Functional Components

React Hooks

Functional Components

- A functional component is just a plain JavaScript function that accepts props as an argument and returns a React element.
- Whereas in a class component requires you to extend from React. Component and create a render function which returns a React element.
- There is no render method used in functional components.

Why we go for Functional Comp

- There are some benefits you get by using functional components in React.
- Functional component are much easier to read and test because they are plain JavaScript functions without state or lifecycle-hooks.
- You end up with less code. That’s help you to use best practices.

Advantages of Functional Comp

- The syntax is less complex than that of class components, and it's easier to read due to knowing how much you can't do with functional components already.
- The use of prop destructuring makes it really beneficial to see what's going on and what's coming out of the component.

React Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

Completely opt-in

You can try Hooks in a few components without rewriting any existing code.

100% backwards-compatible

This means that once something is accepted as valid JS, there will not be a future change to the language that causes that code to become invalid JS.

Why we need Hooks…?

It’s hard to reuse stateful logic between components.

React doesn’t offer a way to “attach” reusable behaviour to a component (for example, connecting it to a store).

- With Hooks, you can extract stateful logic from a component so it can be tested independently and reused.
- Hooks allow you to reuse stateful logic without changing your component hierarchy.
- This makes it easy to share Hooks among many components or throughout the container.

Contd..

- **Complex components become hard to understand**

Hooks let you split one component into smaller functions based on what pieces are related rather than forcing a split based on lifecycle methods.

- **Classes confuse both people and machines**
- **Gradual Adoption Strategy**

State Hook

- It declares a “state variable”.
- useState is a new way to use the exact same capabilities that this.state provides in a class.
- The only argument to the useState() Hook is the initial state.
- It returns a pair of values: the current state and a function that updates it.

You might be wondering: why is useState not named createState instead?

“Create” wouldn’t be quite accurate because the state is only created the first time our component renders. During the next renders, useState gives us the current state. Otherwise it wouldn’t be “state” at all!

Contd..

What Do Square Brackets Mean?

ex: *const [count, setCount] = useState(0);*

- The names on the left aren’t a part of the React API. You can name your own state variables.
- This JavaScript syntax is called “array destructuring”. It means that we’re making two new variables count and setCount.
- When we declare a state variable with useState, it returns a pair — an array with two items. The first item is the current value, and the second is a function that lets us update it.

Effect Hook

- The *Effect Hook* lets you perform side effects in function components.
- Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.
- If you’re familiar with React class lifecycle methods, you can think of useEffect Hook as ***componentDidMount***, ***componentDidUpdate***, and **componentWillUnmount** combined.

Effect Types

There are two common kinds of side effects in React components: those that don’t require clean-up, and those that do.

Sometimes, we want to run some additional code after React has updated the DOM. Network requests, manual DOM mutations, and logging are common examples of effects that don’t require a clean-up. We say that because we can run them and immediately forget about them.

What it will do..?

By using useEffect Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we’ll refer to it as our “effect”), and call it later after performing the DOM updates.

By default, it runs both after the first render *and* after every update.

Effects with Clean-up

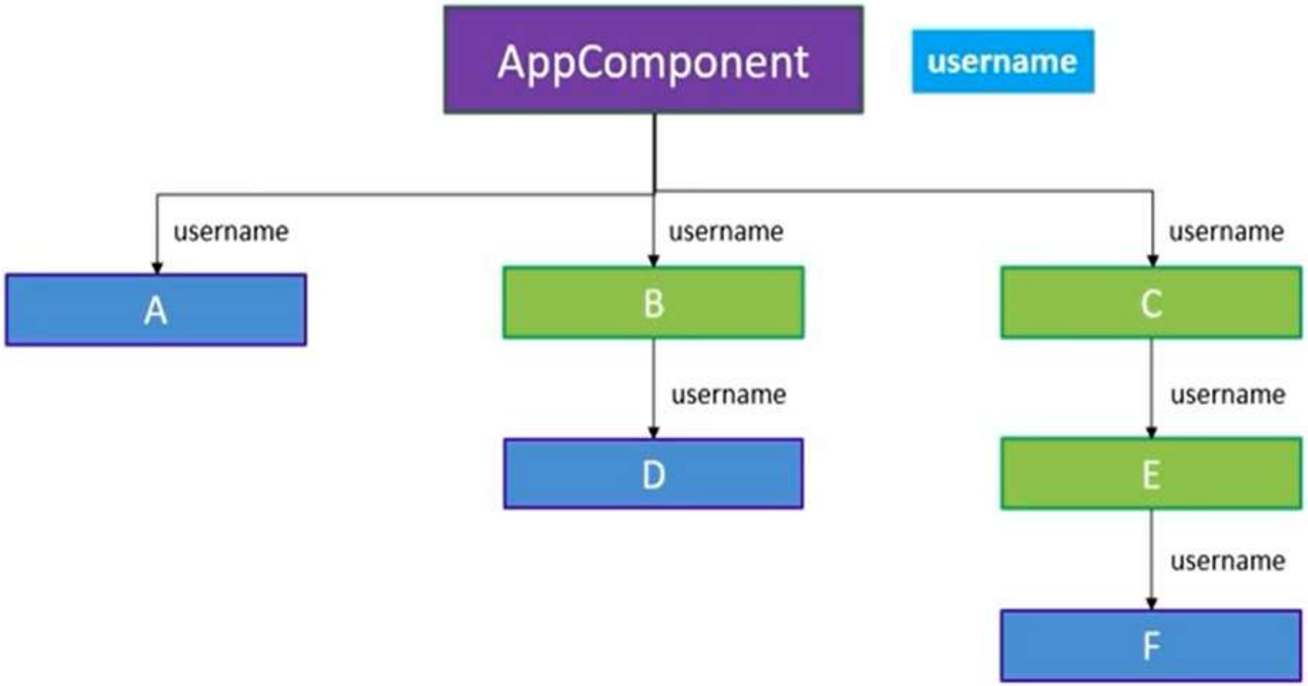
- React performs the clean-up when the component unmounts. However, as we learned earlier, effects run for every render and not just once.
- This is why React *also* cleans up effects from the previous render before running the effects next time.
- It is important to clean up so that we don’t introduce a memory leak!.
- It is similar to componentWillUnmount in class component.

useContext Hook

Context

- Context provides a way to pass data through the component tree
- without having to pass props down manually at every level.
- It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

Context


AppComponent
username
username
username
username
A
B
C
username
username
D
E
username
F

useCallback Hook

Every callback function should be memorized to prevent useless re-rendering of child components that use the callback function.

USECALLBACK(CALLBACK, DEPENDENCIES)

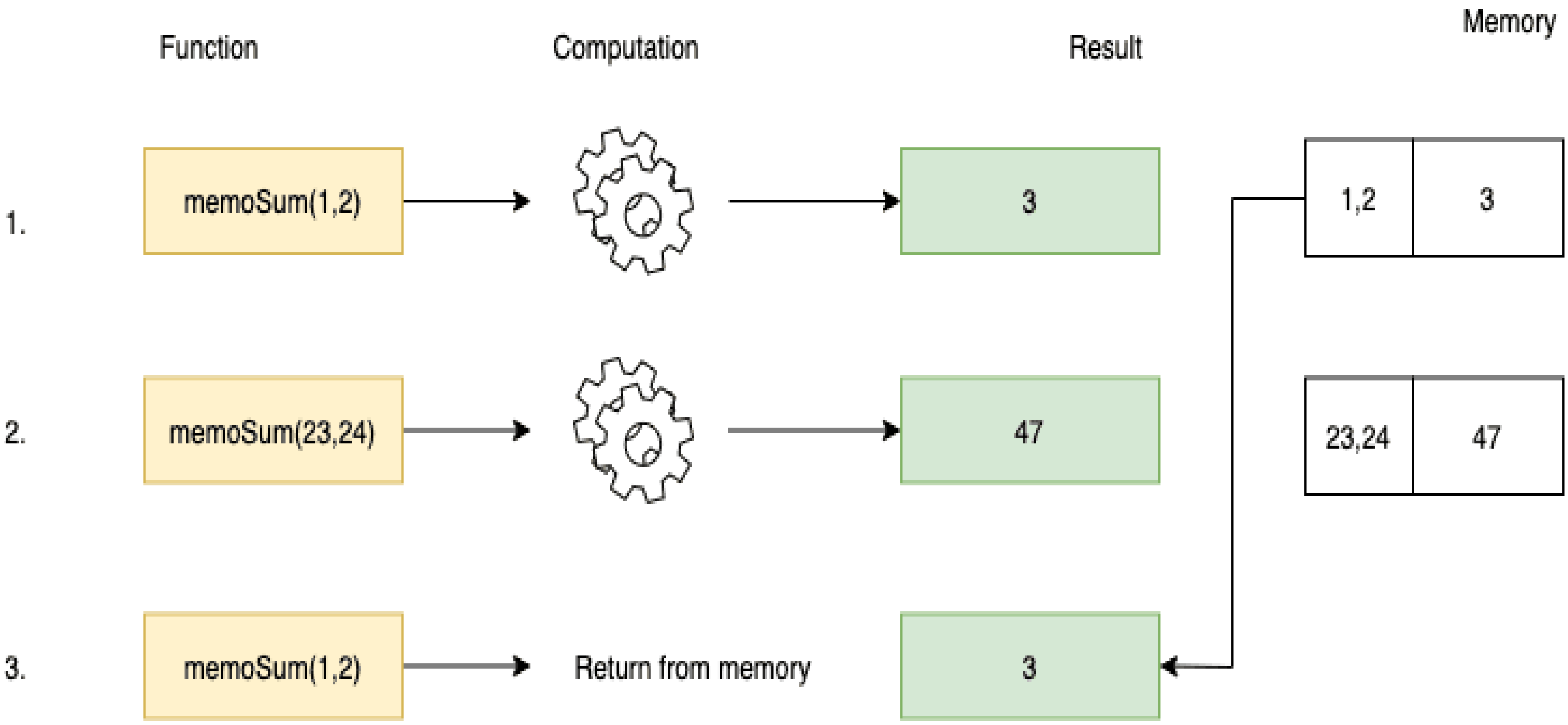
- The useCallback Hook only runs when one of its dependencies update. This means that instead of recreating the function object on every re-render, we can use the same function object between renders.

Contd..

- Applying useCallback doesn’t memoize the result of a function’s invocation. Rather it memoizes the function object itself.

Memoize

- Memoization is a technique where, whenever the parent component re-renders, the child component re-renders only if there’s a change in the props.
- If there’s no change in the props, it won’t execute the render method and will return the cached result.
- Since the render method isn’t executed, there won’t be a virtual DOM creation and difference checks — thus giving us a performance boost.



Function

Computation

Result

Memory

1. memoSum(1,2) → 3

2. memoSum(23,24) → 47

3. memoSum(1,2) → Return from memory → 3

1,2	3
23,24	47

When..?

- Hence, a useCallback hook should be used when we want to memoize a callback, and to memoize the result of a function to avoid expensive computation we can use useMemo.
- useEffect is used to produce side effects to some state changes. One thing to remember is that one should not overuse hooks.

useRef Hook

- The useRef Hook allows you to persist values between renders.
- It can be used to store a mutable value that does not cause a re-render when updated.
- It can be used to access a DOM element directly.
- useRef() only returns one item. It returns an Object called current.
- When we initialize useRef we set the initial value: useRef(0).

Why Ref..?

- Refs are a function provided by React to access the DOM element and the React element that you might have created on your own.
- They are used in cases where we want to change the value of a child component, without making use of props and all.

Why refs are not recommended..?

- That is because you're not building the app the React Way .
- React requires that you communicate between components through props (not refs). That is what makes React, React.
- A production-ready React app with the previous functionality will take a different approach.

useMemo

- It’s used to optimize the performance of a React application.
- useMemo enables memoization in React hook functions. Memoization is a programming strategy where the function remembers the output of previous executions and uses it for further executions.
- The basic usage of this hook is to improve the performance of React applications by memoizing the output and related input parameters of commonly used functions.

Contd..

- If the memoized function is called again with the same parameters, it doesn’t re-execute the function. Rather, the initial cached value is returned from the function, reducing the overhead of executing the same function again.
- Allows you to cache a value between renders.

When to use *useCallback /useMemo*

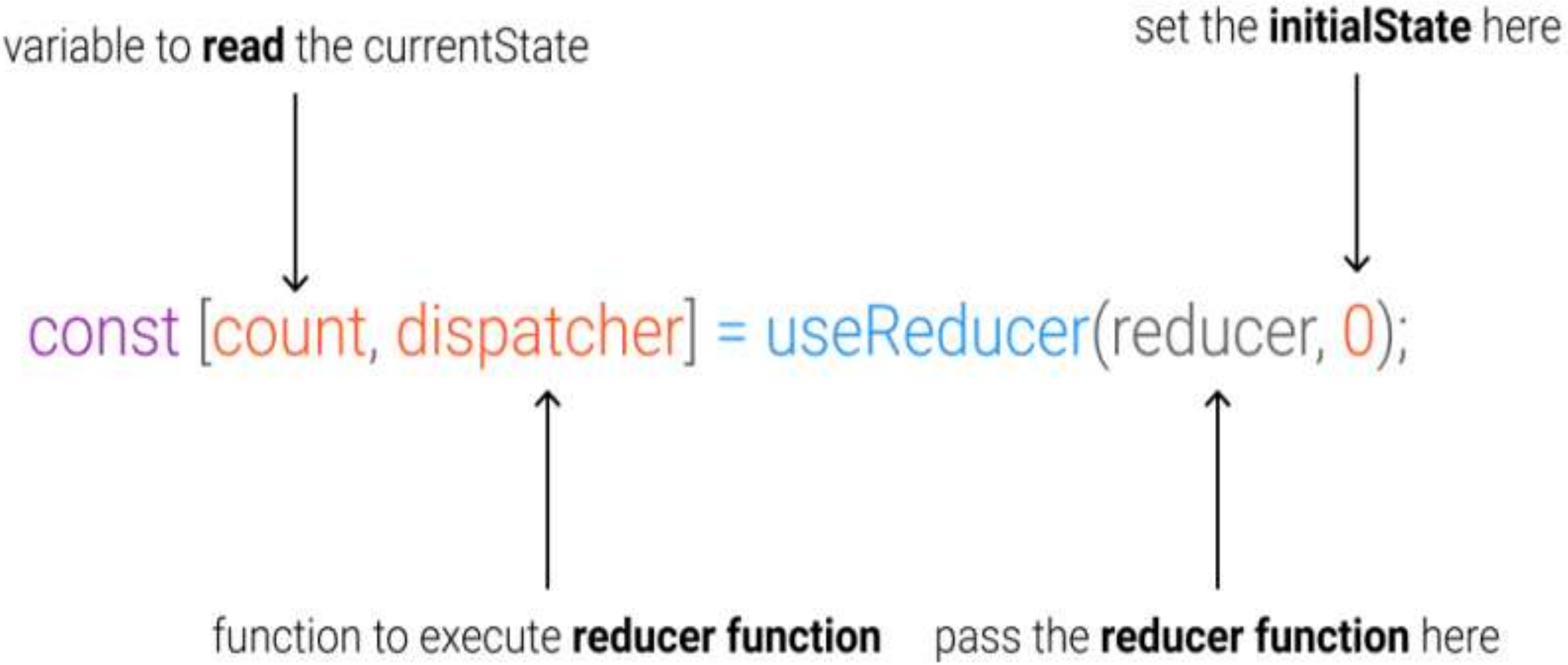
- Adding ***useCallback*** and ***useMemo*** hooks everywhere in your component, please don’t.
- Both of these hooks add some extra complexity to your code and they require a lot of things working under the hood.

  1) Processing large amounts of data
  2) Working with interactive graphs and charts
  3) Implementing animations
  4) Incorporating component lazy loading

(useMemo specifically)

useReducer

- The useReducer() Hook allows you to have a state in the functional component. It is an alternative to useState() Hook.
- The useState() Hook is implemented using useReducer() Hook. It means that useReducer() is primitive, and you can use it for everything that you can do with useState().
- The useReducer() Hook is preferred over useState() when you have complex state logic that involves multiple sub value or when the next state depends on the previous state.



variable to **read** the currentState

set the **initialState** here

```
const [count, dispatcher] = useReducer(reducer, 0);
```

function to execute **reducer function**

pass the **reducer function** here

Contd..

The useReducer() Hook takes two arguments. The first argument is the reducer() method, and the second argument is the initial state.

The useReducer() returns an array. The first element of an array is the variable that stores the current state, and the second element of an array is a dispatch() method that calls the reducer() method.

- Functional Component
- React Hooks
  - State hooks
  - Effect hooks
  - Effect Types
    - With Clean-up
    - With Clean-up
  - useContext
  - Context API
  - useCallback
  - useRef

QUERIES