

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





¿DUDAS DEL ON-BOARDING?

[MIRALO AQUI](#)



Clase 04. REACT JS

COMPONENTES I



OBJETIVOS DE LA CLASE

- Comprender qué son los componentes y qué problemas resuelven.
- Conocer los tipos de componentes.
- Implementar los componentes vistos.

GLOSARIO:

Clase 3

Sugar Syntax: refiere a la sintaxis agregada a un lenguaje de programación con el objetivo de hacer más fácil y eficiente su utilización. Favorece su escritura, lectura y comprensión.

Webpack: es un module bundler o empaquetador de módulos.

Eject: es una acción permanente, que permite tener un control más específico del bundling, a costa de que de ahora en adelante tendremos que encargarnos de mantenerlo.

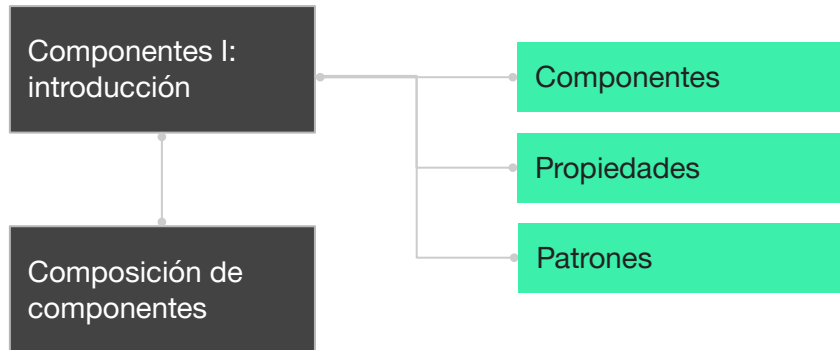
Transpiling: es el proceso de convertir código escrito en un lenguaje, a su representación en otro lenguaje.

JSX: es una extensión de sintaxis de Javascript que se parece a HTML. Oficialmente, es una extensión que permite hacer llamadas a funciones y a construcción de objetos. No es ni una cadena de caracteres, ni HTML.

MAPA DE CONCEPTOS

MAPA DE CONCEPTOS CLASE 5

¡Para
recordar!



CRONOGRAMA DEL CURSO

Clase 3



JSX y Transpiling



EJEMPLOS EN VIVO



MENÚ E-COMMERCE

Clase 4



Componentes I



EJEMPLOS EN VIVO



CREA TU LANDING

Clase 5



Componentes II



EJEMPLOS EN VIVO



CLICK TRACKER

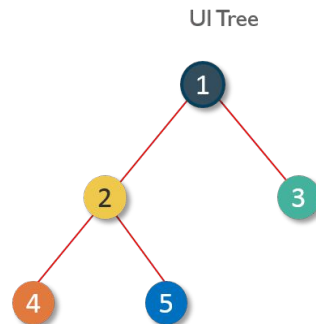
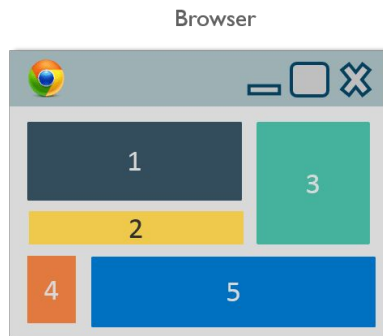


CONTADOR CON BOTÓN

COMPONENTES I:- INTRODUCCIÓN

Las aplicaciones en React se construyen mediante **componentes**.

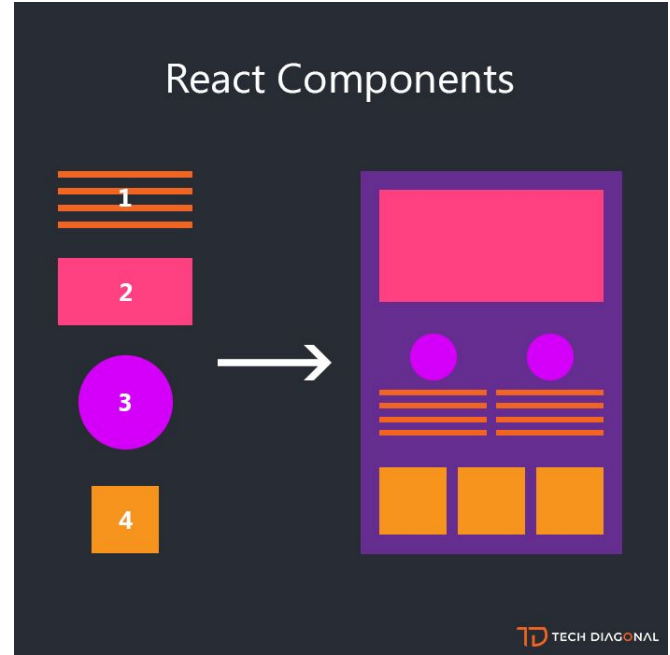
El potencial de este funcionamiento consiste en que podemos crear aplicaciones completas de una manera **modular** y de fácil mantenimiento, a pesar de su complejidad.



DISEÑO MODULAR

Los componentes permiten separar la interfaz de usuario en piezas independientes, reutilizables y pensar en cada pieza de forma aislada.

Al desarrollar crearemos componentes para resolver pequeños problemas, que son fáciles de visualizar y comprender.



DISEÑO MODULAR

Luego, unos componentes se apoyarán en otros para solucionar problemas mayores y al final **la aplicación será un conjunto de componentes que trabajan entre sí.**

Este modelo de trabajo tiene varias ventajas, como la facilidad de mantenimiento, depuración, escalabilidad, etcétera.

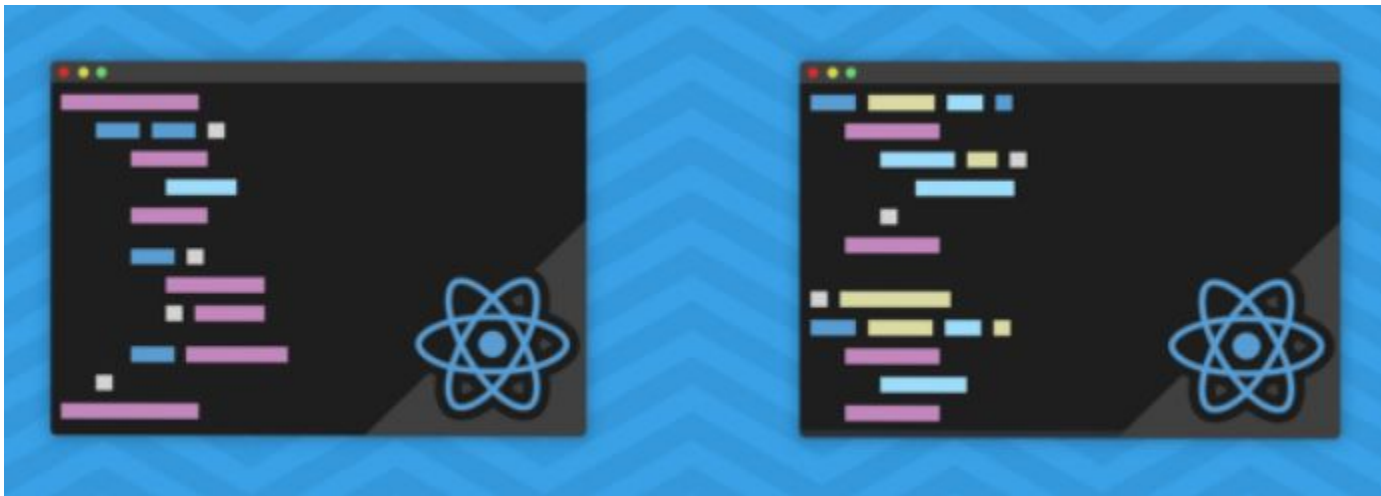
VENTAJAS DEL ENFOQUE

- Favorece la **separación de responsabilidades**: cada componente debe tener una única tarea.
- Al tener la lógica de estado y los elementos visuales por separado, es **más fácil reutilizar** los componentes.
- Se simplifica la tarea de hacer **pruebas unitarias**.
- Puede **mejorar el rendimiento** de la aplicación.
- La aplicación es **más fácil de entender**.

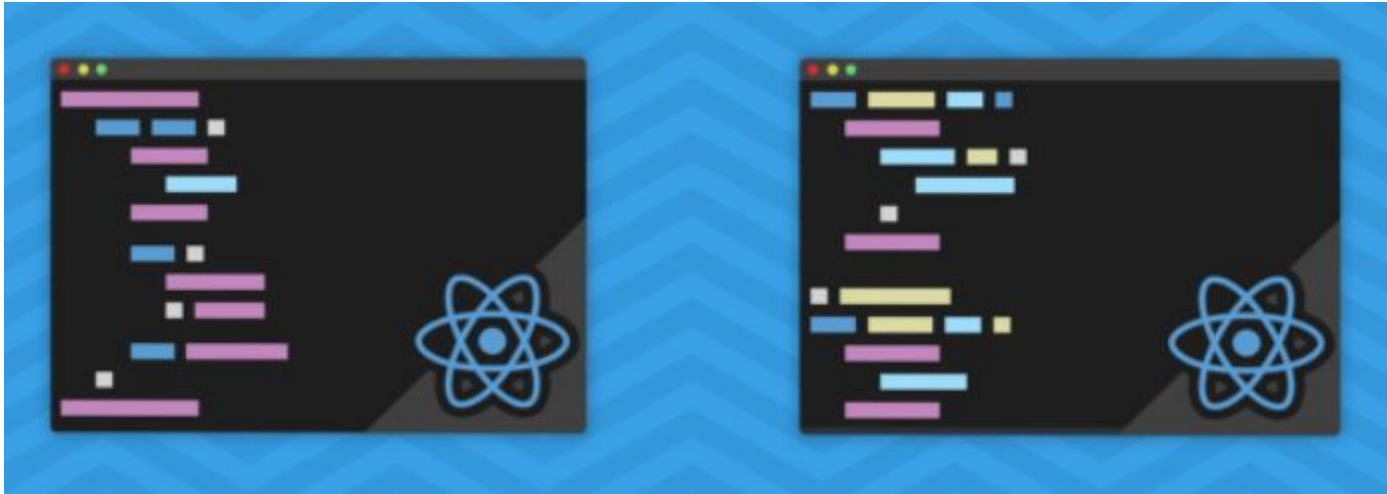
COMPONENTES

En React JS existen dos maneras de entender los componentes, que varían según desde dónde nos paremos para analizarlo.

Vamos a decir que existen **tipos de componentes** y **patrones**.



La confusión se acentúa cuando no somos capaces de identificar las diferencias.



Los dividiremos en estas **dos representaciones**, que después servirán de base para implementar múltiples patrones.

Class based components

componentes basados en clases

```
class App extends Component {  
  render() {  
    return (  
      <p>  
        Class based :)  
      </p>  
    );  
  }  
}  
  
render(<App />, document.getElementById('root'));
```

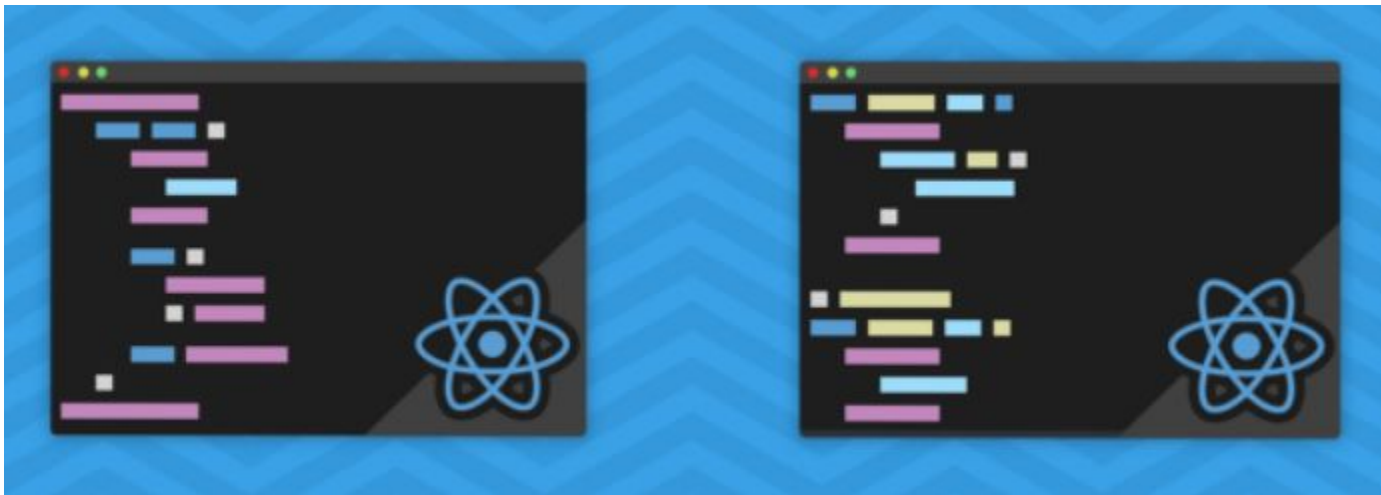
Function based components

componentes basados en funciones

```
const App = () => {  
  return (  
    <p>  
      Function based :)  
    </p>  
  );  
}  
  
render(<App />, document.getElementById('root'));
```

PUNTOS EN COMÚN

- Pueden recibir propiedades (**props**).
- Tienen la capacidad de hacer **render** de un único elemento*.



* Aunque este elemento pueda tener muchos elementos dentro ;).

VAMOS AL CÓDIGO



CODER HOUSE

PROPIEDADES

PROPIEDADES

Las propiedades son la forma que tiene React para **pasar parámetros** de un componente superior a sus **children**.

Es la manera de implementar el **flujo de datos unidireccional**.

Si alguna prop es una función, el **child component** puede llamarla para provocar efectos secundarios en el **parent**.

PROPIEDADES

Class based components
componentes basados en clases

Las propiedades enviadas al componente las recibiremos a través de **this.props** para acceder a un objeto en el cual tendremos todas las propiedades disponibles.

Function based components
componentes basados en funciones

Simplemente se reciben como parámetro de la función

```
( { name } ) => <p>{name}</p>
```

PROPIEDADES

```
const App = (props) => {  
  return (  
    <p>  
      ¡Vamos {props.name}! :)  
    </p>  
  );  
}  
  
render(<App name="coderhouse" />,  
document.getElementById('root'));
```

La propia función es el equivalente al método `render()` que teníamos al crear componentes por medio de una clase ES6. Por lo tanto, **devuelve el JSX para representar el componente.**

Al definir la función se prescinde del método **render**, porque no estamos haciendo una clase.

```
const App = ({ name }) => {  
  return (  
    <p>  
      ¡Vamos {name}! :)  
    </p>  
  );  
}  
  
render(<App name="coderhouse" />,  
document.getElementById('root'));
```

PROPIEDADES

Imaginemos que a nuestro componente le pasamos dos propiedades, llamadas "**nombre**" y "**app**".

Entonces podremos usar esas propiedades de la siguiente manera:

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

export default class FeedbackMessage extends Component {
  render() {
    return (
      <div>
        Bienvenido {this.props.nombre} a {this.props.app}
      </div>
    )
  }
}
```


PROPIEDADES

```
import React, { Component } from 'react'
import FeedbackMessage from './FeedbackMessage'

class App extends Component {
  render() {
    return (
      <div className="App">
        <FeedbackMessage nombre="Leonardo DaVinci" app="Mi App React" />
      </div>
    );
  }
}
```

Este sería un código donde se usa el componente **FeedbackMessage**, indicando los valores de sus props.

Los valores son indicados como atributos del componente "nombre" y "app", que son las props usadas en el ejemplo anterior.

PROPIEDADES

`this.props.nombre` contendrá el valor pasado en la propiedad "nombre" y `this.props.app` el valor de la propiedad "app".

Nuestras propiedades se encuentran encerradas entre llaves { }

Las llaves son importantes, porque es la manera con la que se **escapa un código JSX**, permitiendo colocar dentro sentencias Javascript "nativo".

Aquello que devuelvan esas sentencias se volcará como contenido en la vista.

PATRONES

CODER HOUSE

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 DE PRESENTACIÓN

- Orientados al **aspecto visual**.
- No tienen dependencia con fuentes de datos (ej. Flux).
- **Reciben callbacks** por medio de props.
- Pueden ser descritos como **componentes funcionales**.
- Normalmente **no tienen estado**.



EJEMPLO DE COMPONENTE DE PRESENTACIÓN (CLASS BASED)

```
class Item extends React.Component {
  render () {
    return (
      <li><a href='#>{this.props.valor}</a></li>
    );
  }
}

class Input extends React.Component {
  render () {
    return (
      <input type='text' placeholder={this.props.valor}/>
    );
  }
}

class Titulo extends React.Component {
  render () {
    return (
      <h1>{this.props.valor}</h1>
    );
  }
}
```

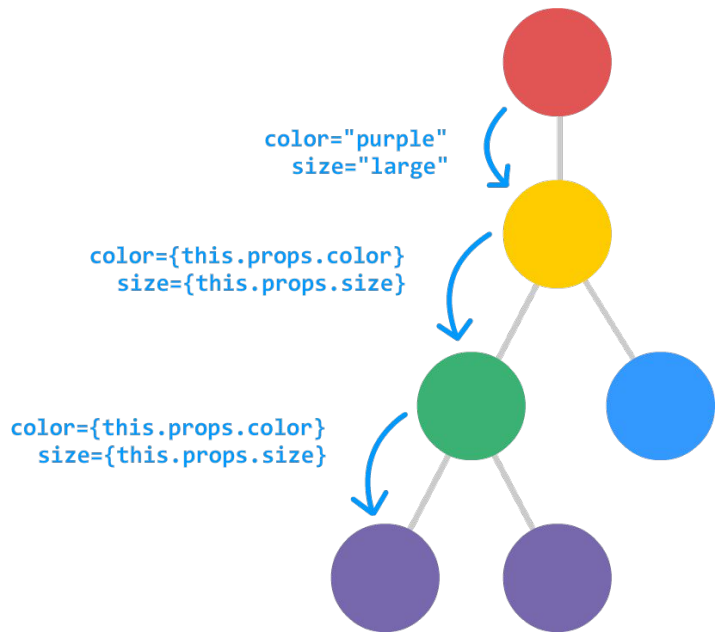
En este fragmento de código definimos algunos componentes de presentación (Item, Input y Header) que son mostrados en la página dentro de un componente contenedor.

COMPONENTES DE PRESENTACIÓN

Los componentes de presentación usualmente no tienen estado, por eso hace más sentido utilizar más simplemente **function** based componentes.

Todo componente puede recibir de su **parent** (superior), **props** y **children**.

(Aunque no sea obligatorio).



COMPONENTES DE PRESENTACIÓN

Usando esta sintaxis, **las propiedades se reciben como parámetros de la función**, y podemos obtener las variables que nos interesan por separado.

```
const Titulo = ({nombre}) => (  
  <h1>{ nombre }</h1>  
);  
  
const Item = (props) => (  
  <li><a href='#'>{ props.valor }</a></li>  
);  
  
const Input = (props) => (  
  <input type='text' placeholder={ props.placeholder} />  
);
```


VAMOS AL CÓDIGO



CODER HOUSE



La ventaja más evidente de estos componentes es la **posibilidad de reutilizarlos** siempre que queramos, sin tener que recurrir a escribir el mismo código una y otra vez.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

COMPOSICIÓN DE COMPONENTES

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).

COMPONENTES CONTENEDORES

- **Orientados al funcionamiento** de la aplicación.
- Contienen componentes de presentación y/u otros contenedores.
- Se **comunican con las fuentes de datos**.
- Usualmente **tienen estado** para representar el cambio en los datos.



EJEMPLO DE COMPONENTE CONTENEDOR

```
class AppContainer extends React.Component {
  constructor (props) {
    super(props);
    this.state = {
      temas: ['JavaScript', 'React JS', 'Componentes']
    };
  }

  render () {
    const items = this.state.temas.map(t => (
      <Item valor={ t } />
    ));
    return (
      <div>
        <Titulo nombre='List Items' />
        <ul>{ items }</ul>
        <Titulo nombre='Inputs' />
        <div>
          <Input placeholder='Nombre' />
          <Input placeholder='Apellido' />
        </div>
      </div>
    );
  }
}
```

El componente contenedor **define los datos contenidos** en la aplicación y también los manipula, creando luego los componentes hijos y mostrándolos con en el método render.

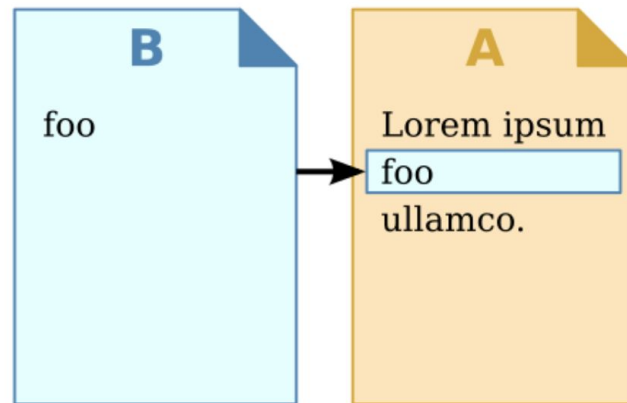
Este tipo de componentes será el encargado de realizar llamadas a las API's externas y/o establecer la lógica a realizar, en función de las acciones que realice el usuario sobre la interfaz.

CHILDREN

CHILDREN

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

```
<Component>  
<ChildComponent/>  
</Component>
```



CHILDREN

Es ideal cuando:

- Necesitamos que un elemento quede dentro de otro, **sin que sepan** el uno del otro.
- Necesitamos implementar patrones más complejos.

```
const Ad = () =>  
  <p style={{ backgroundColor: 'gray' }}>  
    Sponsored by React Team  
  </p>;  
  
const App = ({ children }) => {  
  return (  
    <>  
      <p>  
        ¡Vamos {name}! :)  
      </p>  
      {children}  
    </>  
  );  
}  
  
render(<App name="coderhouse"><Ad /></App>,  
  document.getElementById('root'));
```

¿PREGUNTAS?





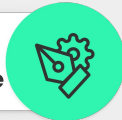
CREA TU LANDING

Crea tu **CartWidget** y tu **ItemListContainer** como los primeros componentes que serán la cara de tu e-commerce, siguiendo las especificaciones del manual.

CREA TU LANDING

Formato: link a último commit en GitHub. Debe tener el nombre “Idea+Apellido”.

Desafío
entregable



>> Consigna:

1. Crea un componente CartWidget.js con un ícono, y ubícalo en el navbar. Agrega algunos estilos con bootstrap/materialize u otro.
2. Crea un componente contenedor ItemListContainer.js con una prop greeting, y muestra el mensaje dentro del contenedor con el styling integrado.

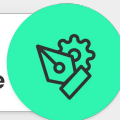
>>Aspectos a incluir en el entregable:

- Parte 1: crea un componente CartWidget.js que haga rendering de un **ícono Cart**, e inclúyelo dentro de NavBar.js para que esté visible en todo momento.
- Parte 2: crea un componente ItemListContainer. Impórtalo dentro de App.js, y abajo de NavBar.js.

CREA TU LANDING

Formato: link al último commit en GitHub.

Desafío
entregable



>> Ejemplo inicial:

1)

```
function NavBar() {  
  return <>  
    // Customiza tu NavBar como prefieras  
    <h3>TU_MARCA</h3>  
    <CartWidget />  
  </>;  
}  
  
</NavBar>
```

(2)

```
function ItemListContainer() {  
  // Incluye aquí el rendering de algún texto o título provisional que luego  
  reemplazaremos por nuestro catálogo  
}
```

CODER HOUSE

CREA TU LANDING

Formato: link al último commit en GitHub.

Desafío
entregable



>> Ejemplo



CODER HOUSE

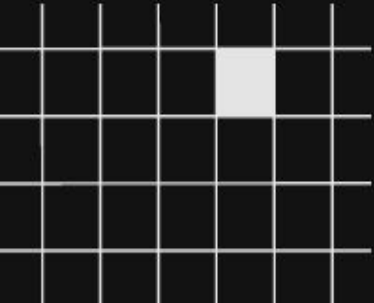
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Introducción y tipos de componentes
 - Children
 - Props
- 



OPINA Y VALORA ESTA CLASE