Building User Interfaces

React 4

Advanced Concepts

Professor Yuhang Zhao

What we will learn today?

- Introducing Hooks
- Optimizing performance in React
- Advanced asynchronous updating
- APIs for advanced interaction

Introducing Hooks

Why do we use Hook?

- Classes could be confusing
- Complex class components are hard to understand
- It's hard to reuse stateful logic between components

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

}
componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
}
...
```

What is a Hook?

Definitions: Hooks are functions that let you "hook into" React state and lifycycle features from function components. Hooks don't work inside classes — they let you use React without classes.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
const element = <Welcome name="Professor Zhao" />;
ReactDOM.render(element, document.getElementById('root'));
```

How to use Hooks?

- Build-in Hooks
  - useState, useEffect, etc.
- Custom Hooks

State Hook

`useState` is a Hook that lets you add React state to function components.

When should I use a State Hook?

If you need some state in a function componenet, you can use State Hook inside the function component!

How to use State Hook?

Import the `useState` Hook from React:

```
import {useState} from 'react';
```

Declare state:

```
const [count, setCount] = useState(0);
```

`useState()` takes in one argument: the intial state, and returns a pair of values: the current state, and a function that updates it.

Class component

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() =>
        this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

Function component with Hook

```
function Example() {
  // Declare a new state variable "count"
  const [count, setCount] = useState(0);


  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Declare multiple state variables

```
[something, setSomething] = useState(value)

function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  ...

  }
```

Effect Hook

`useEffect` lets you perform side effects in function components.

When should we use effect hook?

If you want to involve side effects (e.g., data fetching, setting up a subscription, manually changing the DOM) in a function component, you can use Effect Hook!

It's a combination of `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

How to use Effect Hook?

```
useEffect(function, [state]);
```

By using `useEffect`, you tell React that your component needs to do something after render. React will remember the function you passed in, and call it after performing the DOM updates (e.g., mounting, updating).

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
  }
);
```

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

Class using lifecycle methods

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() =>
        this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

Function with Effect Hook

```
function Example() {

  const [count, setCount] = useState(0);


  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });


  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Example in stackblitz.

Rules of Hooks

- Only call Hooks from React functions, not regular JavaScript functions
- Only call Hooks at the top level, not inside loops, conditions, or nested functions

More about Hooks

- Building your own Hook
- Hook API Reference

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({key: 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}))
    .catch((error) => console.error('Error:', error))
}
```

`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({key: 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();
    }
}
```

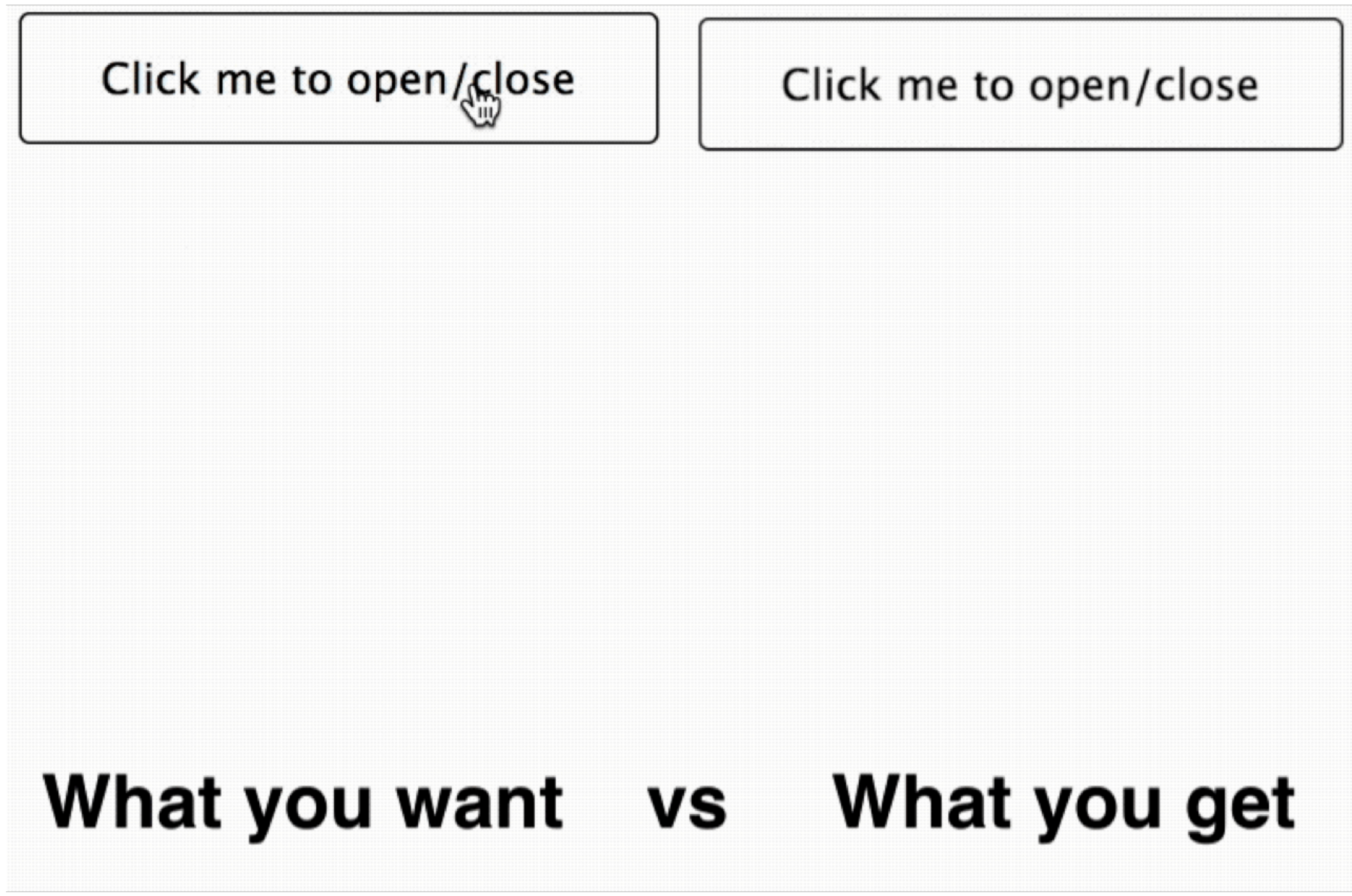
[10] See example in CodePen

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?



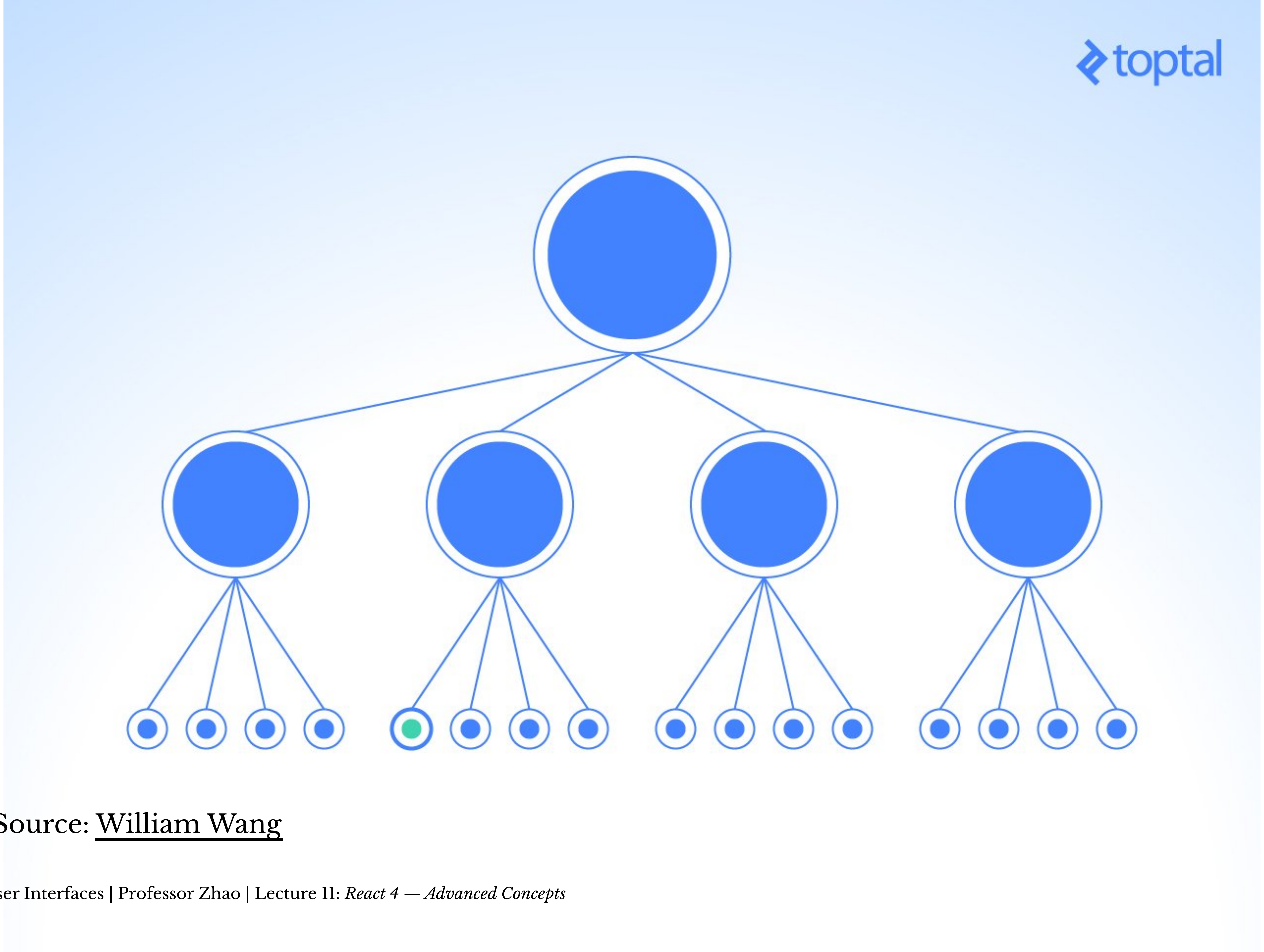
[1] Image Source: Noam Elboim

Why does React do that?

That's how React works!

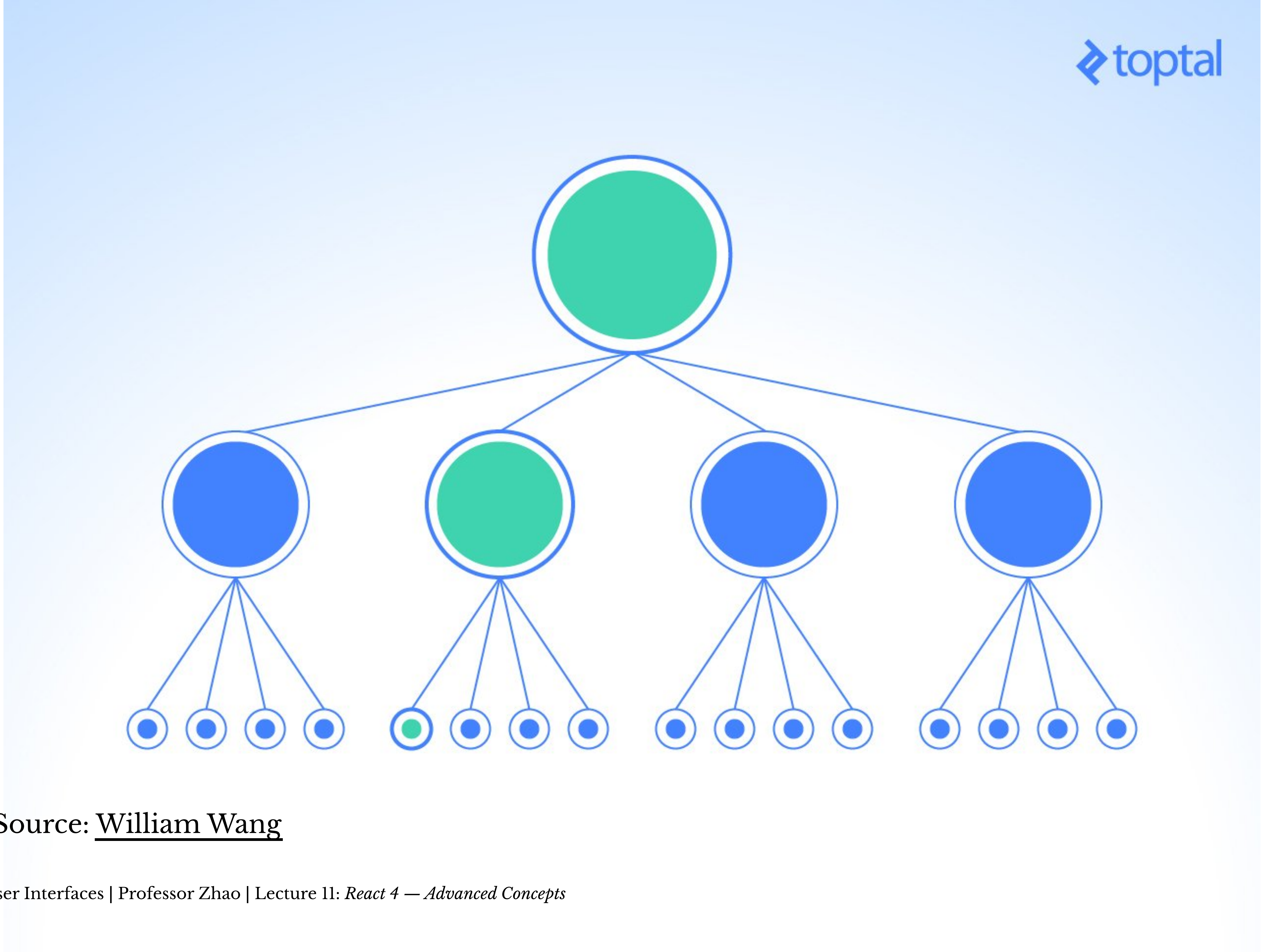
We discussed in React 1 that the diffing within Virtual DOM—*reconciliation*—is what makes it fast, but when things are scaled up, continuous diffing and updating affects performance.

2



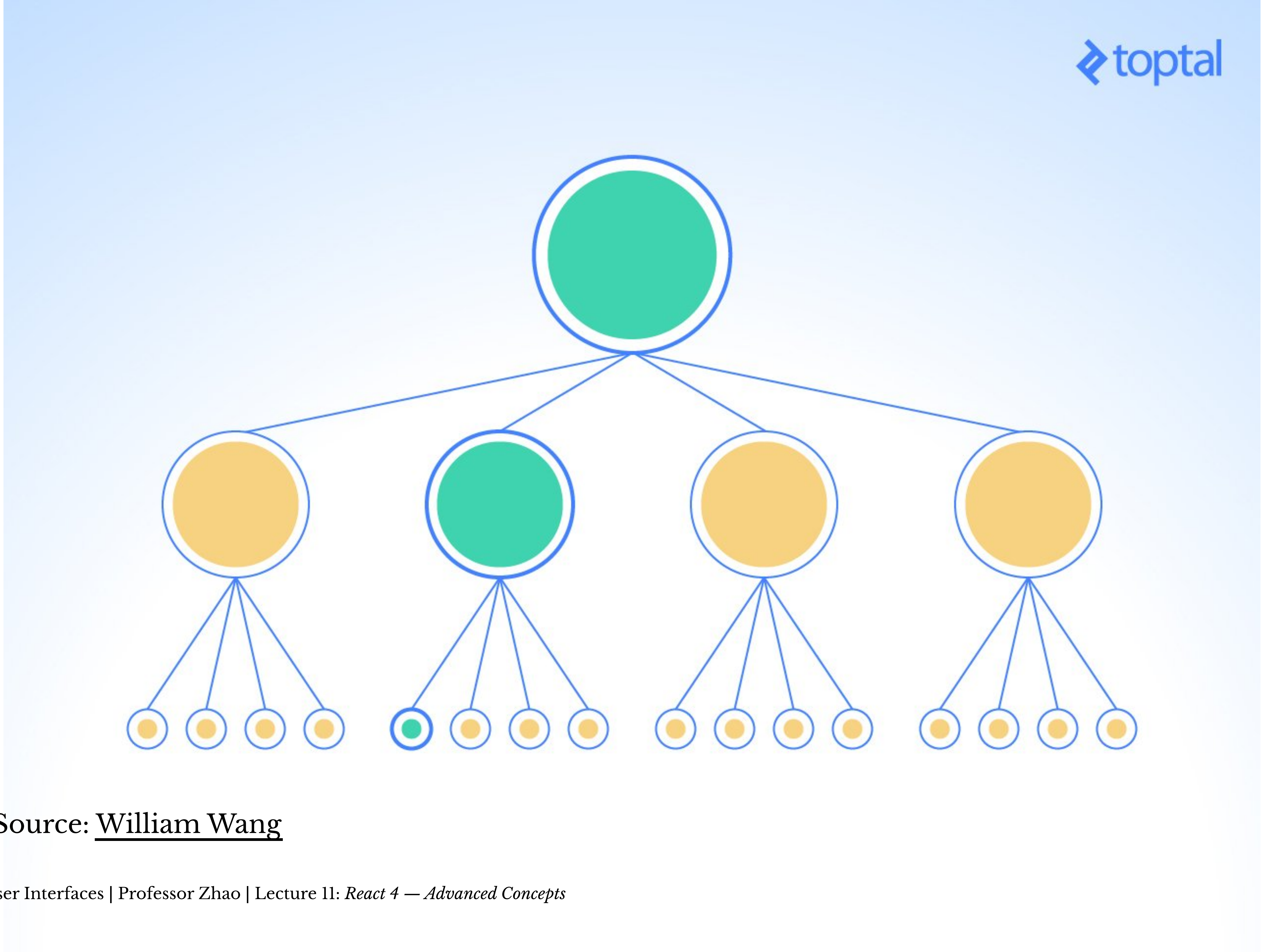
[2] Image Source: William Wang

2



[2] Image Source: William Wang

[2]



[2] Image Source: William Wang

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.

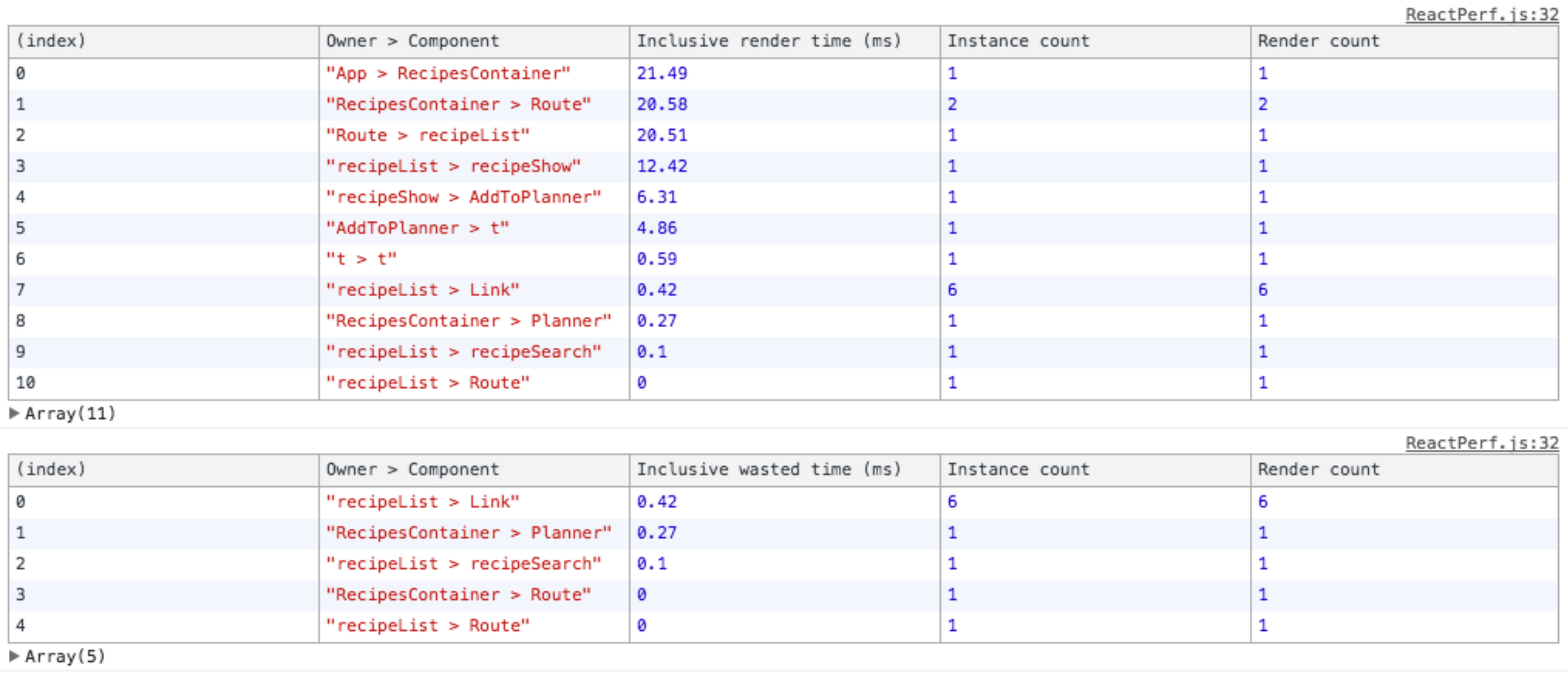
— `Perf.printExclusive()` prints time minus mounting.

— `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]



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)

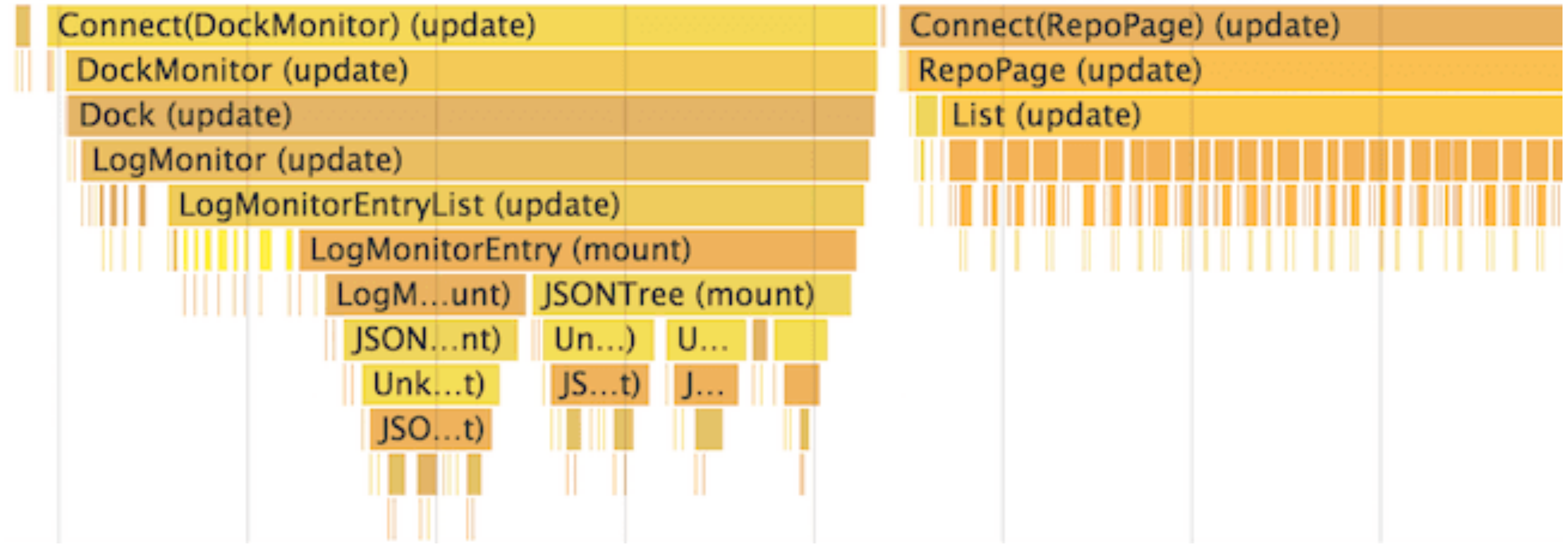
ReactPerf.js:32

(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

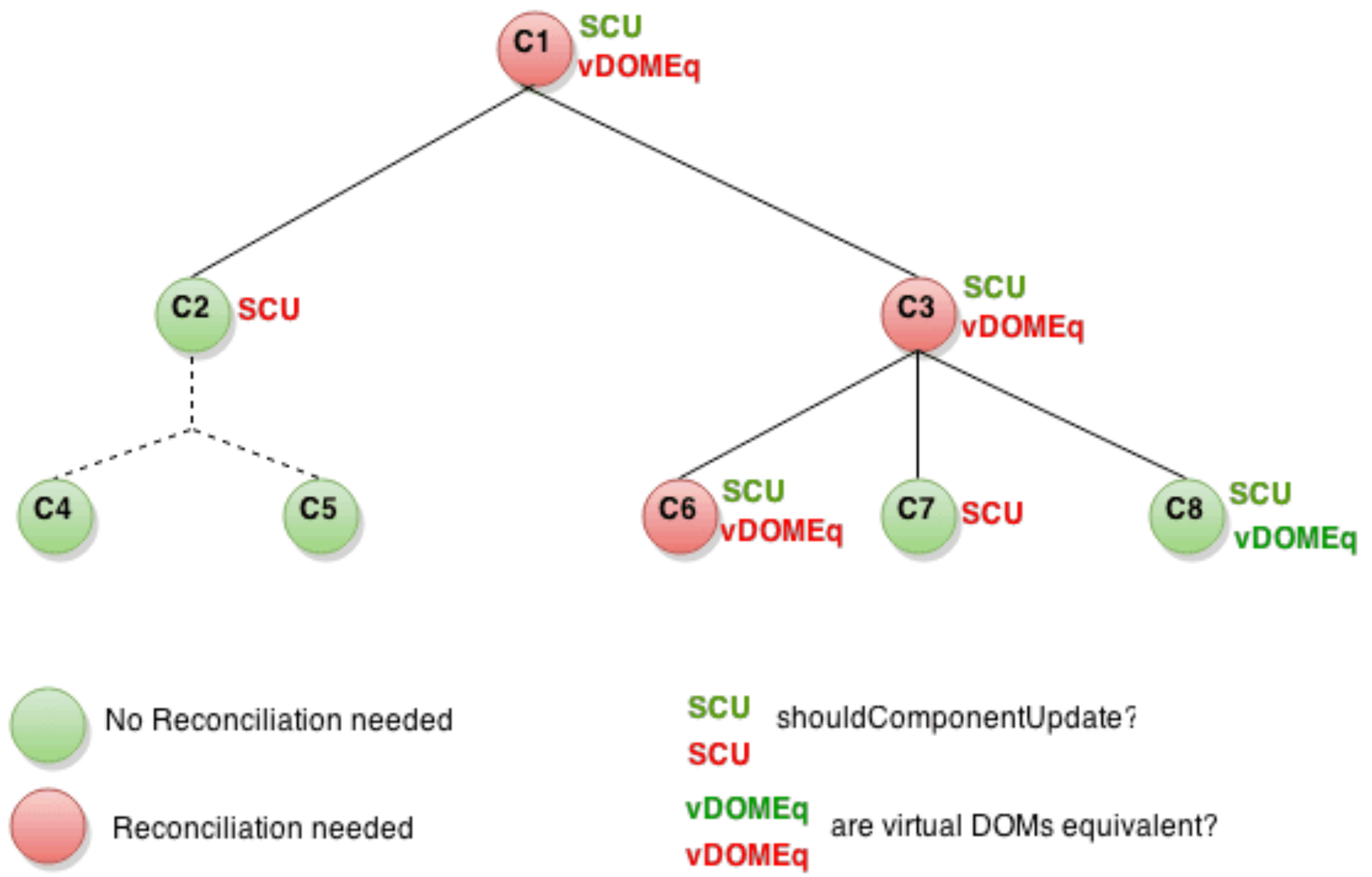
How to eliminate time wasted?

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;
}
```

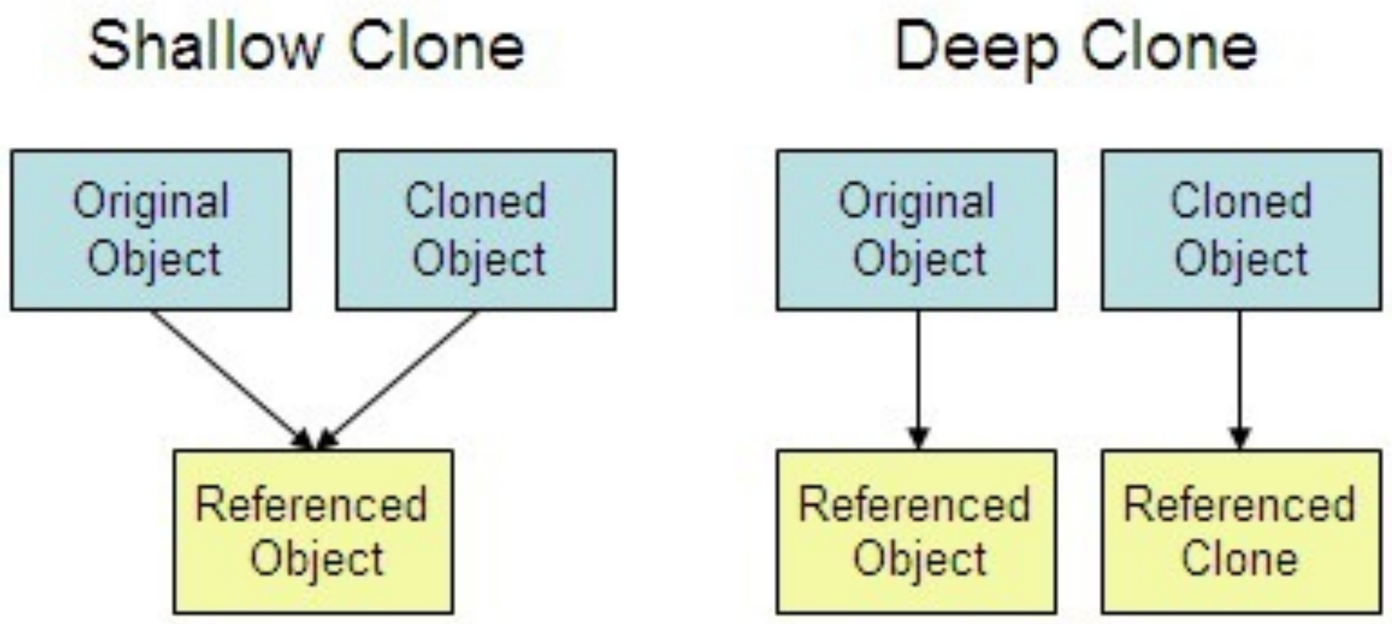
Let's see an example from ReactJS.org...

```
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}
        onClick={() => this.setState(state => ({count: state.count}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

Avoid mutating data

```
 handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }

handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}
```

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

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

A few pieces of advice for assignments

- 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?

- Introducing Hooks
- Optimizing performance in React
- Advanced asynchronous updating
- APIs for advanced interaction