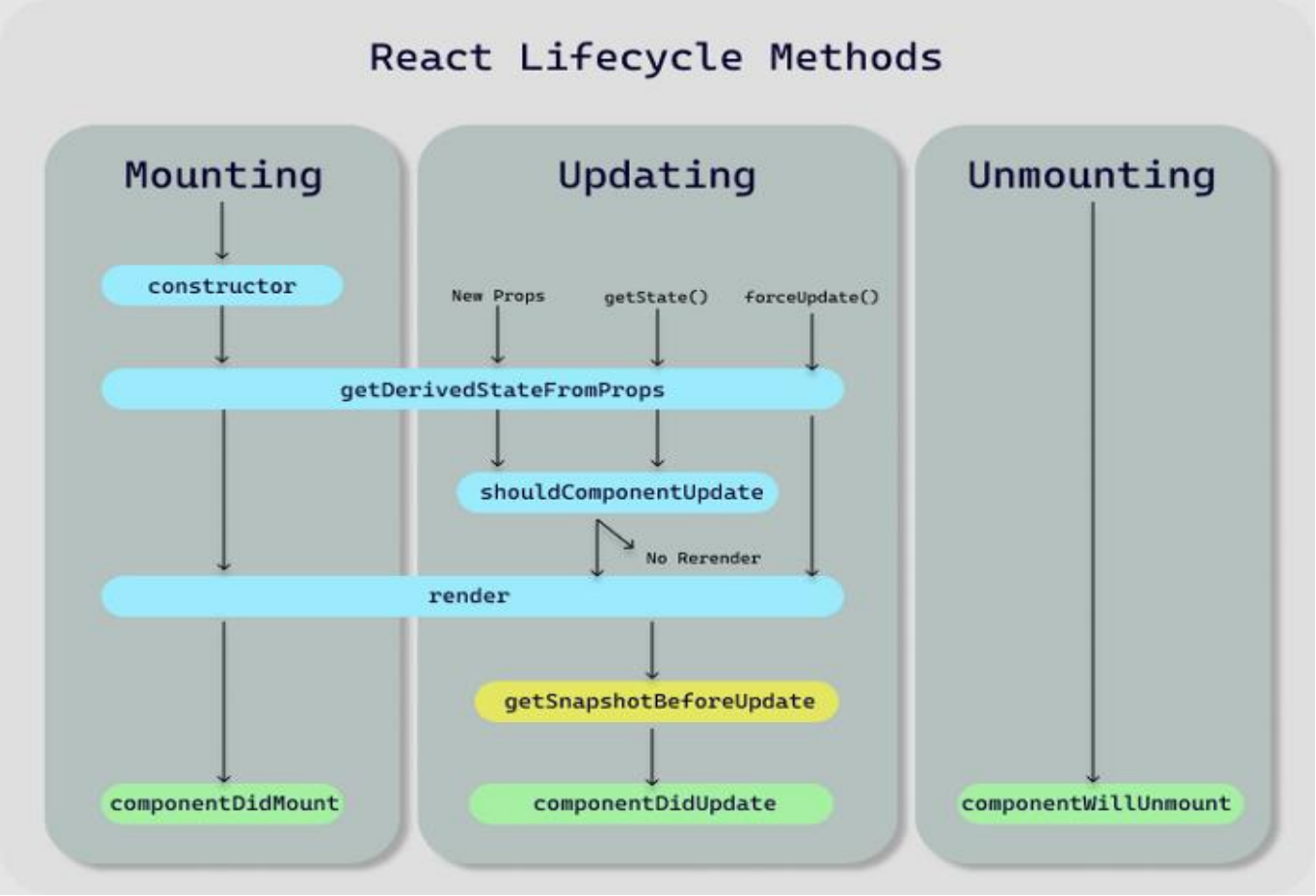
React Function Component Lifecycle

Lifecycle methods are custom functionality that gets executed during the different phases of a component. There are methods available when the component gets created and inserted into the DOM (mounting), when the component updates, and when the component gets unmounted or removed from the DOM.

Here are the lifecycle methods of a class component:



We want to know how each of these methods can be implemented in the component function.

Constructors()

In class-based components, we often see code that uses a constructor to initialize state, like this:

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      counter: 0
    };
  }

  render = () => {
    return (
      <button
        onClick={() =>
          this.setState(prevState => {
            return { counter: prevState.counter + 1 };
          })
        }
      >
        Increment: {this.state.counter}
      </button>
    );
  };
}
```

But the exact same thing can be done like this:

```
class App extends Component {
  state = { counter: 0 };

  render = () => {
    return (
      <button
        onClick={() =>
          this.setState(prevState => {
            return { counter: prevState.counter + 1 };
          })
        }
      >
        Increment: {this.state.counter}
      </button>
    );
  };
}
```

As you see, You can declare initial state directly under the class. when you look at the FAQ for Hooks in reactjs.org, it has a section dedicated to answering, “How do lifecycle methods correspond to Hooks?” The first bullet point in this section says:

`constructor`: Function components don’t ***need*** (emphasis: *mine*) a constructor. You can initialize the state in the `useState` call. If computing the initial state is expensive, you can pass a function to `useState`.

when I think of a constructor, I think of these characteristics.

1. Code that runs *before anything else in the life-cycle of this component*.
2. Code that runs once, *and only once*, for the entire life-cycle of this component.

So despite the Hooks team’s bold assertions, the fact is that there *are* times when I *do* need a constructor (or some equivalent).

Custom Hooks to the Rescue

we can *import* the "constructor-like" functionality into any functional component where it's needed. The code looks like this:

```
const useConstructor(callBack = () => {}) => {
  const [hasBeenCalled, setHasBeenCalled] = useState(false);
  if (hasBeenCalled) return;
  callBack();
  setHasBeenCalled(true);
}

const App = () => {
  useConstructor(() => {
```

```
    console.log(
      "This only happens ONCE.
       it happens BEFORE the initial render."
    );
  });
  const [counter, setCounter] = useState(0);

  return (
    <>
      <div>Counter: {counter}</div>
      <div style={{ marginTop: 20 }}>
        <button onClick={() => setCounter(counter + 1)}>
           Increment
        </button>
      </div>
    </>
  );
};
```

If you place that logic at the very top of your function declaration, it is, effectively, a “constructor”, for all intents and purposes.

GetDerivedStateFromProps()

While you probably [don’t need it](), in rare cases that you do (such as implementing a `<Transition>` component), you can update the state right during rendering. React will re-run the component with updated state immediately after exiting the first render so it wouldn’t be expensive.

Here, we store the previous value of the *row* prop in a state variable so that we can compare:

```
function ScrollView({row}) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);  if (row !==
prevRow) {
    // Row changed since last render. Update isScrollingDown.
    setIsScrollingDown(prevRow !== null && row > prevRow);
    setPrevRow(row);
  }  return `Scrolling down: ${isScrollingDown}`;
}
```

an update during rendering is exactly
what *getDerivedStateFromProps* has always been like conceptually.

Render()

In a class-based component, we had the comfort (IMHO) of a `render()` function. And if a particular bit of logic should *not* run on *every* re-render, the process was pretty straight-forward: Just don't put that logic *in* the `render()` function.

But functional components offer no out-of-the-box equivalent. This is the function component body itself. There *is no* `render()` function. There is only a `return`. The `return` (and all the rest of the code in the body of the function) gets called *every single time this function is called*.

```
const App = () => {
  const [counter, setCounter] = useState(0);

  console.log("This happens on EVERY render.");  return (
    <>
      <div>Counter: {counter}</div>
      <div style={{ marginTop: 20 }}>
        <button onClick={() => setCounter(counter +
1)}>Increment</button>
      </div>
    </>
  );
};
```

ComponentWillMount(), ComponentDidMount(), ComponentDidUpdate(), ComponentWillUnmount()

The *useEffect* hook can express all combinations of these. To understand how we can use *componentWillMount* with functional components, first we need to look at how the component manages mounting with useEffect.

```
const App = () => {
  useEffect(() => {
    console.log(
      "This only happens ONCE. It happens AFTER the initial
render."
    );
  }, []);
};
```

If you add a return function inside the useEffect function, it is triggered when a component unmount from the DOM. This looks like:

```
const App = () => {
  useEffect(() => {
    return () => {
      console.log(
        "This only happens ONCE.
         Anything in here is fired on component UNMOUNT."
      );
     }
   }, []);
};
```

Combining Both Solutions:

This means that you can use componentDidMount and componentWillMount in the same useEffect function call. Dramatically reducing the amount of code needed to manage both life-cycle events. This means you can easily use componentDidMount and componentWillMount within functional components.

Like so:

```
const App = () => {
  useEffect(() => {
    console.log(
      "This only happens ONCE.
       Anything in here is fired on component MOUNT."
    );
    return () => {
      console.log(
        "This only happens ONCE.
         Anything in here is fired on component UNMOUNT."
      );
    }
  }, []);
};
```

componentWillMount *and* componentWillUnmount*, are very similar!*

`ShouldComponentUpdate()`

You can wrap a function component with *React.memo* to shallowly compare its props:

```
const Button = React.memo((props) => {
  // your component
});
```

It's not a Hook because it doesn't compose like Hooks do. *React.memo* is equivalent to PureComponent, but it only compares props. (You can also add a second argument to specify a custom comparison function that takes the old and new props. If it returns true, the update is skipped.)

React.memo doesn't compare state because there is no single state object to compare. But you can make children pure too, or even optimize individual children with *useMemo*.

getSnapshotBeforeUpdate(), componentDidCatch() and getDerivedStateFromError()

There are no Hook equivalents for these methods yet, but they will be added soon.

Conclusion

The react component lifecycle can be intimidating at first and it can be a bit confusing but hopefully this has given you a better understanding of this concept in function components.