

***RECUERDA PONER A GRABAR LA
CLASE***





¿DUDAS DEL ON-BOARDING?

MIRALO AQUI



Clase 05. REACT JS

COMPONENTES II



OBJETIVOS DE LA CLASE

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

GLOSARIO:

Clase 4

Componentes: básicamente, las aplicaciones en React básicamente se construyen mediante los mismos. permiten separar la interfaz de usuario en piezas independientes, reutilizables y pensar en cada pieza de forma aislada.

Propiedades: son la forma que tiene React para pasar parámetros de un componente superior a sus children.

Componentes de presentación: son aquellos que simplemente se limitan a mostrar datos y tienen poca o nula lógica asociada a manipulación del estado (por eso son también llamados stateless components).

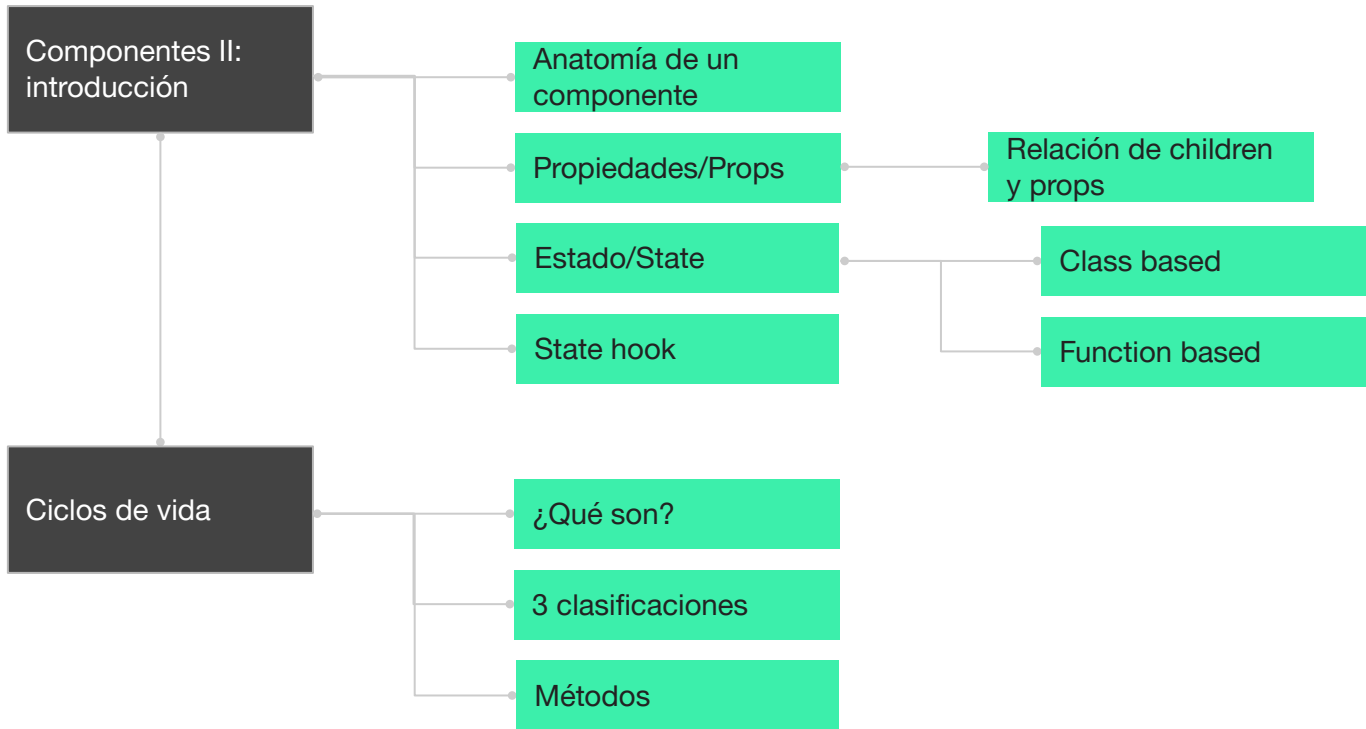
Componentes contenedores: tienen como propósito encapsular a otros componentes y proporcionarles las propiedades que necesitan. Además se encargan de modificar el estado de la aplicación para que el usuario vea el cambio en los datos (por eso son también llamados state components).

Children: es una manera que tiene react de permitirnos proyectar/transcluir uno o más componentes dentro otro.

MAPA DE CONCEPTOS

MAPA DE CONCEPTOS CLASE 5

¡Para
recordar!



CRONOGRAMA DEL CURSO

Clase 4



Componentes I



EJEMPLOS EN VIVO



CREA TU LANDING

Clase 5



Componentes II



EJEMPLOS EN VIVO



CLICK TRACKER



CONTADOR CON BOTÓN

Clase 6



Promises, asincronía y MAP



EJEMPLOS EN VIVO



MOCK ASYNC SERVICE



CATÁLOGO CON MAPS Y PROMISES

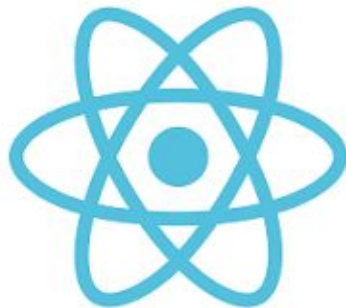
COMPONENTES II:- INTRODUCCIÓN

ANATOMÍA DE UN COMPONENTE

ANATOMÍA DE UN COMPONENTE

Props

State



DOM Sync

Lifecycle

PROPIEDADES/PROPS

CODER HOUSE

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

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!

Console



☒ Clear console on reload

~~Console was cleared~~

▶ {title: "Hey!"}

CODER HOUSE

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.

RENDER PROPS

Si pasamos un componente por prop... Podemos usarlo en otro

componente sin que, como en este caso, **SuperForm** sepa realmente la implementación del render 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}

>

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

ESTADO/STATE: CLASS BASED

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!

CLASS BASED

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!

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.

ESTADO/STATE: FUNCTION BASED

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'));
```

FUNCTION BASED

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 ▾

```
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
```

Console

2

2

>

FUNCTION BASED

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!

hook

```
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])`

Devuelve 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.

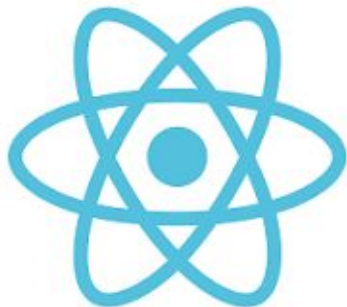
VAMOS AL CÓDIGO



CODER HOUSE

RESUMIENDO

CODER HOUSE



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

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

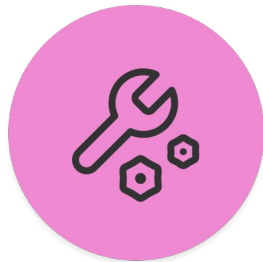
- Cambio en las **props** `<Title text="newtext"/>`

- 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

Crea en **stackblitz** un componente que registre qué cantidad de veces lo clickeamos.



¡A PRACTICAR!

Crea 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.

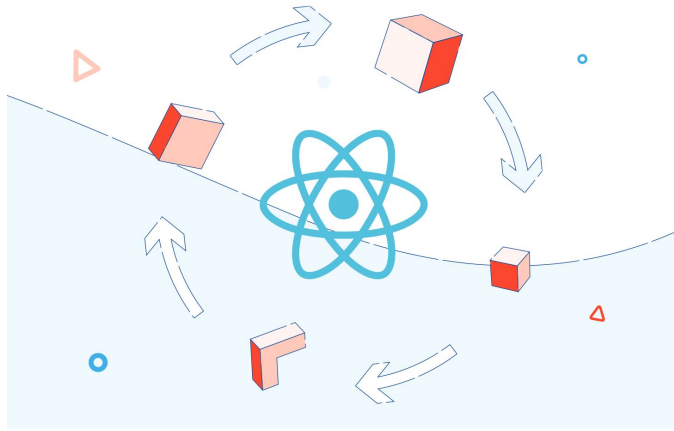
Cuentas con 25 minutos para hacer la actividad.



BREAK

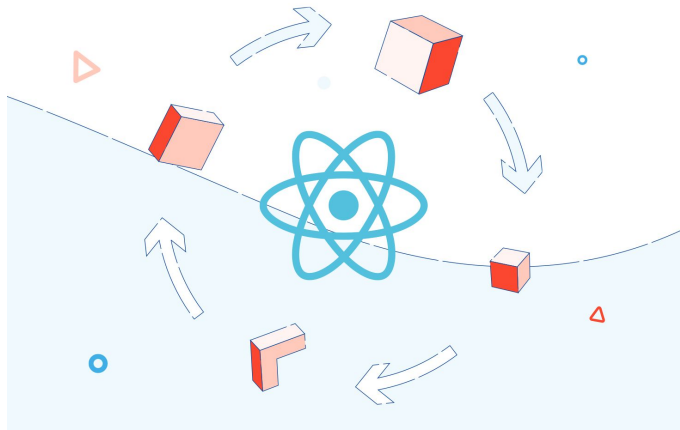
¡5/10 MINUTOS Y VOLVEMOS!

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

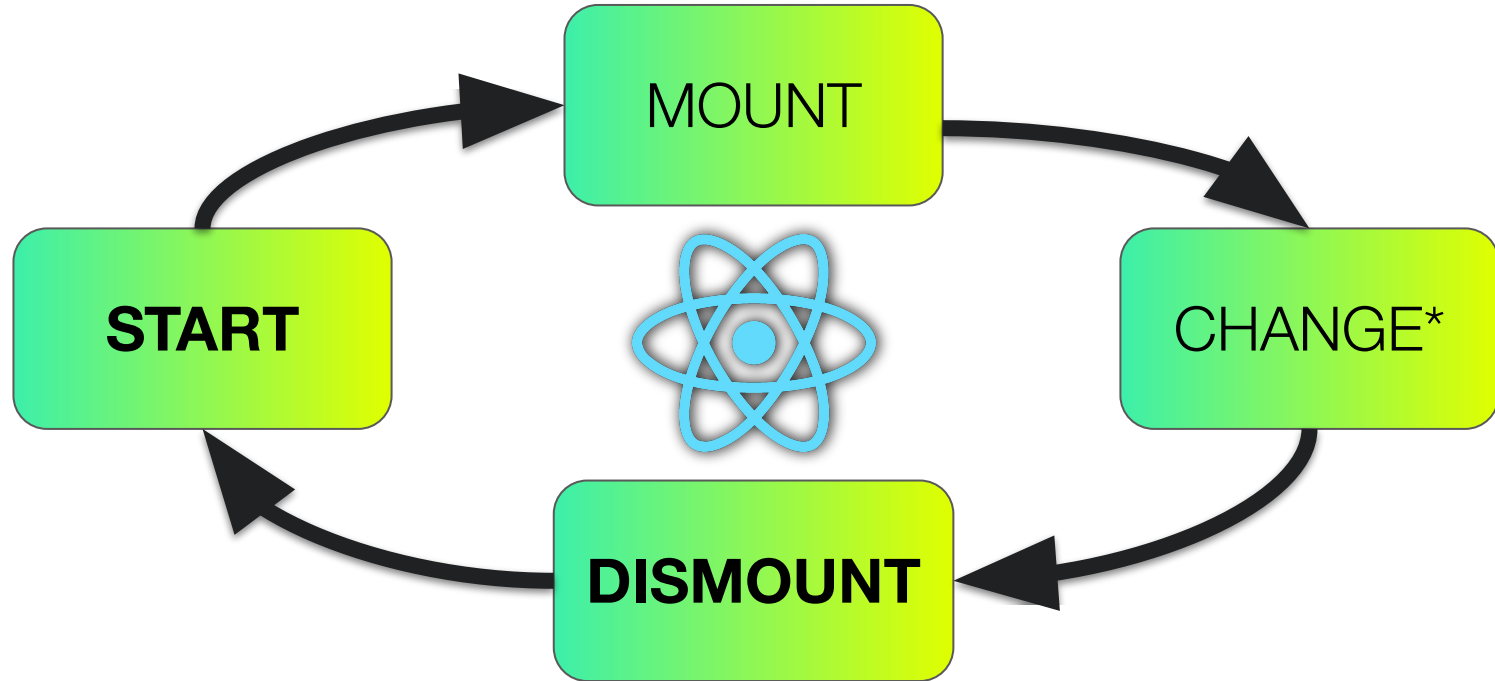
Hay más reglas pero por ahora tengamos en mente las más básicas:

- No bloquear el rendering con tareas pesadas y sincrónicas.
- 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.

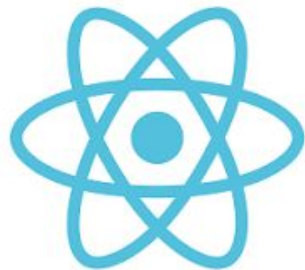
Además, dependiendo del estado actual de un componente y lo que está ocurriendo con él, se producirán **grupos diferentes de etapas del ciclo de vida.**

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

Etapas del ciclo de vida

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		

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

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.

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.

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'); 2
    return () => {
      console.log('Will unmount');
    }
  }, []);
  console.log('Will render') 1
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}
```

Hello ReactClass!

Console

☒ Clear console on reload

Console was cleared

Will render

App mounted

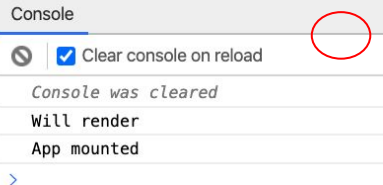
HOOK DE EFECTO

useEffect

```
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'); 2
    return () => {
      console.log('Will unmount');
    }
  }, []);
  console.log('Will render') 1
  return (
    <div onClick={() => setName('ReactFunction')}>
      <Hello name={name} />
    </div>
  );
}
```

Hello ReactClass!



Si queremos reemplazar el lifecycle `componentDidMount()` podemos utilizar el hook de efecto con el mismo resultado

[] => On mount

VARIANTES/FILTROS

useEffect

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

VARIANTES useEffect

[] => 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

>

CLEANUP useEffect

Si devuelves una función `return`
`() => {}`

se ejecutará el clean que quieras
(ajax call, remover 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.

EJEMPLOS/CHEATSHEET

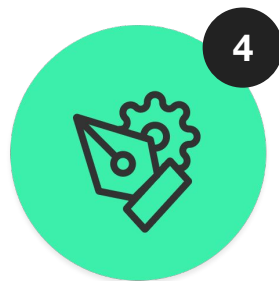
useEffect

Si declaro useEffect(() => { //Accion; return cleanup-fn })	Si mi acción se ejecuta el montado y en cada render , mi limpieza se ejecuta en cada render .
Si declaro useEffect(() => { return cleanup-fn }, [])	Si mi acción se realiza al montar , la limpieza será únicamente al desmontar el componente
Si declaro useEffect(() => { return cleanup-fn }, [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

EJEMPLOS/CHEATSHEET

useEffect

- 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.



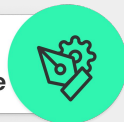
CONTADOR CON BOTÓN

Crear el componente **ItemCount**, para incrementar o decrementar los productos a añadir en el carrito

CONTADOR CON BOTÓN

Formato: link al último commit de tu repositorio en GitHub. Debe tener el nombre "Idea+Apellido".

Desafío
entregable



>> **Consigna:** crea un componente `ItemCount.js`, que debe estar compuesto de un botón y controles, para incrementar y decrementar la cantidad requerida de ítems.

>> **Aspectos a incluir en el entregable:**

Componente **`ItemCount.js`** con los respectivos controles de la consigna.

>> **Ejemplo:**

Camisa tiger

-

1

+

Agregar al carrito

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

CODER HOUSE

CONTADOR CON BOTÓN

Formato: link al último commit de tu repositorio en GitHub.

Desafío
entregable



>> A tener en cuenta:

- El número contador nunca puede superar el stock disponible
- De no haber stock el click no debe tener efecto y por ende no ejecutar el callback onAdd
- Si hay stock al clickear el botón se debe ejecutar onAdd con un número que debe ser la cantidad seleccionada por el usuario.

Detalle importante: como sabes, todavía no tenemos nuestro detalle de ítem, y este desarrollo es parte de él, así que por el momento puedes probar e importar este componente dentro del **ItemListContainer**, sólo a propósitos de prueba. Después lo sacaremos de aquí y lo incluiremos en el detalle del ítem.

CONTADOR CON BOTÓN

Formato: link al último commit de tu repositorio en GitHub.

Desafío
entregable



>> Ejemplo inicial:

```
function ItemCount({ stock, initial,  onAdd }) {  
  // Desarrollar lógica  
}
```

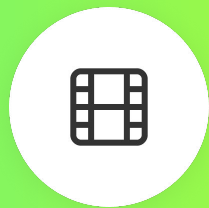
- Adicionalmente tendremos un número inicial (initial) de cantidad de ítems, de tal modo que si lo invoco del siguiente modo:

```
<ItemCount stock="5" initial="1" />
```

debería ver el contador inicializado en 1 por defecto

¿PREGUNTAS?





***¿QUIERES SABER MÁS? TE DEJAMOS
MATERIAL AMPLIADO DE LA CLASE***

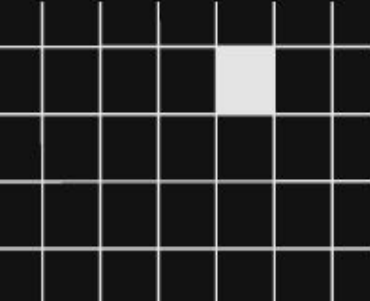


- <https://es.reactjs.org/docs/react-component.html> | **React**
- <https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html>
| **React**



¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Propiedades, Estados y Eventos
 - Ciclo de vida
- 



OPINA Y VALORA ESTA CLASE