Why React Hooks

 React.js is all about **state management** and **side effects**

Principles of React

1. **Declarative**

   Instead of telling React how to do things, we tell it what we want it to do.
2. **Component-based**
3. **Learn once, write anywhere**

In React, there are two types of components:

- **Function components**: JavaScript functions that take the props as an argument, and return the user interface (usually via JSX)
- **Class components**: JavaScript classes that provide a `render` method, which returns the user interface (usually via JSX)

Motivation for using React Hooks:

1. **Confusing classes**

- this

- In a method, `this` refers to the class object (instance of the class).
- In an event handler, `this` refers to the element that received the event.
- In a function or when standing alone, `this` refers to the global object. For example, in a browser, the global object is the `Window` object.
- In strict mode, `this` is `undefined` in a function.
- Additionally, methods such as `call()` and `apply()` can change the object that `this` refers to, so it can refer to any object.

- classes sometimes require us to write code in multiple places at once,

  For example, if we want to fetch data when the component renders, or the data updates, we need to do this using two methods: once in

componentDidMount, and once in **componentDidUpdate**.

```
class Example1 extends React.Component {
    componentDidMount () {
        fetch(`http://my.api/${this.props.name}`)
            .then(...)
    }
    componentDidUpdate (prevProps) {
        if (this.props.name !== prevProps.name) {
            fetch(`http://my.api/${this.props.name}`)
                .then(...)
        }
    }
}
```

```
class Example2 extends React.Component {
    componentDidMount () {
        fetch(`http://my.api/${this.props.name}`)
            .then(...)
    }
    componentDidUpdate (prevProps) {
        if (this.props.name !== prevProps.name) {
            fetch(`http://my.api/${this.props.name}`)
                .then(...)
        }
    }
}
```

---

2. Wrapper hell

if we wanted to encapsulate state management logic, we had to use **higher-order components** and **render props.**

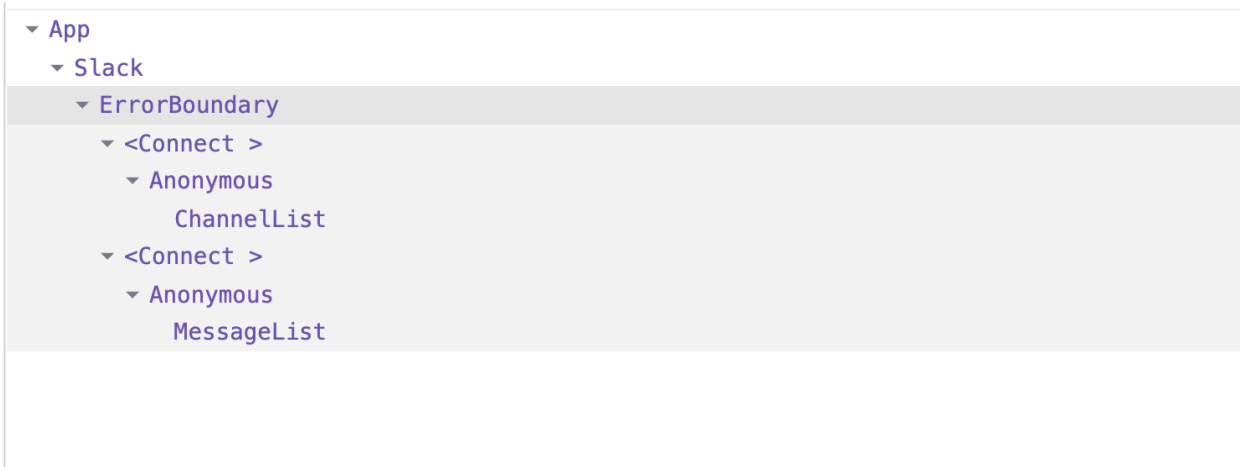
```
<AuthenticationContext.Consumer>
    {user => (
```

```
        <LanguageContext.Consumer>
            {language => (
                <StatusContext.Consumer>
                    {status => (
                        ...
                    )}
                </StatusContext.Consumer>
            )}
        </LanguageContext.Consumer>
    )}
</AuthenticationContext.Consumer>
```

This is not very easy to read or write, and it is also prone to errors if we need to change something later on

Furthermore, the wrapper hell makes debugging hard, because we need to look at a large component tree, with many components just acting as wrappers/containers.

our earlier example as well 😔



Hooks to the rescue!

Using **Hooks**, we can solve all the previously mentioned problems.

We do not need to use class components anymore, because Hooks are simply functions that can be called in function components.

We also do not need to use higher-order components and render props for contexts anymore, because we can simply use a Context Hook to get the data that we need.

Furthermore, Hooks allow us to reuse stateful logic between components, without creating higher-order components.

```
function Example ({ name }) {
    useEffect(() => {
        fetch(`http://my.api/${this.props.name}`)
            .then(...)
    }, [ name ])
    // ...
}
```

```
const user = useContext(AuthenticationContext)
const language = useContext(LanguageContext)
const status = useContext(StatusContext)
```

Now that we know which problems Hooks can solve, let's get started using Hooks in practice!
