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
    - As `dispatch()`
    - In abstract action functions
        - `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

React "Context"

- Allows access to a value
- Returned from a hook
    - Not passed as a prop

Used to avoid prop drilling

- Bad to overuse
    - Hides where value comes from
- Balance where to have complexity

Context Parts

3 Parts to using context

- Creating the context object
- Making a value on context available
    - To part of React Component tree
- A component getting the available value

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 React from 'react'; // We use "React" below

const MyContext = React.createContext(defaultValue);
```

- `MyContext` is a BAD variable name!
    - Better: `TodoContext`, etc
- MixedCase naming but NOT a Component
- `defaultValue` is a "should not happen" case
    - Give it values highlighting an error

Providing Context

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

Provider Component makes Context available

```
<MyContext.Provider value="someValueHere">
  <SomeComponent/>
</MyContext.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 MyContext from './MyContext'; // NOT a component

function SomeComponent() {  // value not from a prop
  const someValue = useContext(MyContext);
  return (
    <div>
      { someValue }
    </div>
  );
}
```

Descendants of a Context Provider

- Can get the value of the context
- Must have the Context 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?

- The "value" in the context can be anything
    - Including state, or setters, OR BOTH
    - Recall a "value" can be an object or array

```
// App.jsx, assume a CatContext

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

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

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

Example of passing props

- Can pass state
- Can pass setter
- Can pass wrapper functions

```
const [theme, setTheme] = useState('dark');
return (
  <SomeChild
    darken={ () => setTheme('dark') }
    lighten={ () => setTheme('light') }
  />
);
```

Abstract setters in context

You can also pass callbacks with Context:

```
const [theme, setTheme] = useState('dark');
const darken = () => setTheme('dark');
const lighten = () => setTheme('lighten');
return (
  <ThemeContext.Provider value={ {theme, darken, lighten} }>
    <SomeChild/>
  </ThemeContext.Provider>
);
```

```
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

Avoiding Context

Context/useContext:

- Good to avoid coupling via prop-drilling
- Additional abstraction/complexity
- Hides dependencies
    - Props previously showed all dependencies
- All consumers rerender on context value change
    - New object, same content? Rerender!
    - New object, the parts you use unchanged? Rerender!

Rendering children

JSX element contents?

- Passed as special prop `children`

```
return (
  <SomeWrapper>
    <p>Some Content</p>
    <SomeThing value={catInfo}/>
  </SomeWrapper>
);
```

```
const SomeWrapper = ({ children }) => {
  return (
    <div className={'wrapped'}>
      <h1>Title Here</h1>
      {children}
    </div>
  );
};
```

Alternatives to Context: Components as Children

- Create descendants directly

```
// Any Component

const [stateToDrill, setStateToDrill] = useState('');

return (
  <>
    <Content>
      <TodoList todos={stateToDrill } />
    </Content>
  </>
);
```

- `<Content>` isn't passed the `stateToDrill` prop
- `<Content>` gets and can render `children` prop
- The contents of `children` (TodoList) have the prop

Alternatives to Context: Redux

Common Question: useContext vs Redux?

- "It depends"
- Redux is better performance
    - Avoids unnecessary rerenders
- Redux is extra layer of abstraction/complexity
    - More complex than useContext
- What state to have in Redux?
    - Common answer is "all"
        - Not often the best answer

Thinking in State and Actions

`useReducer` and `useContext`

- Easier if you think in terms of **state** and **actions**
- State
    - UI state and App state
    - One or many variables
- Actions
    - Changes to state for a reason
- Data models are the way to think about code
- Good to refector code as you write!

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 + export `React.createContext()`
    - Default value to notice lack of Provider
- Component imports and renders `<YOURCONTEXT.Provider>`
    - `value` prop is context value
        - Changes on render of Provider
    - Wraps descendants that access context
- Descendant imports context
    - uses `useContext(YOURCONTEXT)` to get value
- You can have many nested Providers

Summary - Avoiding Context

- Context isn't BAD
    - It just has costs
    - Use when benefit outweighs costs
- Alternative: pass descendant directly
- Alternative: Redux and other state mgmt libs

Summary - Thinking about State

- Initial State?
    - `useState` for all?
    - `useReducer` for some?
    - Switch to reducer once complexity happens?
- Passing Props
    - Assume Context?
    - Add once/if prop-drilling occurs?