations are abstracted inside of *map* and *reduce*. It's also more readable (once you get used to *map* and *reduce*, of course).

Now what about #3? Well I cheated a little bit, and am using React — but note that all three imperative mistakes are still fixed. The real beauty of React is that you can create these declarative user interfaces. By looking at our Btn component, I'm able to easily understand what the UI is going to look like. Another benefit is instead of state living in the DOM, it lives in the React component itself.

Another less-spoken-of benefit to declarative code is that your program can be context-independent. This means that because your code is concerned with what the ultimate goal is— rather than the steps it takes to accomplish that goal — the same code can be used in different programs, and work just fine.

Look at all three of our examples above. We can consume both functions and component in any program we want. They're program agnostic. This is hard to do with imperative code because often times, by definition, imperative code relies on the context of the current state.

One thing that I didn't go too far into is how functional programming is a subset of declarative programming. If you haven't already, I highly recommend getting more familiar with functional programming techniques in JavaScript. Start with *.map*, *.reduce*, *.filter* and work your way up from there.

Odds are there isn't a lower hanging fruit to improve your codebase than making it more functional.

Here are some other definitions that I've found on the internet that may or may not be helpful.

> *Declarative programming is "the act of programming in languages that conform to the mental model of the developer rather than the operational model of the machine".*

> *Declarative Programming is programming with declarations, i.e. declarative sentences.*

> *The declarative property is where there can exist only one possible set of statements that can express each specific modular semantic.The imperative property is the dual, where semantics are inconsistent under composition and/or can be expressed with variations of sets of statements.*

> *Declarative languages contrast with imperative languages which **specify explicit manipulation of the computer's internal state**; or procedural languages which specify an explicit sequence of steps to follow.*

> *In computer science, declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow.*

👋 I'm Tyler. I'm a Google Developer Expert and a partner at React Training where we teach React online, in person, and build OSS like React Router.