



REACT.js

CLASE 4



¿Qué es esto?

```
class MiBoton extends Component{  
  render(){  
    return(  
      <button>Clickeame!</button>  
    );  
  }  
}  
  
export default MiBoton;
```

¿Qué es esto?

```
class MiBoton extends Component{  
  render(){  
    return(  
      <button>Clickeame!</button>  
    );  
  }  
}  
  
export default MiBoton;
```

¡Componente!

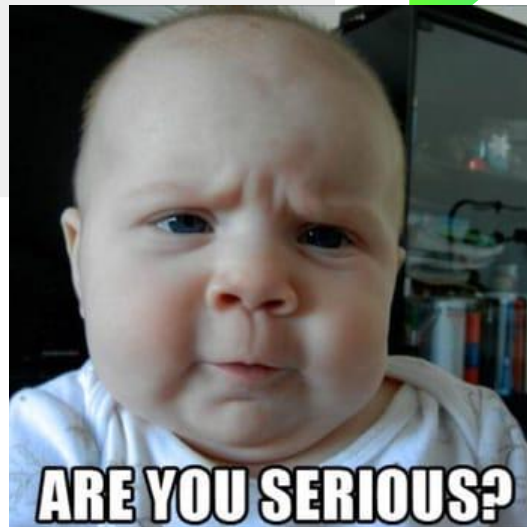
¿Y esto?

```
const saludar = () => {  
  return <h1> Hola! </h1>  
}  
  
export default saludar;
```

¿Y esto?

```
const saludar = () => {  
  return <h1> Hola! </h1>  
}  
  
export default saludar;
```

iComponente!



Componentes ++

Existen dos tipos de componentes:

- ◆ Statefull components
(Componentes de clase)
- ◆ Stateless components
(Componentes funcionales)

Componentes de clase



Componentes de clase

Los componentes de clase permiten mantener **datos propios** a lo largo del tiempo e implementar distintos comportamientos durante su **ciclo de vida**.

Al conjunto de datos internos del componente se conocen como **estado «state»** y es una característica disponible solo para los componentes definidos como clases. Es similar a las props, pero es privado y está completamente controlado por el componente.

State

```
class Counter extends Component{  
  
  constructor(){  
    super();  
    this.state = {  
      count: 1  
    }  
  }  
  
  render(){  
    return(  
      <h1>{this.state.count}</h1>  
    );  
  }  
}  
  
export default Counter;
```

State

```
class Counter extends Component{  
  
  constructor(){  
    super();  
    this.state = {  
      count: 1  
    }  
  }  
  
  render(){  
    return(  
      <h1>{this.state.count}</h1>  
    );  
  }  
}  
  
export default Counter;
```

El método constructor es necesario para poder definir la estructura del estado de un componente

State

```
class Counter extends Component{
```

```
  constructor(){  
    super();  
    this.state = {  
      count: 1  
    }  
  }  
}
```

```
  render(){  
    return(  
      <h1>{this.state.count}</h1>  
    );  
  }  
}
```

```
export default Counter;
```

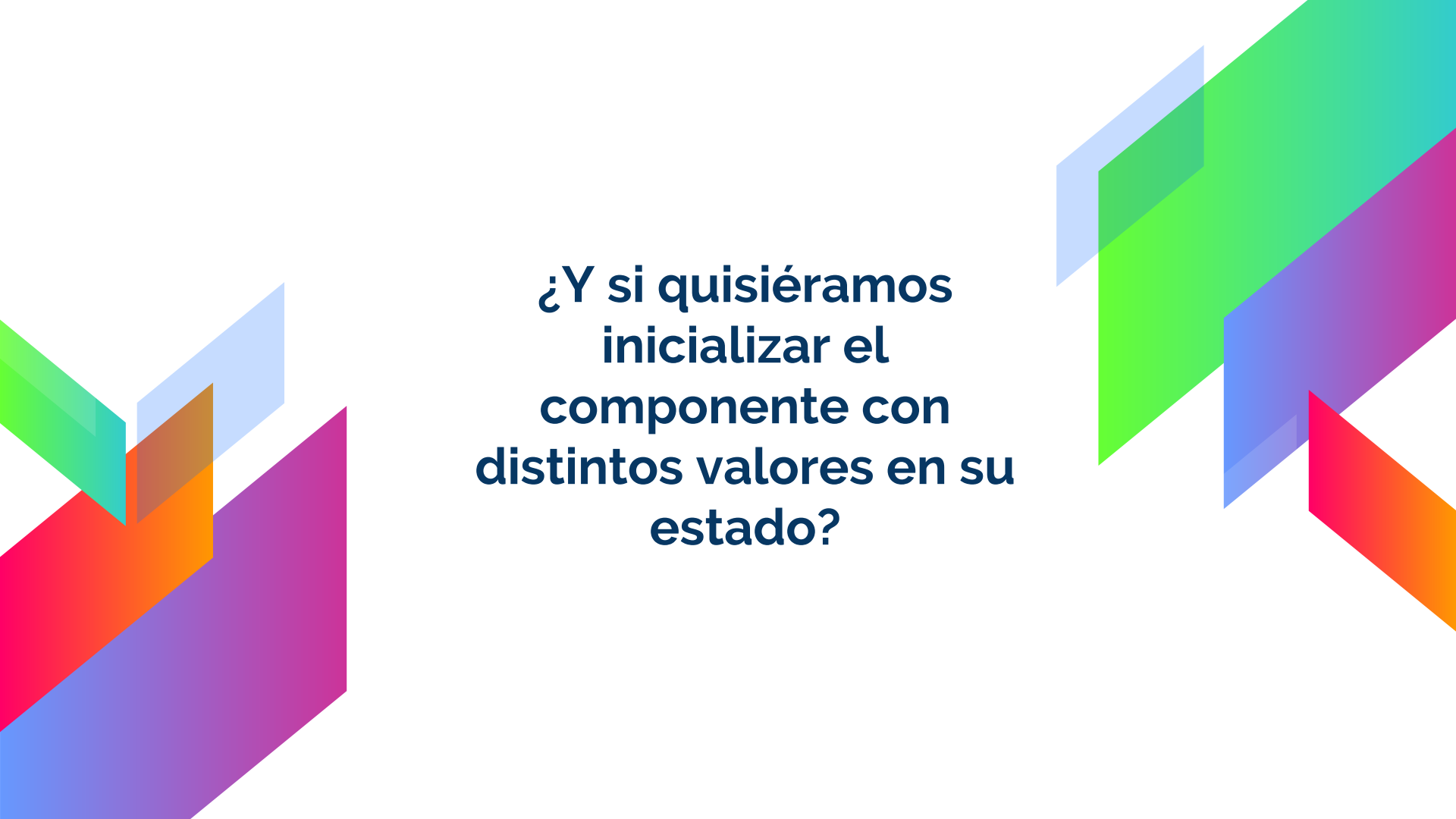
**Llamar a la función
super() en el
constructor es
necesario en React**

State

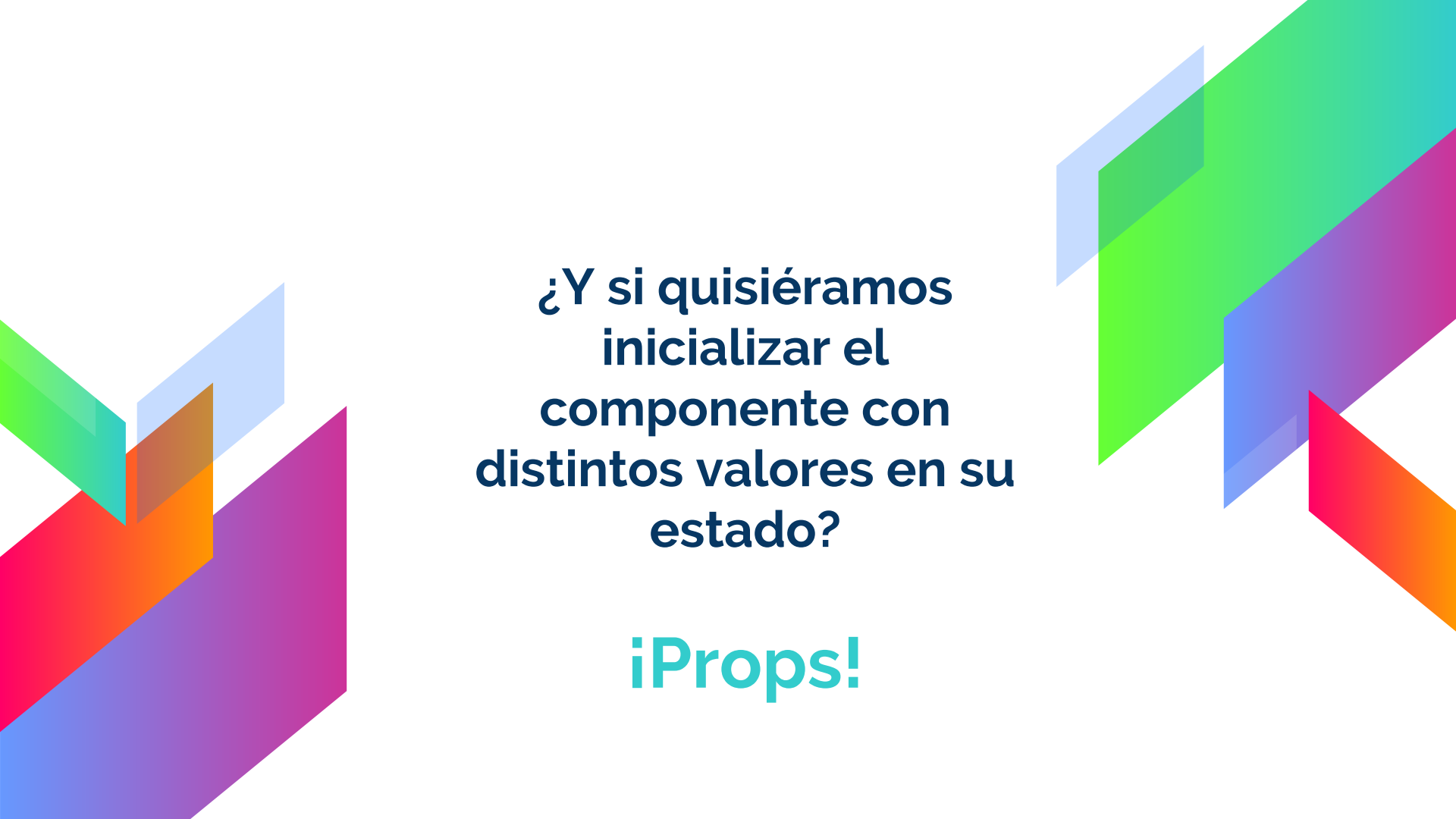
```
class Counter extends Component{  
  
  constructor(){  
    super();  
    this.state = {  
      count: 1  
    }  
  }  
  
  render(){  
    return(  
      <h1>{this.state.count}</h1>  
    );  
  }  
}  
  
export default Counter;
```

El estado de un componente será un objeto literal de JS

El cual luego puede ser utilizado en cualquier método de la clase tanto para lectura como para escritura

The image features abstract geometric shapes in the corners. On the left, there are overlapping shapes in shades of green, blue, orange, and purple. On the right, there are overlapping shapes in shades of green, blue, purple, and orange. The central text is in a dark blue, sans-serif font.

**¿Y si quisiéramos
inicializar el
componente con
distintos valores en su
estado?**

The image features abstract geometric shapes in the corners. On the left, there are overlapping triangles in shades of green, blue, orange, and purple. On the right, there are similar shapes in shades of green, blue, purple, and red. The central text is in a dark blue, sans-serif font.

**¿Y si quisiéramos
inicializar el
componente con
distintos valores en su
estado?**

iProps!

State

```
class Counter extends Component{  
  
  constructor(props){  
    super(props);  
    this.state = {  
      count: props.init;  
    }  
  }  
  
  render(){  
    return(  
      <h1>{this.state.count}</h1>  
    );  
  }  
}  
  
export default Counter;
```

¡Podemos recibir las props en el constructor para luego utilizarlas!

Es buena práctica utilizarlas al llamar super()

State

```
<Counter init="10" />
```

```
class Counter extends Component{  
  constructor(props){  
    super(props);  
    this.state = {  
      count: props.init;  
    }  
  }  
  ...  
}  
  
export default Counter;
```


setState

```
setState(nextState)
```

ó

```
setState(callback)
```

Es buena práctica cambiar el estado a través de setState

setState recibe un objeto literal con los atributos modificados del estado.

Ó puede recibir un callback que debe retornar un objeto literal

setState

```
class Counter extends Component{
  constructor(){
    super()
    this.state = {
      count: 1
    }
  }
  const increment = () => {
    this.setState({count: this.state.count + 1});
  }

  render(){
    return(
      <div>
        <h1>{this.state.count}</h1>
        <button onClick={this.increment}/>
      </div>
    );
  }
}

export default Counter;
```

setState

```
class Counter extends Component{
  constructor(){
    super()
    this.state = {
      count: 1
    }
  }
  const increment = () => {
    this.setState(state => ({count: state.count+1}));
  }

  render(){
    return(
      <div>
        <h1>{this.state.count}</h1>
        <button onClick={this.increment}/>
      </div>
    );
  }
}

export default Counter;
```

setState - onChange de input

```
class Search extends Component{
  constructor() {
    super()
    this.state = {
      inputValue: '',
    }
  }

  handleOnChangeInput = e => {
    this.setState({
      inputValue: e.target.value
    })
  }

  render(){
    return (
      <input value={this.state.inputValue} onChange={this.handleOnChangeInput} />
    )
  }
}

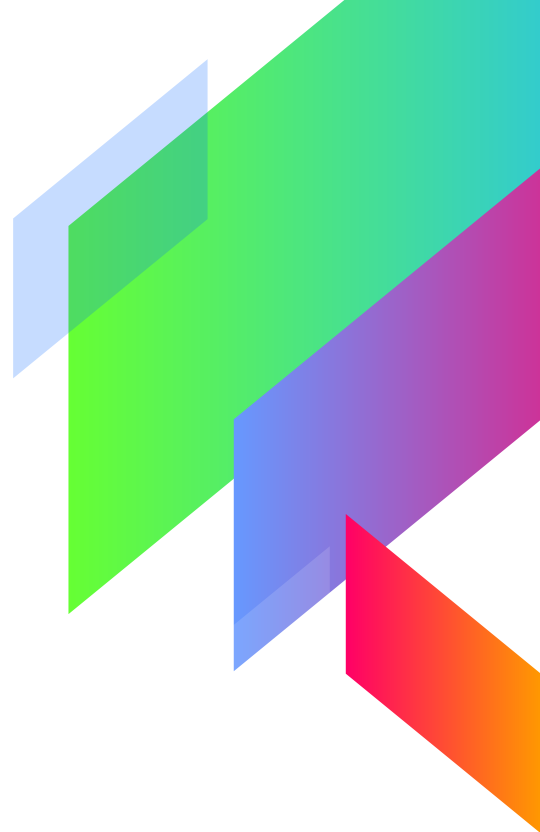
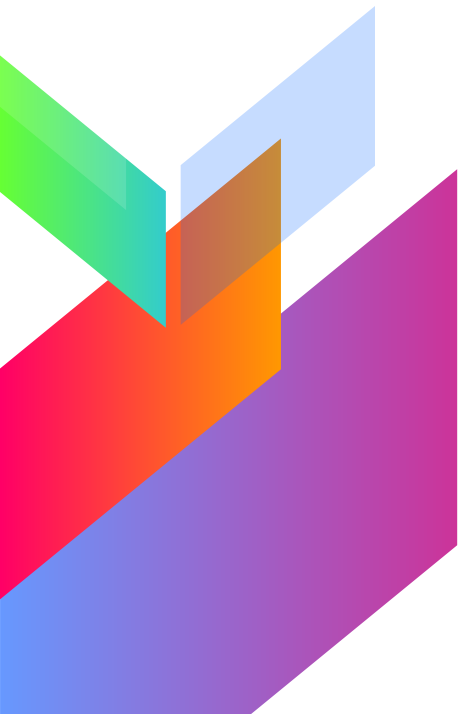
export default Counter;
```



Ejercicios

Punto 1 de la ejercitación

Componentes sin estado



Componentes funcionales

Los componentes funcionales son aquellos de presentación, es decir que se limitan a mostrar datos y no guardan la lógica asociada a manipulación del estado.

Este tipo de componentes básicamente son funciones que reciben como parámetro las props y retorna un elemento.



Componentes funcionales


```
const Msg = (props) => {  
  return <p>{props.msg}</p>  
}
```

```
export default Msg;
```

← "this.props" por "props"

Componentes funcionales

Destructuramos las props!



```
const Msg = ({ msg }) => {  
  return <p>{msg}</p>  
}  
  
export default Msg;
```

The slide features decorative geometric shapes in the corners. On the left, there are overlapping triangles in green, blue, purple, and orange. On the right, there are overlapping triangles in purple, green, red, and blue. In the center, a light gray rounded rectangle contains the text.

*“Try to keep as many of your components
as possible stateless”*

- Dan Abramov



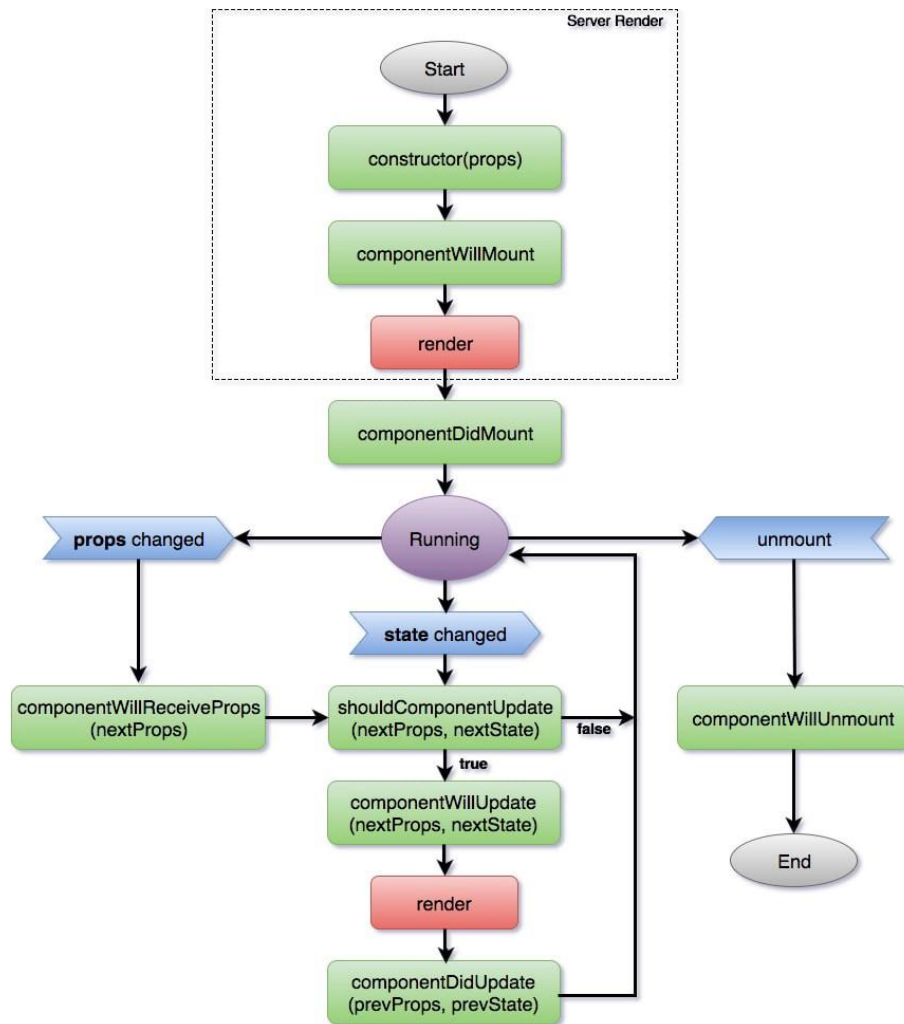
Ejercicios

Punto 2 de la ejercitación



Ciclo de vida

iExclusivo
para statefull
components!



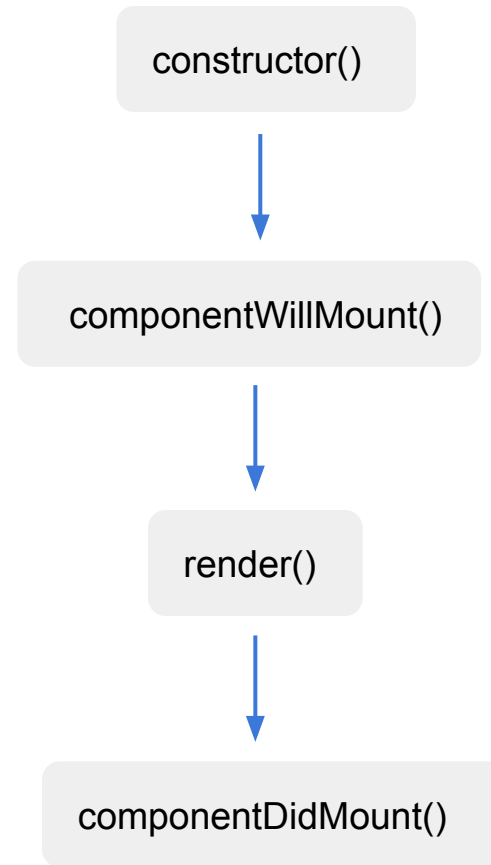
Ciclo de vida

1. Montado del componente
2. Actualización en las propiedades
3. Actualización en el estado
4. Desmontado del componente

Ciclo de vida

1. Montado del componente

2. Actualización en las propiedades
3. Actualización en el estado
4. Desmontado del componente



Montado del componente

```
class Counter extends Component{  
  constructor(props){  
    super(props);  
    this.state = { count: 0 }  
  }  
  render(){  
    return(  
      <h1>{this.state.count}</h1>  
    );  
  }  
}  
  
export default Counter;
```

Pasamos las propiedades al constructor del componente de clase. En el constructor es donde inicializamos el estado del componente.

Montado del componente

```
componentWillMount(){  
  //Code...  
}
```

El método `componentWillMount()` se llama justo antes de ejecutar el método `render()`. La documentación oficial de React recomienda utilizar directamente el constructor.

Montado del componente

```
render(){  
  //Code...  
}
```

El método render()

Montado del componente

```
componentDidMount(){  
  Axios.get(URL)  
    .then((r)=>console.log(r))  
    .catch((e)=>console.log(e));  
}
```

Se invoca esta función una vez ya ejecutado el método render(). Como el DOM ya es accesible podemos en este método realizar cualquier manipulación sobre él.

Se estila en este método hacer pedidos a endpoints via AJAX, inicializar timers o generar suscripciones a servicios.

Ciclo de vida

1. Montado del componente

2. Actualización en las propiedades

3. Actualización en el estado

4. Desmontado del componente

```
componentWillReceiveProps()
```

Actualización en las propiedades

```
componentWillReceiveProps(newProps){  
  this.setState({  
    prop: newProps.prop  
  });  
}
```

Si en un componente padre, cambian las **props** de un componente hijo, se ejecuta **componentWillReceiveProps** en el componente hijo.

Este método nos permite validar y agregar lógica para cambiar el estado del componente si es que se modifican sus props.

OBSERVACIÓN

Cuando recibimos nuevas propiedades podemos actualizar el estado del componente.

Pero esto NO implica que un cambio de estado vaya a generar un cambio en las props.

Es decir, el método `componentWillReceiveProps()` es llamado cuando hay nuevas props pero NO cuando hay un cambio de estado.

Ciclo de vida

1. Montado del componente
2. Actualización en las propiedades
- 3. Actualización en el estado**
4. Desmontado del componente

`shouldComponentUpdate()`



`componentWillUpdate()`



`componentDidUpdate()`

Actualización en el estado

```
shouldComponentUpdate(newProps, newState){  
  return newsProps.prop !== this.state.prop  
}
```

Este método se invoca antes de volver a renderizar cuando se reciben nuevos props o estados.

El método devuelve un valor booleano.

Por defecto retorna true. En caso de devolver false, no se llama a los métodos render(), componenteWillUpdate() y componentDidUpdate().

Actualización en el estado

```
componentWillUpdate(){  
  //code...  
}
```

Se llama justo antes de llamar a `render()`, cuando se reciben los nuevos props o estados. Aquí podemos hacer preparaciones antes de que ocurra la actualización.

NO podemos utilizar `this.setState()` en este método.

Actualización en el estado

```
componentDidUpdate(prevProps,  
prevState){  
  //code...  
}
```

Se llama justo después de render después que todos los cambios han sido hechos en el DOM. Puedes utilizar este componente para hacer alguna operación el DOM después que el componente se haya actualizado.

Ciclo de vida

1. Montado del componente
2. Actualización en las propiedades
3. Actualización en el estado
- 4. Desmontado del componente**

```
componentWillUnmount()
```

Desmontado del componente

```
componentWillUnmount(){  
  //code...  
}
```

Es llamado justo antes de que el componente sea removido del DOM, es útil para hacer cualquier operación de limpieza, tales como invalidar timers, eliminar elementos que se hayan creados durante `componentDidMount` y cualquier otra operación pendiente.

Desmontado del componente

```
class Button extends Component {  
  componentWillUnmount() {  
    alert('El componente será desmontado')  
  }  
  
  render(){  
    return (  
      <button onClick={this.props.onClick}>Guardar cambios</button>  
    )  
  }  
}  
  
Button.propTypes = {  
  onClick: PropTypes.func.isRequired,  
}
```

Desmontado del componente

```
...
import Button from './Button'

class App extends Component {
  constructor() {
    super()
    this.state = {
      buttonVisible: true
    }
  }

  handleClickButton = () => {
    this.setState({
      buttonVisible: false
    })
  }

  render() {
    return (
      this.state.buttonVisible && <Button onClick={this.handleClickButton}/>
    )
  }
}
```

El desmontado se ejecuta cuando se quita un componente montado. Cuando cambiamos el estado "buttonVisible" a false, se desmonta el componente Button

Desmontado del componente

```
...
import Button from './Button'

class App extends Component {
  constructor() {
    super()
    this.state = {
      buttonVisible: true
    }
  }

  handleClickButton = () => {
    this.setState({
      buttonVisible: false
    })
  }

  render() {
    return (
      { this.state.buttonVisible
        ? <Button onClick={this.handleClickButton}/>
        : <p> Sin botón </p>
      }
    )
  }
}
```



Ejercicios

Punto 3 de la ejercitación





Gracias!

¿Preguntas?

