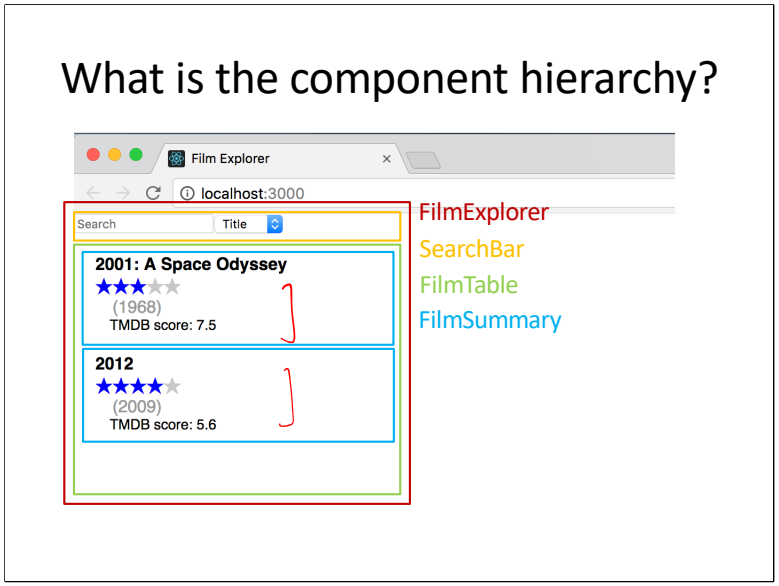
Recall: "Thinking in React"

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://react.dev/learn/thinking-in-react

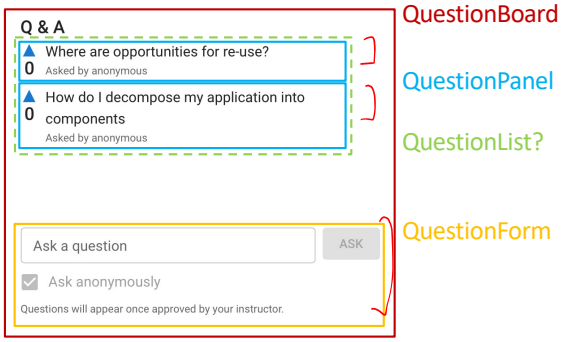


To start, we don't actually need to create multiple components – technically. It will work to have one giant React component. But it will be a nightmare to maintain. That doesn't mean the alternate extreme, extracting lots of fine grain components is the right approach either (moderation in all things). A suggestion: Start from the top, with "simple components" (a term we will talk about in a second), and only extract/split components when needed. When is it needed? Some signs: repeated content, repeated interaction, and the components gets too "big" to the review it all at one time on the screen.

https://www.developerway.com/posts/components-composition-how-to-get-it-right

As a starting point we can look for repetition. We have two film entries. Those are are likely a component that implements the same view (what is changing are the props the specify the tile, rating, etc.). We will have a component to instantiate that array of films. The question will be should that component incorporate the search bar, etc. or not. I suspect that we already get the sense that doing so as one component will be too much and we should likely extract out the search bar as a separate component.

How would you decompose this UI?



https://github.com/csci312-common-v2/class-interactor

How would you decompose this view from the ”class interactor”?

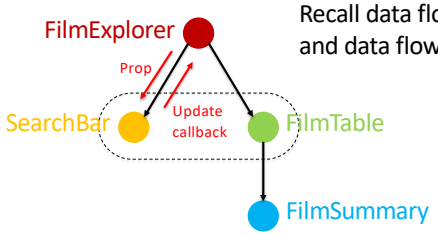
- What are the repeated elements? QuestionPanel?
- Should the QuestionForm be a separate component, or part of the whole? Probably separate to minimize complexity of the overall QuestionBoard.

[click]

Depending on the implementation approach there might also be a QuestionList component that wraps the array of QuestionPanels. The actual application does not use one, but I could imagine doing so if we started to have customized sorting logic, etc.

As a note, the class-interactor is partly a test-bed/demonstrator for this class. I encourage you to check out its code as model of the kinds of things we are working towards this semester.

Review: React state placement



Recall data flows "down" via props and data flows "up" via callbacks

- SearchBar and FilmTable both need the "search term" and "sort type"
- State should "live" in the nearest common ancestor, i.e., FilmExplorer

From Dan Abromov: https://overreacted.io/writing-resilient-components/#principle-4-keep-the-local-state-isolated

If you're not sure whether some state is local, ask yourself: "If this component was rendered twice in different places, should this interaction reflect in the other copy?" Whenever the answer is "no", you found some local state. …

Consider a social media Post component. It has a list of Comment threads (that can be expanded) and a NewComment input….
For example, imagine we rendered the same Post twice. Let's look at different things inside of it that can change.

- *List of comments.* This is similar to post content. We'd want adding a new comment in one tree to be reflected in the other tree too. So ideally, we would use some kind of a cache for it, and it **should not** be a local state of our Post.
- *Expand/Collapse.* I would be weird in expanding/collapsing in one view changes the other, so this be local to the comment threads.
- *The value of new comment input.* It would be odd if typing a comment in one input would also update an input in another tree. Unless inputs are clearly grouped together, usually people expect them to be independent. So, the input value **should** be a local state of the NewComment component.

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 (although A could be the right choice depending on our goal).

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 …”

What about A? It depends on how we conceive of the color update. Should dragging the sliders change the pen color immediately? Or do we want to have a specific update step? For the former, we would want to hoist state up, for the latter, we would likely want separate state within the ColorPicker component, itself.

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.

What are some roles for components?

- Container vs. Presentational[1]
  - Containers implement state & logic
  - Presentational (typically) renders DOM
- Implement vs. Compose[2]
- Simple vs. Container[2] (specific vs. generic?)
  - Simple explicitly render children
  - Container offer generic composition via children prop, etc.
- ~~Stateful (class) vs. stateless (functional)~~

[1]https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
[2]https://www.developerway.com/posts/components-composition-how-to-get-it-right

As you are considering your component hierarchy, here are some potential considerations (and certainly not the only…).

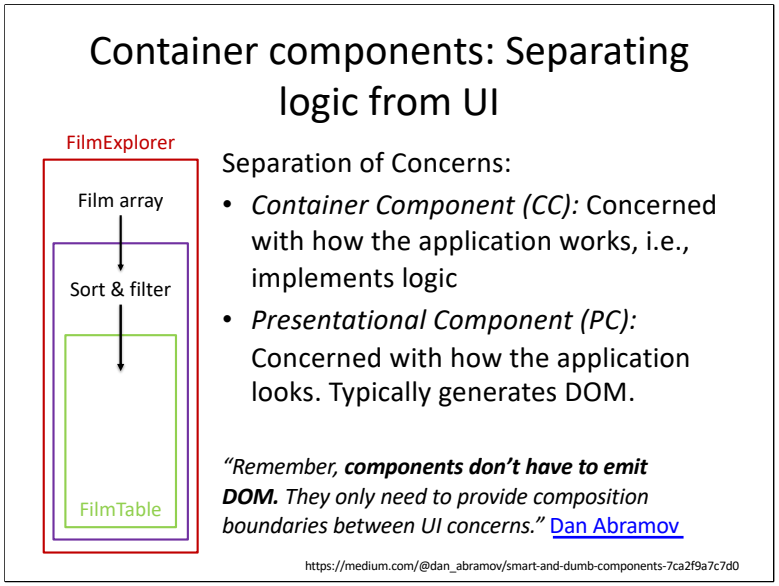
The first encourages us to think about whether a component is responsible for the ”views” seen by the user (presentational) or the logic that underlies the interaction (container). Making that distinction encourages separating those two concerns.

The next consideration is that components should generally either implement specific functionality or compose (group) other components together. From the blog post: ‘A component should be described either as a “component that implements various stuff” or as a “component that composes various components together”, not both.’

The third names are terrible. Perhaps a better description is specific vs. generic. We are considering whether a component is implementing functionality specific to this use case or might be generic/reusable. An example might be a toggle feature that not is specific to any particular use case.

The last used to be a very important technical consideration in the era of classes vs. functional components, which is less (no longer) relevant to in the hooks “era”. Now functional components (components implemented as a

function) can be stateful and we default to functional components for everything. What is a hook? They are “functions that let you “hook into” React state and lifecycle features from function components.” The useState function we saw previously is an example of a hook (they typically have names starting with “use”). They are mechanisms for maintain state within functional components, effectively across renders.



Some of the role of container components has been taken over by custom hooks which can collect logic (for reuse). Dan Abramov, who proposed this notion in 2015, updated the post in 2019 with

"I wrote this article a long time ago and my views have since evolved. In particular, I don't suggest splitting your components like this anymore. If you find it natural in your codebase, this pattern can be handy. But I've seen it enforced without any necessity and with almost dogmatic fervor far too many times. The main reason I found it useful was because it let me separate complex stateful logic from other aspects of the component. Hooks let me do the same thing without an arbitrary division. This text is left intact for historical reasons but don't take it too seriously."

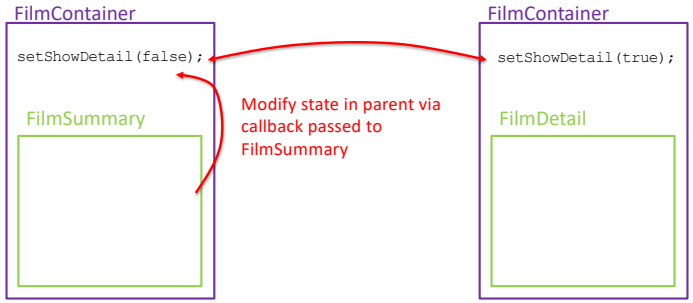
For example, his update would suggest a "sorting" hook that encapsulates the sorting operation in FilmExplorer, i.e., in FilmExplorer we would have something like

[films, setFilms, setSearchString, setSortField] = useSortedFilms(data);

Personally, I think think there is value in this consideration and applying in your design process. Whether that process turns into components or hooks, the underlying considerations are similar.

Container/Composed applied?

How would you apply this design pattern to the toggling between FilmSummary and FilmDetail?



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 FilmContainter) that implements the switch and maintains the corresponding state (a boolean). That container then implements conditional rendering.

Interlude: Sequences in React

```
function FilmTable(props) {
  const films = props.films.map(film => (
    <FilmContainer
      key={film.id}
      {...film}
      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

https://react.dev/learn/rendering-lists

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).

Note that keys are powerful tools outside of sequences. For example, we can use keys when we want to "reset" a component (https://react.dev/learn/you-might-not-need-an-effect#resetting-all-state-when-a-prop-changes)

Interlude: Conditional rendering

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

A React function is code and so you can use conditionals to change views

Some other common conditional patterns:

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

https://react.dev/learn/conditional-rendering

The first other pattern utilizes short circuit evaluation in the and (&&) operation. If the first operand 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://react.dev/learn/conditional-rendering

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. Or now in the hooks era, we could use a hook to fetch the data from the server (effectively serving in the "container" role) and our component would be responsible for rendering the comments as a list.

Simple/Specific vs. Container/Generic

Functional component rendering DOM

```
const Button = ({ title, onClick }) => <button onClick={onClick}>{title}</button>;
```

What if I want a button with an icon?

```
const Button = ({ children, onClick }) =>
     <button onClick={onClick}>{children}</button>;

<Button onClick={onClickHandler}>
  <Icon />
  <span>Create</span>
</Button>
```

Special prop with all child components

] children

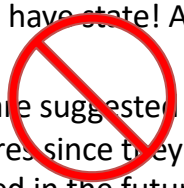
https://www.developerway.com/posts/components-composition-how-to-get-it-right

Or more generally, should a button care what its children are? Not really…

Note that are other, even more sophisticated composition patterns, that we won't get into here.

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)

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

Interlude: Rules of Hooks

- Only call Hooks at the top level of a function

  Don't call Hooks inside loops, conditions, or nested functions

- Only call Hooks from React functions or custom Hooks

  Don't call Hooks from regular JavaScript functions

https://reactjs.org/docs/hooks-rules.html

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 our skeletons that will check some aspects of these rules (but no linter rule is perfect…).

https://reactjs.org/docs/hooks-rules.html

What might go wrong here?

```
const [films, setFilms] = useState([]);
…
const setRating = (filmid, rating) => {
  const index = films.findIndex((film) => film.id === filmid);
  films[index].rating = rating;
  setFilms(films);
}
```

Calling setter with same object may not trigger a re-render since React doesn't think anything changed

```
films.sort(...);
return (<FilmTable films={films} … />);
```

Sorts in place, so React may not know that FilmTable's props have changed, and thus not re-render

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.

[click]

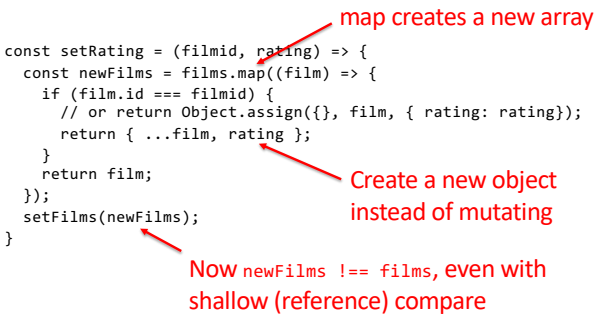
What about the lower snippet? sort is in place. If FilmTable compares its new props to previous props it may think nothing as changed and thus not re-reender.

[click]

In short, we don't want to mutate props or state objects.

https://react.dev/learn/updating-objects-in-state
https://react.dev/learn/updating-arrays-in-state

Make copies instead of mutating state or props



Instead, we make copies. Here we are making a copy of the films array with map. Further we making a copy of the specific object we are modifying. As a result, everything that has changed, the array and the modified film, point to new locations in memory.

To make a copy of the object, we are using the spread operator. The spread operator (the ellipses) works by populating the new object literal with all the properties of the film object and then overwrites that with rating (this concise syntax is short for `rating: rating`). The comment shows how to do the same with Object.assign.

[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.]

Wait, wait I hear you saying. Isn't this inefficient (and verbose/awkward)? Yes, but it may not matter. First, and most importantly, we don't want to start optimizing unless we know something is a problem. In many cases, it won't matter. For us, updating the screen is much more expensive that manipulating objects; minimizing/optimizing re-renders can be more important. If we do observe performance problems, we can look towards caching techniques (e.g., useMemo hook) or immutable data structures to speedup and simplify updates for complex objects.

Take home message: Don't mutate state or props, create new objects

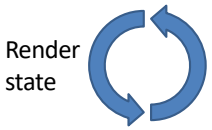
- Mutated props/state will not compare as different objects and so may not trigger a re-render
- Assigning to state does not trigger a re-render

```
// Wrong (in a class component)
this.state.comment = 'Hello';

// Correct (in a class component)
this.setState({ comment: 'Hello' });

// Hooks prevents the above error (but not calling setComment
// with the same object)
const [comment, setComment] = useState('');
comment = 'Hello'; // Javascript error
```

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

Review: React controlled components




```
function Example(props) {
  const [title, setTitle] = useState('');

  return (<input
    type="text"
    value={title}
    onChange={(event) => setTitle(event.target.value})}
  />);
}
```

Input value determined by React state

Change updates state, which re-renders input with new value

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. Doing so makes the component “predictable”, we know it will always show the state we specified and enables us to access those values for validation and other uses.

React: Controlled vs. Uncontrolled

+ Single source of truth
- Lots of callbacks

(Familiar?) Controlled component:

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

Uncontrolled component:

Reference to real DOM element

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

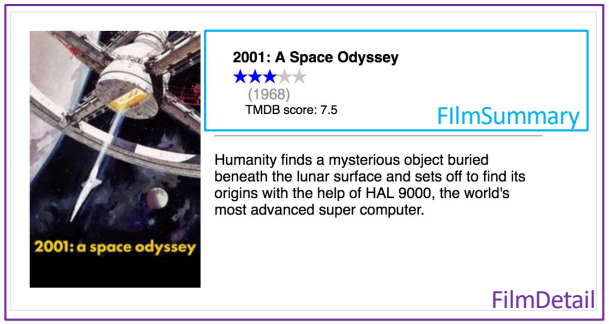
Feature	Controlled	Uncontrolled
One–time retrieval, e.g., on submit	✓	✓
Validating on submit	✓	✓
Instant validation	✓	✗
Conditionally disabling submit	✓	✗
Several inputs for one piece of data	✓	✗
Dynamically modify data (e.g., capitalize)	✓	✗
<input type="file" />	✗	✓

The ”con” for controlled components is lots of callbacks because we need to implement onChange and other handlers 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/

React: Composition vs. Inheritance?



If implemented as classes, should FilmDetail inherit from FilmSummary or contain a FilmSummary?

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 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 a more formal mechanism for thinking about inheritance vs. composition.

Formalizing subtyping: Liskov Substitution Principle

Let φ(x) be a property provable about objects *x* of type *T*. Then φ(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 *t*ype *T*, should also work on any subtype of *T*