React uses the order in which hooks are called to maintain the mapping between state and useState calls. Thus the order needs to be same every time the React function is invoked (conditions and loops are likely to violate this assumption). The second rule ensures that all stateful logic in a component is clearly visible from its source code. There are ESLint rules included in CRA that will check some aspects of these rules (but no linter rule is perfect...).

https://reactjs.org/docs/hooks-rules.html
Here is a representative implementation of React. One question is where is React state actually stored? In a closure (here useState closes over the local hooks array in the PseudoReact function).

Notice that hooks are tracked by index. Thus all hooks need to execute in the same order every time. Loops, conditions, nested functions, etc. all have the potential to change to order in which the hooks are invoked. And thus we should invoke a hook inside any of those constructs.

How does React know what functions to invoke? It keeps track of all of the components you previously rendered starting with the ReactDOM.render call ”kicks” off React and inserts the results in the web page.

Hooks aren't the only way. Prior to hooks, and still, React implemented stateful components with classes. There is a nearly one to one mapping between the state we created with hooks and what we would implement with classes. The obvious distinction is that with hooks we create discrete instances of state (with the corresponding setters) instead of having the combined state and single setter. The end result is similar.
Why hooks? We choose Hooks in part based on the React team recommendations. Our components so far have been very simple, but they won’t stay that way. One challenge is that as our components grow in complexity, we can’t break them down any further into smaller, more manageable and more maintainable pieces because the logic is tightly bound with the state. As Dan Abramov describes we end up with:

- Huge components that are hard to refactor and test.
- Duplicated logic between different components and lifecycle methods.
- Complex patterns like render props and higher-order components.

All bad things. Hooks, are an attempt to address those challenges by “organizing logic inside a component into reusable isolated units”.

As a practical matter, hooks are also an important part of React native (used on mobile devices) and we want to prepare you for mobile classes.

More generally, it is entirely possible that in your project you will not need the capabilities that make Hooks useful (i.e. the ability to create custom hooks), but as is often the case in this class, we want to introduce the concepts that may be useful in the future.
Recall: “Thinking in React”

1. Break the UI into a component hierarchy
2. Build a static version in React
3. Identify the minimal (but complete) representation of state
4. Identify where your state should live
5. Add “inverse” data flow (data flows down, callbacks flow up)

https://reactjs.org/docs/thinking-in-react.html
As a starting point we can look for repetition. We have two film entries. Those are likely a component that implements the same view (what is changing are the props that specify the title, rating, etc.). We have a list of films so that is likely another component. The “search bar” is different from the film list and so will be its own component.

In general we are looking for repetition and “boundaries” between the roles of different aspects of our UI.
Review: React state placement

- SearchBar and FilmTable both need the “search term” and “sort type”
- State should “live” in the nearest common ancestor, i.e. 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” (termed “prop drilling”) burden.
Some of the role of container components has been taken over by custom hooks which can collect logic (for reuse). For example we could create a “sorting” hook that encapsulates the sorting operation in FilmExplorer, i.e. in FilmExplorer we would have something like

```
[films, setFilms, setSearchString, setSortField] = useSortedFilms(data);
```
Recall that React is trying to figure the minimal number of edits to apply when updating the browser screen. If you insert an element of the array it might seem to React that all of the elements in the array have changed because now `oldArray[0] !== newArray[0]`. And thus React might do a lot more work re-rendering all the elements. But in reality the rendering of all the remaining elements can be reused. Using keys in this context helps React realize that elements just shifted (and thus can be reused).
How can we apply this same idea to FilmSummary/FilmDetail? The “logic” is switching between the two components. Let’s pull that into a container component FilmContainer that implements the switch and maintains the corresponding state (a boolean). That container then implements conditional rendering.
**Interlude: Conditional rendering**

function FilmContainer(props) {
    const [showDetail, setShowDetail] = useState(false);
    if (showDetail) {
        return <FilmDetail
            {...props} onClick={() => setShowDetail(false)} />
    }
    return <FilmSummary
        {...props} onClick={() => setShowDetail(true)} />
}

Some other common conditional patterns:

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

The first other pattern utilizes short circuit evaluation in the and (&&) operation. If the first time is falsy JS won’t evaluate the second expression. And React will not render anything for {false}. The second pattern is the ternary operator which is effectively an inline if-else expression. If the Boolean predicate evaluates to truthy it will evaluate to Component1 (before the colon), if falsy it will evaluate to Component2 (after the colon).

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 an 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 could be split into a container component that fetches the data and a CommentList component that displays the comment list UI.
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.

Function components are suggested in all situations (using Hooks if stateful)

CC vs. PC is not a technical distinction, it is a dichotomy of purpose. What are some technical dichotomies?

Prior to hooks, State could only be implemented in classes. Function components could only used for stateless components (for which they were recommended over classes). Now with hooks function components can be stateful, and are recommended in all but a few highly specialized situations.

Adapted from Dan Abramov
Although we mutated one of the elements in the films array, the films variable still points to the same array object. The state setter compares the new and old object when deciding to re-render. The comparison rules are lengthy, but generally simple values like integers are compared via equality while objects are compared by reference. In this case, since it is the same object (old films and new films point to the same array in memory), React may not trigger a re-render.

In general, we don’t want to mutate props or state objects (and this is why).

The spread operator works by populating the new object literal with all of the properties of the film object and then overwrites that with rating (this concise syntax is short for `rating: rating`). 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 object, overwrites with the properties in film and then overwrites the rating property with the new rating.

Another approach to this problem is to use immutable data structures (that prevent mutation).
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).

Immutable data structures help by always returning a new object from any expression that mutates the data structure such that upon shallow compare the structures are unequal, e.g. map1 and map2 point to different objects in memory and thus map1 !== map2 would be evaluate to true.
So should we always use immutable data structures? Not necessarily. Think of them as an optimization when working with deeply nested data structures that would otherwise be awkward to copy elegantly. Most of the time the techniques we saw earlier will work fine, but useful to know we have these libraries in our toolbox if we need them. In general, we should always try to do the simple thing first.

https://reactjs.org/docs/state-and-lifecycle.html#do-not-modify-state-directly
By default HTML input components have their own internal state and “update” loop, i.e. dragging the slider updates that internal state. Controlled components override that internal update loop with React’s update loop. Dragging the slider triggers the onChange event which updates the states which triggers a re-render which moves the slider, … The motivation is to maintain that single source of state, that is everything (the logic and the UI) is “controlled” by the same React state.
The “con” for controlled components is lots of callbacks because we need to implement onChange to update value (triggering the re-render). But there are a lot of advantages that come from being able to act on the input state in the component logic.

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/
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 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 FilmDetail “is a” FilmSummary or “has a” FilmSummary?

I think it is more natural to say that FilmDetail has a FilmSummary. Further, as we see more formally, it is not clear that we could or should use a FilmDetail everywhere a FilmSummary is expected. This latter reasoning is our more formal mechanism for thinking about inheritance vs. composition.
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 $T$
The property/assumption of rectangles is that we can change one of the width or height independently of the other. That same property should apply to any subtypes, e.g. squares. But that is not the case for squares. Changing the width also changes the height. Thus this inheritance relationship does not follow the LSP.
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 an LSP violation.