Building User Interfaces

React 4

Advanced Concepts

Professor Bilge Mutlu

What we will learn today?

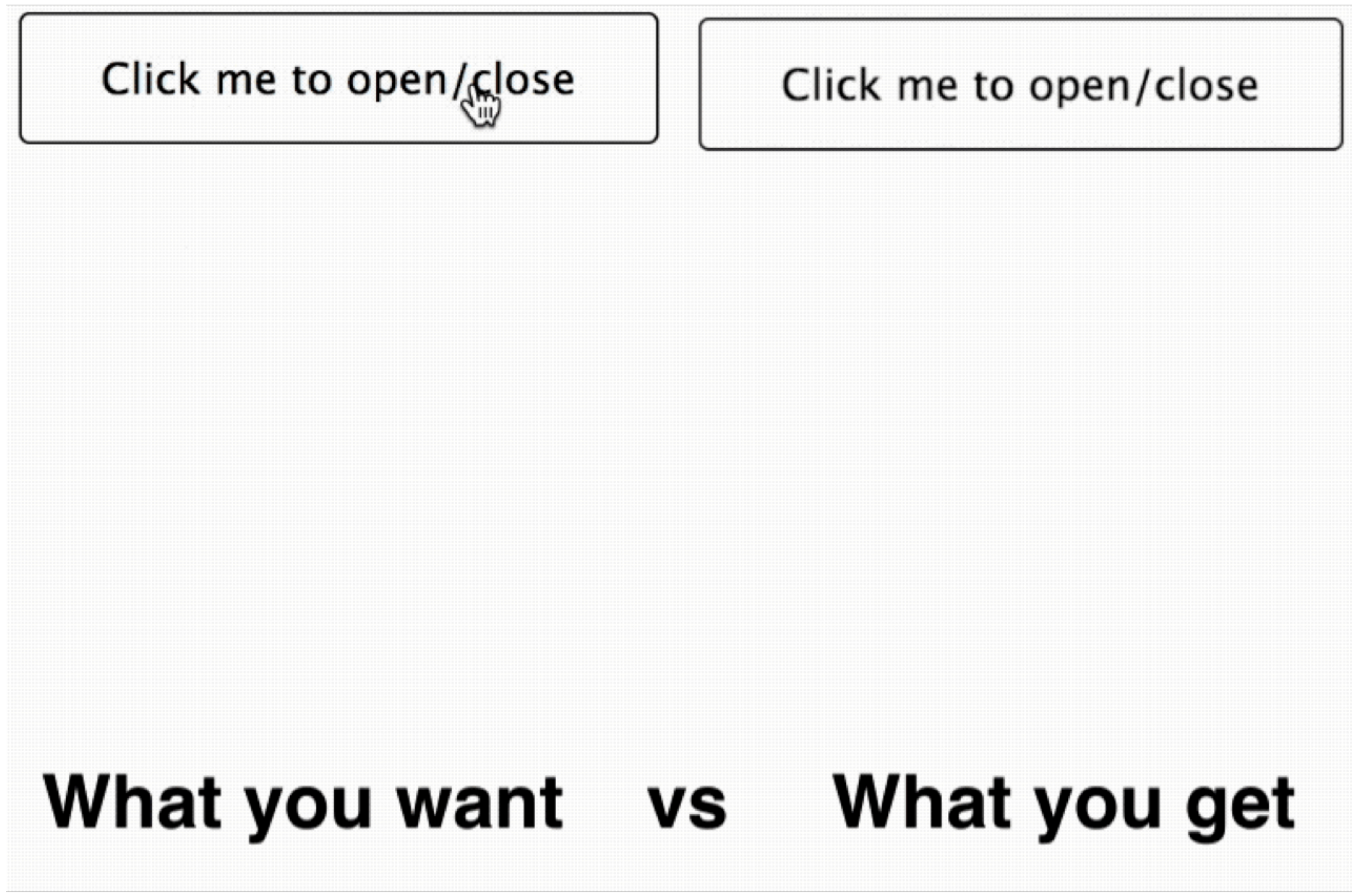
- Optimizing performance in React
- Advanced asynchronous updating
- APIs for advanced interaction
- Assignment Preview

Optimizing *Performance* in React

Why do we need to worry about performance?[1]

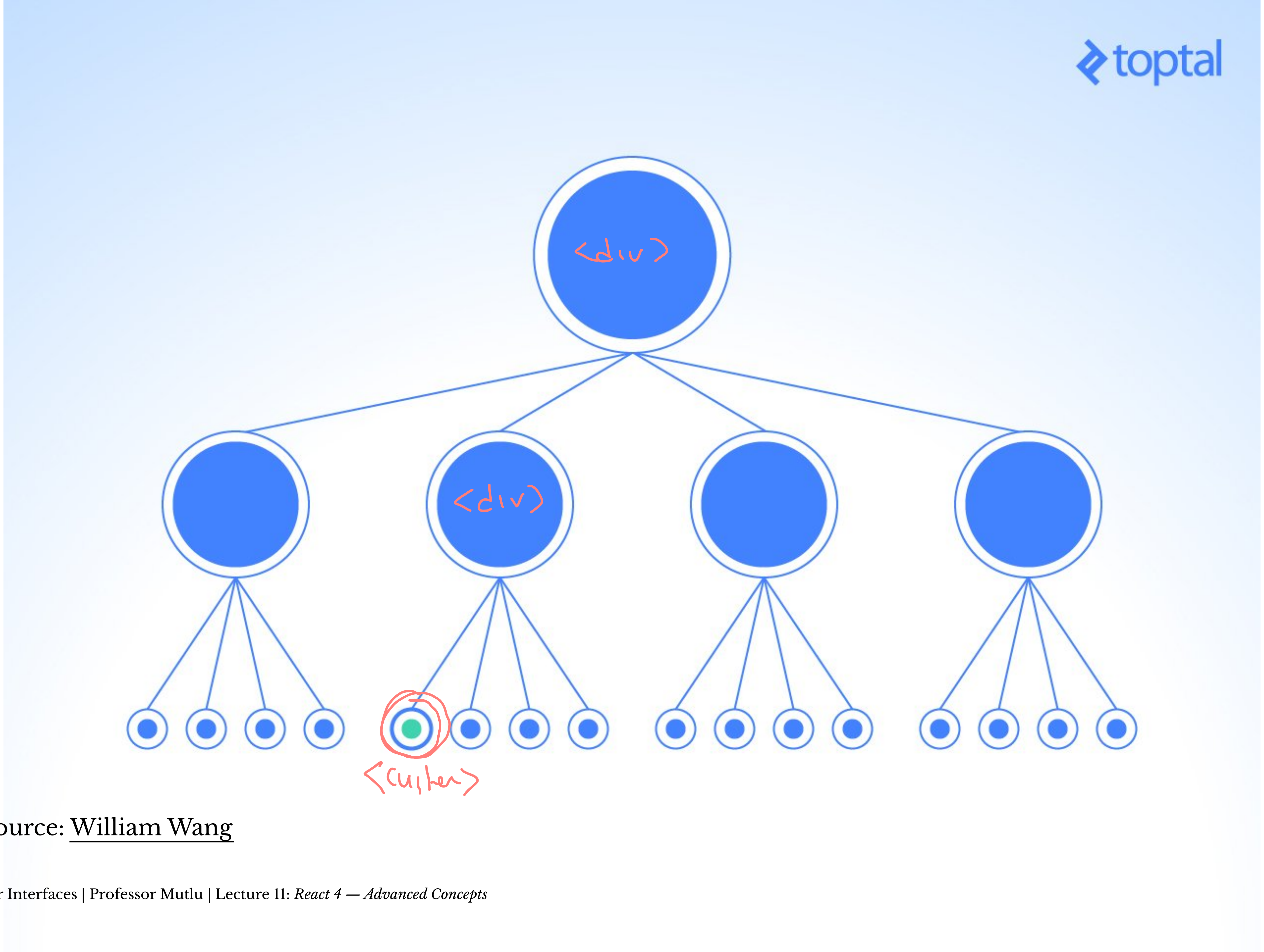
As the complexity of your application scales, performance will necessarily degrade.

Why? And what do we do about it?



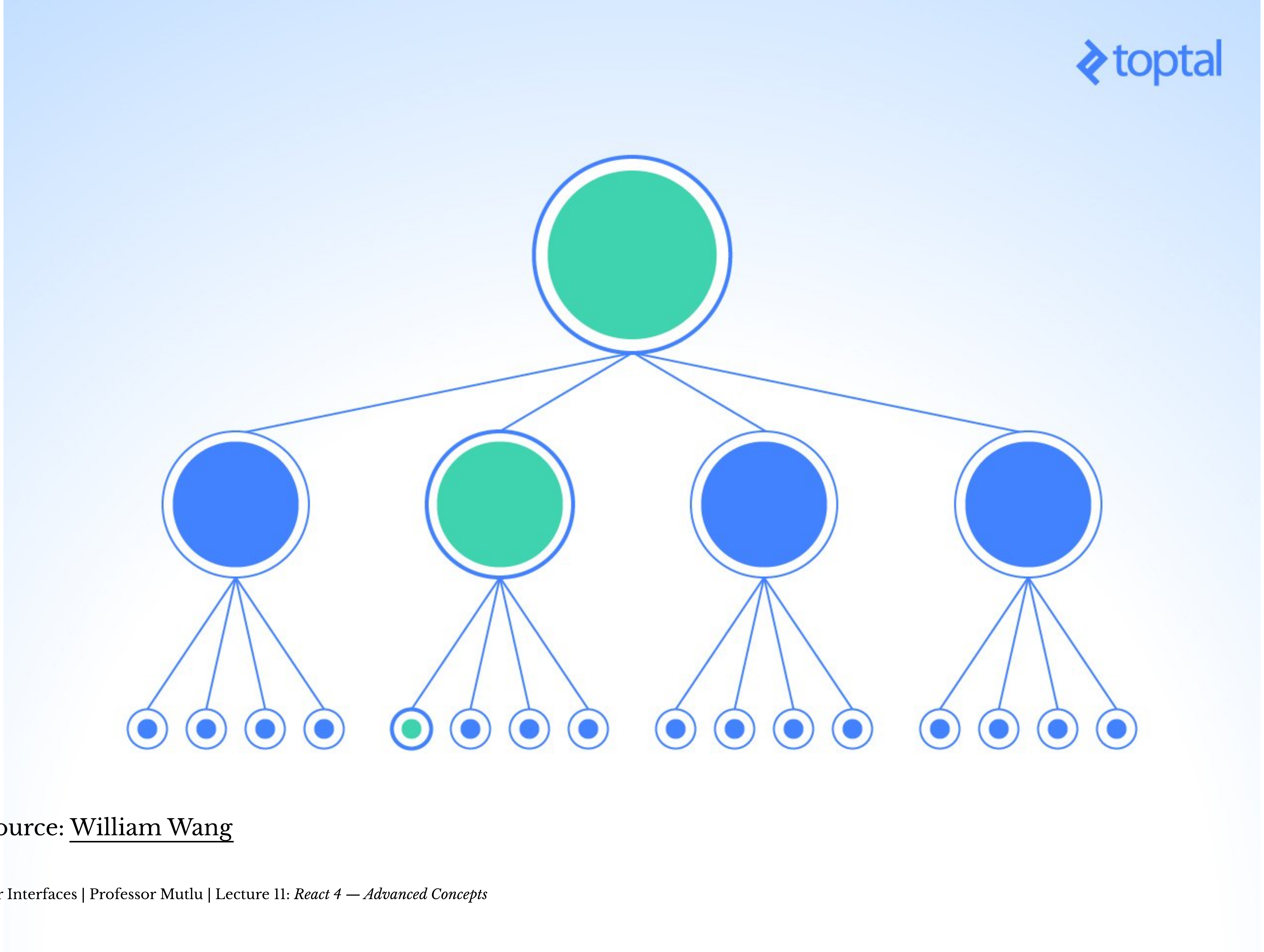
[1] Image Source: Noam Elboim

2



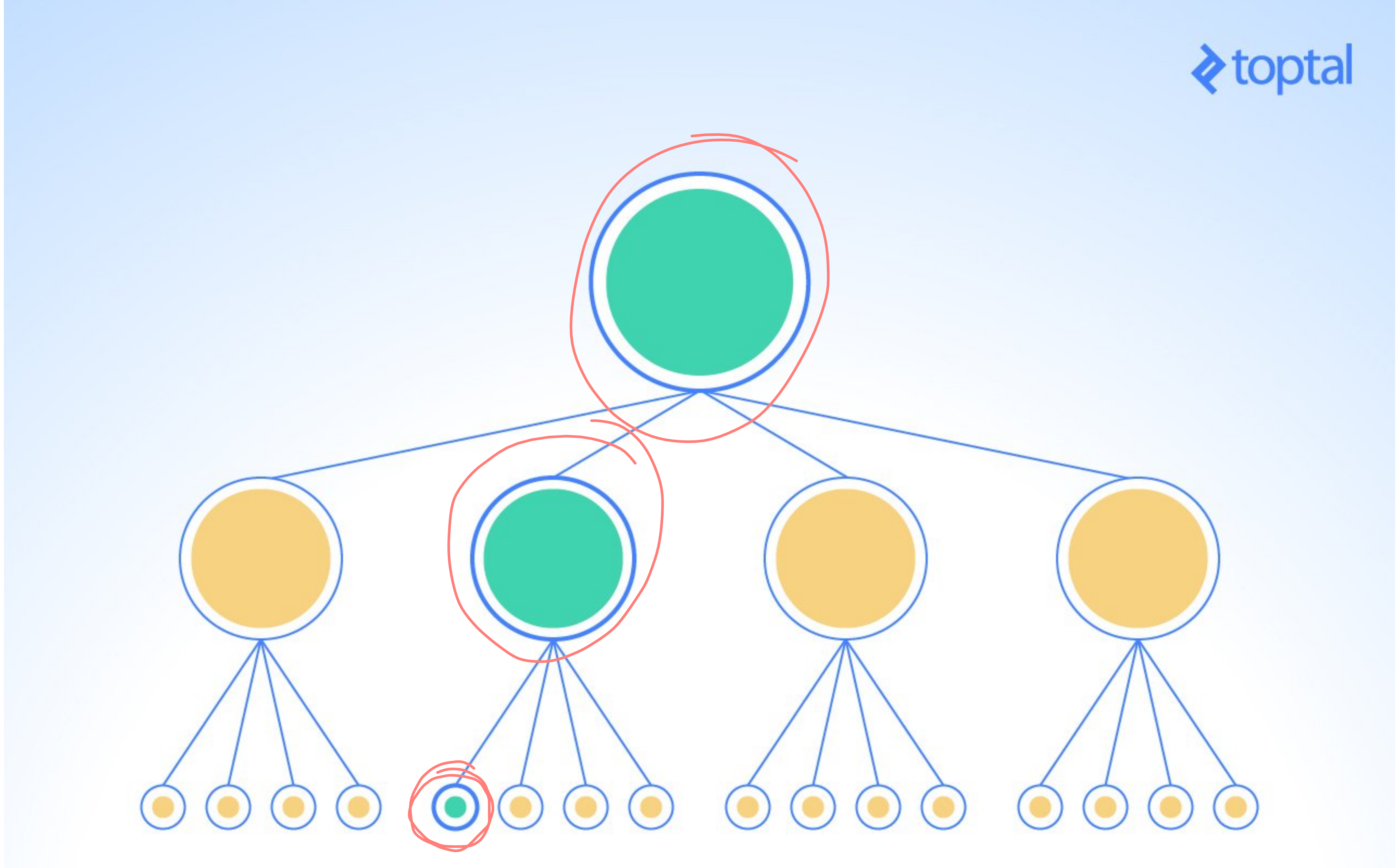
[2] Image Source: William Wang

2



[2] Image Source: William Wang

2

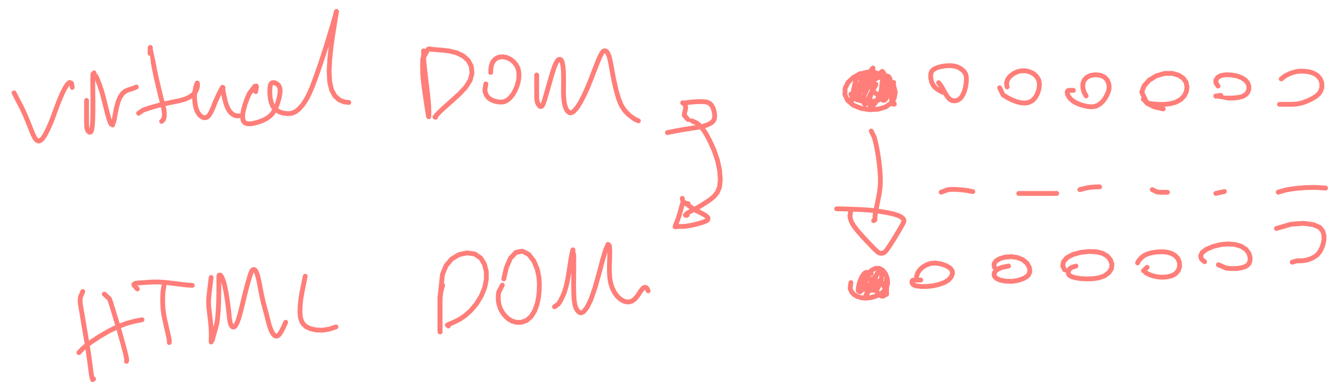


[2] Image Source: William Wang

Why does React do that?

That's how React works!

We discussed in React 1 that the diffing within Virtual DOM—*reconciliation*—is what makes it fast, but when things are scaled up, continuous diffing and updating affects performance.



How do we know that?

Performance tools: React provides a powerful library, `react-addons-perf`,[3] for taking performance measurements.

```
import Perf from 'react-addons-perf';
Perf.start()
// Our app
Perf.stop()
```

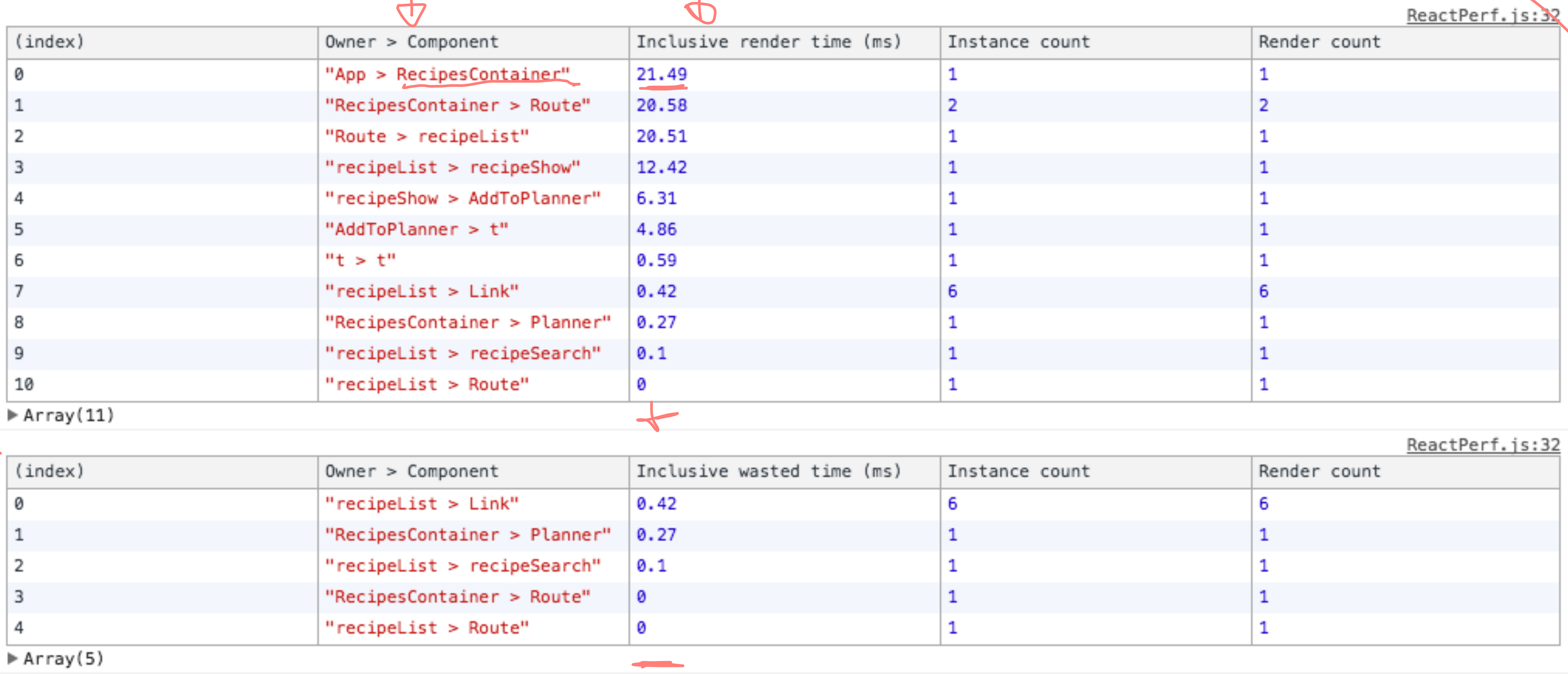
[3] ReactJS.org: Performance tools

Useful `Perf` methods

- `Perf.printInclusive()` prints overall time taken.
- `Perf.printExclusive()` prints time minus mounting.
- `Perf.printWasted()` prints time *wasted* on components that didn't actually render anything.
- `Perf.printOperations()` prints all DOM manipulations.
- `Perf.getLastMeasurements()` prints the measurement from the last `Perf` session.

`Perf.printInclusive()` and `Perf.printWasted()` output:[4]

owners


ReactPerf.js:32

(index)	Owner > Component	Inclusive render time (ms)	Instance count	Render count
0	"App > RecipesContainer"	21.49	1	1
1	"RecipesContainer > Route"	20.58	2	2
2	"Route > recipeList"	20.51	1	1
3	"recipeList > recipeShow"	12.42	1	1
4	"recipeShow > AddToPlanner"	6.31	1	1
5	"AddToPlanner > t"	4.86	1	1
6	"t > t"	0.59	1	1
7	"recipeList > Link"	0.42	6	6
8	"RecipesContainer > Planner"	0.27	1	1
9	"recipeList > recipeSearch"	0.1	1	1
10	"recipeList > Route"	0	1	1

▶ Array(11)

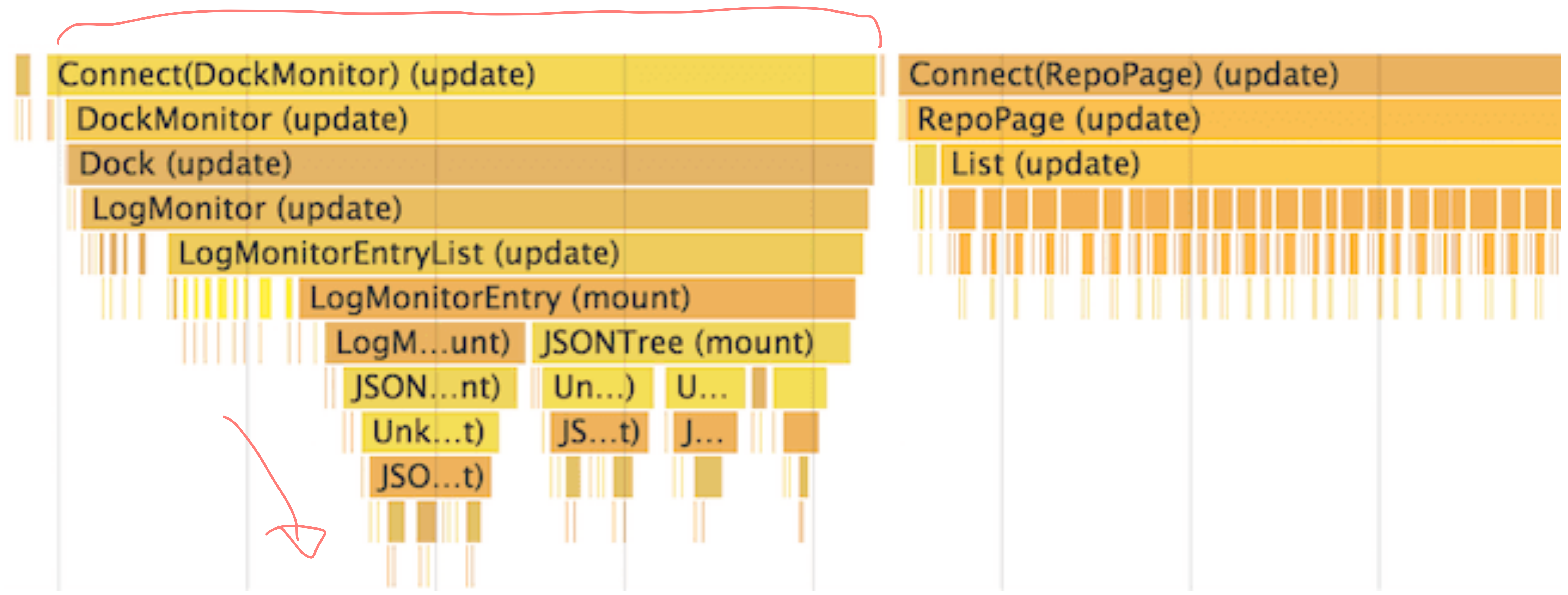
ReactPerf.js:32

(index)	Owner > Component	Inclusive wasted time (ms)	Instance count	Render count
0	"recipeList > Link"	0.42	6	6
1	"RecipesContainer > Planner"	0.27	1	1
2	"recipeList > recipeSearch"	0.1	1	1
3	"RecipesContainer > Route"	0	1	1
4	"recipeList > Route"	0	1	1

▶ Array(5)

[4] Image Source: Daniel Park

We can also visualize the performance of all components:[5] [6]



[5] An advanced guide to profiling performance using Chrome Devtools

[6] Image source

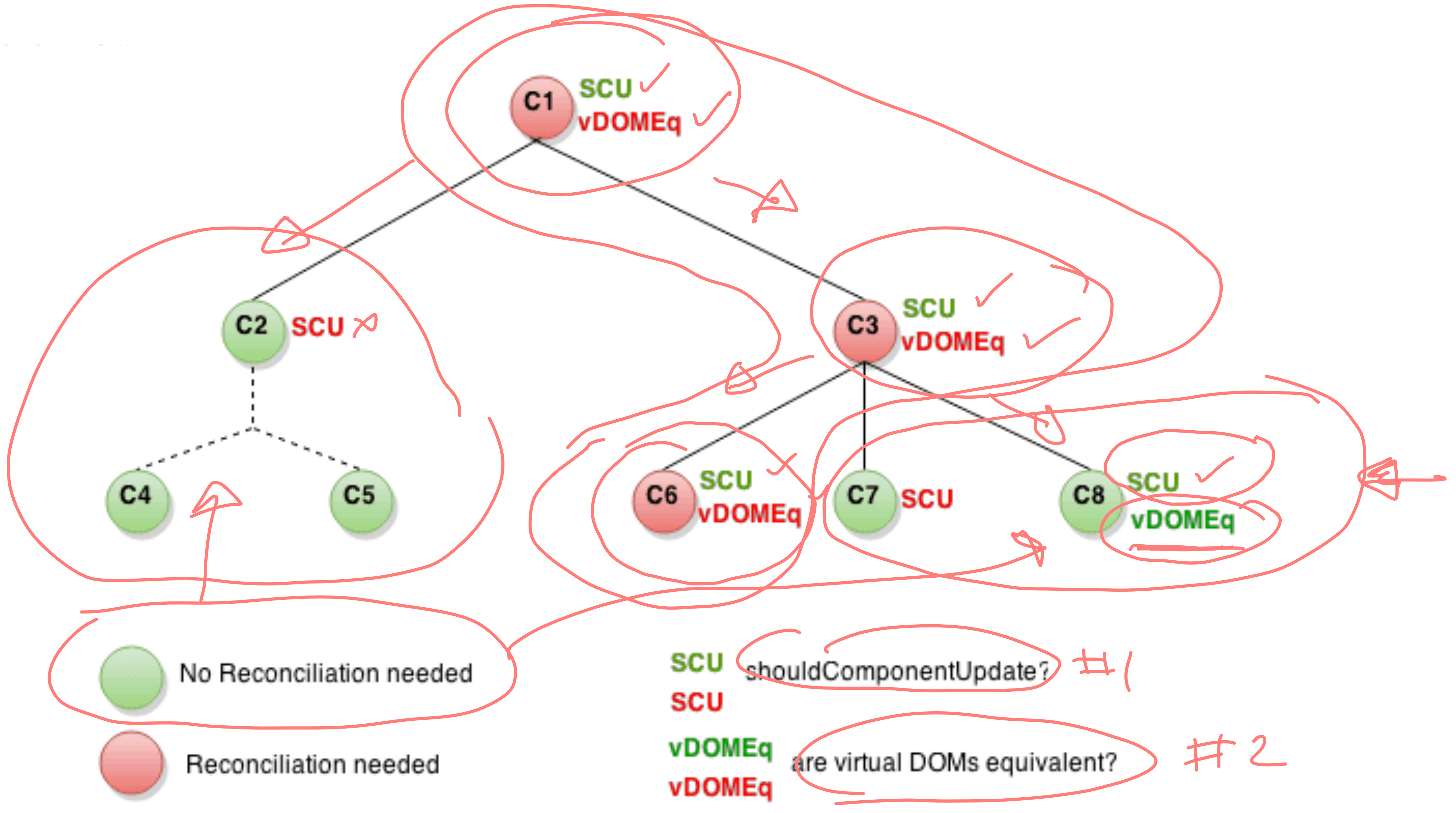
How to eliminate time wasted?

By avoiding reconciliation, i.e., only rendering when there is actually an update, using `shouldComponentUpdate()`.

Definition: For components that implement `shouldComponentUpdate()`, React will only render if it returns `true`.

```
function shouldComponentUpdate(nextProps, nextState) {
    return true;
}
```

7



[7] Image source

An example of *shallow* comparison to determine whether the component should update:

```
shouldComponentUpdate(nextProps, nextState) {
    return this.props.color !== nextProps.color;
}
```

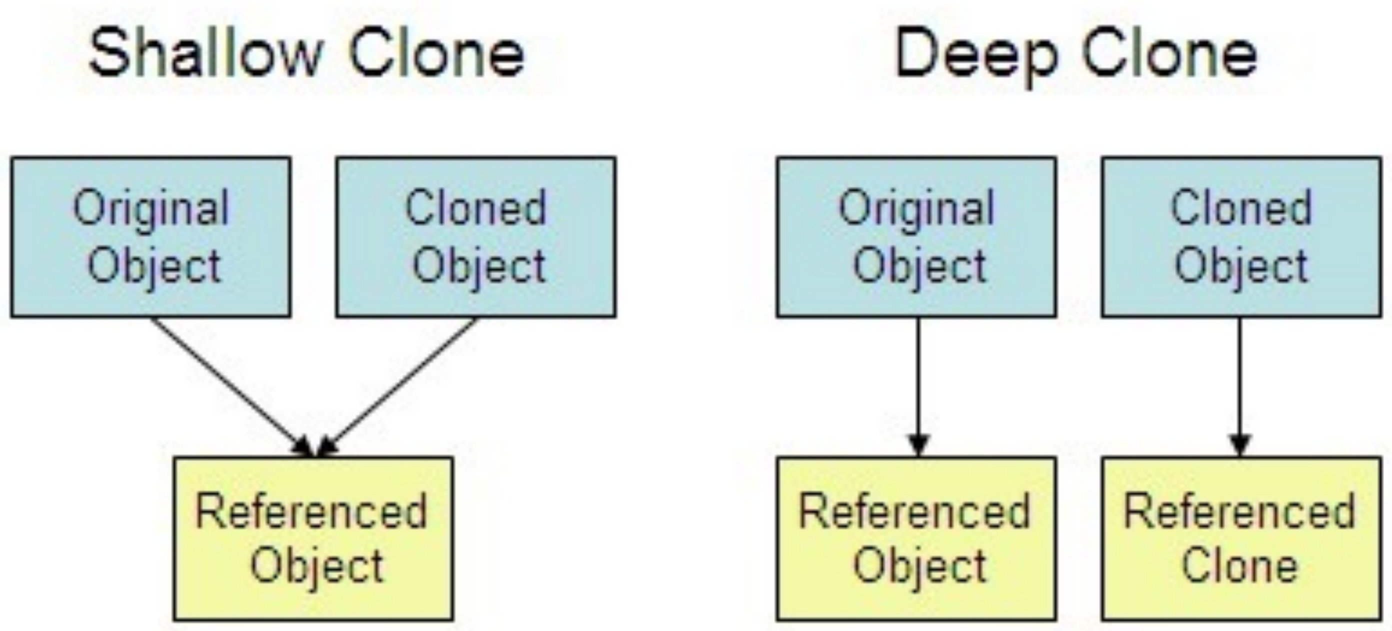
Let's see an example from ReactJS.org...

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

Detour: Shallow vs. Deep Comparison[8]

Shallow Comparison: When each property in a pair of objects are compared using *strict* equality, e.g., using ===.

Deep Comparison: When the properties of two objects are recursively compared, e.g., using Lodash `isEqual()`.



[8] Image source

`React.PureComponent`

React provides a component called `PureComponent` that implements `shouldComponentUpdate()` and only diffs and updates when it returns `true`.

Note that any child of `PureComponent` must be a `PureComponent`.

Let's see an example from ReactJS.org...

```
class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color} #1
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count} #2
      </button>
    );
  }
}
```

Other Ways of Optimizing Performance

- Not mutating objects (see *The Power of Not Mutating Data*, Immer, `immutability-helper`)
- Using immutable data structures (see more on data immutability)
- Using the `production` build of React
- Many more...

Further Reading on React Performance

— 21 Performance Optimizations for React Apps ✓
— Efficient React Components: A Guide to Optimizing React Performance ✓
— ReactJS.org: Optimizing Performance

Quiz 1

Complete the Canvas quiz.

Quiz 2

Complete the Canvas quiz.

Advanced *Asynchronous* Updating

Getting data within `componentDidMount()`

Ideally, we want to interact with the server in the following way. What would happen here?

```
componentDidMount() {
  const res = fetch('https://example.com')
  const something = res.json()
  this.setState({something})
}
```

But we end up following up `fetch()` with a series of `then()`s.

```
componentDidMount() {
  fetch('https://example.com')
    .then((res) => res.json())
    .then((something) => this.setState({something}))
}
```

`then()` allows us to program asynchronously (by allowing `componentDidMount()` to wait for the `Promise` to be resolved). Although, this syntax can be unintuitive and not readable.

Programming asynchronously using `async/await`

`async/await` provides syntax to program asynchronously in an intuitive and clean way.

Usage:

— `async function()` denotes that the `function()` will work asynchronously.

— `await expression` enables the program to wait for `expression` to be resolved.

Example:[9]

```
async componentDidMount() {
  const res = await fetch('https://example.com')
  const something = await res.json()
  this.setState({something})
}
```

[9] See in CodePen

async Functions[10]

Any function can be asynchronous and use async. Useful where the function has to wait for another process.

```
async addTag(name) {
    if(this.state.tags.indexOf(name) === -1) {
        await this.setState({tags: [...this.state.tags, name]});
        this.setCourses();
    }
}
```

[10] See example in CodePen

Quiz 3

Complete the Canvas quiz.

APIs for advanced interaction

Interaction Libraries

- react-beautiful-dnd: Examples
- react-smooth-dnd: Demo
- React DnD: Examples

Component Libraries

- Material UI
- Material Kit React: Demo
- Rebass
- Grommet
- React Desktop : Demo

Managing Data

- React Virtualized: Demo

Assignment Preview

React 3

Deliverable options for React 2 α:

A course recommender application

Problem 1: Definition

Load in a json file of previous courses located. The user should be able to view the contents at this url as courses that the user has previously taken.

Problem 1: Suggested Workflow

1. Fetch data from server
2. Create a new component to represent previously taken courses. This component will look somewhat like the Course component, but it will be simpler and won't have options to add the course to the cart.
3. Create a new component to hold the previously taken courses. Make this component accessible from the app (maybe another tab on the top or a tab within the cart page).

Problem 2

Create a rating system for previously taken courses. The user should be able to rate some or all of the previously taken courses loaded from the `json` file.

Problem 2: Suggested Workflow

1. Create a component for rating a specific course.
2. Create a component for holding all of the rating components.
3. Make the holder accessible from the search tab.
4. Save the data from the holder in the state of the lowest ancestor of any component that will need it

Problem 3

Create a way for the user to select areas of interest that you define. These areas can be general or specific. Some examples might be `computer science`, `artificial intelligence`, or `science`.

Problem 3: Suggested Workflow

1. Generate a list of interest areas based on the course data.
2. Create a component for the user to select interest areas as defined in step 1.
3. Make this component accessible from the search tab.

Problem 4

Recommend courses to the user based off of the rated previously taken courses and the user's specified interest areas.

Problem 4: Suggested Workflow

1. Create the recommender algorithm that takes in the rated courses and interest areas, searches through subjects and keywords, and returns courses most similar to the highly rated courses and the courses that match the most interest areas.
2. Display the recommended courses to the user.

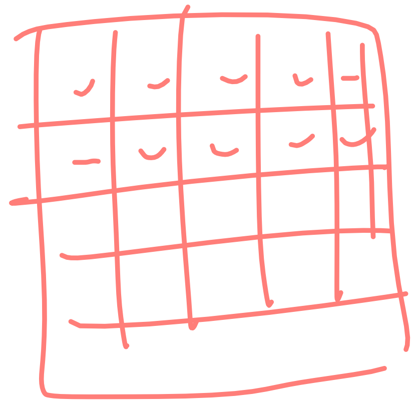
Implementations will be evaluated for:

— In α: efficiency, elegance, clean, and readable

— In β: usability, visual design, navigation model

My Simple Recommender

My simple recommender, *constraints*



Requirements:

- Users are given a set of elements to evaluate <
- Evaluations are standardized into a ~~ranking~~ rating scheme <
- The ranking scheme is used to look up matches <
- Top match is returned as a recommendation <

My simple recommender, *design decisions*

Design decisions:

- Give the user a *randomly generated* set of swatches
- Allow users to provide *like/dislike* ratings
- *Average out* the colors of liked swatches
- Give the user a recommend swatch *with the average*

Italics indicate the simplest possible implementation.

My simple recommender, *component structure*

```
<App />
    <Swatch />  — Com
    ...
    <RecommendedSwatch />  Rec Com
```

Plus, possibly a function component for ranking.

A few pieces of advice

- Start early
- Google (or Bing, DuckDuckGo, etc.) is your friend
  - E.g., even if we cover correct syntax in class, slides are not useful for debugging
- Use debugging tools
  - Compiler errors, React Development Tools, `console.log()`
- Come to office hours (early)

What did we learn today?

— Optimizing performance in React

— Advanced asynchronous updating

— APIs for advanced interaction

— Assignment Preview