React Lifecycles and Hooks

Ben Wilhelm
Nimble Moose, LLC

at Tandem
December 16, 2019

Hi 👋

Ben Wilhelm

Nimble Moose, LLC

ben.wilhelm@nimblemoose.com

Send me questions!

- 10 years as a software developer, technical lead, and manager
- Mostly with agencies about the size of Tandem
- Teaching JavaScript and React for the last 2 years

Today

- 10:00 - Arrive, settle, intros
- 10:15 - Lecture and Demo of all lifecycles
- 10:45 - Hands on pairing exercise applying lifecycles
- 12:15 - Lunch (60min)
- 1:15 - Lecture and Demo of React Hooks
- 2:00 - Hands on pairing exercise using existing hooks and building our own reusable hooks (will include short break)
- 4:00 - Short overview of Context API, followed by exercise
- 5:00 - End of Day

Who are you?

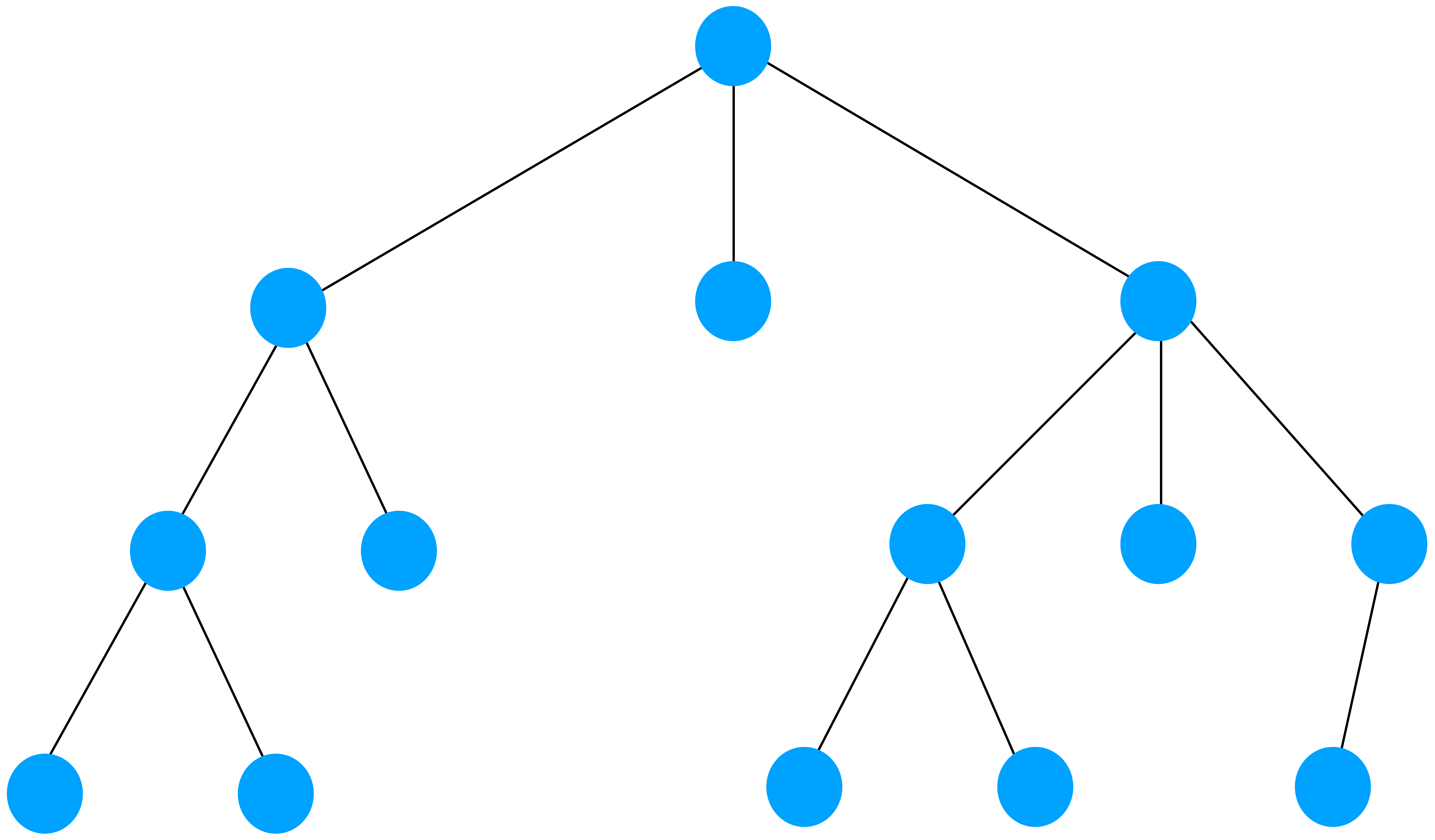
Why I Love React

- Lightweight tool
- Easy to grasp the basics, you can get productive very quickly
- It’s declarative!
- Its abstraction rarely leaks
- Excellent documentation and community

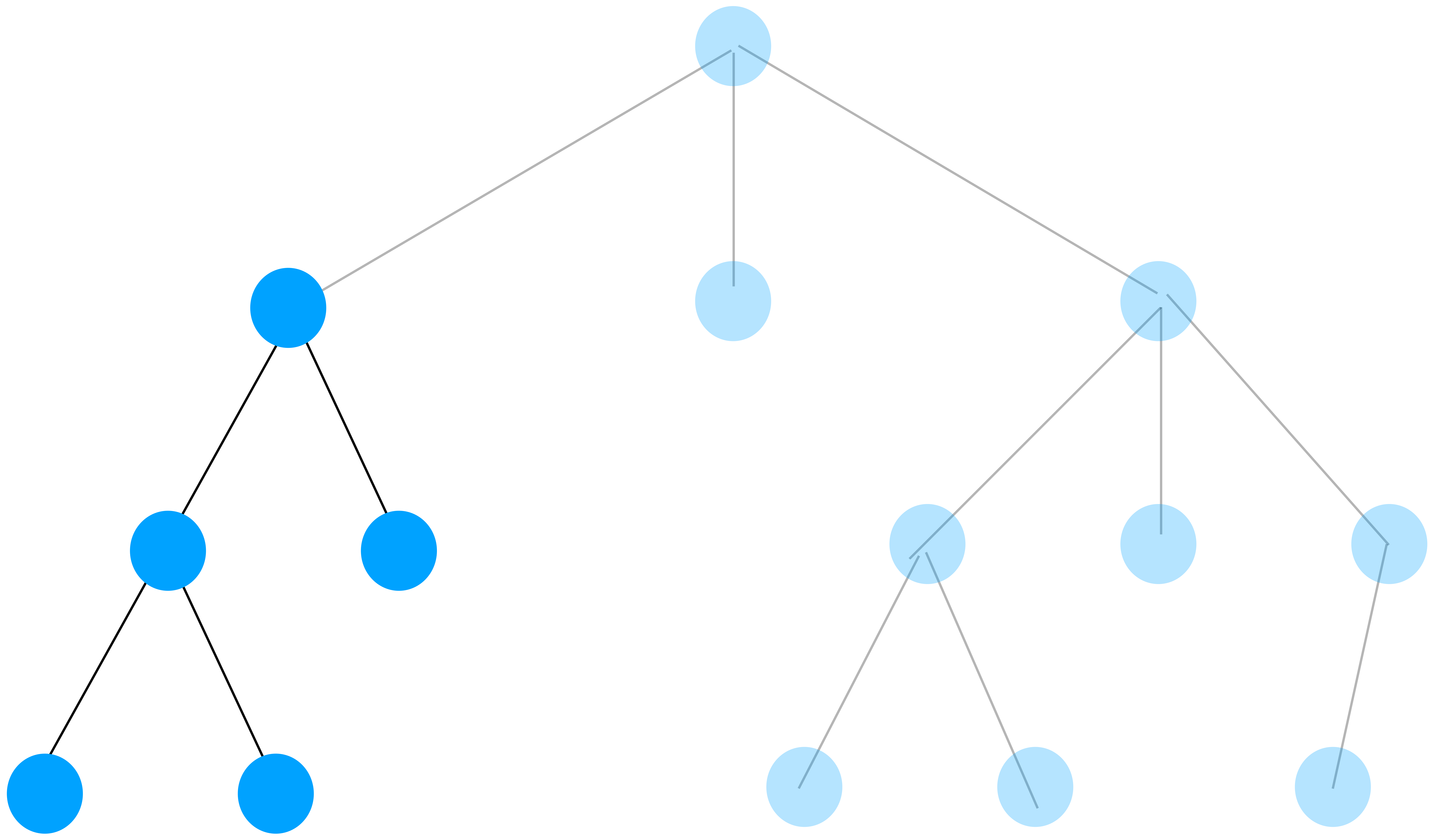
React is Declarative

What does that mean?

Trees!



Trees are Trees of Trees!



Reconciliation

The method by which React diffs the component tree and determines how to efficiently update the the UI

Two Assumptions

- Two elements of different types will produce different trees.
- The developer can hint at which child elements may be stable across different renders with a key prop.

reference: https://reactjs.org/docs/reconciliation.html

Assumption 1 Explained

If root elements are of same type:

Keep root element, update its attributes/props

```
<Game difficulty={HIGH} >
  <Counter count={1} />
</Game>
```

```
<Game difficulty={LOW} >
  <Counter count={2} />
</Game>
```

Components will be maintained, but props updated
True for both DOM elements and Component Elements

If root elements are of different type:

Rebuild tree from the ground up

```
<div>
  <Counter count={2} />
</div>
```

```
<p>
  <Counter count={2} />
</p>
```

Entire tree will be rebuilt.
<Counter> will be unmounted and remounted even though props haven’t changed

Assumption 2 Explained

Rendering a list is easy:

Simply iterate over the elements

```
<ul>
{things.map(thing => (
  <li>{thing.name}</li>
))}
</ul>
```

```
<ul>
  <li>Thing 1</li>
  <li>Thing 2</li>
</ul>
```

Appending to the list is inexpensive, but what
if you want to insert an element or sort the list?

But how do we persist elements across renders?

Use a `key` attribute

```
<ul>
{things.map(thing => (
  <li key={thing.id}>{thing.name}</li>
))}
</ul>
```

```
<ul>
  <li key="3">Thing 3</li>
  <li key="1">Thing 1</li>
  <li key="2">Thing 2</li>
</ul>
```

Now React will use existing elements where appropriate

Demo

The Lifecycle of a React Component

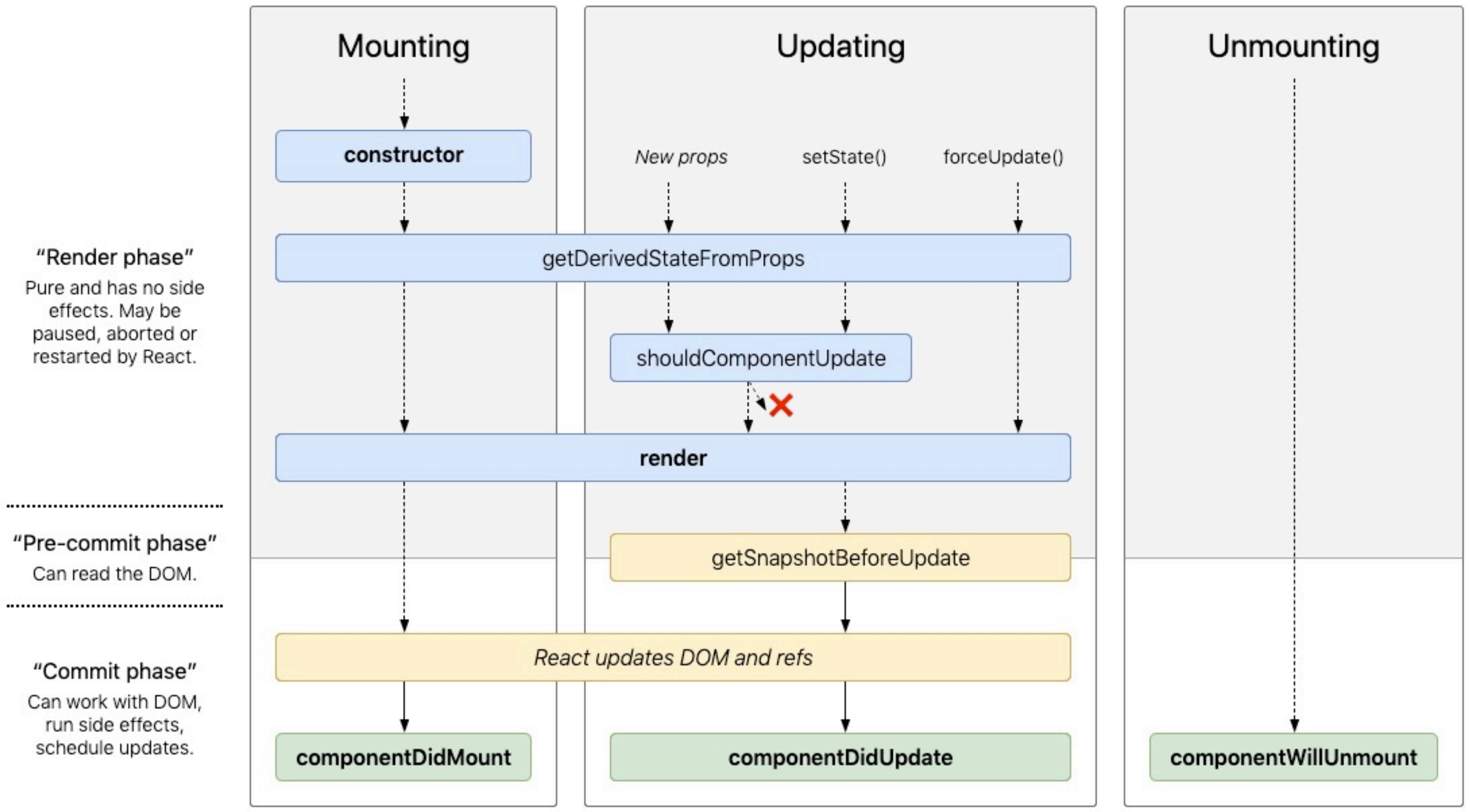


Image credit: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

Legacy Lifecycles

- UNSAFE_componentWillMount()
- UNSAFE_componentWillUpdate()
- UNSAFE_componentWillReceiveProps()

These lifecycles were deprecated in React 16.3
due to the forthcoming async rendering mode.

They will be removed completely in React 17

Pair Exercise!