Ian Mundy Follow
Software Engineer at Faithlife Corporation. Fan of all things Javascript—especially React Native.
Feb 20, 2017 · 4 min read

Declarative vs Imperative Programming

Or wrong ways I was thinking about React

If you've read the React docs, you probably have seen that React is declarative. This is normally the type of thing I skip over—a not so good habit of mine. (Give me the code samples already!) But as someone who started learning React soon out of college, I was often thinking about problems incorrectly.

This isn't surprising. College CS can be an exercise is writing hacky scripts to solve one off problems. That was great for my coding interviews, but not so great for web development at scale. And that's not a knock on my education—it provided me with a lot of the tools I needed to get started. It's my job to continue to grow and become a better developer.

What are Declarative and Imperative Programming?

According to The Information:

> ***Declarative programming*** *is a programming paradigm … that expresses the logic of a computation without describing its control flow.*

> ***Imperative programming*** *is a programming paradigm that uses statements that change a program's state.*

"Programming Paradigm" sounds super pretentious and is definitely a phrase some of my college profs *loved.* If you've had a formal CS education, these definitions may not be that cryptic, but still I find that examples go a long way in helping.

Before I get too far though, I want to stress that there is nothing wrong with either of these approaches. They absolutely have strengths and weaknesses, but one approach will always be right. For instance, declarative programming can result in less direct control, which may not be correct for applications like embedded systems where "the right answer delivered too late becomes the wrong answer."

Alright here's a metaphor.

Declarative Programming is like asking your friend to draw a landscape. You don't care how they draw it, that's up to them.

Imperative Programming is like your friend listening to Bob Ross tell them how to paint a landscape. While good ole Bob Ross isn't exactly commanding, he is giving them step by step directions to get the desired result.

Tell Me In Code

Alright, alright, here's some code examples. Here we just have a button that changes it's color on click. I'll start with the imperative example:

```
const container = document.getElementById('container');
const btn = document.createElement('button');


btn.className = 'btn red';
btn.onclick = function(event) {
 if (this.classList.contains('red')) {
   this.classList.remove('red');
   this.classList.add('blue');
 } else {
   this.classList.remove(‘blue’);
   this.classList.add(‘red’);
 }
};

container.appendChild(btn);
```

And our declarative React example:

```
class Button extends React.Component{

  this.state = { color: 'red' }

  handleChange = () => {
    const color = this.state.color === 'red' ? 'blue' :
'red';
    this.setState({ color });
  }

  render() {
    return (<div>
      <button
         className=`btn ${this.state.color}`
         onClick={this.handleChange}>
      </button>
    </div>);
  }
}
```

The differences here may be subtle. We still have logic that says if red then blue, but there’s one huge difference. The React example never actually touches an element. it simply declares an element *should* be rendered given our current state. It does not actually manipulate the DOM itself.

Reaching for direct DOM manipulation is a mistake I felt myself making often when I first started working with React. It’s also a problem I’ve noticed in developers coming from a background that’s heavy in jQuery.

When writing React, it’s often good not to think of *how* you want to accomplish a result, but instead *what* the component should look like in it’s new state. This sets us up for a good control flow where state goes through a series of predictable and replicable mutations. This doesn't just apply to components either, but also to application state. Libraries like Redux will help enforce this method of architecting, but they aren't necessary to achieve it.

To me, application structure is the most important consideration when you want to embrace a declarative approach. It's often easy to follow on a single component, but becomes harder at scale. It's tempting to make two components communicate with each other in a hacky way, or to settle for passing information only halfway back up the tree. But a consistent approach to state management will make your application easier to reason about as a whole.

Things To Avoid

I've either made, or allowed all of these mistakes in code I've worked on. They're just pieces of advice; do with them what you will.

- Avoid Refs. Sure, sometimes they're necessary, but if you feel yourself reaching for refs then it can often mean you're doing something wrong. (There are some legitimate uses for refs in the React docs).
- Attempting to manipulate DOM directly (this goes *very* closely with Refs)
- Instead of returning functions that render a component, prefer to return functions that return the necessary information to render a component. In the first we are instructing what to do(render precisely this thing), while in the second we're just returning some information (use this information to do something).
- Communicating between siblings, instead of through components. Try to *only* communicate with other components through props.
- Use pure functional components where possible. Because these components don't have lifecycle methods, they require you to rely on a declarative, props-based approach. And they can also provide performance improvements.