

Introduction to

# React Hooks

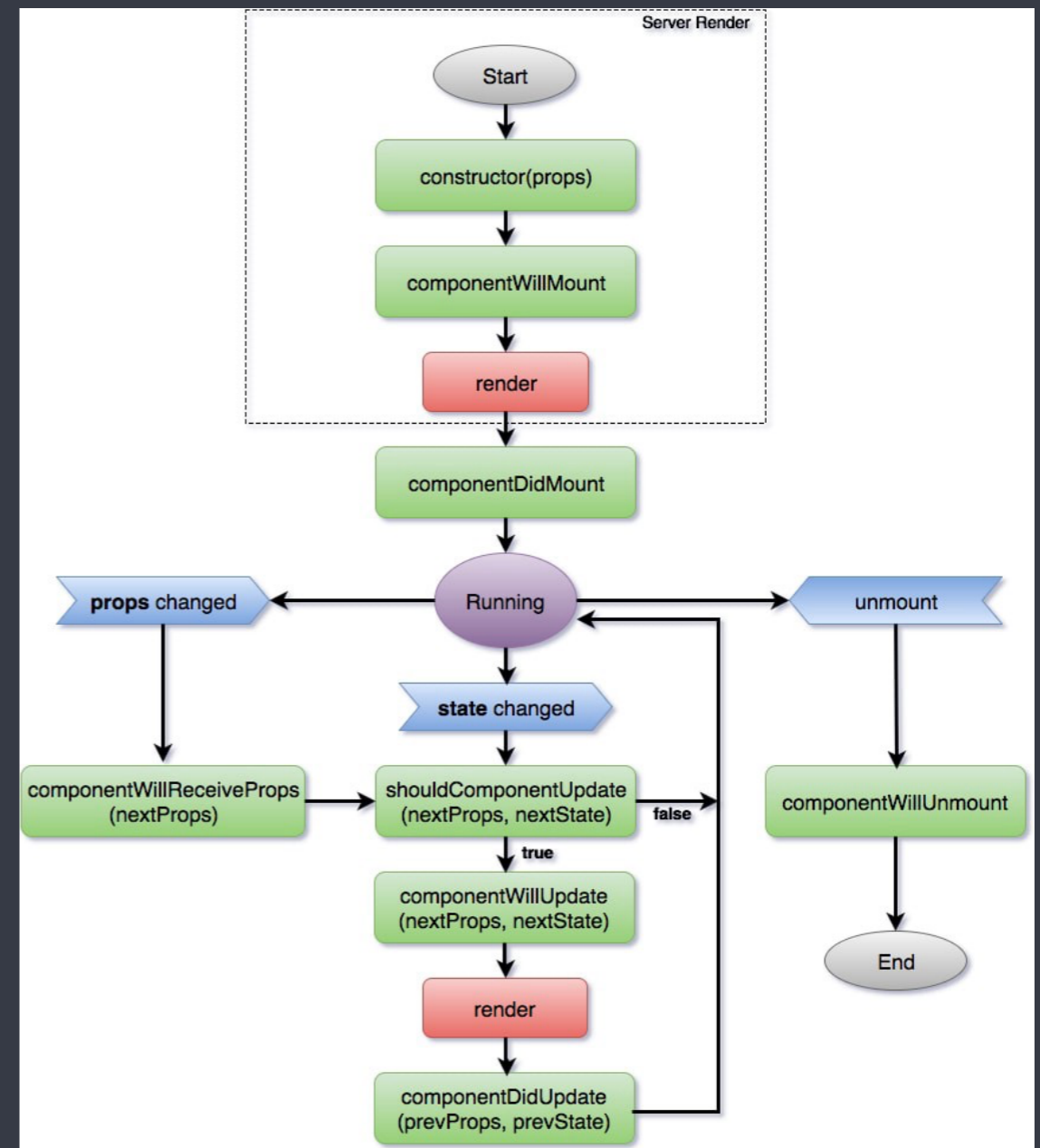
# The Problem

**You can't use React state without classes!**

*But there's more...*

# The Component Lifecycle is Complicated & Redundant

- On component initialization, there's `constructor`, `getDerivedStateFromProps`, `componentWillMount`, `render`, and `componentDidMount`.
- On update, there's `getDerivedStateFromProps` again, `componentWillReceiveProps`, `shouldComponentUpdate`, `getSnapshotBeforeUpdate`, `componentWillUpdate`, `render` again, and `componentDidUpdate`.
- There's also `componentWillUnmount`, `componentDidCatch`, and `getDerivedStateFromError`.
- And typically, the implementations for these methods are almost exactly the same.



# Component Logic Isn't Easily Reusable

- Often, we have code that interacts with the lifecycle and state that we'd like to use in multiple React components.
- But, doing so typically requires ugly patterns, like render props or higher order components.
- These patterns lead to bigger and bigger component classes, shared state between components, and poor readability.
- Commonly, we mitigate these problems by extracting code into wrapper components, but wrappers come with their own set of issues...



# The Solution

# The State Hook

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

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

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



# Functional Updates

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

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

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

# Lazy Initialization

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

function Example() {
  const [count, setCount] = useState(() => 0);

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

# The Effect Hook

```
import React, { useState, useEffect } from 'react';

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(prevCount => prevCount + 1)}>
        Click me
      </button>
    </div>
  );
}
```

# Conditional Effects

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(() => 0);

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

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

# Clean Up Functions

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(() => 0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
    return () => console.log(`Last count was ${count}`);
  }, [count]);

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

# Bonus Hooks

- `useDebugValue` - Allows you to expose helpful data on components in the React debugger
- `useContext` - Accepts a context object and returns the current context value for that context
- `useReducer` - Accepts a reducer function and returns the current state paired with a dispatch method, similar to Redux
- `useCallback` - Accepts a function and a list of dependencies then returns a memoized version of the function that only executes when the dependencies change
- `useMemo` - Similar to `useCallback`, but returns a value instead of a function
- `useLayoutEffect` - Just like `useEffect`, but executes prior to render for easier DOM mutation
- `useRef` - Old fashioned refs + a convenient default value prior to render
- `useImperativeHandle` - A terrible, awful thing that should never be used

# Questions

# Thanks

Rose Karr

[RosalineKarr.com](https://RosalineKarr.com)  
[github.com/rosalinekarr](https://github.com/rosalinekarr)  
[twitter.com/rosalinekarr](https://twitter.com/rosalinekarr)  
[keybase.io/rosalinekarr](https://keybase.io/rosalinekarr)