Announcements

- 🏆 **MILESTONE 2** due next Wednesday, Jan 24 at 6 PM SHARP
- 🌆 **HACKATHON** TONIGHT 7 PM - 1 AM in 32-082 YAY
    - There will be food and drinks! 🍕 🥤
    - Nick will also give a mini lecture on how to get started on your project!
- 🤓 **Homework 3: Setting up Render** required for deployment
    - Let us know if you didn't receive an email with a Render code
- 🎥 **Lecture recordings** are up at weblab.is/recordings
- 💖 **Subject Evaluations** are open Monday-Friday next week!
    - Special stickers for ppl who fill out their subject evals 👉👈 (take a screenshot of your completion page heh)

State Management

Mark Tabor and Jay Hilton

State in React

- useState(initialValue) -> (state, setState)
    - After setState called, components using state re-render

State in React

- useState(initialValue) -> (state, setState)
    - After setState called, components using state re-render
- How do we get other components to use / update state?

State in React

- useState(initialValue) -> (state, setState)
  - After setState called, components using state re-render
- How do we get other components to use / update state?
  - Pass state and setState as props!

State in React

- useState(initialValue) -> (state, setState)
    - After setState called, components using state re-render
- How do we get other components to use / update state?
    - Pass state and setState as props!
- What if the component that needs the state is deep in the tree?

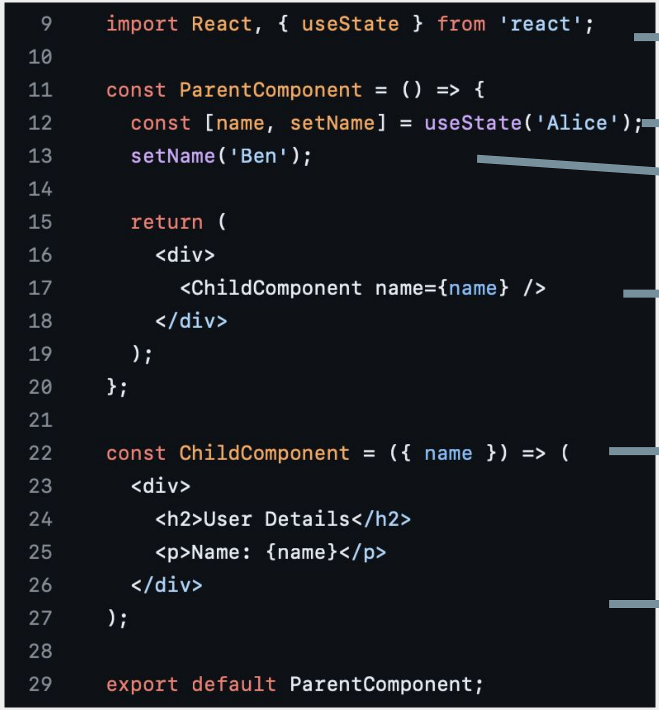
State in React

- useState(initialValue) -> (state, setState)
    - After setState called, components using state re-render
- How do we get other components to use / update state?
    - Pass state and setState as props!
- What if the component that needs the state is deep in the tree?
    - Have to pass it through many components
    - We’re sad…

State in React

- useState(initialValue) -> (state, setState)
    - After setState called, components using state re-render
- How do we get other components to use / update state?
    - Pass state and setState as props!
- What if the component that needs the state is deep in the tree?
    - Have to pass it through many components
    - We’re sad…
- Is there a better way?

A Concrete Example



```
import React, { useState } from 'react';

const ParentComponent = () => {
  const [name, setName] = useState('Alice');
  setName('Ben');

  return (
    <div>
      <ChildComponent name={name} />
    </div>
  );
};

const ChildComponent = ({ name }) => (
  <div>
    <h2>User Details</h2>
    <p>Name: {name}</p>
  </div>
);

export default ParentComponent;
```

Import useState

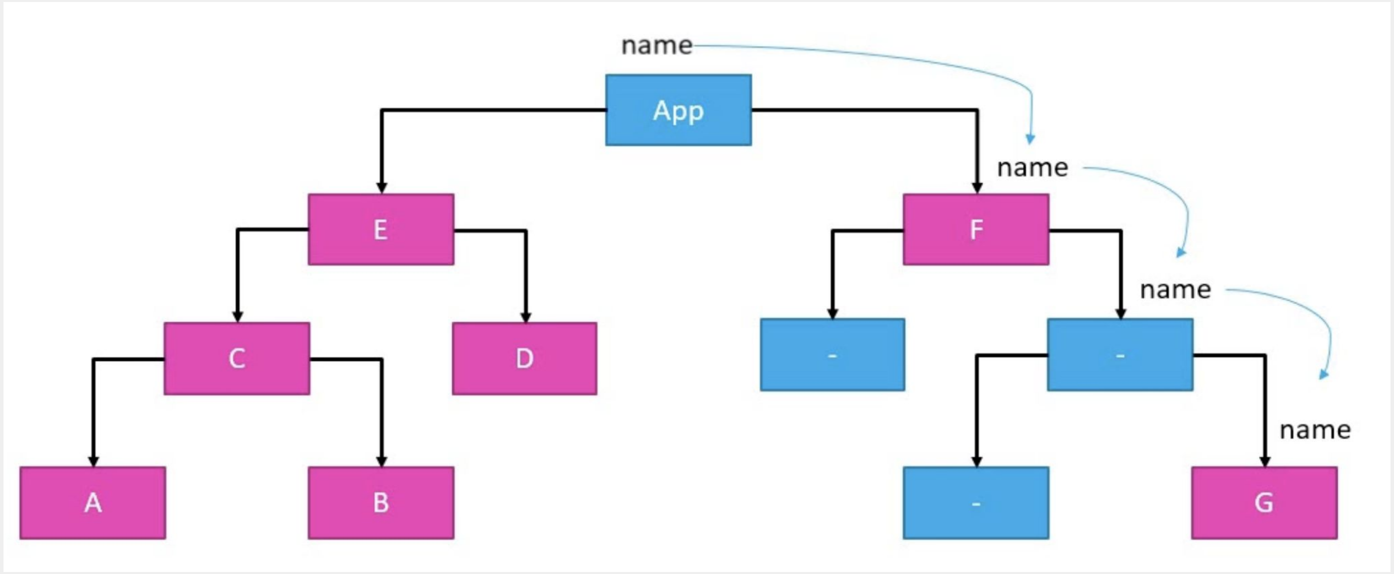
Create State and setState

setState to ‘Ben’

Pass state as prop

ChildComponent has user as state and renders

A Concrete Example



Contexts to the Rescue!

- Create a context C
- Can use context C in any descendant of C.Provider

Contexts to the Rescue!

- Create a context C
- Can use context C in any descendant of C.Provider
- How do I change C?

Contexts to the Rescue!

- Create a context C
- Can use context C in any descendant of C.Provider
- How do I change C?
    - useState!

Contexts to the Rescue!

- Create a context C
- Can use context C in any descendant of C.Provider
- How do I change C?
    - useState!
- How do I pass around the setter for C?

Contexts to the Rescue!

- Create a context C
- Can use context C in any descendant of C.Provider
- How do I change C?
    - useState!
- How do I pass around the setter for C?
- useState hook as context too?

Contexts to the Rescue!

- Create a context C
- Can use context C in any descendant of C.Provider
- How do I change C?
  - useState!
- How do I pass around the setter for C?
- useState hook as context too?
  - How do I know where C is changed?
  - Centralized updates would be nice for debugging

Contexts to the Rescue!

- Create a context C
- Can use context C in any descendant of C.Provider
- How do I change C?
  - useState!
- How do I pass around the setter for C?
- useState hook as context too?
  - How do I know where C is changed?
  - Centralized updates would be nice for debugging

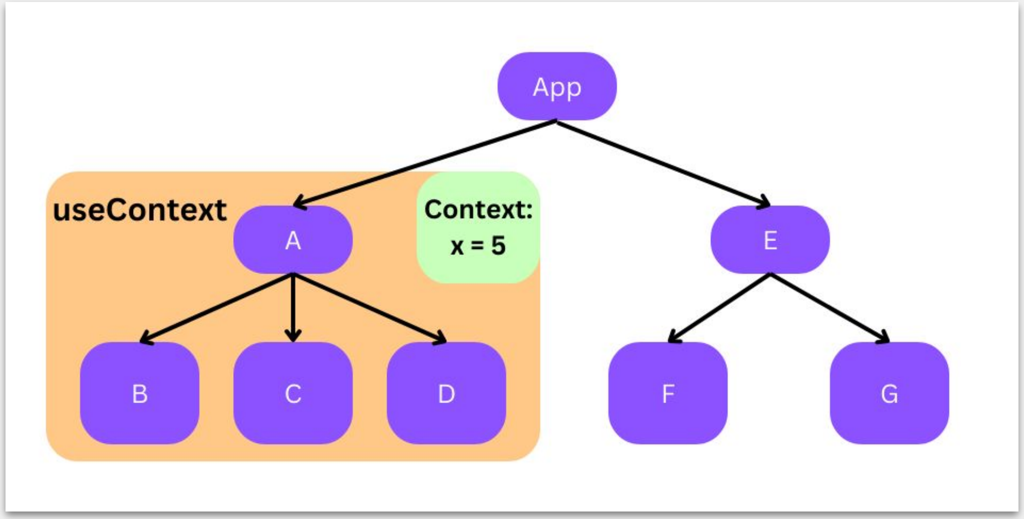
This is a form of dependency injection, if you've heard of it.

If not, that's okay too!

Examples

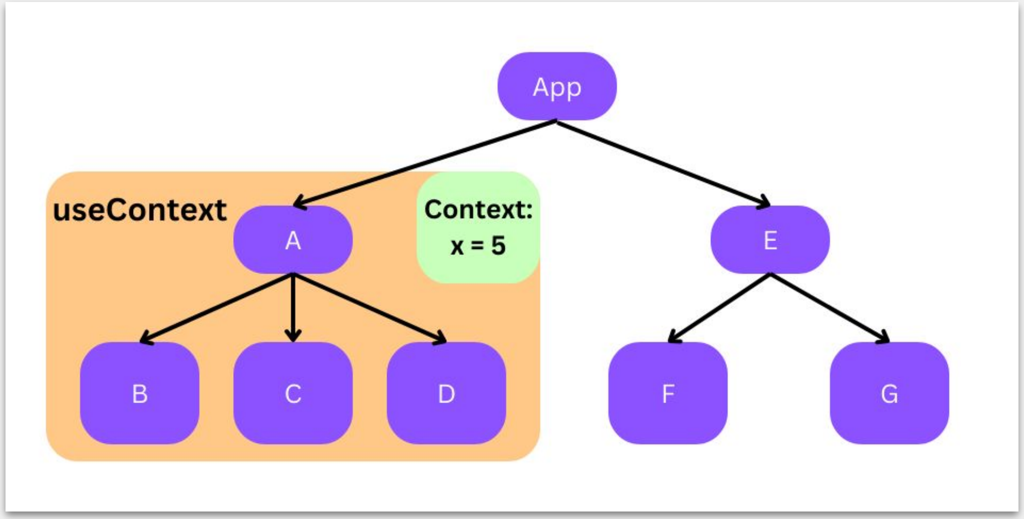
useContext Example 1

- Here's a component tree:



useContext Example 1

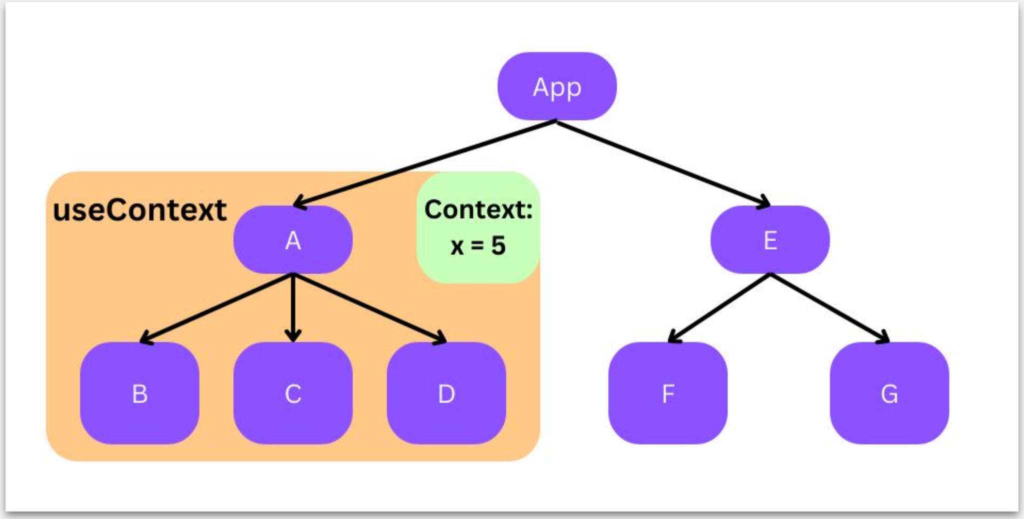
- Here’s a component tree:



- What is the value of the variable x in A? What about D?

useContext Example 1

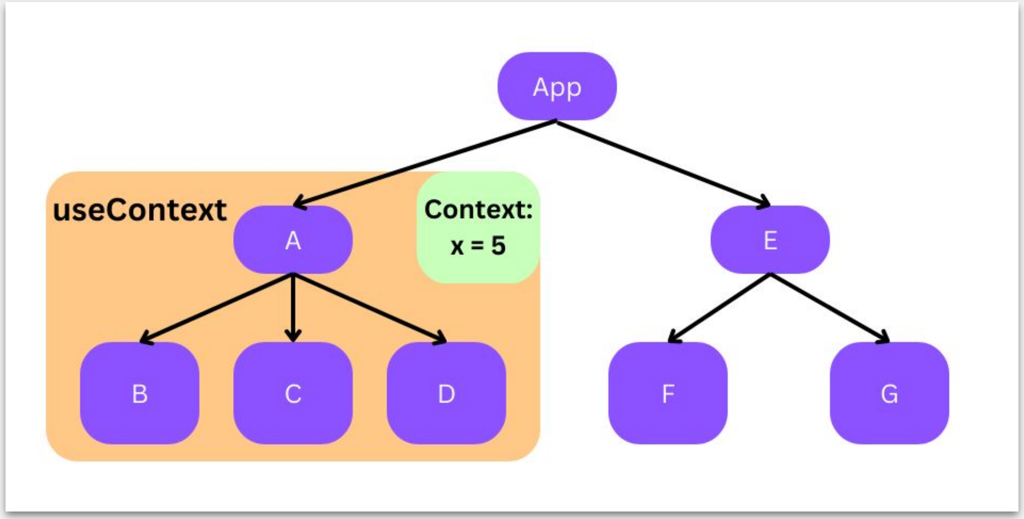
- Here’s a component tree:



- What is the value of the variable x in A? What about D?
    - Answer: 5

useContext Example 1

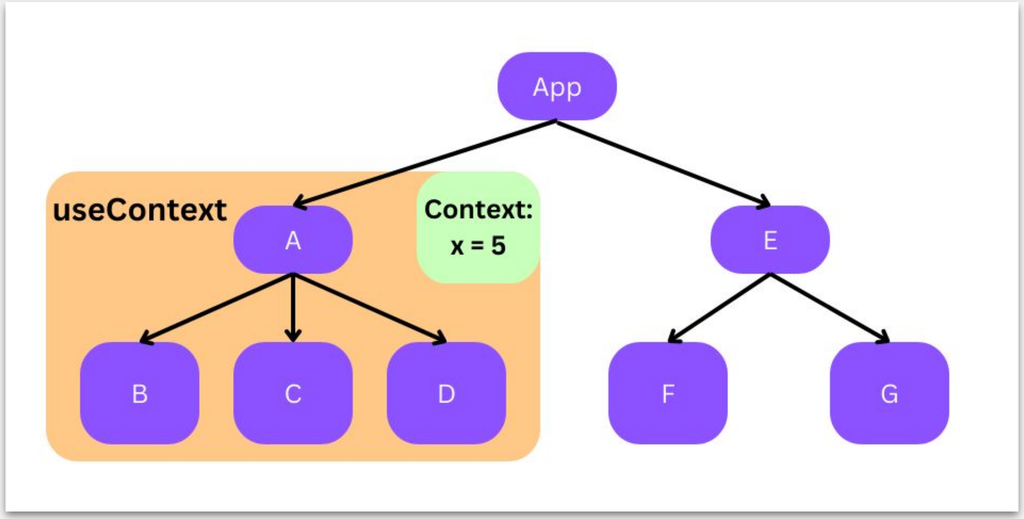
- Here's a component tree:



- What is the value of the variable x in A? What about D?
  - Answer: 5
- What about in F?

useContext Example 1

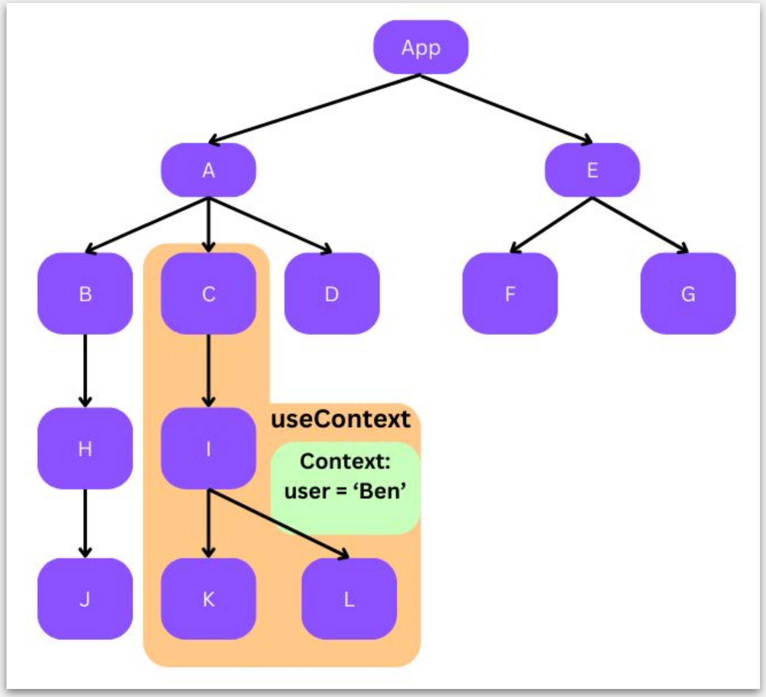
- Here's a component tree:



- What is the value of the variable x in A? What about D?
    - Answer: 5
- What about in F?
    - We don't know!

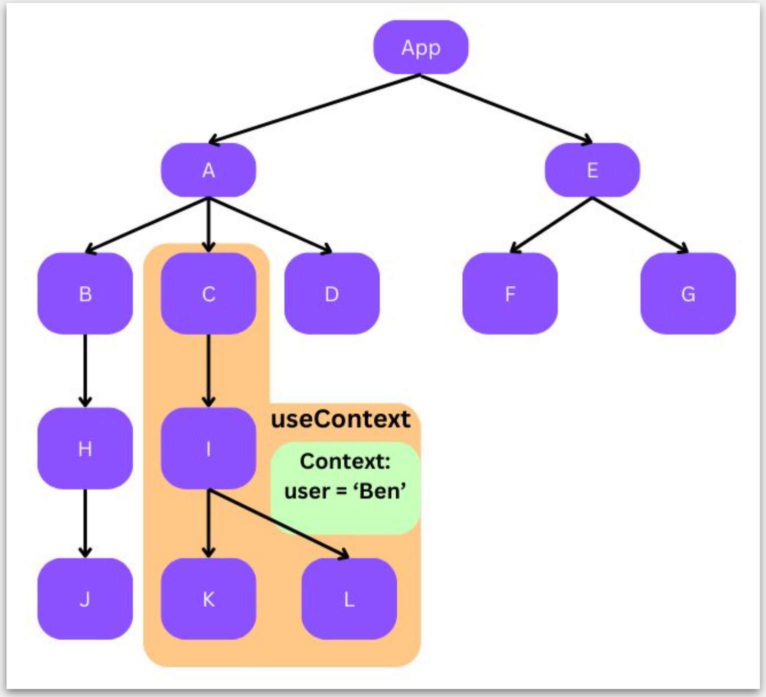
useContext Example 2

- Now you try!
- Here's a different component tree!



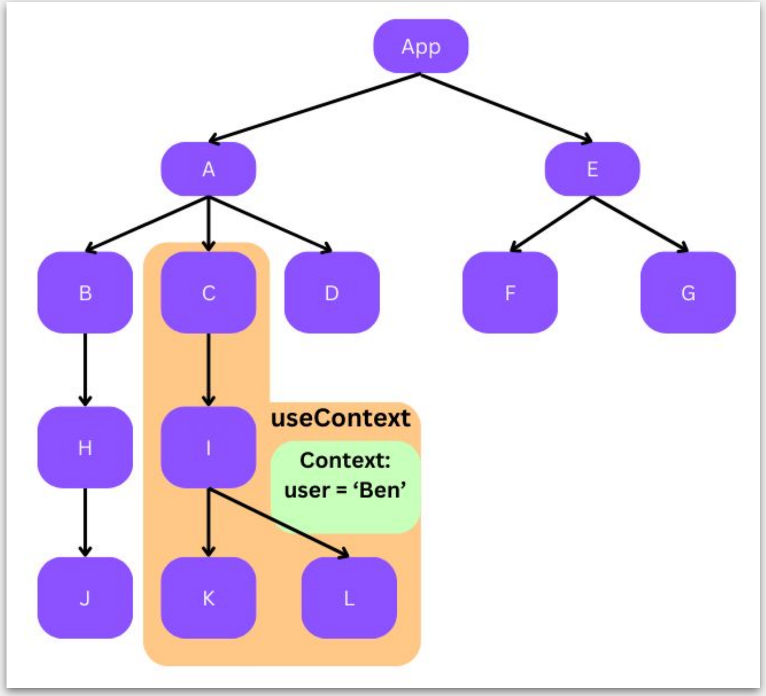
useContext Example 2

- Now you try!
- Here's a different component tree!
- What's the value of the variable user in B?



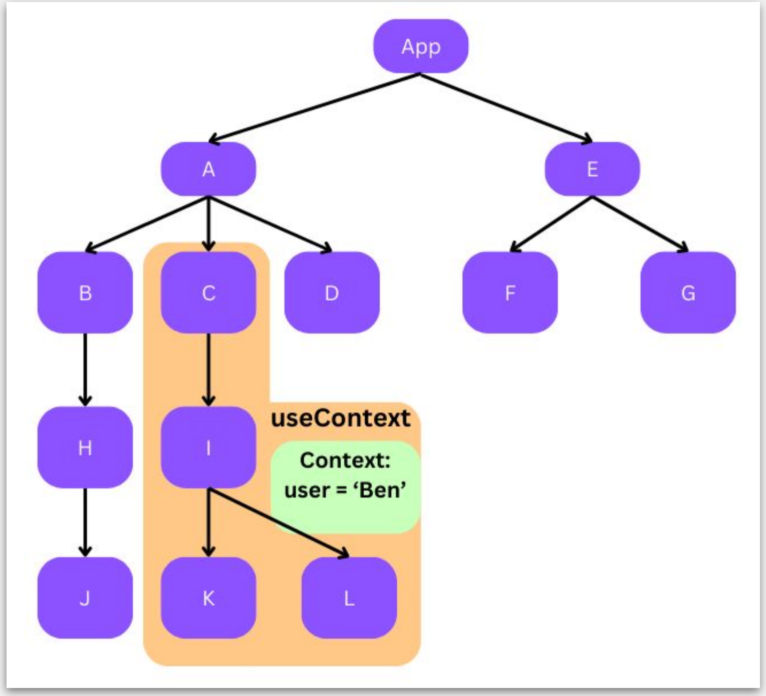
useContext Example 2

- Now you try!
- Here’s a different component tree!
- What’s the value of the variable user in B?
  - Answer: We don’t know!



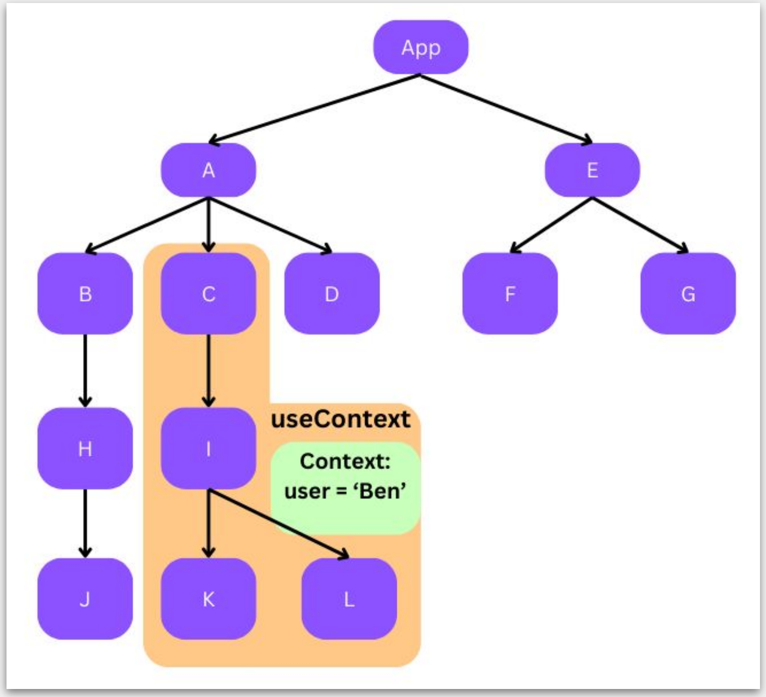
useContext Example 2

- Now you try!
- Here's a different component tree!
- What's the value of the variable user in B?
  - Answer: We don't know!
- What about in C?



useContext Example 2

- Now you try!
- Here's a different component tree!
- What's the value of the variable user in B?
  - Answer: We don't know!
- What about in C?
  - Answer: 'Ben'

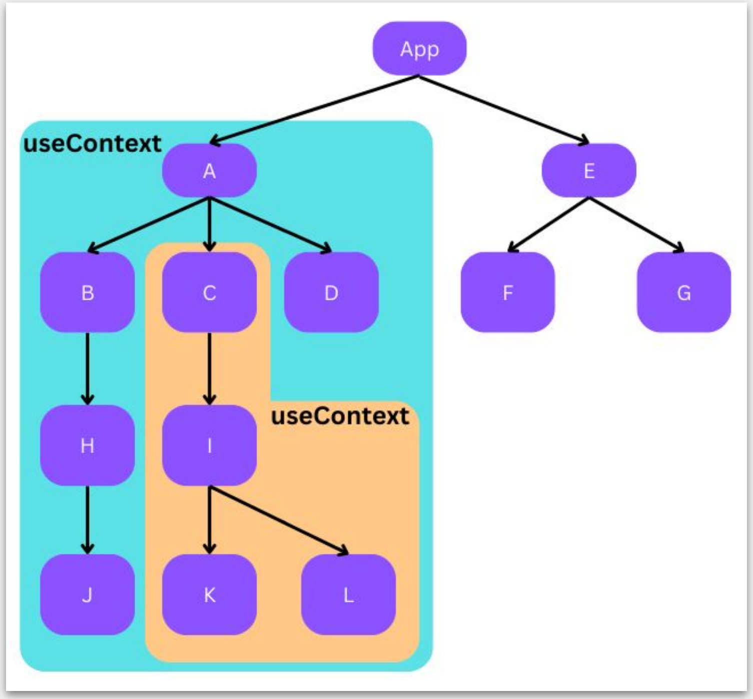


Many Contexts

- Can we have multiple contexts?

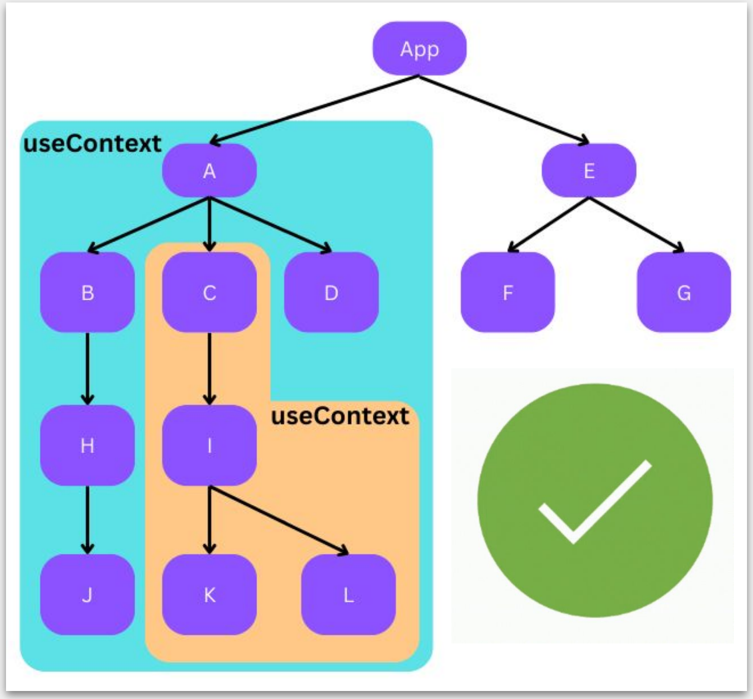
Many Contexts

- Can we have multiple contexts?
- Maybe something like:



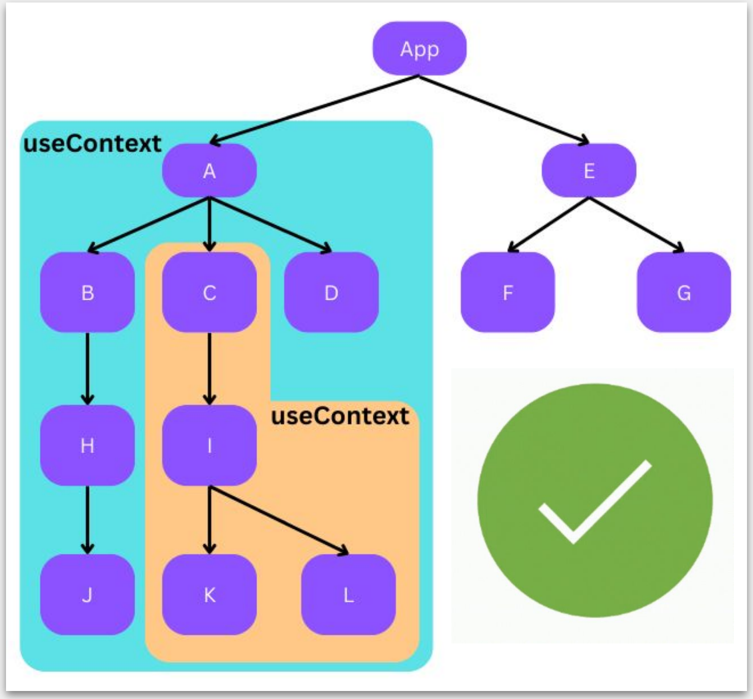
Many Contexts

- Can we have multiple contexts?
- Maybe something like:



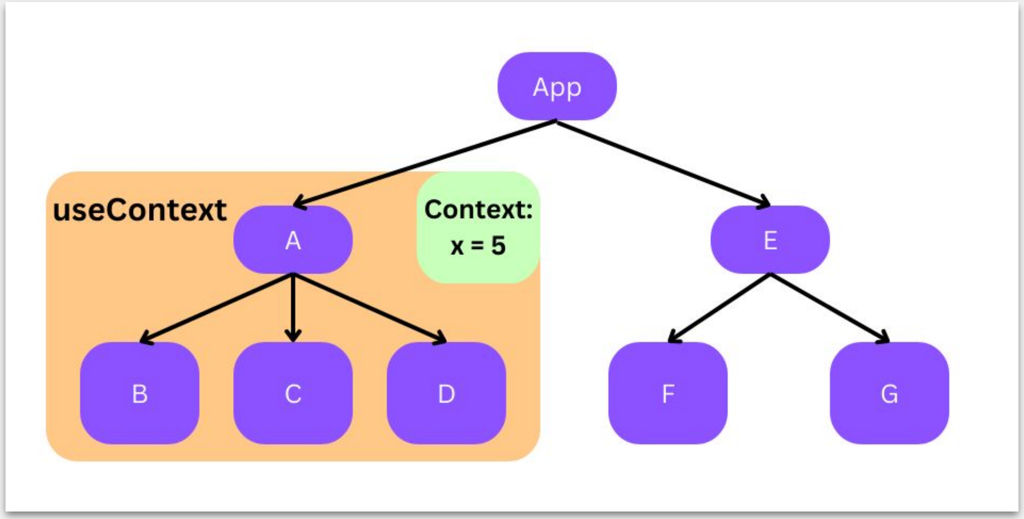
Many Contexts

- Can we have multiple contexts?
- Maybe something like:
- TODO: (MAYBE MORE TEXT ABOUT IT?)



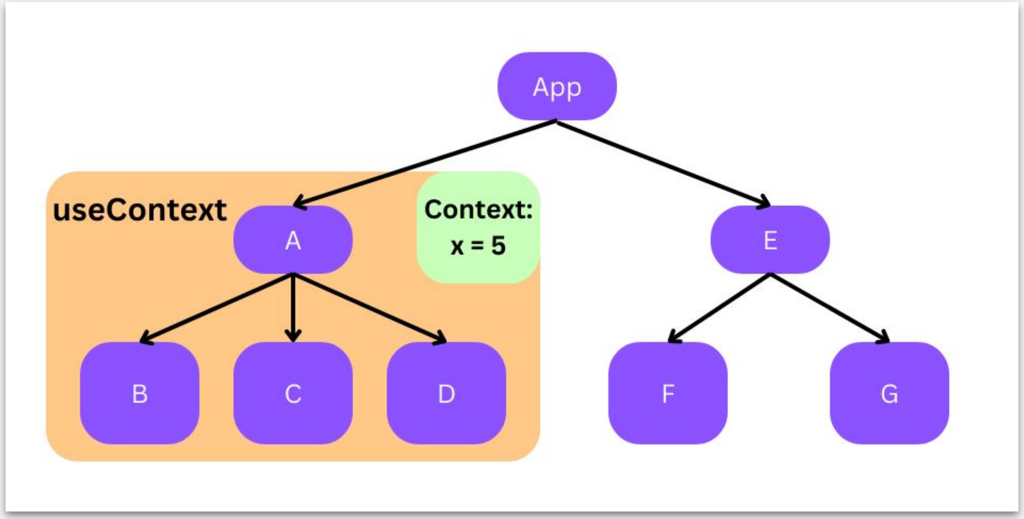
useContext / useReducer Example 1

- Here’s a component tree:



useContext / useReducer Example 1

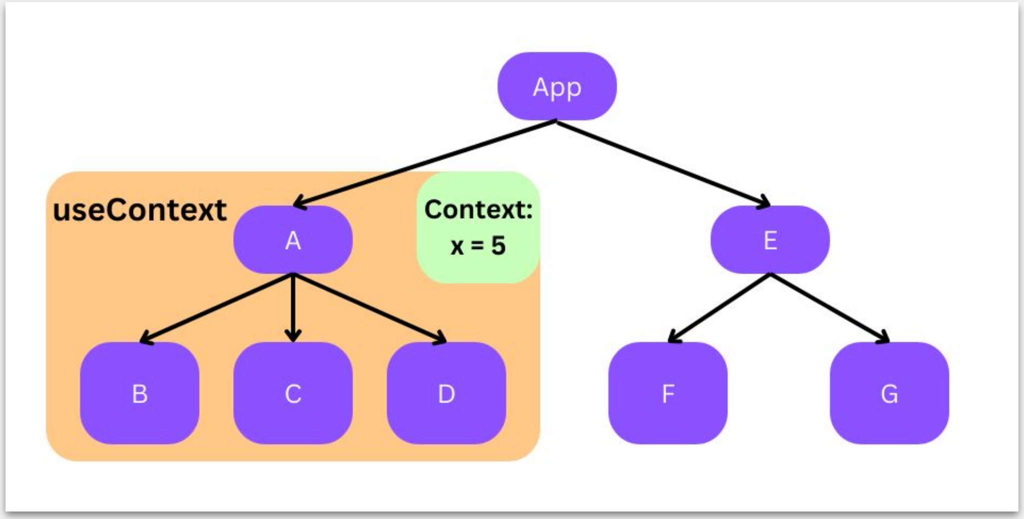
- Here's a component tree:



- What is the value of the variable x in A? What about D?

useContext / useReducer Example 1

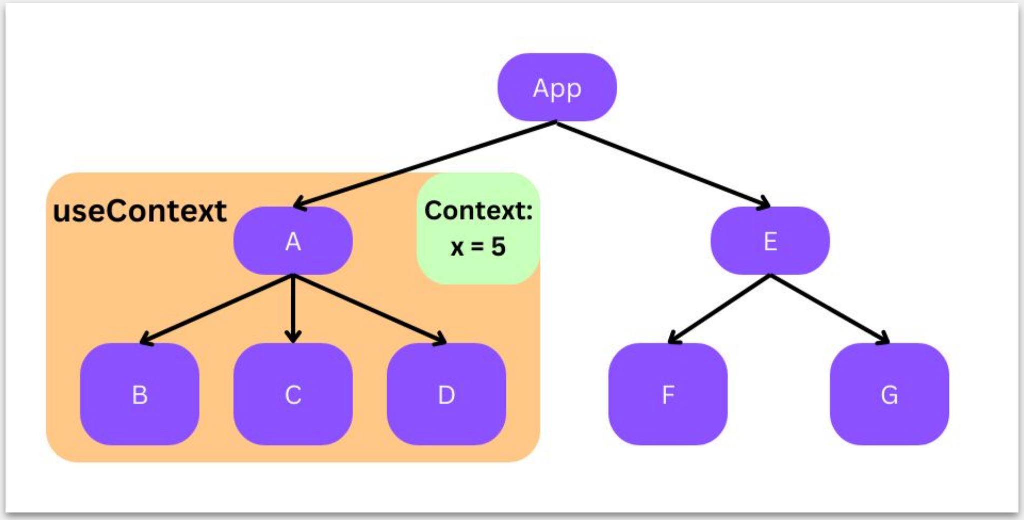
- Here’s a component tree:



- What is the value of the variable x in A? What about D?
    - Answer: 5

useContext / useReducer Example 1

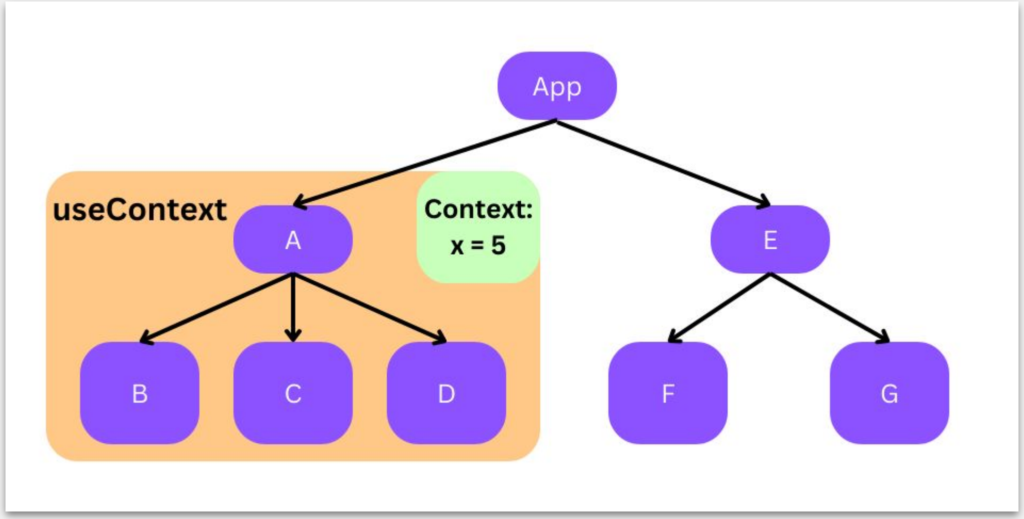
- Here's a component tree:



- What is the value of the variable x in A? What about D?
  - Answer: 5
- What about in F?

useContext / useReducer Example 1

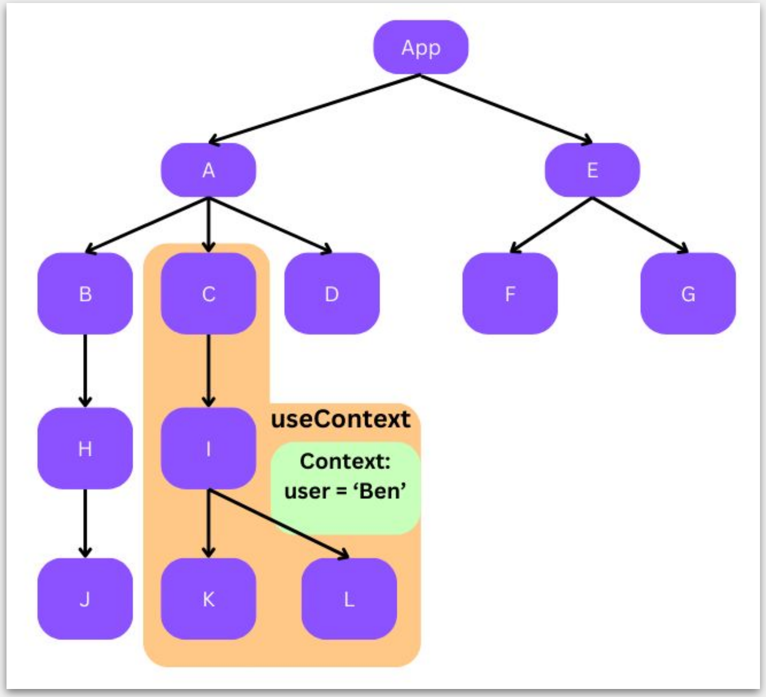
- Here's a component tree:



- What is the value of the variable x in A? What about D?
  - Answer: 5
- What about in F?
  - We don't know!

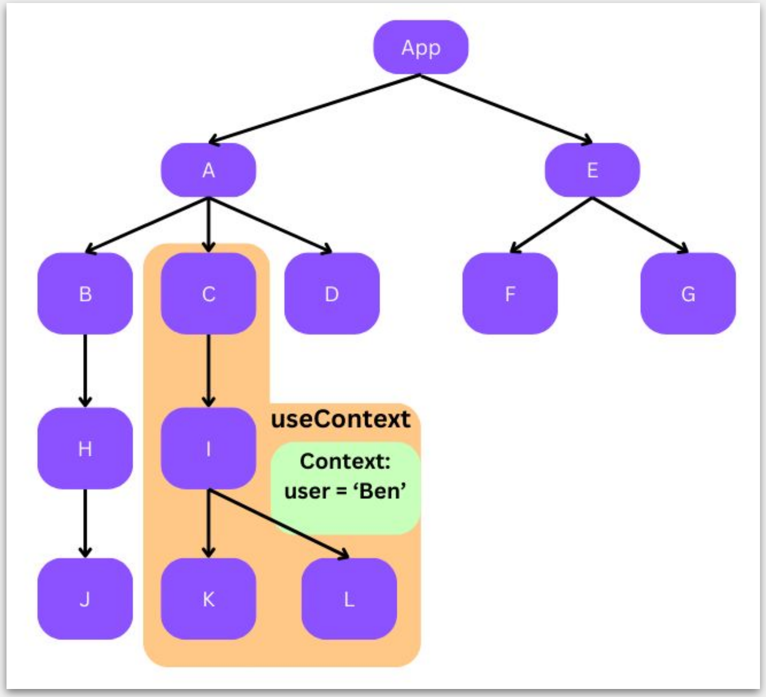
useContext / useReducer Example 2

- Now you try!
- Here’s a different component tree!



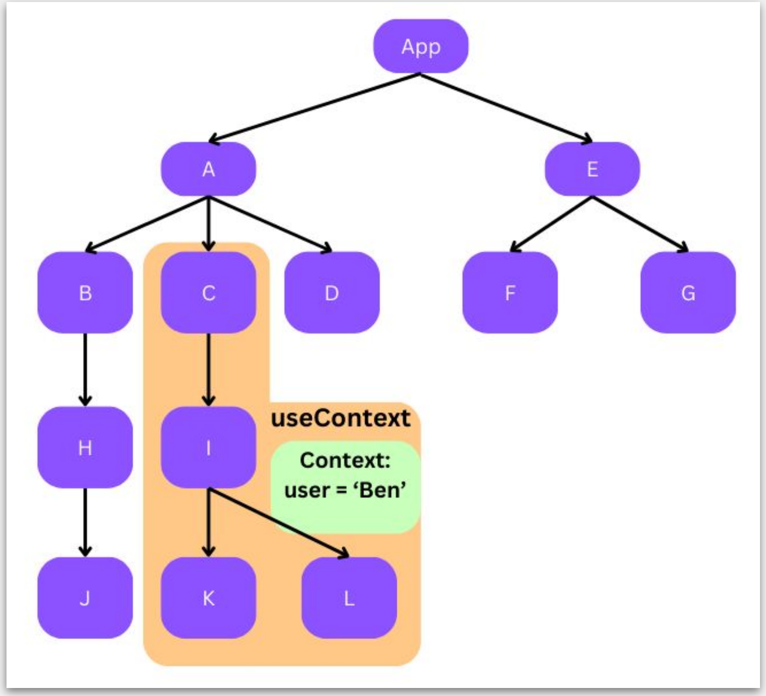
useContext / useReducer Example 2

- Now you try!
- Here's a different component tree!
- What's the value of the variable user in B?



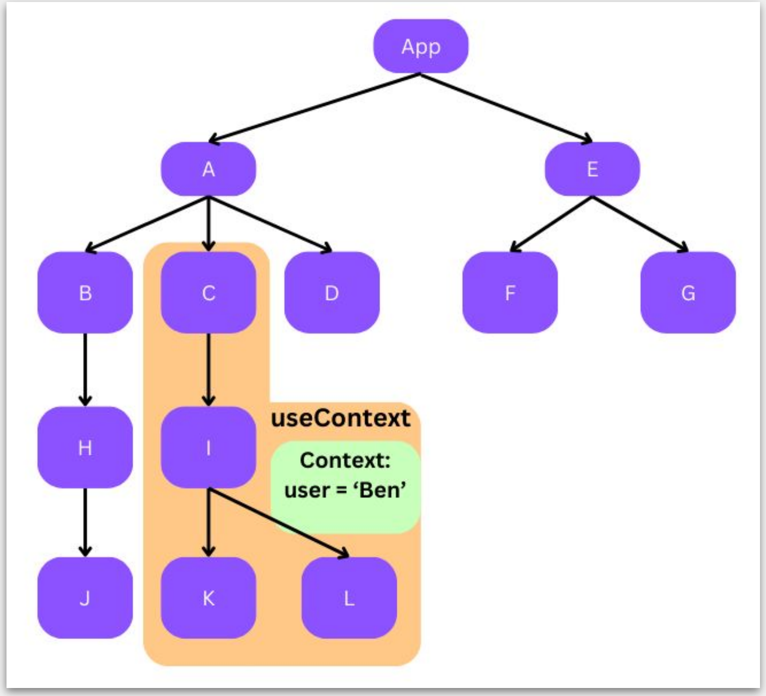
useContext / useReducer Example 2

- Now you try!
- Here's a different component tree!
- What's the value of the variable user in B?
    - Answer: We don't know!



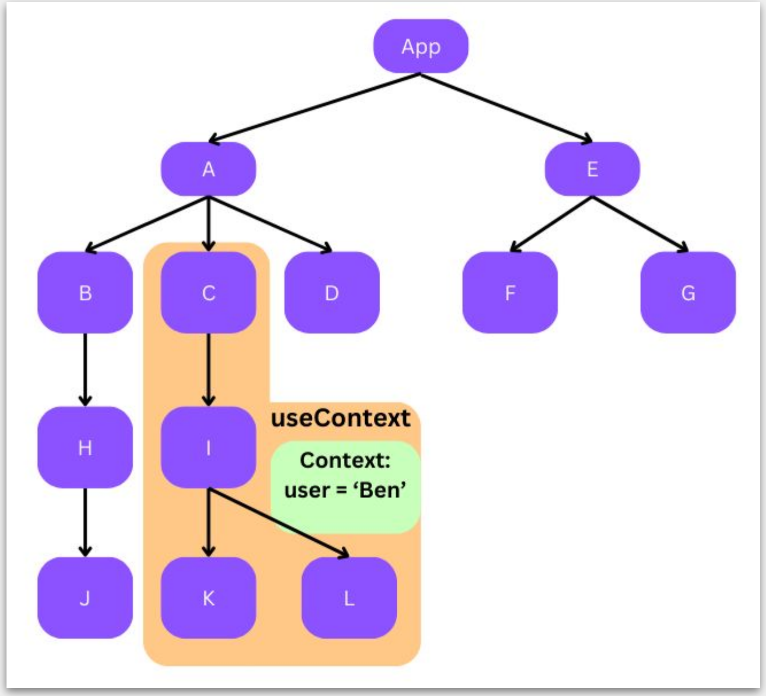
useContext / useReducer Example 2

- Now you try!
- Here's a different component tree!
- What's the value of the variable user in B?
    - Answer: We don't know!
- What about in C?



useContext / useReducer Example 2

- Now you try!
- Here’s a different component tree!
- What’s the value of the variable user in B?
  - Answer: We don’t know!
- What about in C?
  - Answer: ‘Ben’

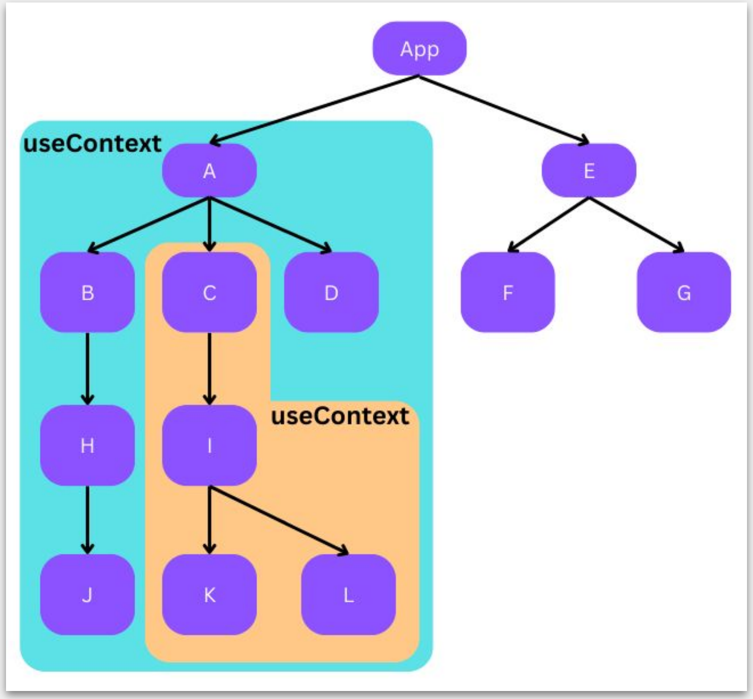


Many Contexts

- Can we have multiple contexts?

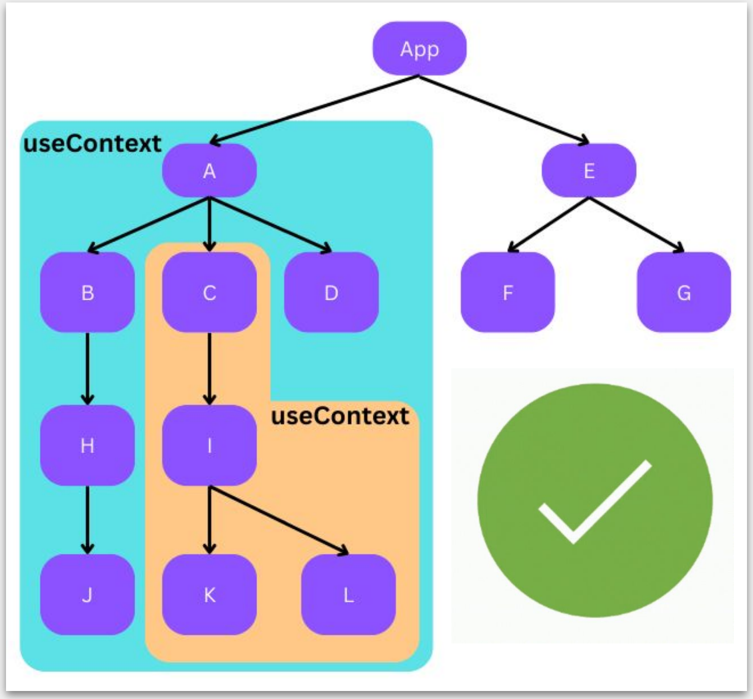
Many Contexts

- Can we have multiple contexts?
- Maybe something like:



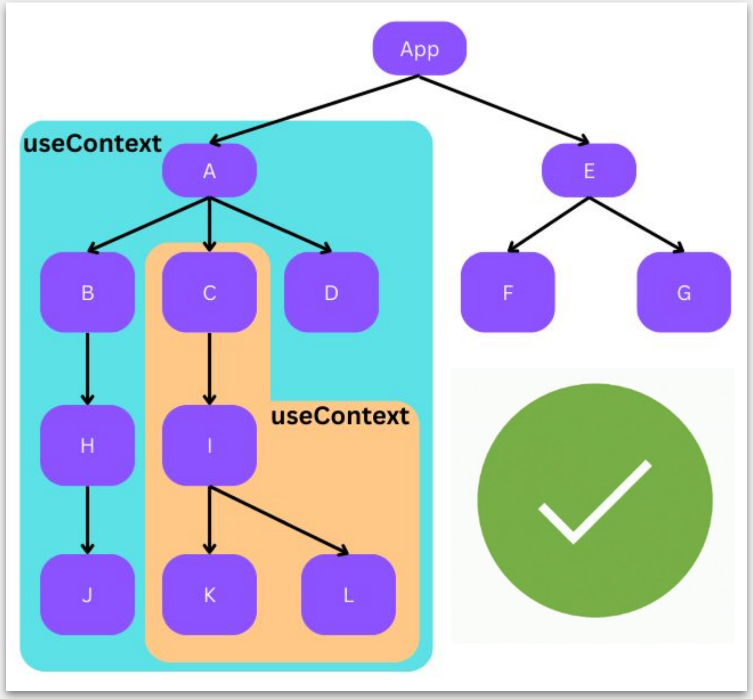
Many Contexts

- Can we have multiple contexts?
- Maybe something like:



Many Contexts

- Can we have multiple contexts?
- Maybe something like:
- TODO: (MAYBE MORE TEXT ABOUT IT?)



Questions?

State Machines

Background: State Machines

- Initial state
- next(state, action) -> newState
    - Transition function
- Examples?

Background: State Machines

- Initial state
- next(state, action) -> newState
  - Transition function
- Examples?
  - Washing machine: off, rinsing clothes, done
  - Garage door: closed, opening, open, closing
  - Oven: off, warming up, on (at some temperature), cooling down

Background: State Machines

- Initial state
- next(state, action) -> newState
    - Transition function
- Examples?
    - Washing machine: off, rinsing clothes, done

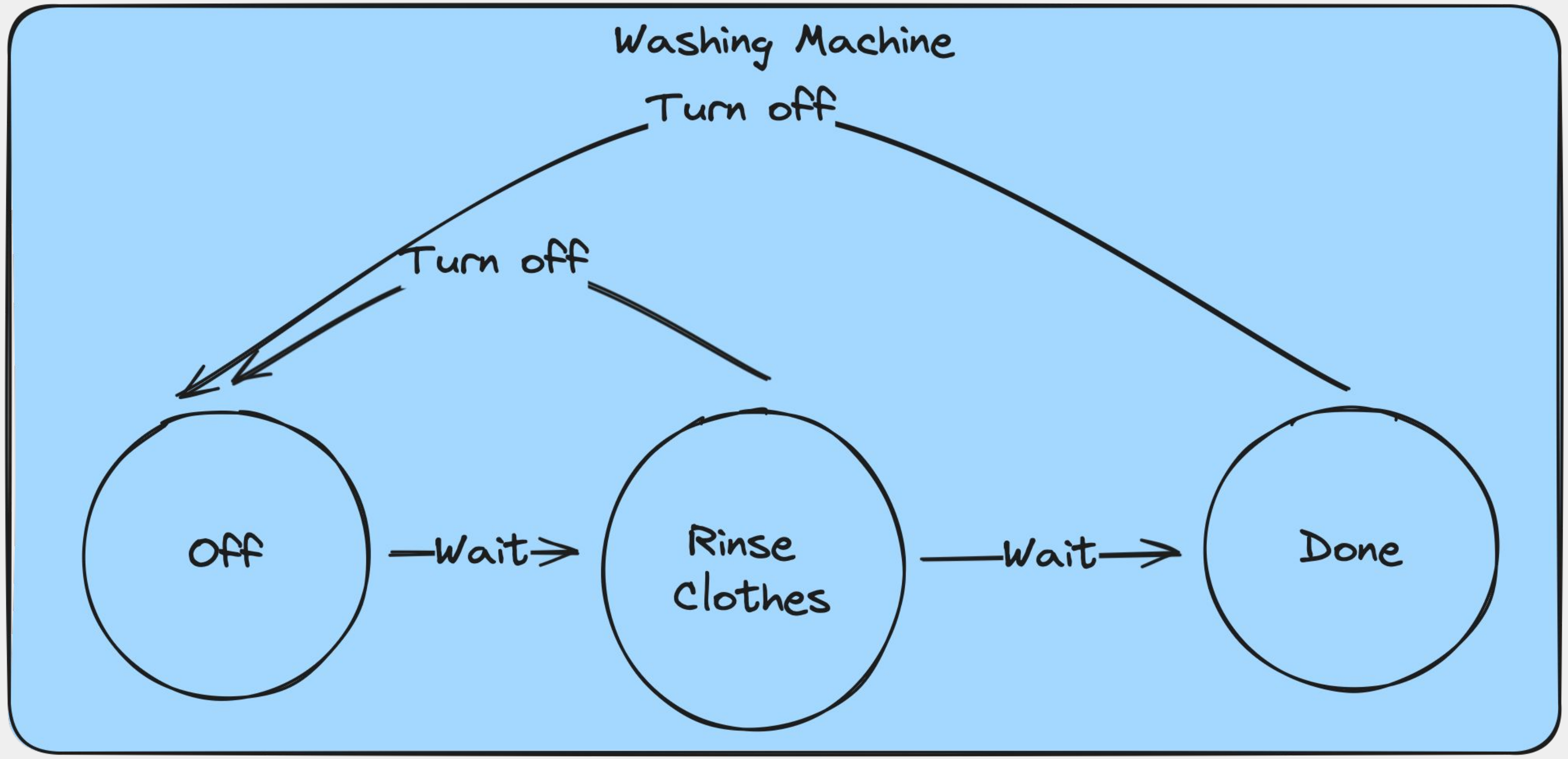
Background: State Machines

- Initial state
- next(state, action) -> newState
  - Transition function
- Examples?
  - Washing machine: off, rinsing clothes, done
  - Garage door: closed, opening, open, closing

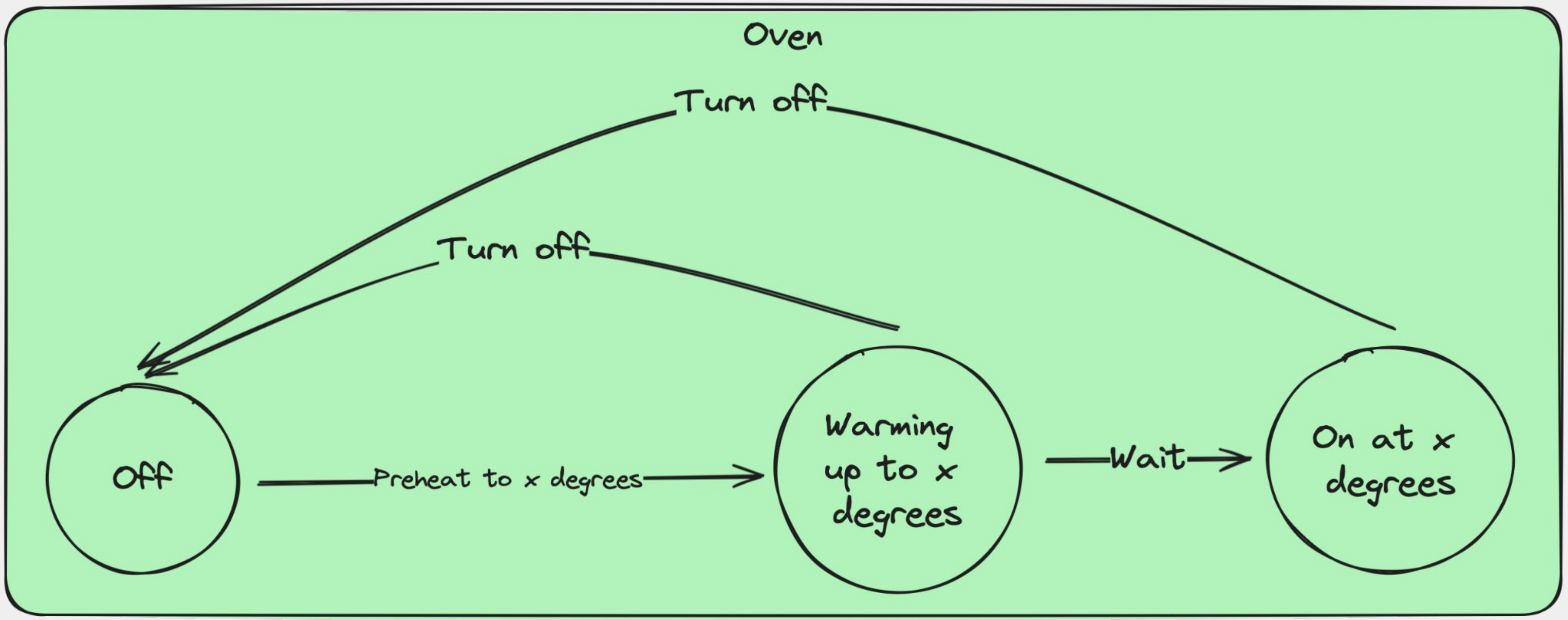
Background: State Machines

- Initial state
- next(state, action) -> newState
    - Transition function
- Examples?
    - Washing machine: off, rinsing clothes, done
    - Garage door: closed, opening, open, closing
    - Oven: off, warming up, on (at some temperature), cooling down

Background: State Machine Diagrams



Background: State Machine Diagrams



Reducers

~~Contexts~~ Reducers to the Rescue!

- Reducer: f(state, action) -> newState
  - Some initial state
  - Don’t mutate state, just return the new state
    - Mutating makes it hard to see the past state
    - Seeing past state is nice for logging and debugging

~~Contexts~~ Reducers to the Rescue!

- Reducer: f(state, action) -> newState
  - Some initial state
  - Don’t mutate state, just return the new state
- Reducers are just state machines!
  - They often name the transition function `dispatch`

~~Contexts~~ Reducers to the Rescue!

- Reducer: f(state, action) -> newState
    - Some initial state
    - Don’t mutate state, just return the new state
- Reducers are just state machines!
    - They often name the transition function `dispatch`
- How do we call `dispatch`?

~~Contexts~~ Reducers to the Rescue!

- Reducer: f(state, action) -> newState
    - Some initial state
    - Don’t mutate state, just return the new state
- Reducers are just state machines!
    - They often name the transition function `dispatch`
- How do we call `dispatch`?
    - Import it! It’s just like a function

~~Contexts~~ Reducers to the Rescue!

- Reducer: f(state, action) -> newState
    - Some initial state
    - Don’t mutate state, just return the new state
- Reducers are just state machines!
    - They often name the transition function `dispatch`
- How do we call `dispatch`?
    - Import it! It’s just like a function
- Why might this be better?

~~Contexts~~ Reducers to the Rescue!

- Reducer: f(state, action) -> newState
  - Some initial state
  - Don’t mutate state, just return the new state
- Reducers are just state machines!
  - They often name the transition function `dispatch`
- How do we call `dispatch`?
  - Import it! It’s just like a function

~~Contexts~~ Reducers to the Rescue!

- Why might this be better?

~~Contexts~~ Reducers to the Rescue!

- Why might this be better?
    - We can keep the update logic in one place!
    - Only worry about dispatching the right actions
    - Atomic updates: an action happens all at once if it’s not async

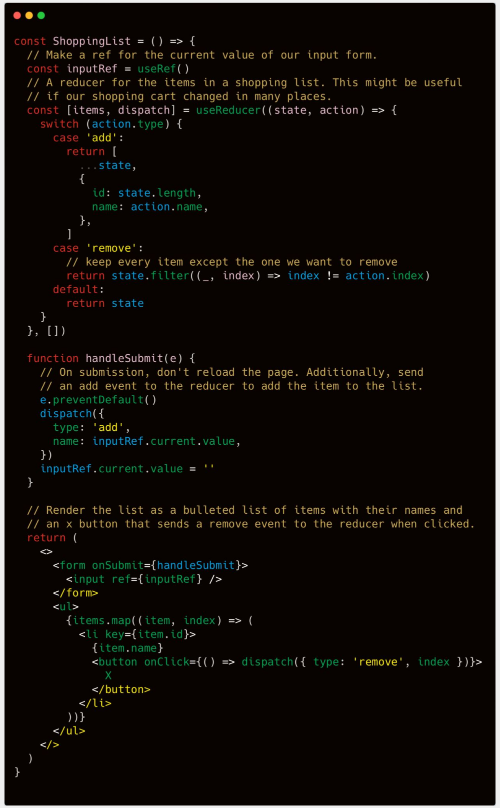
~~Contexts~~ Reducers to the Rescue!

- Why might this be better?
  - We can keep the update logic in one place!
  - Only worry about dispatching the right actions
  - Atomic updates: an action happens all at once if it's not async
- Why are they named that?
  - Similar to Array.reduce
    - `Array.reduce((previousState, newItem) => { const nextState = f(previousState, newItem); return nextState });`
    - The function we use as a reducer is like the function we pass to Array.reduce

Questions?

Some Example Reducer Code

*don't worry if the code is small, we'll talk about each part in detail



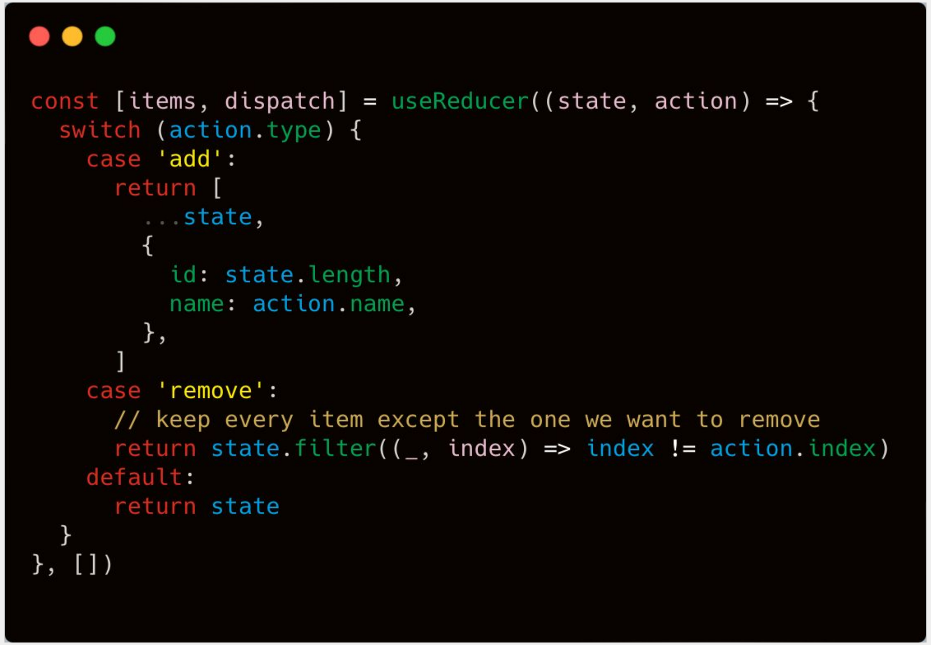
```
const ShoppingList = () => {
  // Make a ref for the current value of our input form.
  const inputRef = useRef()
  // A reducer for the items in a shopping list. This might be useful
  // if our shopping cart changed in many places.
  const [items, dispatch] = useReducer((state, action) => {
    switch (action.type) {
      case 'add':
        return [
          ...state,
          {
            id: state.length,
            name: action.name,
          },
        ]
      case 'remove':
        // keep every item except the one we want to remove
        return state.filter((_, index) => index != action.index)
      default:
        return state
    }
  }, [])

  function handleSubmit(e) {
    // On submission, don't reload the page. Additionally, send
    // an add event to the reducer to add the item to the list.
    e.preventDefault()
    dispatch({
      type: 'add',
      name: inputRef.current.value,
    })
    inputRef.current.value = ''
  }

  // Render the list as a bulleted list of items with their names and
  // an x button that sends a remove event to the reducer when clicked.
  return (
    <>
      <form onSubmit={handleSubmit}>
        <input ref={inputRef} />
      </form>
      <ul>
        {items.map((item, index) => (
          <li key={item.id}>
            {item.name}
            <button onClick={() => dispatch({ type: 'remove', index })}>
              X
            </button>
          </li>
        ))}
      </ul>
    </>
  )
}
```

Some Example Reducer Code

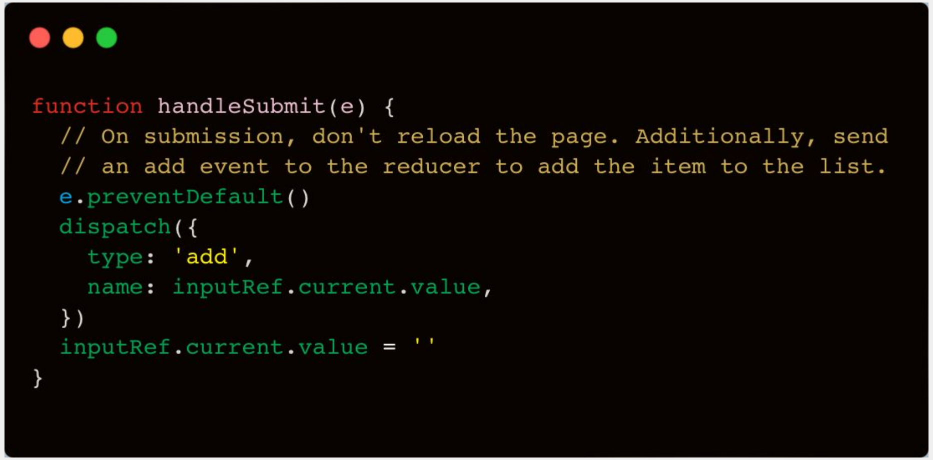
- Here's what a reducer looks like!
- switch is like an if statement here
    - If the type is add, add the item
    - If the type is remove, remove the item
    - For anything else, leave the state unchanged
- The state can be read with items
- But how do we use this code?



```
const [items, dispatch] = useReducer((state, action) => {
  switch (action.type) {
    case 'add':
      return [
        ...state,
        {
          id: state.length,
          name: action.name,
        },
      ]
    case 'remove':
      // keep every item except the one we want to remove
      return state.filter((_, index) => index != action.index)
    default:
      return state
  }
}, [])
```

Some Example Reducer Code

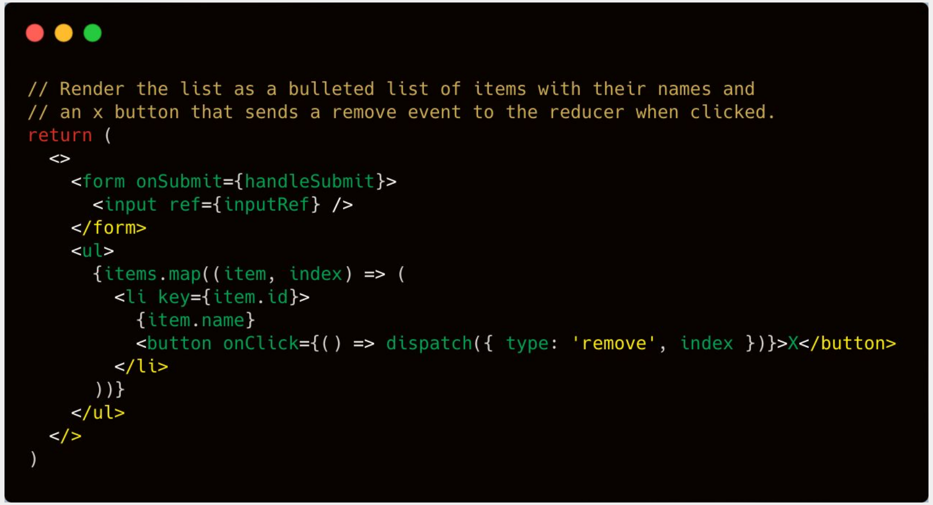
- This function, when called, will dispatch an add event
- Adds current value stored in input box to the list
    - inputRef was created earlier



```
function handleSubmit(e) {
  // On submission, don't reload the page. Additionally, send
  // an add event to the reducer to add the item to the list.
  e.preventDefault()
  dispatch({
    type: 'add',
    name: inputRef.current.value,
  })
  inputRef.current.value = ''
}
```

Some Example Reducer Code

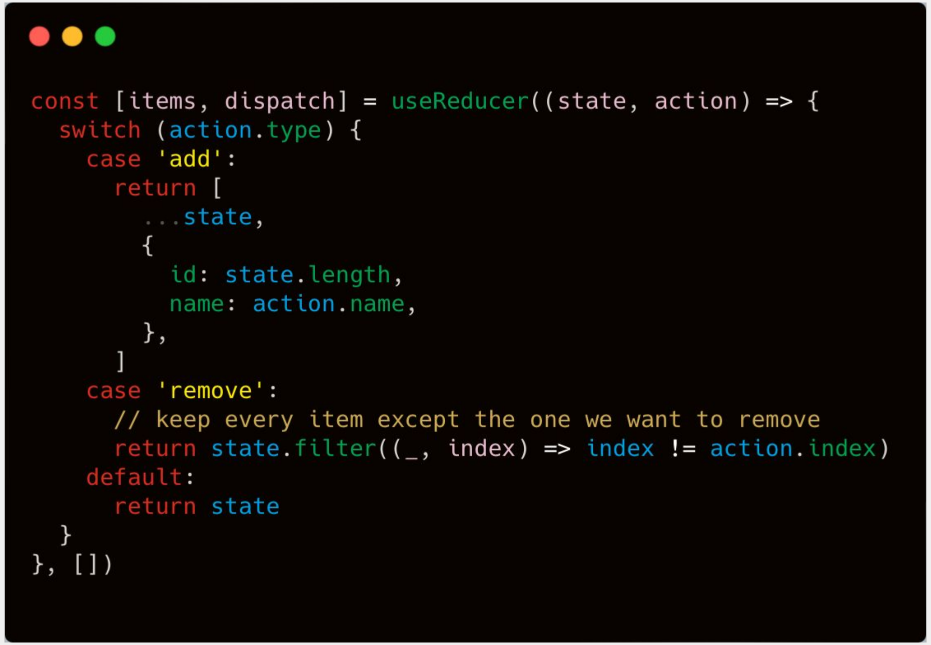
- When the form is submitted, call handleSubmit
    - Adds value in input box to list
- When the button next to an item is clicked, remove it from the list
    - Button has an x inside



```
// Render the list as a bulleted list of items with their names and
// an x button that sends a remove event to the reducer when clicked.
return (
  <>
    <form onSubmit={handleSubmit}>
      <input ref={inputRef} />
    </form>
    <ul>
      {items.map((item, index) => (
        <li key={item.id}>
          {item.name}
          <button onClick={() => dispatch({ type: 'remove', index })}>X</button>
        </li>
      ))}
    </ul>
  </>
)
```

A Reducer Example

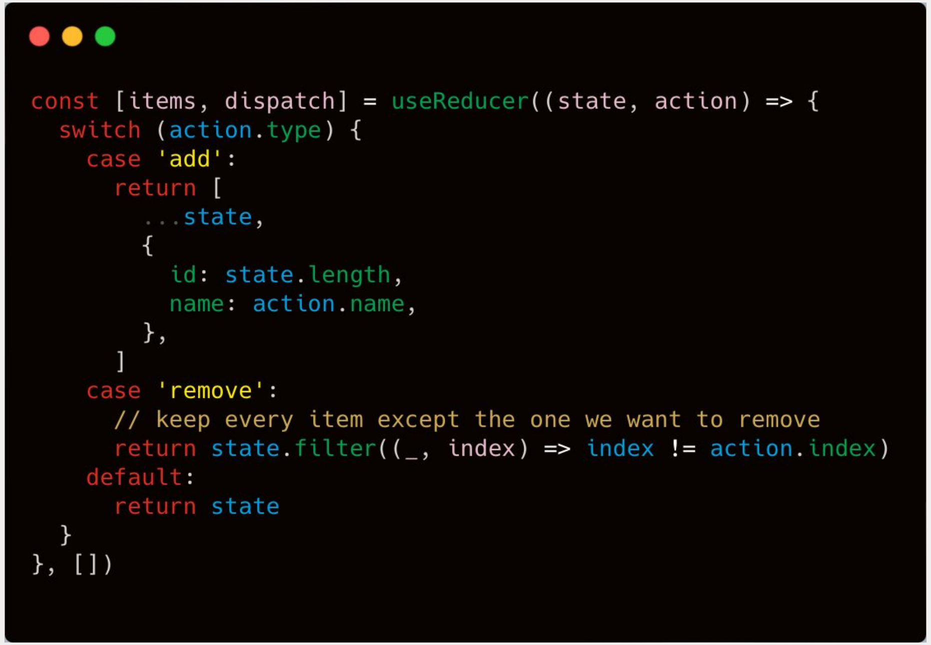
- { type: “add”, name: “cheese” }
- [{id: 0, name: “cheese”}]



```
const [items, dispatch] = useReducer((state, action) => {
  switch (action.type) {
    case 'add':
      return [
        ...state,
        {
          id: state.length,
          name: action.name,
        },
      ]
    case 'remove':
      // keep every item except the one we want to remove
      return state.filter((_, index) => index != action.index)
    default:
      return state
  }
}, [])
```

A Reducer Example

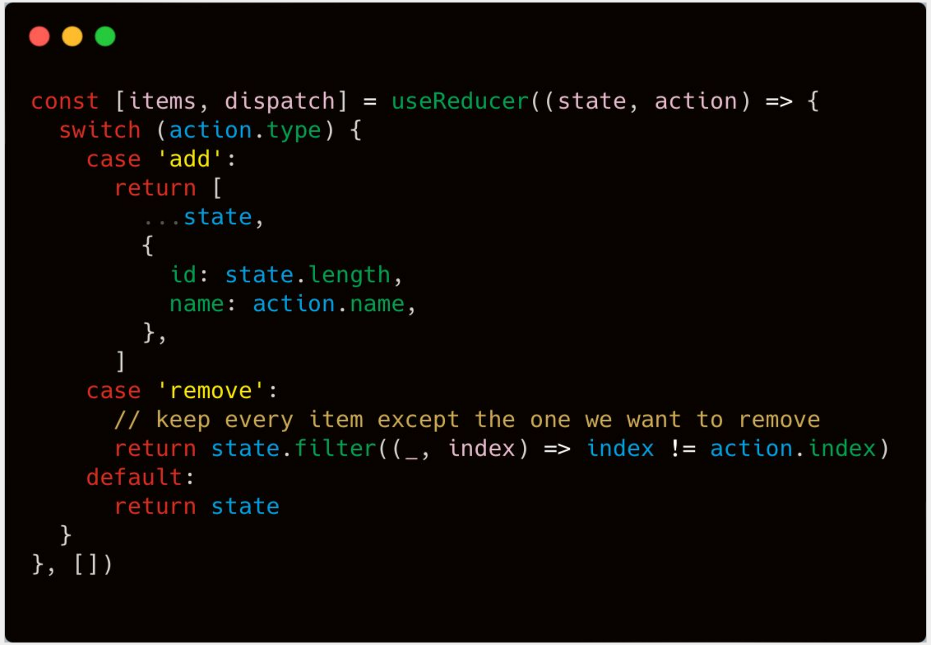
- { type: “add”, name: “cheese” }
- [{id: 0, name: “cheese”}]
- { type: “add”, name: “bread” }



```
const [items, dispatch] = useReducer((state, action) => {
  switch (action.type) {
    case 'add':
      return [
        ...state,
        {
          id: state.length,
          name: action.name,
        },
      ]
    case 'remove':
      // keep every item except the one we want to remove
      return state.filter((_, index) => index != action.index)
    default:
      return state
  }
}, [])
```

A Reducer Example

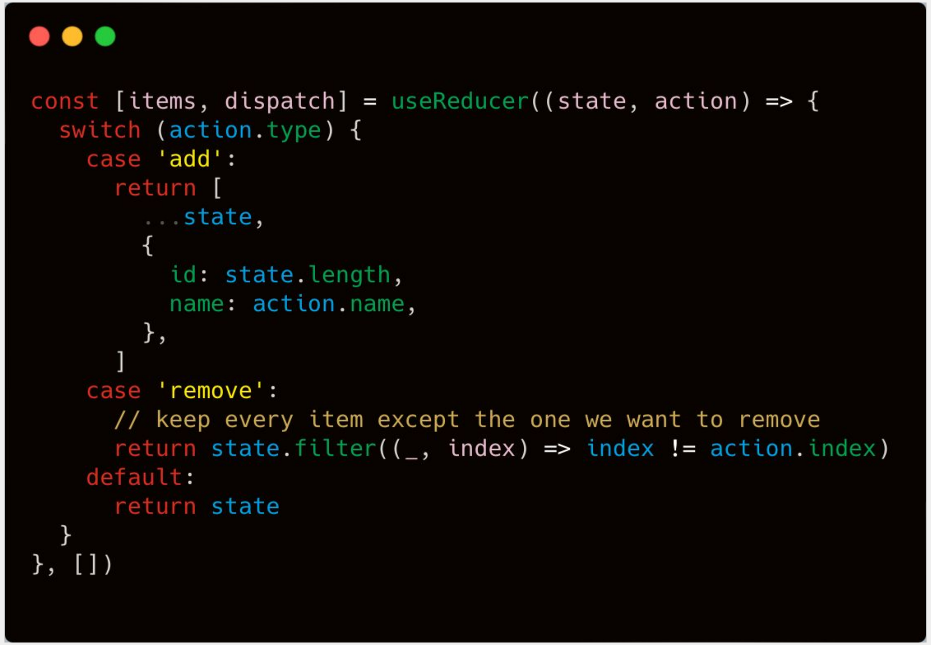
- { type: “add”, name: “cheese” }
- [{id: 0, name: “cheese”}]
- { type: “add”, name: “bread” }
- [{id: 0, name: “cheese”}, {id: 1, name: “bread”}]



```
const [items, dispatch] = useReducer((state, action) => {
  switch (action.type) {
    case 'add':
      return [
        ...state,
        {
          id: state.length,
          name: action.name,
        },
      ]
    case 'remove':
      // keep every item except the one we want to remove
      return state.filter((_, index) => index != action.index)
    default:
      return state
  }
}, [])
```

A Reducer Example

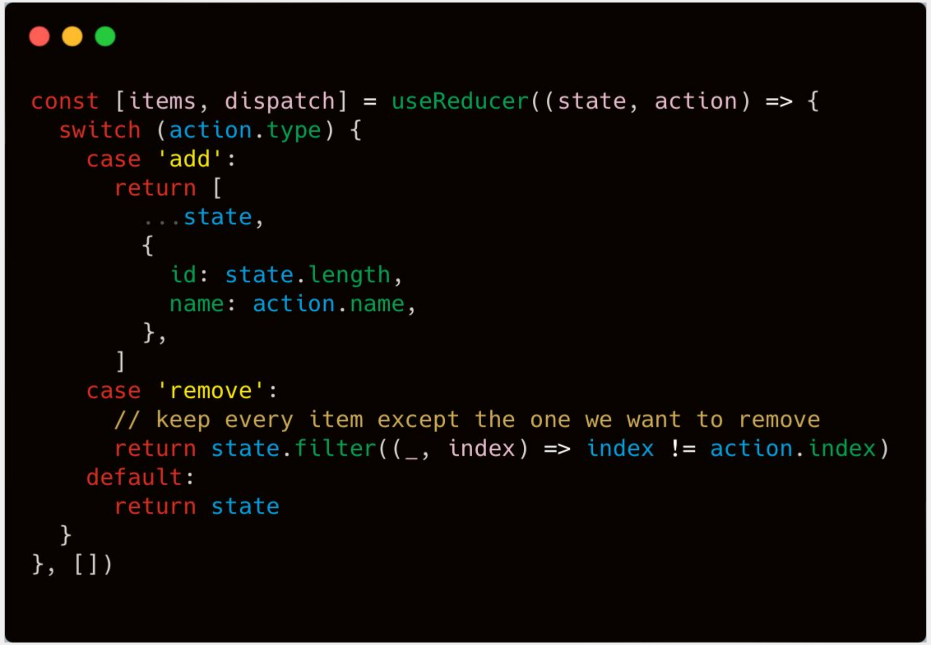
- { type: “add”, name: “cheese” }
- [{id: 0, name: “cheese”}]
- { type: “add”, name: “bread” }
- [{id: 0, name: “cheese”}, {id: 1, name: “bread”}]
- { type: “remove”, index: 0 }



```
const [items, dispatch] = useReducer((state, action) => {
  switch (action.type) {
    case 'add':
      return [
        ...state,
        {
          id: state.length,
          name: action.name,
        },
      ]
    case 'remove':
      // keep every item except the one we want to remove
      return state.filter((_, index) => index != action.index)
    default:
      return state
  }
}, [])
```

A Reducer Example

- { type: “add”, name: “cheese” }
- [{id: 0, name: “cheese”}]
- { type: “add”, name: “bread” }
- [{id: 0, name: “cheese”}, {id: 1, name: “bread”}]
- { type: “remove”, index: 0 }
- [{id: 1, name: “bread”}]



```
const [items, dispatch] = useReducer((state, action) => {
  switch (action.type) {
    case 'add':
      return [
        ...state,
        {
          id: state.length,
          name: action.name,
        },
      ]
    case 'remove':
      // keep every item except the one we want to remove
      return state.filter((_, index) => index != action.index)
    default:
      return state
  }
}, [])
```

Questions?

But how can we share reducer values?

- useContext!
  - Share the value with useContext
- Export the dispatch function from one central place
  - Ex: dispatch.js
  - We can just import it then!

State Management

- useContext / useReducer: simple state management
- Change state in a centralized file without explicitly passing it around
    - Dispatch actions to modify state

State Management

- useContext / useReducer: simple state management
- Change state in a centralized file without explicitly passing it around
  - Dispatch actions to modify state
  - This is more *declarative*: we're saying what we want to do
    - Often more understandable

State Management

- useContext / useReducer: simple state management
- Change state in a centralized file without explicitly passing it around
    - Dispatch actions to modify state
    - This is more *declarative*: we're saying what we want to do
        - Often more understandable
    - Easier to change how we handle updates if logic changes

State Management

- Sneak peek: Redux!
    - More complex state management
    - Leverages immutability more

State Management

- Sneak peek: Redux!
  - More complex state management
  - Leverages immutability more
    - Better tooling

Redux

- Everything you just saw, and more:

Redux

- Everything you just saw, and more:
  - Better dev tools!
    - Can see what actions are emitted, in the browser!

Redux

- Everything you just saw, and more:
    - Better dev tools!
        - Can see what actions are emitted, in the browser!
    - Time travel debugging
        - Replay the actions emitted and watch state change!

Redux

- Everything you just saw, and more:
    - Better dev tools!
        - Can see what actions are emitted, in the browser!
    - Time travel debugging
        - Replay the actions emitted and watch state change!
    - Can easily log actions to see what’s going on in the app

Redux

- Structurally a little different
    - Single global data structure that reducers are looked up in

Redux

- Structurally a little different
  - Single global data structure that reducers are looked up in
  - But global variables are bad!

Redux

- Structurally a little different
    - Single global data structure that reducers are looked up in
    - But global variables are bad!
    - Not so bad if we can very easily tell when they change

Redux

- Structurally a little different
  - Single global data structure that reducers are looked up in
  - But global variables are bad!
  - Not so bad if we can very easily tell when they change
- Adds even more complexity than useContext/useReducer

Which should I use?

- Should we use state management here? If so, which?

Which should I use?

- Should we use state management here? If so, which?
- Example 1: Counter component (like in Catbook!)

Which should I use?

- Should we use state management here? If so, which?
- Example 1: Counter component (like in Catbook!)
  - No State Management!

Which should I use?

- Should we use state management here? If so, which?
- Example 1: Counter component (like in Catbook!)
  - No State Management!
- Example 2: Task management dashboard with a task list, task detail view, and a sidebar with filters

Which should I use?

- Should we use state management here? If so, which?
- Example 1: Counter component (like in Catbook!)
  - No State Management!
- Example 2: Task management dashboard with a task list, task detail view, and a sidebar with filters
  - useContext/useReducer!

Which should I use?

- Should we use state management here? If so, which?
- Example 1: Counter component (like in Catbook!)
    - No State Management!
- Example 2: Task management dashboard with a task list, task detail view, and a sidebar with filters
    - useContext/useReducer!
- Example 3: Large e-Commerce platform with a shopping cart, user authentication, and various product pages

Which should I use?

- Should we use state management here? If so, which?
- Example 1: Counter component (like in Catbook!)
    - No State Management!
- Example 2: Task management dashboard with a task list, task detail view, and a sidebar with filters
    - useContext/useReducer!
- Example 3: Large e-Commerce platform with a shopping cart, user authentication, and various product pages
    - Redux!

Wrap-up

- There's a lot more that we haven't said
  - Redux is complicated
  - Interesting performance optimizations
    - Re-render only when relevant state changes
- State management adds complexity
  - Can make code more maintainable and easier to understand
- Maybe it's a good idea for your project!
- Questions?