Consuming React State so far

- Composable Function based components
- with state and props
- auto-render when state/props change
- state can be
    - multiple or single values
    - simple or complicated
    - directly set or via dispatched actions
- state and setters can be
    - use in same component
    - passed to another component
    - wrapped and the wrapper functions passed

Passing State

Increasing complexity of state when a component:

1. uses its own state
2. passes state to child component as prop
3. passes state to a descendant within itself
4. passes state to a child that passes it on
5. passes component as prop
6. uses state from Context

Direct Passing

Passing as a prop:

```
return (
  <SomeChild value={fromState.value}/>
);
```

Passing directly to a descendant:

```
return (
  <SomeChild>
    <SomeOther value={fromState.value}/>
  </SomeChild>
);
```

Rendering children

The contents of a JSX element are passed as the 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>
  );
};
```

Why pass a component?

When you want:

- a wrapping layer
- That is IGNORANT of (decoupled from) some of the content inside it

You can pass a component to the wrapper

- that is part of the contents to be wrapped

This is just a more general case of passing via `children`

Passing a component

Passing a component as a prop:

```
return (
  <SomeChild thing={<SomeOther value={fromState.value}/>} />
);
```

Using a passed component:

```
const SomeChild = ({ thing }) => {
  return (
    <div>
      <p>Other content</p>
      {thing}
    </div>
  );
};
```

This allows ALL passing of state

Why do we have another option?

Pass a component as a prop when you want:

- wrapper to be ignorant of the content

Use Context when you want:

- wrapper to know the content
- ...but not the state (decouple from state)
- ...because the state doesn't impact the wrapper

Wrapper isn't "generic"

- but is separate from state

What is "context" in React?

Context in React is:

- a way to access one value
    - simple or complex (arrays/objects okay)
- Created via `React.createContext()`
- That can be used in a "Provider" component
- Can be accessed in descendant that uses `useContext`

Each Provider is only 1 context

- can nest providers and consume multiple contexts

Creating Context

```
// OUTSIDE of component function
// probably in a separate file
const CatContext = React.createContext({
  default: 'Overridden by provider value'
});
export default CatContext;
```

```
const App = () => {
  const [catState, setCatState] = useState({
    name: 'Grumpy Cat',
    mood: 'joyful',
  });

  return (
    <CatContext.Provider value={ catState }>
      <div className="app">
        <SomeChild/>
      </div>
    </CatContext.Provider>
  );
};
```

About Creating Context

- **Creating** the context
  - Export from an separate file
  - Has a default, but often isn't the REAL default
- **Providing** the context to descendants
  - A wrapping component
  - Needs the context object
  - sets the value
    - overrides the "default"
    - ...even if null/undefined
    - We use state to hold the value

Context is ACCESS to a value, not automatically State

Consuming Context

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

const SomeChild = () => {
  const catState = useContext(CatContext);
  return (
    <div>
      {catState.name} is feeling {catState.mood}
    </div>
  );
};
```

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

Modifying State via Context

Context is ACCESS to a value, not state

You can provide any setters in the context value

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

```
const [catState, setCatState] = useContext(CatContext);
return (
  <div>
    {catState.name} is feeling {catState.mood}
  </div>
);
```

Context isn't State

You CAN pass a simple state setter as a prop

- But we often pass a callback to abstract state

```
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

They are often a good pairing

- share `state` and `dispatch`

Do not overcomplicate state

Only be as complex as you need

- Start with `useState`
- Pass as props
- Add new state as needed
- Don't plan too far ahead

If too much state/callbacks passed "deep"

- switch to Context and `useContext`

If you find you are changing a lot of state

- switch to `useReducer`

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
- Don't rush to solve the specific issue!