



Clase 04. React JS

# ***COMPONENTES I***



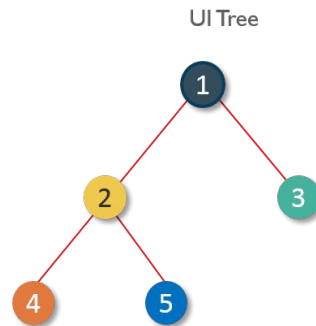
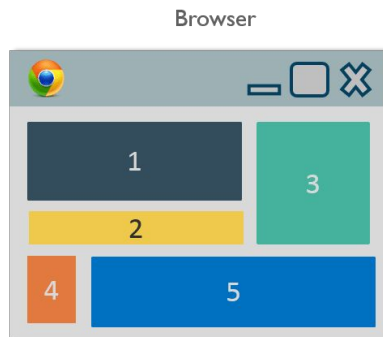
## ***OBJETIVOS DE LA CLASE***

- Comprender qué problemas resuelven los componentes
  - Conocer los tipos de componentes
  - Aprender a implementarlos

# ***COMPONENTES I:- INTRODUCCIÓN***

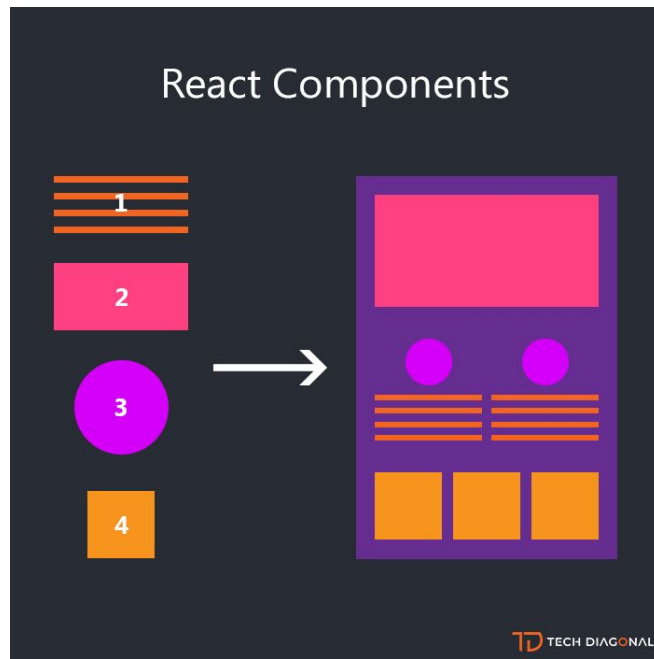
Las aplicaciones en React básicamente se construyen mediante **componentes**.

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



# ***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 resolver, visualizar y comprender.

Luego, unos componentes se apoyarán en otros para resolver 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, etc.



# ***Ventajas del enfoque***

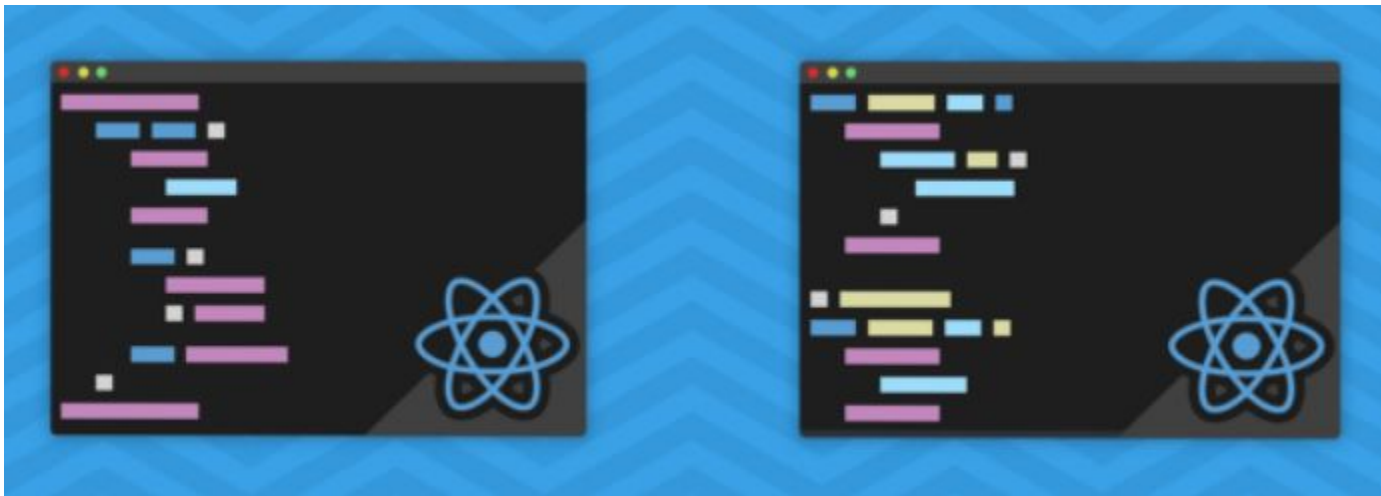
- Favorece la **separación de responsabilidades**: cada componente debe tener una única tarea.
- Al tener la lógica de estado y los elementos visuales por separado, es **más fácil reutilizar** los componentes.
- Se simplifica la tarea de hacer **pruebas unitarias**.
- Puede **mejorar el rendimiento**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

# ***Componentes***

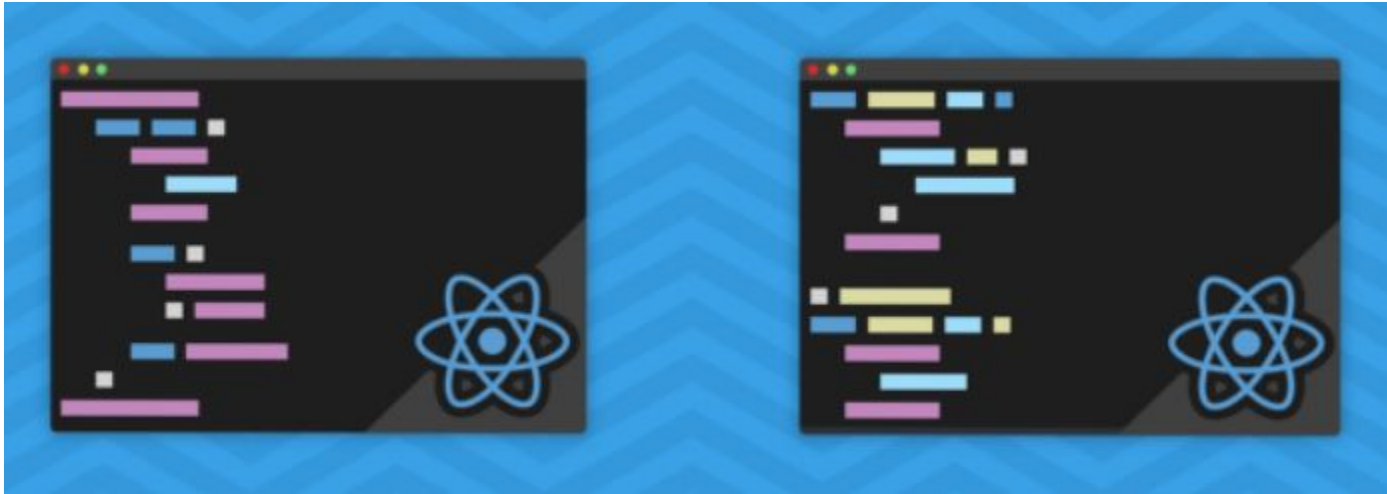


En React JS existen dos maneras de entender los componentes, que varían según desde dónde nos paremos para analizarlo.

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***

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>
```

```
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'));
```



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

Entonces podremos usar esas propiedades de la siguiente manera:

```
112 import React, { Component } from 'react';
113
114
115
116 export default class FeedbackMessage extends Component {
117   render() {
118     return (
119       <div>
120         Bienvenido {this.props.nombre} a {this.props.app}
121       </div>
122     )
123   }
124 }
125
```

# ***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***

## ***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)***

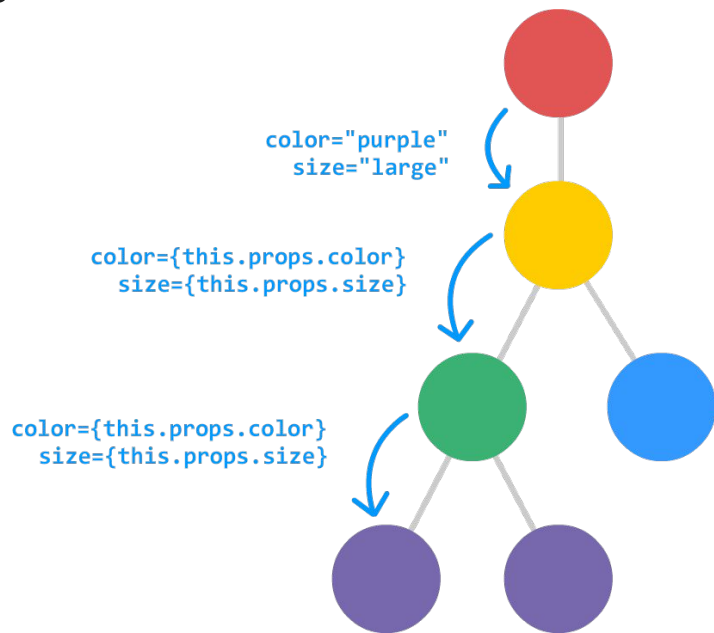
```
136
137 class Item extends React.Component {
138   render () {
139     return (
140       <li><a href='#'>{this.props.valor}</a></li>
141     );
142   }
143 }
144
145 class Input extends React.Component {
146   render () {
147     return (
148       <input type='text' placeholder={this.props.valor}/>
149     );
150   }
151 }
152
153 class Titulo extends React.Component {
154   render () {
155     return (
156       <h1>{this.props.valor}</h1>
157     );
158   }
159 }
160
```

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.

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)



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

```
97
98  const Titulo = ({nombre} = props) => (
99    <h1>{ nombre }</h1>
100  );
101
102  const Item = (props) => (
103    <li><a href='#'>{ props.valor }</a></li>
104  );
105
106  const Input = (props) => (
107    <input type='text' placeholder={ props.placeholder} />
108  );
109
```



***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.

***¿PREGUNTAS?***



***BREAK***



***CODER HOUSE***

# ***COMPOSICIÓN DE COMPONENTES***

# ***Componentes contenedores***



## ***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***

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.

```
136
137 class AppContainer extends React.Component {
138   constructor (props) {
139     super(props);
140     this.state = {
141       temas: ['JavaScript', 'React JS', 'Componentes']
142     };
143   }
144
145   render () {
146     const items = this.state.temas.map(t => (
147       <Item valor={ t } />
148     ));
149     return (
150       <div>
151         <Titulo nombre='List Items' />
152         <ul>{ items }</ul>
153         <Titulo nombre='Inputs' />
154         <div>
155           <Input placeholder='Nombre' />
156           <Input placeholder='Apellido' />
157         </div>
158       </div>
159     );
160   }
161 }
162
```

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.

```
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.

```
const App = (props) => {
  return (
    <p>
      ¡Vamos {props.name}! :)
    </p>
  );
}

render(<App name="coderhouse" />,
document.getElementById('root'));
```

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

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.

```
const App = ({ name }) => {
  return (
    <p>
      ¡Vamos {name}! :)
    </p>
  );
}

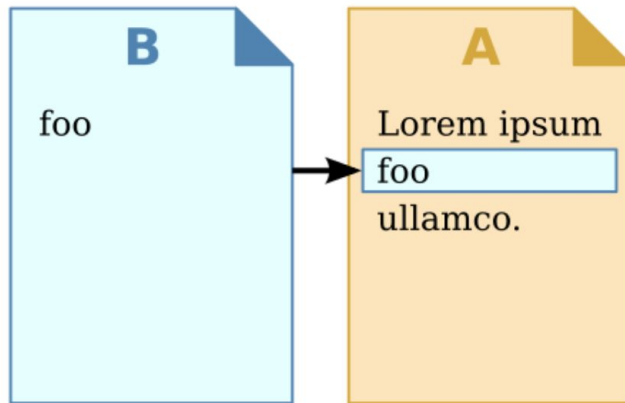
render(<App name="coderhouse" />,
document.getElementById('root'));
```

***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?***







## ***Estilos y Home***

1. Crear componente CartIcon.js con un ícono y ubicarlo en el navbar.  
Agregar algunos estilos con bootstrap/materialize u otro.
2. Crear componente contenedor Home.js con una prop **greeting** y mostrar el mensaje dentro del contenedor con el styling integrado.

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

*Tiempo: 30 minutos*

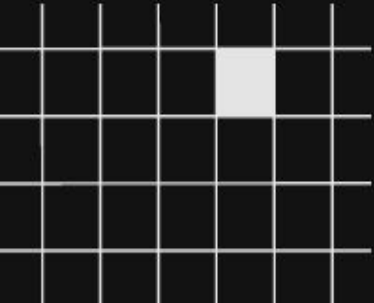
***¿PREGUNTAS?***





# ***¡MUCHAS GRACIAS!***

Resumen de lo visto en clase hoy:

- Introducción y tipos de componentes
    - Children
  - Introducción a props
- 



***OPINÁ Y VALORÁ ESTA CLASE***