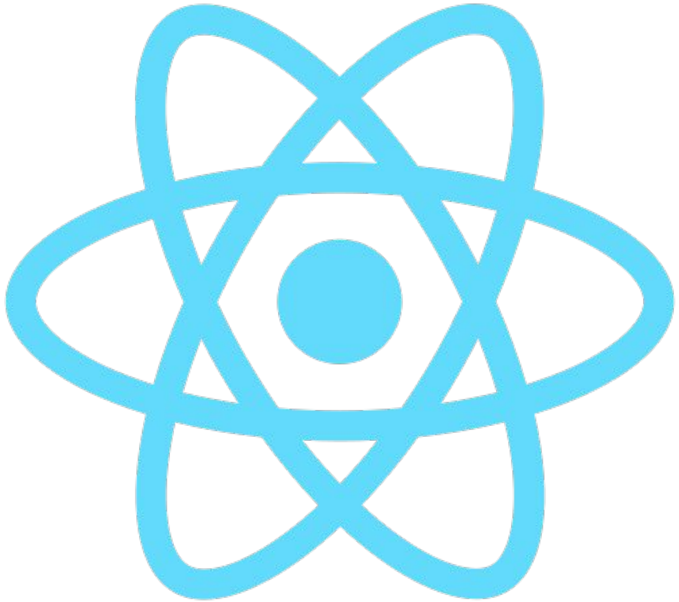
The Component Lifecycle and Hooks-1

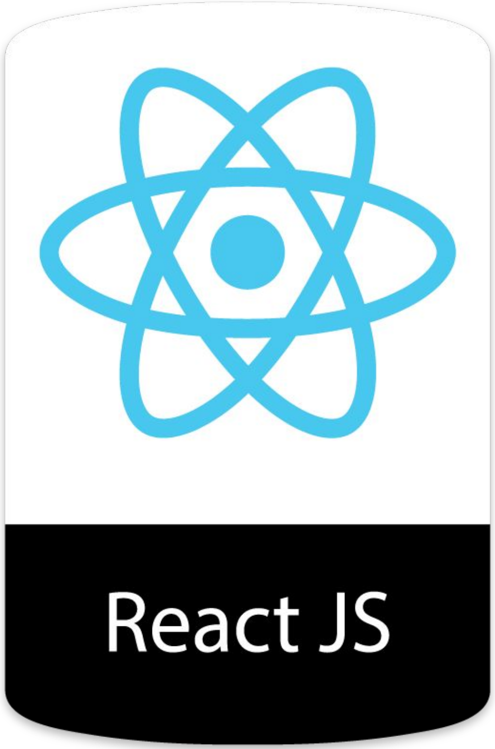
React Session-5



CLARUSWAY
WAY TO REINVENT YOURSELF

Table of Contents

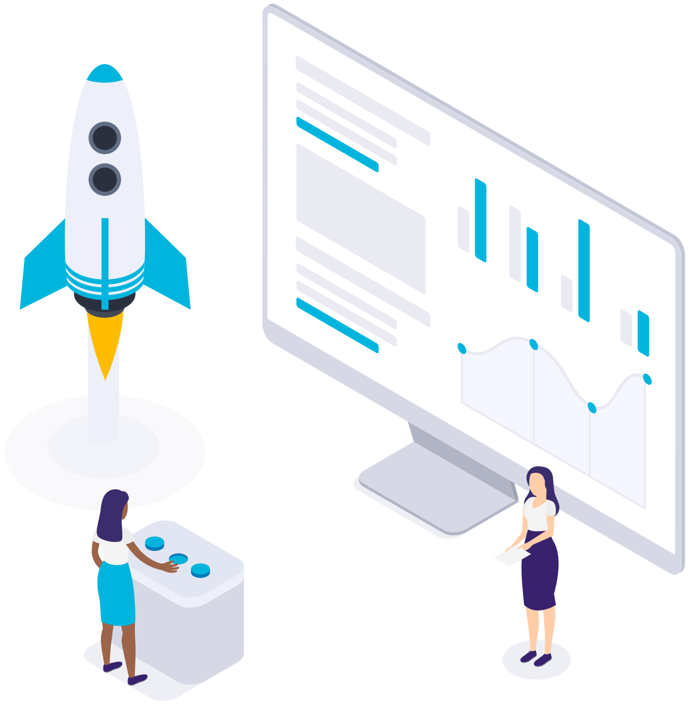
Did you finish React pre-class material?



Kahoot!

1

What is Lifecycle?



What is Lifecycle?

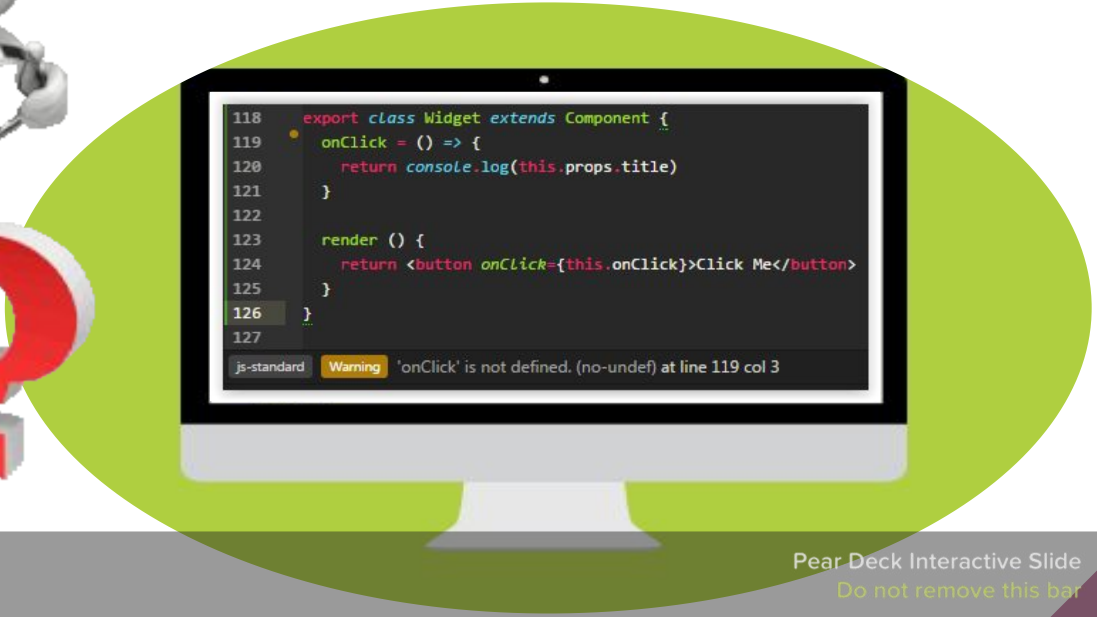
Around us everything goes through a cycle of taking birth, growing and at some of time it will die.

Consider trees, any software application, yourself, a div container or UI component in a web browser, each of these takes birth, grows by getting updates and dies.

The lifecycle methods are various methods which are invoked at different phases of the lifecycle of a component.

CLARUSWAY
WAY TO REINVENT YOURSELF

What is the phases of lifecycle?



```
export class Widget extends Component {
  onClick = () => {
    return console.log(this.props.title)
  }

  render () {
    return <button onClick={this.onClick}>Click Me</button>
  }
}
```

js-standard Warning 'onClick' is not defined. (no-undef) at line 119 col 3

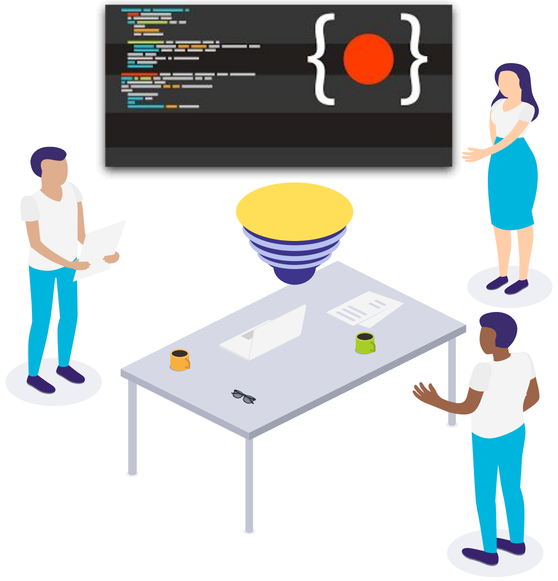
USWAY
REINVENT YOURSELF

Students, write your response!

Pear Deck Interactive Slide
Do not remove this bar

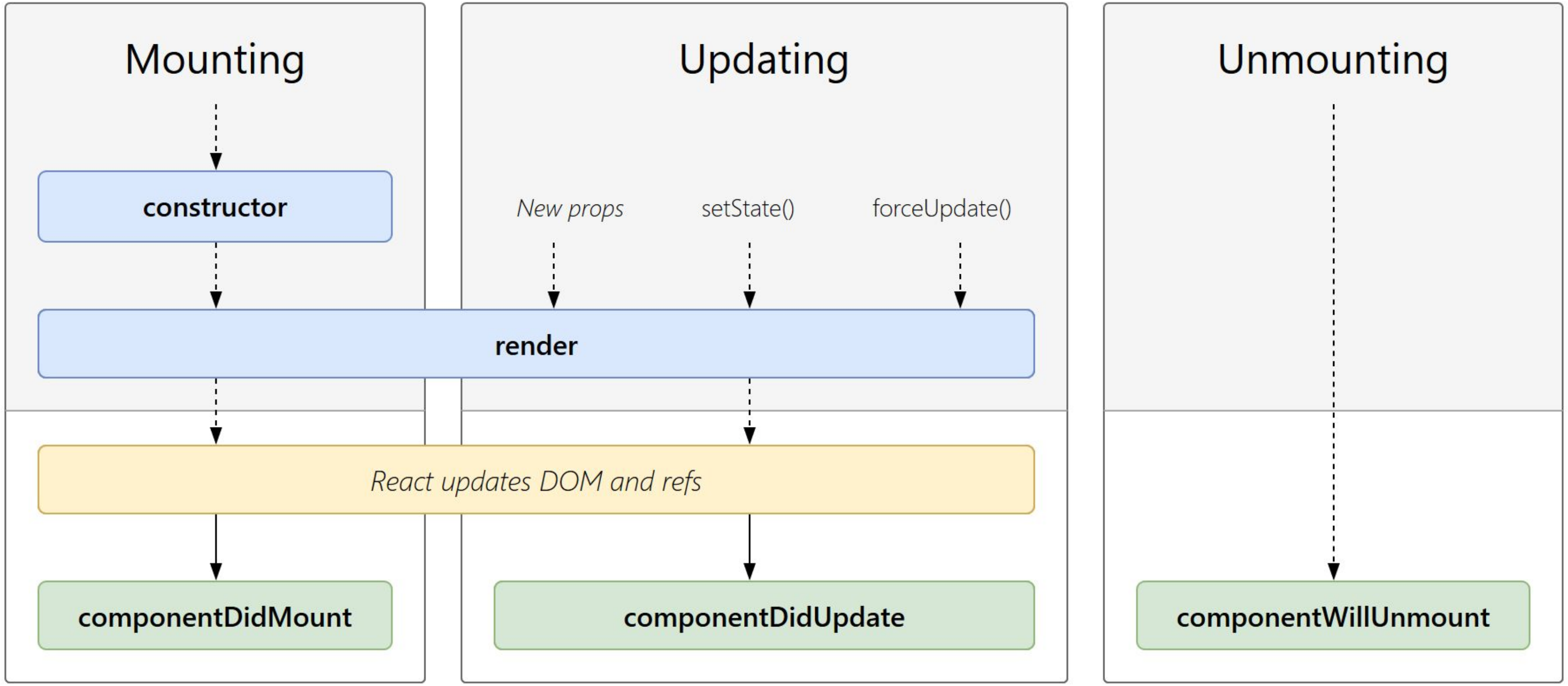
Three phases of React Component

- Mounting
- Updating
- Unmounting



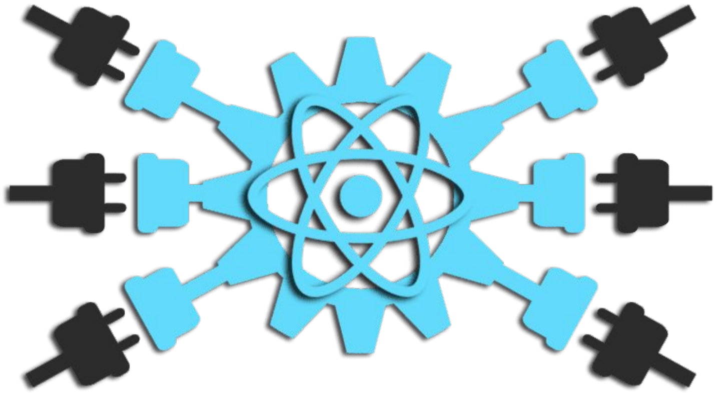
CLARUSWAY
WAY TO REINVENT YOURSELF

Three phases of React Component

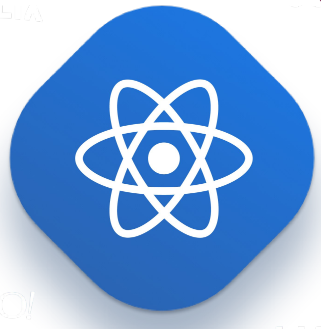


2

Mounting



Mounting



constructor()

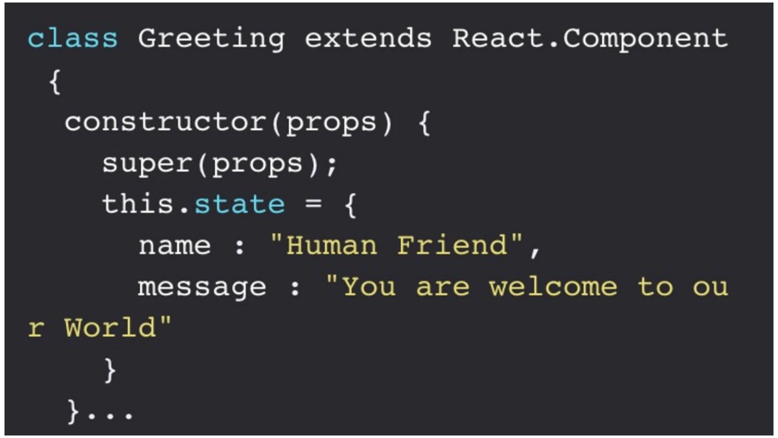
render()

componentDidMount()

CLARUSWAY
WAY TO REINVENT YOURSELF

constructor()

• ***constructor(props)*** is not necessary if you do not initialize state and/or you do not bind methods to a component.
• Called before a component is mounted.
• Should call ***super(props)*** if using constructor before any other statement, otherwise this.props will be undefined in the constructor which can lead to bugs)



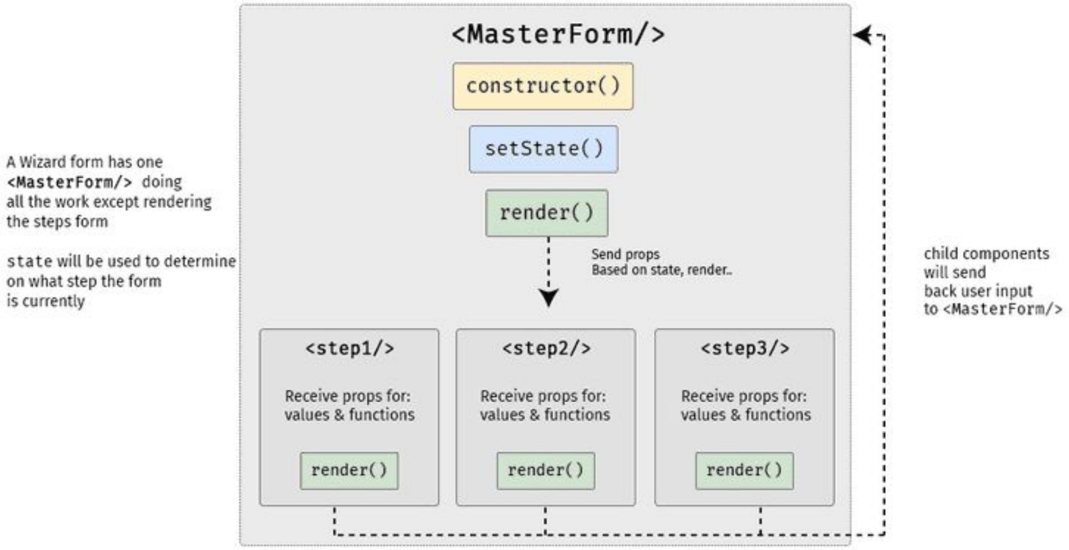
```
class Greeting extends React.Component
 {
  constructor(props) {
    super(props);
    this.state = {
      name : "Human Friend",
      message : "You are welcome to ou
r World"
    }
  }...
```

• Typically, constructors are used for two purposes:
  - Initialize local state by assigning an object to this.state
  - Binding event handler methods to an instance

CLARUSWAY
WAY TO REINVENT YOURSELF

render()

- ***render()*** returns the component markup, which can be a single child component, a set of components, or null or false (in case you don't want anything rendering)


A Wizard form has one <MasterForm/> doing all the work except rendering the steps form
state will be used to determine on what step the form is currently
<MasterForm/>
constructor()
setState()
render()
Send props Based on state, render..
<step1/>
Receive props for: values & functions
render()
<step2/>
Receive props for: values & functions
render()
<step3/>
Receive props for: values & functions
render()
child components will send back user input to <MasterForm/>

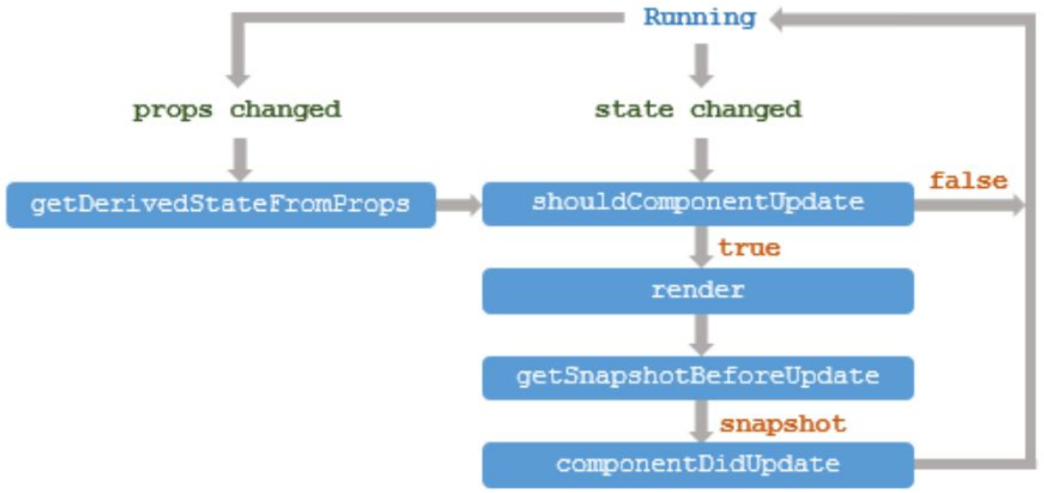
componentDidMount()

- ***`componentDidMount()`*** is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.
- This method is a good place to set up any subscriptions. If you do that, don't forget to unsubscribe in ***`componentWillUnmount()`***.
- You may call ***`setState()`*** immediately in ***`componentDidMount()`***. It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the ***`render()`*** will be called twice in this case, the user won't see the intermediate state. Use this pattern with caution because it often causes performance issues. In most cases, you should be able to assign the initial state in the ***`constructor()`*** instead. It can, however, be necessary for cases like modals and tooltips when you need to measure a DOM node before rendering something that depends on its size or position.

CLARUSWAY
WAY TO REINVENT YOURSELF

2

Updating


Running
props changed
state changed
getDerivedStateFromProps
shouldComponentUpdate
false
true
render
getSnapshotBeforeUpdate
snapshot
componentDidUpdate

componentDidUpdate()

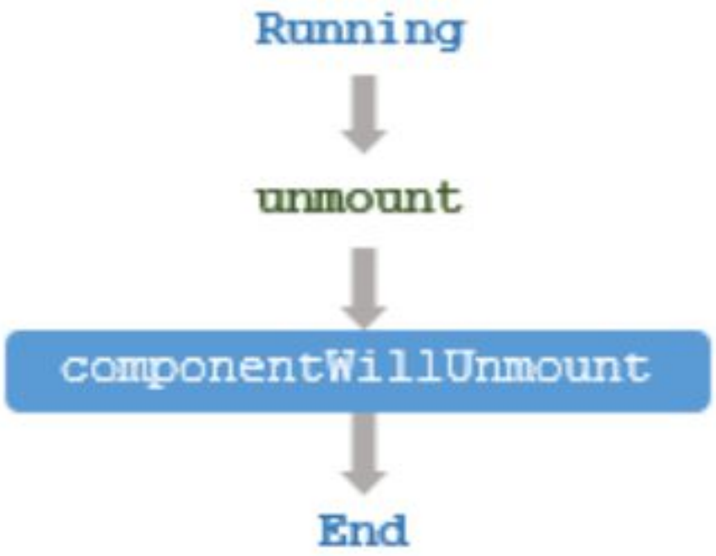
- This lifecycle method is invoked as soon as the updating happens. The most common use case for the ***componentDidUpdate()*** method is updating the DOM in response to prop or state changes.
- You can call ***setState()*** in this lifecycle, but keep in mind that you will need to wrap it in a condition to check for state or prop changes from previous state. Incorrect usage of setState can lead to an infinite loop.
- You can modify the component state within the componentDidUpdate(), but use it with caution.
- Take a look at the example below that shows a typical usage example of this lifecycle method.

```
componentDidUpdate(prevProps) {
   //Typical usage, don't forget to compare the props
   if (this.props.userName !== prevProps.userName) {
     this.fetchData(this.props.userName);
    }
  }
```

- Notice in the above example that we are comparing the current props to the previous props. This is to check if there has been a change in props from what it currently is. In this case, there won't be a need to make the API call if the props did not change.

3

Unmounting



componentWillUnmount()

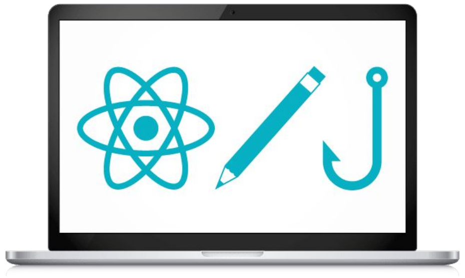
- As the name suggests this lifecycle method is called just before the component is unmounted and destroyed. If there are any cleanup actions that you would need to do, this would be the right spot.
- You cannot modify the component state in ***`componentWillUnmount`*** lifecycle.
- This component will never be re-rendered and because of that we cannot call setState() during this lifecycle method.

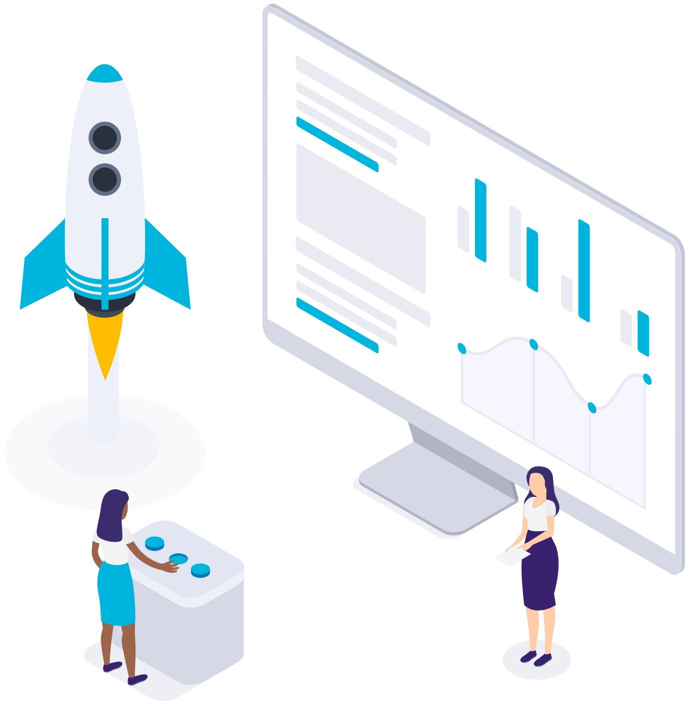
```
componentWillUnmount() {
   window.removeEventListener('resize', this.resizeListener)
}
```

- Common cleanup activities performed in this method include, clearing timers, cancelling api calls, or clearing any caches in storage.

1

What is React Hooks?





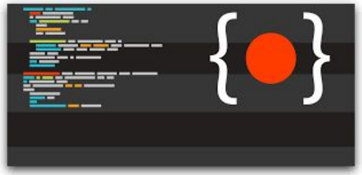
CLARUSWAY
WAY TO REINVENT YOURSELF

What is React Hooks?

Hooks are a new addition in React 16.8. They let you use state and other React features without writing class components.

Hooks are functions that let you “hook into” React state and lifecycle features from function components.

Hooks don’t work inside classes — they let you use React without classes. In fact, they can only be used in functional components.



What is the motivation behind Hooks?

- It’s hard to reuse stateful logic between components. **Hooks allow you to reuse stateful logic without changing your component hierarchy.**
- Complex components become hard to understand. **Hooks let you split one component into smaller functions based on what pieces are related (such as setting up a subscription or fetching data)**, rather than forcing a split based on lifecycle methods.
- Classes confuse both people and machines. **Hooks let you use more of React’s features without classes.**
- Only call Hooks **at the top level**. Don’t call Hooks inside loops, conditions, or nested functions.
- Only call Hooks **from React function components**. Don’t call Hooks from regular JavaScript functions or class components. (There is just one other valid place to call Hooks — your own custom Hooks. More on this later.)

Most Used React Hooks

State Hook - `useState`

Effect Hook - `useEffect`

Context Hook - `useContext` *

Additional Hooks

useReducer *

useCallback

useMemo

useRef

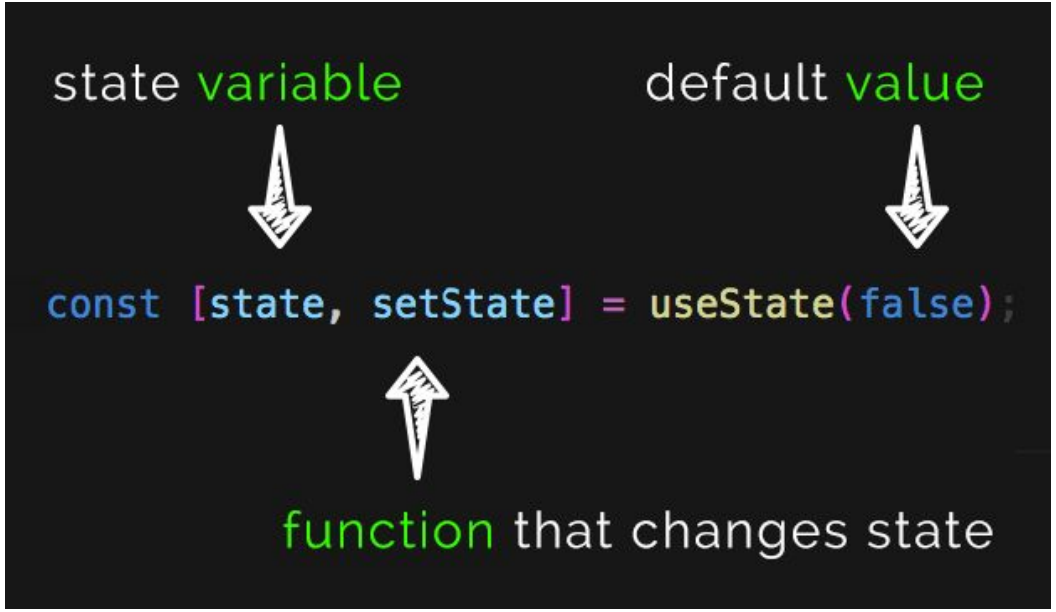
*We will see Context Hook and useReducer in Context chapter

CLARUSWAY
WAY TO REINVENT YOURSELF

2

State Hook

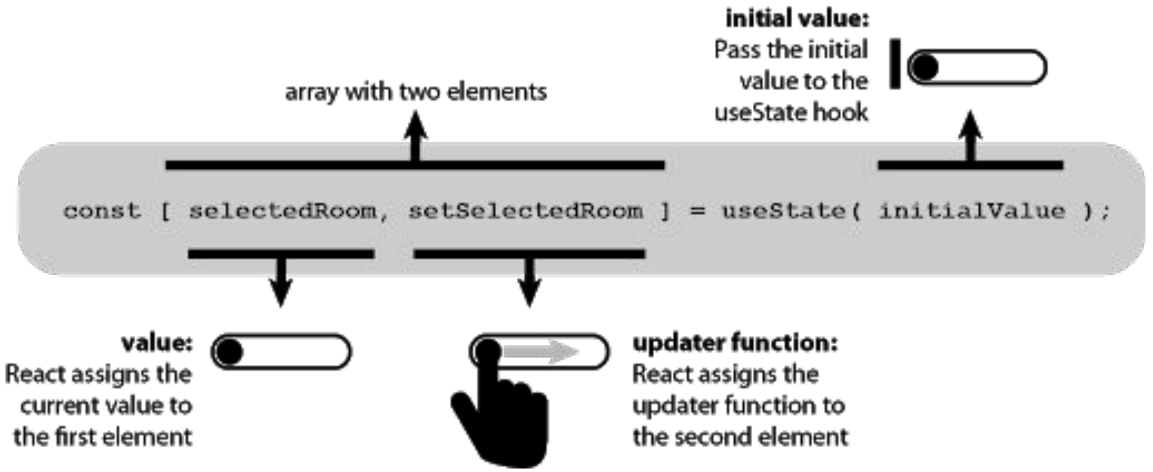
`useState`



```
const [state, setState] = useState(false);
```

State Hook - `useState`

- `useState` allows us to make our components stateful. We call it inside a function component to add some local state to it.



array with two elements
initial value:
Pass the initial value to the useState hook
value:
React assigns the current value to the first element
updater function:
React assigns the updater function to the second element

```
const [ selectedRoom, setSelectedRoom ] = useState( initialValue );
```

- Whereas using state previously required using a class component, hooks give us the ability to write it using just functions. It allows us to have more flexible components.

State Hook - `useState`

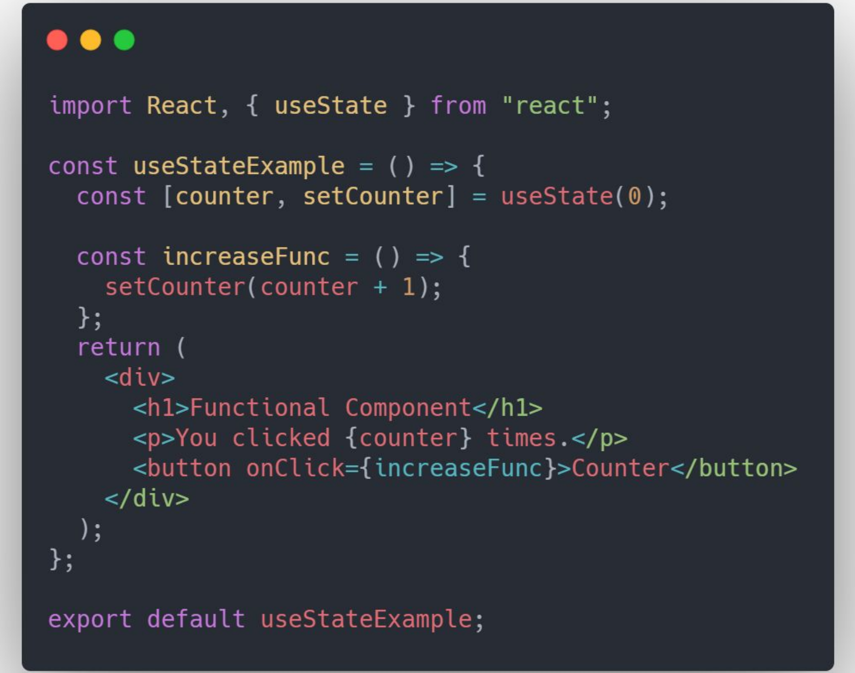
React will preserve this state between re-renders.

- `useState` hooks maintain state by being fed the latest state in a new every render.
- `useState` returns a tuple pair: the *current* state value and a function that lets you update the state.
- The only argument passed to `useState` is the initial state. Unlike `this.state`, the state here doesn't have to be an object — although it can be if you want. The initial state argument is only used during the first render.
- You can have as many hooks in a function you can possibly want; so multiple pieces of state each with their own updater function.

CLARUSWAY
WAY TO REINVENT YOURSELF

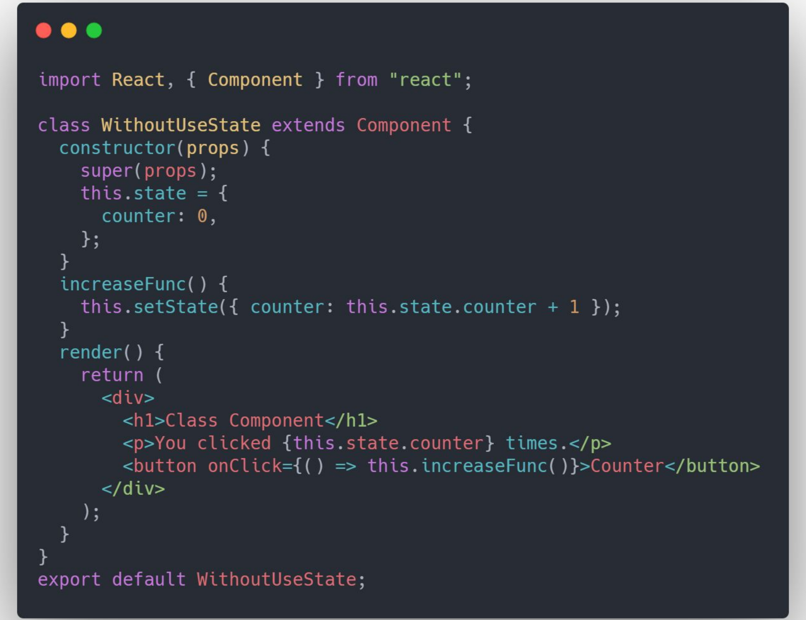
State Hook - useState

Example of **useState**— when you click the button, it increments the value:



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

const useStateExample = () => {
  const [counter, setCounter] = useState(0);

  const increaseFunc = () => {
    setCounter(counter + 1);
  };
  return (
    <div>
      <h1>Functional Component</h1>
      <p>You clicked {counter} times.</p>
      <button onClick={increaseFunc}>Counter</button>
    </div>
  );
};

export default useStateExample;
```

with *useState()*



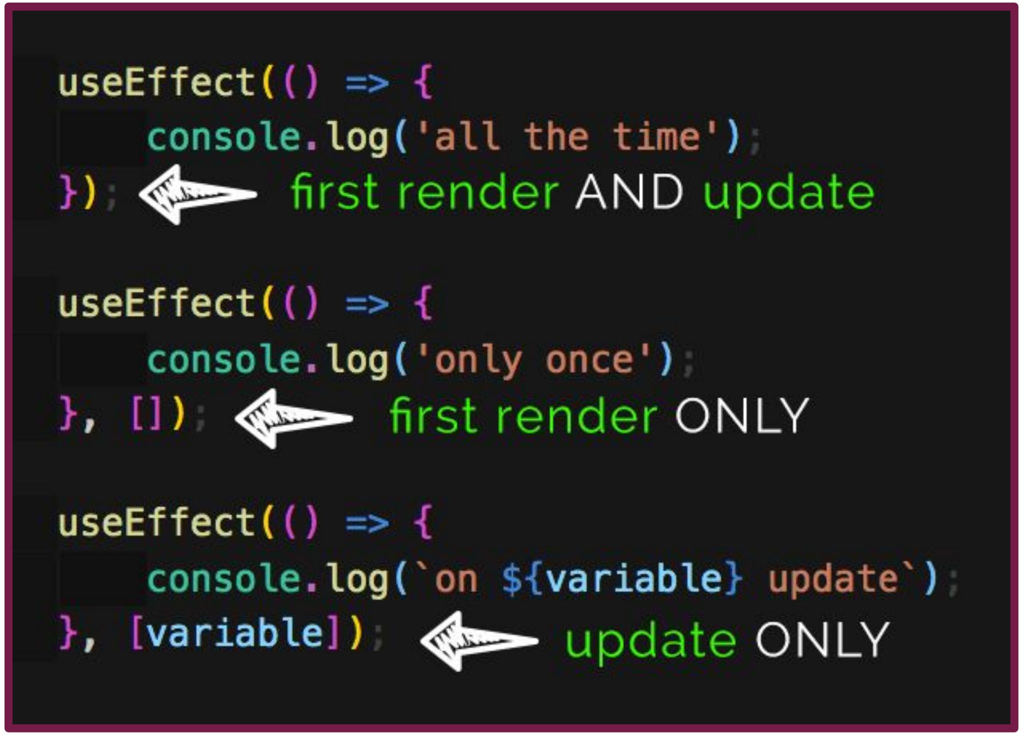
```
import React, { Component } from "react";

class WithoutUseState extends Component {
  constructor(props) {
    super(props);
    this.state = {
      counter: 0,
    };
  }
  increaseFunc() {
    this.setState({ counter: this.state.counter + 1 });
  }
  render() {
    return (
      <div>
        <h1>Class Component</h1>
        <p>You clicked {this.state.counter} times.</p>
        <button onClick={() => this.increaseFunc()}>Counter</button>
      </div>
    );
  }
}
export default WithoutUseState;
```

without *useState()*

3

Effect Hook

useEffect



```
useEffect(() => {
    console.log('all the time');
});
```

first render AND update

```
useEffect(() => {
    console.log('only once');
}, []);
```

first render ONLY

```
useEffect(() => {
    console.log(`on ${variable} update`);
}, [variable]);
```

update ONLY

Effect Hook - `useEffect`

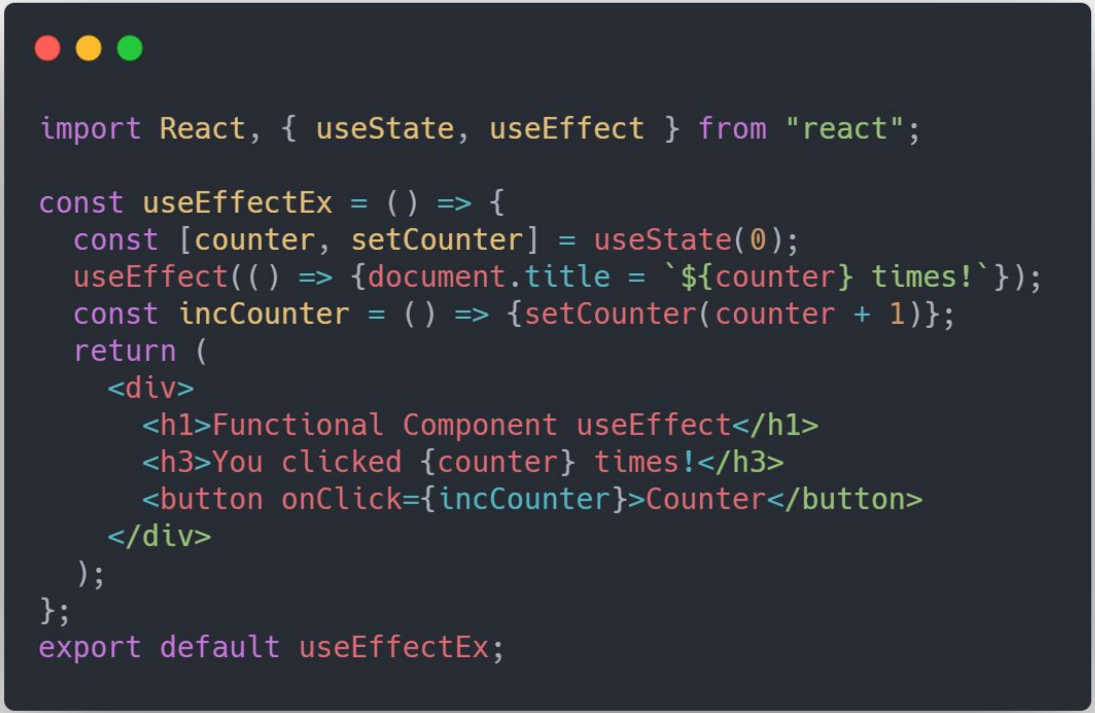
The Effect Hook, `useEffect`, adds the ability to perform side effects from a function component. It serves the same purpose as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in React classes, but unified into a single API.

- When you call `useEffect`, you're telling React to run your "effect" function after flushing changes to the DOM. Effects are declared inside the component so they have access to its props and state. By default, React runs the effects after every render — *including* the first render.
- React is constantly scheduling this.

Effect Hook - `useEffect`

- You can also use a second parameter to give it a list of dependencies. An empty array as a second parameter would run the `useEffect` function once.
- Effects may also optionally specify how to “clean up” after them by returning a function. So if using `setTimeout` return `clearTimeout`. This is so that on unmount React can run the clean-up functions. So with Ajax requests, you can cancel a request.
- The array of dependencies is not passed as arguments to the effect function. Conceptually, though, that’s what they represent: every value referenced inside the effect function should also appear in the dependencies array.

Example of `useEffect` — this component sets the document title after React updates the DOM:



```jsx
import React, { useState, useEffect } from "react";

const useEffectEx = () => {
  const [counter, setCounter] = useState(0);
  useEffect(() => {document.title = `${counter} times!`});
  const incCounter = () => {setCounter(counter + 1)};
  return (
    <div>
      <h1>Functional Component useEffect</h1>
      <h3>You clicked {counter} times!</h3>
      <button onClick={incCounter}>Counter</button>
    </div>
  );
};
export default useEffectEx;
```

CLARUSWAY
WAY TO REINVENT YOURSELF

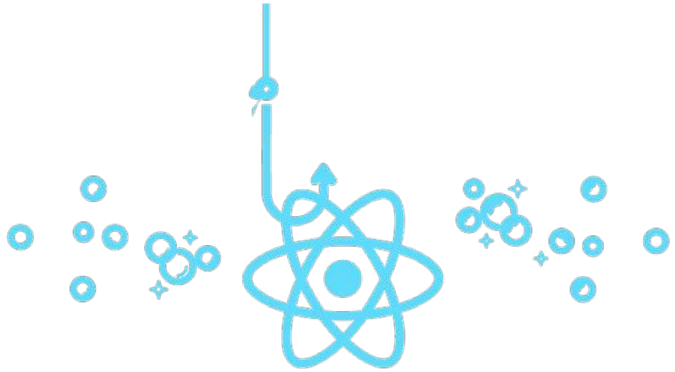
Rules of Hooks

1. **Call Hooks at the top level**

   We shouldn't call hooks inside loops, conditions or nested functions

2. **Only call Hooks from React Function Component**

   We should only call them from React Function Components and from custom Hooks.



CLARUSWAY
WAY TO REINVENT YOURSELF

THANKS!

Any questions?

CLARUSWAY
WAY TO REINVENT YOURSELF