Consuming React State so far

- State defined at some level
    - App-wide state at top-level
    - Scoped state in a component
- State passed as props to descendant components
- Setters passed as props
    - Passed directly
        - `useState` setters or `dispatch()`
    - Passed wrapper functions
        - Around `useState()` setters or `dispatch()`

Prop Drilling

Passing props through multiple layers of components

- When those components don't use the props
    - Pass to some descendants so they have them

This **prop drilling**

- Undesireable
    - Couples components to state they don't use
    - Cognitive overhead

Context - an extremely overused word

In React, a "context"

- Allows a Component to gain access to a value
    - Without that value being passed as prop
- Value must be defined in ancestor **Provider**
- Component must explicitly get value

Used to avoid prop drilling

- Bad to overuse Contexts
    - Hides use and source of value
- Balance where to have complexity

How to make use of Context

You must

1. **Define a Context using** `createContext()`
2. **Pass the value to the context Provider**
3. **Get the value using** `useContext`

Creating Context is a little odd

- React code, but not JSX
- Has a property that IS a Component
- We will still use MixedCase naming style

```
import { createContext } from 'react';

const defaultValue = 'TodoContext missing Provider';
const TodoContext = createContext(defaultValue);
```

- Name context for the value it will hold
    - Big apps may have multiple contexts
- MixedCase naming but NOT a Component
- `defaultValue` is a "should not happen" case
    - Use a value that highlights an error

Providing Context

- Context holds a value
    - Usually a variable value
    - Makes available to other components
    - ...without passing as a prop

Provider Component makes value available

```
<TodoContext.Provider value={someValueHere}>
  <SomeComponent/>
</TodoContext.Provider>
```

Descendants of Provider can access Context value

- Anything outside Provider does not

Consuming Context

The `useContext()` hook gets you the actual value

```
import { useContext } from 'react';
import TodoContext from './TodoContext'; // NOT a component

function SomeComponentDecendant() {  // not passed someValue
  const someValue = useContext(TodoContext);
  return (
    <div>
      { someValue }
    </div>
  );
}
```

Descendants of a Context Provider

- Can get the value from Context `Provider`
- Must `useContext` the Context object itself

About Consuming Content

You:

- **Created** the context
- **Provided** the context to descendants
- **Consumed** the context
  - via `useContext` and context object
  - as a descendant of a provider
  - got the values
  - ...but no setters?

What are the practical benefits?

- "value" can be state, setters, BOTH
    - JS values include arrays and objects

```
const [catState, setCatState] = useState('Jorts');

return (
  <CatContext.Provider value={ [catState, setCatState] }>
    <div className="app">
      <SomeComponent/>
    </div>
  </CatContext.Provider>
);
```

```
function SomeComponentDescendant() {
    const [catState, setCatState] = useContext(CatContext);
    //...
}
```

The Context can provide access to

- Simple State (ex: a string)
- Complex State (ex: an object)
- State and Setter
- Useful functions built from state
- Wrapped Setter functions (such as `onLogin`)

If it could be passed as a prop

- can be in Context

Only use Context to avoid deep prop-drilling

- To keep layers from being coupled
- If they are coupled anyway, pass as props

Passing wrapped setters without Context

```
const [theme, setTheme] = useState('dark');

return (
  <SomeChild
    theme={theme}
    darken={ () => setTheme('dark') }
    lighten={ () => setTheme('light') }
  />
);
```

```
function SomeChildDescendant({ theme, darken, lighten }) {
  return (
    <div>
      Your theme is {theme}
      <button onClick={lighten}>Lighten Up!</button>
      <button onClick={darken}>Brood and scowl</button>
    </div>
  );
}
```

Passing wrapped setters using Context

```
const [theme, setTheme] = useState('dark');
const darken = () => setTheme('dark');
const lighten = () => setTheme('lighten');

return (
  <ThemeContext.Provider value={ {theme, darken, lighten} }>
    <SomeChild/>
  </ThemeContext.Provider>
);
```

```
function SomeChildDescendant() {
  const {theme, darken, lighten} = useContext(ThemeContext);
  return (
    <div>
      Your theme is {theme}
      <button onClick={lighten}>Lighten Up!</button>
      <button onClick={darken}>Brood and scowl</button>
    </div>
  );
}
```

Reducers in Context

Reducers are good for:

- Complex state
- Manipulated from different components

Context is good for:

- Complex state
- Shared among many components

Context works well with Reducers

- share `state` and `dispatch`/actions

Passing state and dispatch via Context

```
const [state, dispatch] = useReducer(reducer, initialState);

return (
  <StateContext.Provider value={ [state, dispatch] }>
    <SomeChild/>
  </StateContext.Provider>
);
```

```
function SomeChildDescendant() {
  const [state, dispatch] = useContext(StateContext);
  const darken = () =>
    dispatch({ type: 'setTheme', theme: 'dark'});
  const lighten = () =>
    dispatch({ type: 'setTheme', theme: 'light'});
  return (
    <div>
      Your theme is {state.theme}
      <button onClick={lighten}>Lighten Up!</button>
      <button onClick={darken}>Brood and scowl</button>
    </div>
  );
}
```

State values and dispatch wrappers via Context

```
const [state, dispatch] = useReducer(reducer, initialState);
const theme = state.theme;
const darken = () =>
  dispatch({ type: 'setTheme', theme: 'dark'});
const lighten = () =>
  dispatch({ type: 'setTheme', theme: 'light'});
return (
  <ThemeContext.Provider value={ {theme, darken, lighten} }>
    <SomeChild/>
  </ThemeContext.Provider>
);
```

```
function SomeChildDescendant() {
  const {theme, darken, lighten} = useContext(ThemeContext);
  return (
    <div>
      Your theme is {theme}
      <button onClick={lighten}>Lighten Up!</button>
      <button onClick={darken}>Brood and scowl</button>
    </div>
```

```
    );
}
```

Summary - State and Context

- Your state is the key to how your app works
    - It will track everything that can change
- App-wide state is share with many components
    - Prop-drilling complicates/couples components
- useContext shares state/actions w/o prop-drilling
- useContext hides dependencies
- useContext can cause unnecessary re-renders

It depends

Summary - Context syntax

- Create using `createContext()`, export result
  - Default value to notice lack of Provider
- Import and render `<SomeContext.Provider>`
  - `value` prop is context value
- Descendant imports context object
  - uses `useContext(someContext)` to get value
- You can have many nested Providers
  - Each a different Context

Summary - Avoiding Context

- Context isn't BAD
    - It has costs
    - Use when benefit outweighs costs