What is the component hierarchy?
Review: React state placement

- SearchBar and MovieTable both need the “search term” and “sort type”
- State should “live” in the nearest common ancestor (FilmExplorer)

Recall data flows “down” via props and data flows “up” via callbacks
You are embedding the color picker in a drawing app (to pick the pen color), where should you maintain the color state?

A. In the ColorPicker, and use a callback to communicate changes to the parent drawing component
B. In the drawing component
C. Neither, I heard I am supposed to use Redux to manage state

Answer: B

The React philosophy to is to maintain one source of truth. Thus there should be one instance of the pen color (in the drawing component that needs it) and it is passed as a prop to the color picker (and updated from the color picker via callback). The tradeoff of this approach is that we may have “lace” that state through many components. There are several ways to mitigate that burden. Redux is one. There are a lot of tools that can be used with React. And the Internet will have strong opinions. But I want to advocate against any change that starts with “I heard that ...” From Dan Abramov of the React team (and creator Redux).

“However, if you’re just learning React, don’t make Redux your first choice. Instead learn to think in React. Come back to Redux if you find a real need for it, or if you want to try something new. But approach it with caution, just like you do with any highly opinionated tool.”

Recent versions of React incorporated Contexts (effectively pseudo-global variables) to reduce the “lacing” burden.
Container components: Separating logic from UI

Separation of Concerns:

- **Container Component (CC):** Concerned with how the application works, i.e. implements logic
- **Presentation Component (PC):** Concerned with how the application looks. Typically generates DOM.

"Remember, **components don't have to emit DOM. They only need to provide composition boundaries between UI concerns.**" — Dan Abramov
Interlude: Sequences in React

function MovieTable(props) {
    const films = props.movies.map(movie => {
        <MovieContainer
            key={movie.id}
            {...movie}
            setRatingFor={props.setRatingFor}
        />
    });
    return <div>{films}</div>;
}

“Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity. Most often you would use IDs from your data as keys” - ReactJS Docs
How can we apply this same idea to MovieSummary/MovieDetail? The “logic” is switching between the two components. Let’s pull that into a container component (MovieContainer) that implements the switch and maintains the corresponding state (a boolean). That container then implements conditional rendering.
Interlude: **Conditional rendering**

```javascript
render() {
  if (this.state.detail) {
    return <MovieDetail {...this.props} onClick={this.handleClick} />;
  }
  return <MovieSummary {...this.props} onClick={this.handleClick} />;
}
```

(render() is code and so can use conditionals to change views)

Some other common conditional patterns:

{boolean && <Component ... />}
{boolean ? <Component1 ... /> : <Component2 ... />}

https://reactjs.org/docs/conditional-rendering.html
You have implemented a CommentList component that fetches an array of comments from your server and renders those comments as a unnumbered list (i.e. `<ul>...</ul>`). CommentList is a:

A. Presentation component  
B. Container component  
C. Both a presentation and container component  
D. Neither a presentation not container component

Answer: C

As described CommentList is both a Presentation Component and Container Component, in that it generates DOM (the `<ul>`) and so is concerned with how the application looks *and* is concerned with how the application works (i.e. gets comments from server). It should be split into a container component that fetches the data and a CommentList component that displays the comment list UI.
CC vs. PC is not a technical distinction, it is a dichotomy of purpose. What are some technical dichotomies?

What are keys features only available in classes? State! That said, because functional components are easier to understand, I suggest you to use them unless you need state, lifecycle hooks, or performance optimizations, which are only available to the class components at this time.

Adapted from Dan Abramov
What challenges lurk inside MovieTableContainer?

What is the potential issues in this code?
```javascript
this.props.movies.sort(...);  // Sorts in place!
return (<MovieTable movies={this.props.movies} />);
```

A fix...
```javascript
let films = this.props.movies.slice();  // Make a deep copy
films.sort(...);
return (<MovieTable movies={films} />);
```

Analogous approach for objects
```javascript
Object.assign({}, oldObject, { property: newValue });
```

Recall props are immutable! How else can tackle this problem, Immutable Data Structures

How does Object.assign work in this context, assign overwrites the properties of its 1st argument with the remaining arguments (in order). Thus this create a new empty objects, overwrites with the properties in oldObject and then overwrites property to be newValue
Interlude: Immutable data structures

- **Immutable**: Once created, a collection cannot be altered
- **Persistent**: Can create new collections from previous collection and a mutation. The original is still valid.
- **Structural Sharing**: New collections use the same structure as the original where possible to reduce copying

*Immutable*: Once created, a collection cannot be altered

*Persistent*: Can create new collections from previous collection and a mutation. The original is still valid.

*Structural Sharing*: New collections use the same structure as the original where possible to reduce copying
Interlude: Immutable data structures

- **Immutable**: Once created, a collection cannot be altered
- **Persistent**: Can create new collections from previous collection and a mutation. The original is still valid.
- **Structural Sharing**: New collections use the same structure as the original where possible to reduce copying

```javascript
const { Map } = require('immutable');
const map1 = Map({ a: 1, b: 2, c: 3 });
const map2 = map1.set('b', 50)
'${map1.get('b')} vs ${map2.get('b')}'  // ? vs ?
```

// 2 vs. 50
// because the set on line 3 does not modify map1 (recall these data structures are immutable).

Adapted from [immutable.js](https://github.com/RRhoey/immutable.js)
Take home message: Don’t mutate state or props

- Mutated props will not compare as different objects and so may not trigger a re-render
- Assigning to state does not trigger a re-render

```javascript
// Wrong
this.state.comment = 'Hello';

// Correct
this.setState({ comment: 'Hello' });
```

https://reactjs.org/docs/state-and-lifecycle.html#do-not-modify-state-directly
Stepping back: We use inheritance to enable customization and facilitate code reuse (e.g. our child gets parent’s methods for “free”).

By inheritance we mean having the same implementation as the parent. Note that inheritance and subtyping are not the quite the same, although in many languages, e.g. Java, they co-occur because the way to create a subtype is via inheritance. JavaScript is not one of those languages.

By composition we mean contains instead of inherits from,

Both could be made to work. However, community best practices are to use composition instead of inheritance. In the context of React, composition is typically is more flexible and can satisfy every potential use case for inheritance. There is value in following those practices to improve readability and maintainability (being a special case is not a benefit in SW development). But I think we can also make more formal arguments about inheritance in this context.
When do we use subtyping (inheritance)?

- Subtyping is described by an “is a” relationship, e.g. a car “is a” vehicle
- Composition is described by a “has a” relationship, e.g. a car “has an” engine

So MovieDetail “is a” MovieSummary or “has a” MovieSummary?
Formalizing subtyping: Liskov Substitution Principle

Let $\varphi(x)$ be a property provable about objects $x$ of type $T$. Then $\varphi(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$.

TL;DR; A method that works on an instance of type $T$, should also work on any subtype of $T$

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
When a Square is not a Rectangle

class Rectangle {
    constructor(w, h) {
        this.w = w;
        this.h = h;
    }
    setWidth(w) { this.w = w; }
}

class Square extends Rectangle {
    constructor(side) {
        super(side, side);
    }
    setWidth(w) {
        this.w = w;
        this.h = h;
    }
}
Which of the following two statements about the Liskov Substitution Principle are true?

a) In duck-typed languages, LSP violations can occur even when inheritance is not used

b) In statically-typed languages, if the compiler does not report any type errors or warnings, then there are no LSP violations

A. Only (a) is true
B. Only (b) is true
C. Both (a) and (b) are true
D. Neither (a) or (b) are true

Answer: A

LSP isn't necessarily tied to inheritance or class-based typing and thus applies to both duck-typed and statically-typed languages. Any time polymorphism is used (in whatever form) the LSP is applicable. We just saw an example with squares and rectangles that would compile in a statically typed language but would still be a LSP violation.
Lots of callbacks because we need to implement onChange to update value.

In React, an `<input type="file" />` is always an uncontrolled component because its value can only be set by a user, and not programmatically.

https://goshakkk.name/controlled-vs-uncontrolled-inputs-react/
Explicitly bind this for handler like shown in React docs ... this would involve adding the following to the constructor

```javascript
this.handleChange = this.handleChange.bind(this);
```

then you can use

```javascript
onChange={this.handleChange}
```

as the callback prop. The advantage is don’t have to create a new function every time the component renders. This disadvantage is the increase boilerplate.