



Clase 05. React JS

# ***COMPONENTES II***



## ***OBJETIVOS DE LA CLASE***

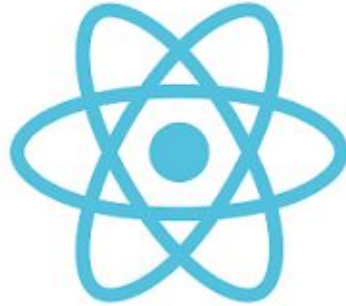
- Conocer los ciclos de vida de un componente
- Aprender a aplicar propiedades, eventos, estados y ciclos de vida en un componente.

# ***COMPONENTES II:- INTRODUCCIÓN***

# ***Anatomía de un componente***

Props

State



DOM Sync

Lifecycle

# ***Propiedades/Props***

## ***Props: ¿Un espacio multipropósito!?***

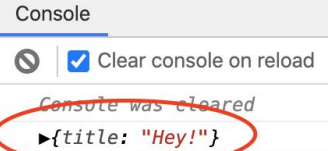
- No están limitadas a ser valores fijos como: 1 / “Alexis” / true
- Pueden ser lo que sea:
  - **Valores comunes**
    - num, bool, array, obj
  - **Funciones**
  - **Componentes** Si los componentes son funciones, ¡entonces puedo pasar componentes! ;)
  - **Children**
  - **Valores inyectados por librerías**
    - location, rutas, traducciones

# Props - Relación de Children y Props

React inyecta automáticamente **children en** las props, sólo si encuentra alguno. En este ejemplo **<SuperForm>** no tiene children

```
function SuperForm (props){  
  console.log(props); // Solo tiene titulo  
  return <>  
    <h1>{props.title}</h1>  
  </>;  
}  
  
function App() {  
  return <SuperForm title="Hey!"></SuperForm>  
}  
  
render(<App />, document.getElementById('root'));
```

Hey!





# Props - Relación de Children y Props

Si le agregamos children en el JSX...

```
function SuperButton() {  
  function doSomething() {  
    console.log('Hey coders!');  
  }  
  return <button type="button" onClick={doSomething}  
    >Click me</button>  
}  
  
function SuperForm(props) {  
  console.log(props); // Tiene titulo y children  
  return <>  
    <h1>{props.title}</h1>  
    {props.children}  
  </>;  
}  
  
function App() {  
  return <SuperForm title="Hey!">  
    <SuperButton />  
  </SuperForm>  
}
```

Hey!

Click me

Console

☐ Clear console on reload

Console was cleared

▶ {title: "Hey!", children: {...}}

Los inyecta **como objeto** si es único o **como array** si son muchos.

**Tener cuidado** para evitar errores del tipo **children[0]**, si espero un grupo de children y **viene uno solo**, cuando hay un único child de tipo object

# Props - Render props

Si pasamos un componente por prop...

```
function SuperButton({ buttonText }) {  
  function doSomething() {  
    console.log('Hey coders!');  
  }  
  return <button type="button" onClick={doSomething}  
    >{ buttonText }</button>  
}  
  
function SuperForm(props) {  
  console.log(props);  
  return <>  
    <h1>{props.title}</h1>  
    {props.render({ buttonText: 'Superform button'})}  
  </>;  
}  
  
function App() {  
  return <SuperForm title="Hey!" render={SuperButton}>  
    </SuperForm>  
}  
  
render(<App />, document.getElementById('root'));
```

Hey!

Superform button

Console

☒ Clear console on reload

Console was cleared

▶ {title: "Hey!", render: f}

>

Podemos usarlo en otro componente sin que, como en este caso, **SuperForm** sepa realmente la implementación del render prop.

*No es obligatorio usar el nombre **render** como está en este ejemplo*

***Estado/State: Class based***

El estado en las clases era “*más simple*” de mantener, porque las clases en sí tienen un contexto propio (this.state) persistente

## **Class** based components

componentes basados en clases

```
class App extends Component {  
  constructor() {  
    super();  
    this.state = {  
      name: 'React'  
    };  
  }  
  
  render() {  
    return (  
      <div>  
        <Hello name={this.state.name} />  
      </div>  
    );  
  }  
}
```

**Hello React!**

Utilizando  
**this.setState** se  
podía guardar en  
**this.state**, que  
persiste entre  
renders, porque la  
clase se crea al  
montar y se destruye  
al desmontar

## Class based components

componentes basados en clases

```
class App extends Component {  
  constructor() {  
    super();  
    this.state = {  
      name: 'ReactClass'  
    };  
  }  
  
  updateName = () => {  
    this.setState({ name: 'ReactFunction' });  
  }  
  
  render() {  
    return (  
      <div onClick={this.updateName}>  
        <Hello name={this.state.name} />  
      </div>  
    );  
  }  
}
```

Hello ReactClass!

# ***Estado/State: Function based***

El problema es que **las funciones viven únicamente durante el tiempo que son ejecutadas.**

## Function based components

componentes basados en funciones

```
function App() {  
  const state = 'Esto morirá al finalizar la función :(';  
  
  return <p>{state}</p>  
}  
  
render(<App />, document.getElementById('root'));
```

Esto deriva de la manera en la que ocurren las cosas en JS

Al terminar la ejecución de **addOne(num)**, **a** y **b** serán puestas a disposición del **garbage collector**

## Function based components

componentes basados en funciones

JavaScript ▾

Console

2

2

>

```
function addOne(number) {  
  let a = Number(number);  
  let b = 1 + a;  
  return b;  
}  
  
let number = 1;  
console.log(addOne(number)); // 2  
console.log(addOne(number)); // 2
```



Todas las constantes o variables que declare para “intentar” mantener el estado, morirán y serán reiniciadas en cada render

## **Function** based components

componentes basados en funciones

```
function App() {  
  const state = 'Esto morirá al finalizar la función :(';  
  
  return <p>{state}</p>  
}  
render(<App />, document.getElementById('root'));
```

**Cada evento** que ocurra cumpliendo ciertas características invocará el completo de la función una vez por cada re-render

***State: ???***

Ok, ya entendí

¿Entonces dónde guardamos  
nuestro estado?



# ***State Hook***

# State hook

Antes

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

class ClassApp extends Component {
  constructor() {
    super();
    this.state = {
      name: 'ReactClass'
    };
  }

  updateName = () => {
    this.setState({ name: 'ReactFunction' });
  }

  render() {
    return (
      <div onClick={this.updateName}>
        <Hello name={this.state.name} />
      </div>
    );
  }
}

render(<ClassApp />, document.getElementById('root'));
```

Hello ReactClass!



```
import React, { Component, useState } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function FnApp() {
  const [name, setName] = useState('ReactClass');
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}
```

Simplificado con hooks

# ***State hook: Estructura básica***

Se usan de la siguiente manera:

```
useState([valorInicial])
```

Devuelven un array:

[0] => valor (ref)

[1] => setName (fn)

```
import React, { Component, useState } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function App() {
  const [name, setName] = useState('ReactClass');
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}

render(<App />, document.getElementById('root'));
```

# State hook: Estructura básica

Los declaramos con **spread syntax** para simplificar

Reglas:

- El value es constante
  - No puedo hacer `name = x`
- Se cambia con `setName`
  - `setName('Nuevo valor')`
- No llamar `setName` entre la declaración del hook y el render

```
import React, { Component, useState } from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function App() {
  const [name, setName] = useState('ReactClass');
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}

render(<App />, document.getElementById('root'));
```

# ***Reglas generales de los hooks***

- Deben ejecutarse **SIEMPRE**
- Esto implica que no pueden ser ejecutados dentro de otras estructuras, como IF, FOR, ó **ternary A ? B : C**
- Se ejecutan en orden y nunca en simultáneo



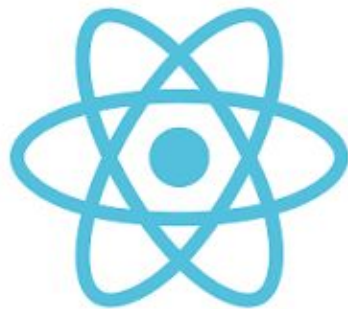
***VAMOS AL CÓDIGO***



***CODER HOUSE***



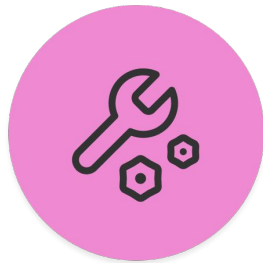
***Resumiendo***



¿Entonces qué correlación hay entre el **render**, las props, el estado y los eventos?

Para saber **qué debe renderizar**, React busca ciertas condiciones específicas:

- Cambio en las **props** `<Title text="newtext"/>`
- Cambio en el **estado**
  - **`this.setState({count: 2})`** / Class based
  - **`setCount(2)`** / Fn + Hooks
- **Eventos:**
  - Al ocurrir eventos, programáticamente modificaremos el estado, lo cual detona los dos primeros puntos



# ***CLICK TRACKER***

Crear en [stackblitz](https://stackblitz.com) un componente que registre qué cantidad de veces lo clickeamos, y lo muestre en pantalla en conjunto con la fecha/hora del último click, usando la librería **Date** de js

*Tiempo: 25 minutos*

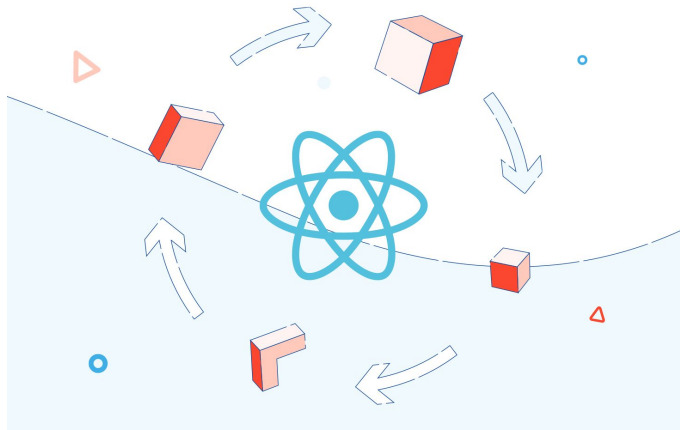
***CODER HOUSE***

***BREAK***



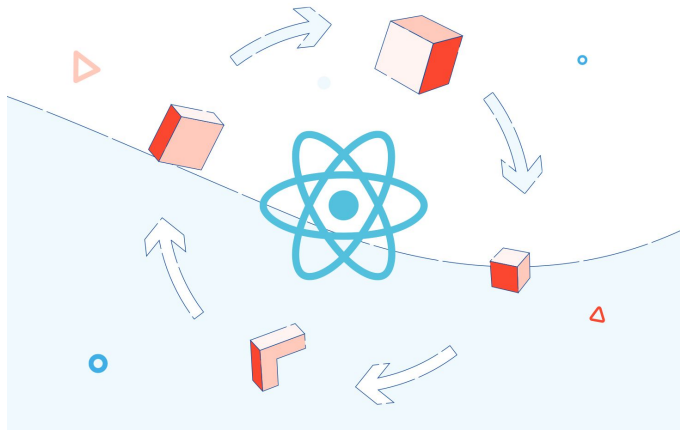
***CODER HOUSE***

# ***CICLOS DE VIDA***



El ciclo de vida no es más que una **serie de estados** por los cuales **pasa todo componente a lo largo de su existencia.**

Esos estados tienen correspondencia en diversos métodos, que podemos implementar para realizar acciones cuando se van produciendo.



En React es fundamental el ciclo de vida, porque hay determinadas **acciones que necesariamente debemos realizar en el momento correcto de ese ciclo.**

Conocer estos ciclos nos ayudará a optimizar la aplicación, siguiendo las reglas básicas que pone React



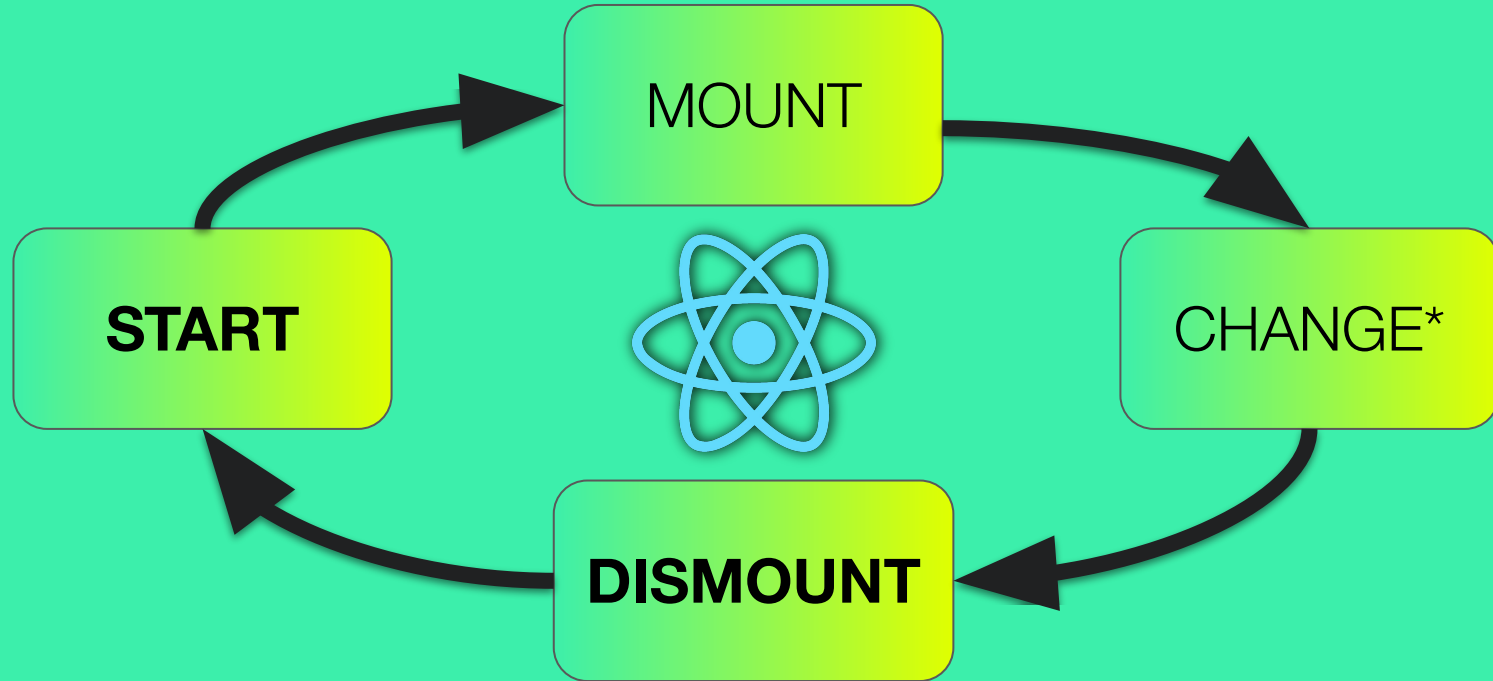
Hay más reglas pero por ahora tengamos en mente las más básicas:

1. No bloquear el rendering con tareas pesadas y sincrónicas
2. Ejecutar tareas asincrónicas con **efectos** secundarios luego del montaje (**mount**)

# ***Las tres clasificaciones de estados dentro de un ciclo de vida***

- El **montaje** se produce la primera vez que un componente va a generarse, incluyéndose en el DOM.
- La **actualización** se produce cuando el componente ya generado se está actualizando.
- El **desmontaje** se produce cuando el componente se elimina del DOM.

# Resumiendo



\*El hijo tendrá la posibilidad de cambiar todas las veces que quiera hasta que el componente que lo generó lo destruya

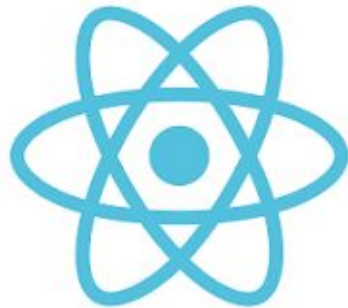
Además, dependiendo del estado actual de un componente y lo que está ocurriendo con él, se producirán **grupos diferentes de etapas del ciclo de vida.**

En la siguiente imagen puedes ver un resumen de esta diferenciación.

Primer renderizado	Cambios en las propiedades	Cambio en el estado	Componente se desmonta
getDefaultProps *	componentWillReceiveProps <small>DEPRECADO</small>	shouldComponentUpdate	componentWillUnmount
getInitialState *	shouldComponentUpdate	componentWillUpdate	
componentWillMount <small>DEPRECADO</small>	componentWillUpdate <small>DEPRECADO</small>	render *	
render *	render *	componentDidUpdate	
componentDidMount	componentDidUpdate		

Etapas del ciclo de vida

# ***Métodos de ciclos de vida ( class based )***



Si bien hoy en día con **componentes funcionales** tendremos reemplazos para varios de los **lifecycles**, a continuación encontrarás una referencia para que los conozcas, con la **consideración** de que en **React 17.x** [serán deprecados](#):

- `componentWillMount`
- `componentWillReceiveProps`
- `componentWillUpdate`

## ***componentWillMount()\****

**DEPRECADO**

Este método del ciclo de vida es de **tipo montaje**.

Se **ejecuta justo antes del primer renderizado** del componente.

Si dentro de este método seteas el estado del componente con `setState()`, el primer renderizado mostrará ya el dato actualizado y el componente se renderizará sólo una vez .



# ***componentDidMount()***

Método de montaje, que **solo se ejecuta en el lado del cliente**. Se produce **inmediatamente después del primer renderizado**.

Una vez se invoca este método ya están disponibles los elementos asociados al componente en el DOM.

Si se tiene que realizar llamadas Ajax, setInterval, y similares, éste es el sitio adecuado.

# ***componentWillReceiveProps***

DEPRECADO

**Método de actualización** que se invoca cuando las **propiedades** se van a **actualizar**, aunque **no** en el **primer renderizado** del componente, por lo tanto no se invocará antes de inicializar las propiedades por primera vez.

Tiene como particularidad que **recibe el valor futuro del objeto de propiedades que tendrá**.

El valor anterior es el que está todavía en el componente, pues este método se invocará antes de que esos cambios se hayan producido.

# ***shouldComponentUpdate (nextProps, nextState)***

Es un **método de actualización** y tiene una particularidad especial con respecto a otros métodos del ciclo de vida, que consiste en que **debe devolver un valor booleano**.

Se invocará tanto cuando se producen **cambios de propiedades** o **cambios de estado** y es una oportunidad de decirle a react si queremos que actualice la vista

## ***componentWillUpdate (nextProps, nextState)***

DEPRECADO

Este **método de actualización** se invocará justo **antes de que el componente vaya a actualizar su vista**.

Es indicado para **tareas de preparación** de esa inminente renderización causada por una actualización de propiedades o estado.

## ***componentDidUpdate (prevProps, prevState)***

**Método de actualización** que se ejecuta justamente **después de haberse producido la actualización del componente.**

En este paso los cambios ya están trasladados al DOM del navegador, así que podríamos operar con el DOM para hacer nuevos cambios.

Como **parámetros** en este caso **recibes el valor anterior de las propiedades y el estado.**

# ***componentWillUnmount()***

Este es el único **método de desmontado** y se ejecuta en el momento que el **componente se va a retirar del DOM**.

Este método es muy importante, porque es el momento en el que se debe realizar una **limpieza** de cualquier cosa que tuviese el componente y **que no deba seguir existiendo** cuando se retire de la página.

# ***Métodos de ciclos de vida ( function based )***

# Hook de efecto/useEffect

El hook de efecto sirve para:

1. controlar el ciclo de vida
2. mutaciones (props, estado)

Piénsalo como un filtro:

`useEffect(fn, filter)`

```
import React, { Component, useState, useEffect }
from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function App() {
  const [name, setName] = useState('ReactClass');
  useEffect(() => {
    console.log('App mounted');
    return () => {
      console.log('Will unmount');
    }
  }, []);
  console.log('Will render');
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}
```

Hello ReactClass!

Console



☒ Clear console on reload

Console was cleared

Will render

App mounted



# Hook de efecto/useEffect

Si queremos reemplazar el  
lifecycle  
componentDidMount()  
podemos utilizar el hook de  
efecto con el mismo  
resultado  
[ ] => On mount

```
import React, { Component, useState, useEffect }  
from 'react';  
import { render } from 'react-dom';  
import Hello from './Hello';  
  
function App() {  
  const [name, setName] = useState('ReactClass');  
  useEffect(() => {  
    console.log('App mounted');  
    return () => {  
      console.log('Will unmount');  
    }  
  }, []);  
  console.log('Will render')  
  return (  
    <div onClick={() => setName('ReactFunction')}>  
      <Hello name={name} />  
    </div>  
  );  
}
```

Hello ReactClass!

Console

☒ Clear console on reload

Console was cleared

Will render

App mounted

# ***useEffect: variantes/filtros***

Variantes/filtros:

[ ] => On mount

[prop] => On mount y por cada cambio de prop

[prop1, prop2] => On mount y en cada cambio en  
prop1 o prop2

undefined => **useEffect**( ()=>{ }) => Mount y en  
cada render

***VAMOS AL CÓDIGO***



***CODER HOUSE***

# *useEffect- Variantes*

[ ] => On mount

[prop] => On mount y por cada cambio de name

[prop1, prop2] => On mount y en cada cambio en prop1 ó prop2

```
function App({ defaultName }) {  
  const [name, setName] = useState('ReactClass');  
  useEffect(() => {  
    console.log('App mounted');  
    return () => {  
      console.log('Will unmount');  
    }  
  }, [name]);  
  useEffect(() => {  
    console.log('Received prop ', defaultName)  
    return () => {  
      console.log('Will receive new prop: name');  
    }  
  }, [defaultName]);  
  console.log('Will render')  
  return (  
    <div onClick={() => setName('ReactFunction')}>  
      <Hello name={name} />  
    </div>  
  );  
}  
  
render(<App defaultName="otherName" />,  
document.getElementById('root'));
```

Console

☒ Clear console on reload

Console was cleared

Will render

App mounted

Received prop otherName

>

# *useEffect: Cleanup*

Si devuelves una función

`return () => {}`

se ejecutará el clean que  
quieras (ajax call, remove  
una suscripción, librería, etc)

```
import React, { Component, useState, useEffect }
from 'react';
import { render } from 'react-dom';
import Hello from './Hello';

function App() {
  const [name, setName] = useState('ReactClass');
  useEffect(() => {
    console.log('App mounted');
    return () => {
      console.log('Will unmount');
    }
  }, []);
  console.log('Will render')
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}
```

## Hello ReactClass!

Console

☒ Clear console on reload

Console was cleared

Will render

App mounted

# ***IMPORTANTE***

Tanto los **callbacks** como los **cleanups**:

- Se ejecutan **en el orden en que se hayan declarado** los otros hooks respectivos
- Recuerda que **la función se destruye en cada ejecución**, si tienes actividad pendiente hay que cerrarla en cada cleanup y volver a suscribirla

# ***Comportamiento simétrico***

Los hooks se comportan simétricamente tanto con los valores observados **props** como con el **state**

Acción => Limpieza => Acción => Limpieza  
y nunca

Acción => Acción => Acción => Limpieza

Cualquier acción en un **effect** tiene una acción opuesta de limpieza, que será ejecutada antes de poder volver a ejecutar la acción.

# ***useEffect: Ejemplos/Cheatsheet***

Si declaro useEffect(() => { //Accion; return <b>cleanup-fn</b> })	Si mi acción se ejecuta el montado y <b>en cada render</b> , mi limpieza se ejecuta <b>en cada render</b> .
Si declaro useEffect(() => { return <b>cleanup-fn</b> }, [])	Si mi acción se realiza <b>al montar</b> , la limpieza será <b>únicamente al desmontar</b> el componente
Si declaro useEffect(() => { return <b>cleanup-fn</b> }, [prop])	Mi acción se realizará al montar, y antes del próximo cambio de prop se hará una limpieza y recién ahí se ejecutará la acción

- Toda acción del effect-hook se ejecuta al montar
- Ningún efecto bloquea el render
- Todas las acciones y limpiezas se realizan en orden
- Si modifico el state incluido en los filtros propios habrá un loop infinito



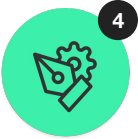


## ***Contador con botón***

Crearás un componente ItemCount para incrementar o decrementar los productos a añadir en el carrito.

*Formato de entrega: carpeta comprimida con los archivos del proyecto*

*Tiempo: 30 minutos*



# Contador con botón

Crea un componente `<ItemCount initial="" min="" max="" onAdd="">` que tenga un botón y controles para **incrementar** (+) y **decrementar** (-) el **count inicial** pero nunca irse de los **límites** min/max.

Al clicar el botón debe invocar el callback de `onAdd(count)` pasando el count del counter

Camisa tiger

- 1 +

Agregar al carrito

*No es necesario usar este estilo, sirve a modo de orientación*



## ***RECURSOS DE LA CLASE***

- <https://es.reactjs.org/docs/react-component.html>
- <https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html>

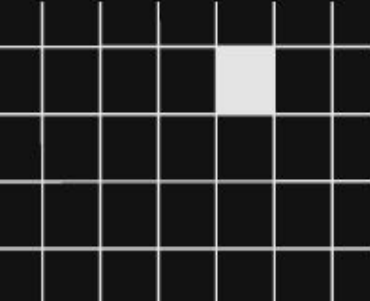
***¿PREGUNTAS?***





# ***¡MUCHAS GRACIAS!***

Resumen de lo visto en clase hoy:

- Propiedades, Estados y Eventos
  - Ciclo de vida
- 



***OPINA Y VALORA ESTA CLASE***