What is the component hierarchy?

FilmExplorer
SearchBar
MovieTable
MovieSummary
Review: React state placement

Recall data flows “down” via props and data flows “up” via callbacks

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

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

“Arrays” need key to uniquely identify components

“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
CC applied: MovieContainer

How would you apply this design pattern to the toggling between MovieSummary and MovieDetail?

```javascript
this.state.detail = false;
```

Modify state in parent via callback passed to MovieSummary

```javascript
this.state.detail = true;
```
Interlude: **Conditional rendering**

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

Some other common conditional patterns:

```javascript
{boolean && <Component ... />}
{boolean ? <Component1 ... /> : <Component2 ... />}
```
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
What are some React technical “dichotomies”?

Stateful vs. Stateless

CC are typically stateful, PC typically stateless (but not always)

Class vs. Functional Components

Classes can have state! And lifecycle methods. Functions are suggested unless you need Class features since they are simpler and may be optimized in the future
What challenges lurk inside MovieTableContainer?

What is the potential issues in this code?

```javascript
this.props.movies.sort(...);
return (<MovieTable movies={this.props.movies} />);
```

Sorts in place!

A fix...

```javascript
let films = this.props.movies.slice();
films.sort(...);
return (<MovieTable movies={films} />);
```

Make a deep copy

Analogous approach for objects

```javascript
Object.assign({}, oldObject, { property: newValue });
```
Interlude: Immutable data structures

• Immutable
• Persistent
• Structural Sharing
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 ?
```

Adapted from 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

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

// Correct
this.setState({ comment: 'Hello' });
React: Composition vs. Inheritance?

Should MovieDetail inherit from MovieSummary or contain a MovieSummary?
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 $T$.

Turing Award Winner
Barbara Liskov

TL;DR; A method that works on an instance of type $T$, should also work on any subtype of $T$
When a Square is not a Rectangle

```javascript
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;
    }
}
```

Assumption is that changing width doesn’t change height
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
React: Controlled vs. Uncontrolled

(Familiar?) Controlled component:

```jsx
<input type="text" value={{...}} onChange={{...}}/>
```

- + Single source of truth
- - Lots of callbacks

Uncontrolled component:

```jsx
<input type="text" ref={(input) => this.input = input} />
```

Reference to real DOM element

<table>
<thead>
<tr>
<th>Feature</th>
<th>Controlled</th>
<th>Uncontrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>One–time retrieval, e.g. on submit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validating on submit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instant validation</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>Conditionally disabling submit</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>Several inputs for one piece of data</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>Dynamically modify data (e.g. capitalize)</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td><code>&lt;input type=&quot;file&quot; /&gt;</code></td>
<td></td>
<td>×</td>
</tr>
</tbody>
</table>
Review: React controlled components

```jsx
class Example extends Component {
  constructor(props) {
    super(props);
    this.state = { title: '' };  
  }

  render() {
    return (<input  
      type="text"
      value={this.state.title}  
      onChange={({event) => this.setState({title: event.target.value})})  
    />);
  }
}
```

Update state on every input

Render state

Subtleties with `this` in JSX callback
- User “fat arrow”, like above
- Explicitly bind `this` for handler like shown in React docs