UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Desarrollo de Software
Notas de Clase

Clase 19 - React

React es una librería JavaScript para la construcción de interfaces de usuario, desarrollada por Facebook en el año 2013 y ha sido extensamente utilizada en los últimos años. Cabe aclarar que React no es un framework (como sí lo es, por ejemplo, Angular), esto significa que React no marca la manera en la que nuestros proyectos se organizan, ni impone reglas específicas sobre las convenciones de código, lo que le permite a los equipos adoptar React de la manera que les resulte más conveniente.

Además de React, que se utiliza para la construcción de aplicacions web, existen otras librerías y frameworks derivados como: *React Native* para aplicaciones móviles y *React 360* para aplicaciones de realidad virtual.

El objetivo principal de React es minimizar los errores que ocurren cuando los desarrolladores construyen interfaces de usuario. Esto lo hace mediante el uso de componentes — piezas de código lógicas y auto-contenidas que describen una parte de la interfaz del usuario. Estos componentes se pueden juntar para crear una interfaz de usuario completa, y React abstrae la mayor parte del trabajo de renderizado, permitiéndonos enfocarnos en el diseño de la interfaz.

A medida que se presenten los ejemplos de este material (y si recordamos de qué manera hemos construído las primeras interfaces hasta ahora), veremos cuánto más rápido y sencillo resulta trabajar con React.

Características principales de React

- **Declarativo:** Permite crear interfaces de usuario de manera sencilla. Debemos diseñar vistas simples para cada estado en nuestra aplicación, y React se encargará de actualizar y renderizar de manera eficiente los componentes correctos cuando los datos cambien.
- **Basado en Componentes:** La interfaz de usuario se construye mediante componentes, la lógica del componente está codificada en javascript, y se puede mantener el estado fuera del DOM. Una interfaz de usuario comunmente va estar constituído por múltiples componentes generalmente interelacionados entre si.
- **ReactDOM:** Permite realizar manipulaciones en un DOM Virtual, sin modificar el DOM del browser aumentanto considerablemente el tiempo de respuesta del renderizado de la aplicación. En lugar de manipular directamente el DOM del navegador, React crea un DOM virtual en la memoria, donde realiza toda la manipulación necesaria antes de realizar los cambios en el DOM del navegador, que lo hace mucho mas veloz.

Requerimientos

- Instalar la última versión Node.js LTS (Long Term Support, o soporte a largo plazo) (https://nodejs.org/es/download/). Nodejs incluye el gestor de paquetes de Node llamado **NPM:** (Node Package Manager) y **npx** que es el ejecutor de scripts de paquetes de node. Para tutorial de instalación

en windows ingresar aquí:
https://docs.google.com/document/d/1AAuuftGEitM5YMRLimvgBASUMlZ9sFxebtOTfLhcVOE/edit

- Conocimimientos de HTML 5, Javascript, ES6, DOM.
- Browser modernos como Edge, Firefox, Chrome, Safari. No soportado IE11

¿Cómo React usa JavaScript?

Al ser una librería JS, el código React puede ser escrito en este lenguaje. Sin embargo, siendo que queremos construír interfaces web con HTML y CSS, React permite el uso de la sintaxis JSX, que amplía la sintaxis de JavaScript para introducir código de marcado, haciéndolo más similar al código que se terminará renderizando en el browser. Por ejemplo:

```
const heading = <h1>Mozilla Developer Network</h1>;
```

Esta constante "heading" se conoce como una expresión JSX. React puede usarla para representar la etiqueta h1 en nuestra aplicación.

Supongamos que, por razones semánticas, queremos envolver nuestro encabezado en una etiqueta header. El enfoque JSX nos permite anidar nuestros elementos entre sí, tal como lo hacemos con HTML:

```
const header = (
  <header>
    <h1>Mozilla Developer Network</h1>
  </header>
)
```

Habrá casos donde, por ejemplo, el texto del header no sea algo fijo, sino que el mismo se obtenga de alguna variable javascript. Para estos casos, la utilización de llaves permite combinar el lenguaje de marcado con JS.:

```
const titulo = 'Mozilla Developer Network';
const header = (
  <header>
    <h1>{titulo}</h1>
  </header>
)
```

Este código tendrá el mismo efecto que el caso anterior, pero el valor del texto se puede ajustar de forma dinámica.

JSX es más estricto que HTML. Se deben cerrar etiquetas como
.

Más información sobre JSX puede ser encontrada en el siguiente enlace: https://react.dev/learn/writing-markup-with-jsx

Crear una aplicación con React

- En windows, desde la consola CMD ejecutar en la carpeta donde deseemos crear la aplicación, el siguiente comando:

```
npx create-react-app my-app
```

Para ejecutar la aplicación, desplazarse a la carpeta **my-app** y ejecutar el siguiente comando:

```
npm start
```

Se visualizarán en la cosola unos comandos similares a:

```
> my-react-app@0.1.0 start
> react-scripts start

(node:32536) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE]
DeprecationWarning: 'onAfterSetupMiddleware' option is deprecated. Please use the
'setupMiddlewares' option.
(Use `node --trace-deprecation ...` to show where the warning was created)
(node:32536) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE]
DeprecationWarning: 'onBeforeSetupMiddleware' option is deprecated. Please use the
'setupMiddlewares' option.
Starting the development server...
Compiled successfully!

You can now view my-react-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.110:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
Compiling...
Compiled successfully!

You can now view my-react-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.110:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

Estructura de la aplicación (*)

create-react-app nos provee todo lo que necesitamos para desarrollar una aplicación React. Su estructura inicial de archivos luce así:

```
my-app
├── README.md
├── node_modules
├── package.json
├── package-lock.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── serviceWorker.js
```

La carpeta **src** es donde reside el código fuente de nuestra aplicación.

La carpeta **public** contiene archivos estáticos que serán leidos por el navegador; el archivo que funciona como punto de entrada es *index.html*. React introduce nuestro código en este archivo de manera que el navegador pueda ejecutarlo. En esta carpeta también se pueden almacener imágenes estáticas o archivos que van a ser utilizadas por toda la aplicación. La carpeta *public* también será publicada cuando se cree y se despliegue la versión para producción de la aplicación.

El archivo **package.json** contiene información sobre nuestro proyecto, los paquetes que utiliza, comandos que se pueden ejecutar con npm. Este archivo no es exclusivo de las aplicaciones React sino es de cualquier aplicación que utilice npm. El script create-react-app le agrega entradas según necesidad del proyecto.

El archivo **App.css** contiene la hoja de estilo del componente App

El archivo **App.js** contiene la definición del componente principal o inicial de React que representa una parte de nuestra aplicación. El archivo App.js se compone de tres partes principales:

- algunas declaraciones import en la parte superior,
- el componente App en el medio,
- y una declaración export en la parte inferior. La mayoría de los componentes de React siguen este patrón.

El archivo **index.js** es el archivo que inicia la ejecución de la librería de React y del componente App como indica el código a continuación.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

```
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
<React.StrictMode>
    <App />
</React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

El archivo **App.test.js** contiene las pruebas del componente App.

Declaraciones import (*)

Las declaraciones import en la parte superior del archivo le permiten a App.js utilizar código que ha sido definido en otra de la aplicación o en otra biblioteca o libería.

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
```

La primera declaración importa la biblioteca React como tal. Dado que React convierte el JSX que escribimos en React.createElement(), todos los componentes de React deben importar el módulo React. Si omitimos este paso, nuestra aplicación producirá un error.

La segunda declaración importa un logotipo de './logo.svg'. Observemos el uso de ./ al principio de la ruta y la extensión .svg al final — estos nos indican que el archivo es local y que no es un archivo JavaScript. De hecho, el archivo logo.svg reside en nuestra carpeta raíz.

No hace falta proveer una ruta o extensión al importar el módulo React, ya que este no es un archivo local.

La tercera declaración importa el CSS relacionado con nuestro componente App. Se debe notar que no hay nombre de variable ni de directiva from. Esta sintaxis de importación en particular no es propia de la sintaxis de módulos de JavaScript. Esta proviene de Webpack, la herramienta que create-react-app usa para agrupar todos nuestros archivos JavaScript y enviarlos al navegador.

Componentes en React (*)

Los componentes son una pieza fundamental en React. Tienen la característica que son independientes, pueden mantener estado y son piezas reusables de código. Trabajan de manera aislada y generan el HTML que terminará siendo renderizado en el navegador. Un componente es una pieza de UI (siglas en inglés de interfaz de usuario) que tiene su propia lógica y apariencia. Un componente puede ser tan pequeño como un

botón, o tan grande como toda una página. El programador tiene la libertad de generar los componentes de acuerdo a sus necesidades, considerando la posibilidad de reutilización de los mismos. Por dar un ejemplo, si estuviéramos programando una plataforma de comercio electrónico, tal vez sería buena idea considerar como un componente a cada producto del listado de resultados (y este componente tendría una descripción del producto, sus imágenes, precio, etc.).

Si recordamos los conceptos de DOM y SPA de las primeras fichas, notaremos que, en el código de inicio generado anteriormente por npx create-react-app, el body de *index.html* contiene un único elemento div, y que el compontente App se renderiza en él. En este caso *App*, es el componente que representa a toda la aplicación, que puede estar compuesta, a su vez, por otros componentes como cuadros de texto, botones, imágenes, etc.

Ahora bien, ¿cómo se define un componente en React?. Hay dos maneras de hacer esto: definir los componentes mediante **clases** o mediante **funciones**.

Definición mediante clases

La definición es más compleja comparada con la definición mediante funciones, incluye un constructor y un método para renderizar que devuelven el HTML del elemento. Un componente de clase extiene la clase **Component** de React, permite guardar el estado y controlar el ciclo de vida con los métodos **componentDidMount** o **componentWillUnmount**

```
import React from 'react';

class MateriaUniversitariaComponent extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    const materia = 'Matemáticas'; // Constante local
    return (<div><h1> Materia: {materia} - Cantidad Estudiantes
{this.props.numeroDeEstudiantes} </h1></div>);
  }
}

export default MateriaUniversitariaComponent;
```

No entraremos en mayor detalle aquí ya que, como se comentará más adelante, la forma preferida en la actualidad es definir los componentes mediante funciones.

Definición mediante funciones

Para definir un componente React mediante una función, hay que crear una función, cuyo nombre será el nombre del componente a definir, y la misma debe devolver el código de markup que se quiere renderizar. Básicamente, tiene que devolver lo mismo que el método *render()* explicado en la sección anterior. Por ejemplo, para declarar un componente llamado *MyButton*, que terminará siendo renderizado como un botón html y el texto "I'm a button", la función correspondiente sería la siguiente:

```
function MyButton() {
  return (
    <button>I'm a button</button>
  );
}
```

Se debe notar que se está utilizando JSX y que la función está definida como cualquier otra función de JavaScript.

Ahora que hemos declarado MyButton, podemos anidarlo en otro componente:

```
export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton />
    </div>
  );
}
```

Los nombres de los componentes de React siempre deben comenzar con mayúscula, mientras las etiquetas HTML deben estar minúsculas.

Profundizando con componentes funcionales

Son funciones de javascript que reciben como parámetro un objeto con propiedades (al que generalmente llamaremos **props**) y que devuelven un elemento **ReactNode**, un **HTML**, **string**, etc. para renderizar.

Propiedades (props): Es un parámetro a la función que estamos creando que contiene propiedades para el componente y otros valores para realizar el enlace con otros componentes.

ReactNode: Es cualquier valor que se pueda renderizar en un componente de React, como un elemento JSX, un componente de React, una cadena de texto, un número, un booleano, entre otros.

Los componentes se deben definir con la primera letra en mayúscula, para diferenciarlos de elementos html 5 (los elementos en html 5 van siempre en minúsculas).

Definición del componente MateriaUniversitariaComponent

```
  //Componente MateriaUniversitariaComponent
  function MateriaUniversitariaComponent({numeroDeEstudiantes}) {
      const materia = "Desarrollo de Software";
      return (<div><h1> Materia: {materia} - Cantidad Estudiantes
{numeroDeEstudiantes} </h1></div>);
  }

  export default MateriaUniversitariaComponent;
```

Definición del componente App

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import MateriaUniversitariaComponent from
'./components/MateriaUniversitariaComponent';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <>
        <MateriaUniversitariaComponent numeroDeEstudiantes="80">
</MateriaUniversitariaComponent>
    </>
);
```

Mostar datos

JSX te permite poner marcado dentro de JavaScript. Las llaves permiten «escapar de nuevo» hacia JavaScript de forma tal que se pueda incrustar una variable en el código y mostrársela al usuario. Por ejemplo, esto mostrará *Tu edad es: 20*:

```
function MiComponente(props) {
  const edad = 20;
  return <p>Tu edad es: {edad}</p>;
}
```

Props en componentes

En React, los componentes funcionales pueden recibir argumentos llamados "props" (abreviatura de "propiedades") que les permiten recibir datos de un componente padre y utilizarlos en su renderizado.

Las props se pasan como argumentos en la función del componente y se pueden acceder a ellas dentro del cuerpo de la función. Por ejemplo, supongamos que tenemos un componente Saludo que recibe un nombre como prop y lo muestra en pantalla:

```
function Saludo(props) {
  return <h1>Hola, {props.nombre}!</h1>;
}
```

En este ejemplo, el componente Saludo recibe un objeto props como argumento y accede a la propiedad nombre del objeto para mostrar el saludo en pantalla.

Para utilizar el componente Saludo y pasarle una prop nombre, podemos hacer lo siguiente:

```
<Saludo nombre="Juan" />
```

En este ejemplo, estamos creando una instancia del componente Saludo y pasándole la prop nombre con el valor "Juan". Cuando el componente se renderice en la página, mostrará el saludo "Hola, Juan!".

Las props son una forma de pasar datos de un componente a otro en la jerarquía de componentes de React, y se pueden utilizar para personalizar y configurar el comportamiento y el renderizado de los componentes en tu aplicación.

Desestructuración de Props

En React, puedes desestructurar las props que recibe un componente para acceder a sus valores de una manera más conveniente y legible. La desestructuración es una técnica de JavaScript que permite extraer valores de objetos y arreglos en variables individuales.

Para desestructurar las props en un componente funcional, puedes hacer lo siguiente:

```
function MiComponente(props) {
  const { prop1, prop2, prop3 } = props;
  // Utiliza prop1, prop2, prop3 como variables en el cuerpo de tu componente
  return (
    // Código JSX para el renderizado del componente
  );
}
```

En este ejemplo, estamos desestructurando las props recibidas en el componente MiComponente y extrayendo las propiedades prop1, prop2 y prop3 del objeto props. Luego, podemos utilizar estas variables dentro del cuerpo del componente para acceder a los valores de las props.

También podemos desestructurar las props directamente en la lista de argumentos de la función del componente, de la siguiente manera:

```
function MiComponente({ prop1, prop2, prop3 }) {
  // Utiliza prop1, prop2, prop3 como variables en el cuerpo de tu componente
  return (
    // Código JSX para el renderizado del componente
  );
}
```

En este ejemplo, estamos desestructurando las props directamente en la lista de argumentos de la función del componente, lo que nos permite acceder a sus valores utilizando variables con los mismos nombres que las propiedades en el objeto props.

La desestructuración de props es una forma útil de simplificar el código de tus componentes y hacer que sea más fácil de leer y mantener.

Creación de componentes: Consideraciones adicionales

Es necesario comprender que, a diferencia de los objetos, las funciones no tienen estado... por eso, si se define un componente mediante una función, será necesario el uso de los llamados "hooks" (tema que estudiaremos más adelante). Desde la introducción de los hooks en React (una nueva forma de administrar el estado en componentes funcionales), se ha vuelto más fácil manejar el estado y otros comportamientos de los componentes sin necesidad de utilizar clases. Además, los componentes funcionales tienen un rendimiento más rápido y son más fáciles de testear.

Si bien existen estas dos opciones (crear una clase por componente o crear una función por componente), en la actualidad se prefiere la segunda forma, entonces en las fichas de estudio utilizaremos **funciones** para definir los componentes. A partir de React versión 16.8.0, con la implementación de los hooks, los componentes de tipo clase quedaron anticuados.

Finalmente, se debe notar que en los ejemplos se utilizó JSX lo cual, como ya se mencionó, no es obligatorio, pero sí recomendable. Para entender porqué, basta con analizar los siguientes ejemplos:

Ejemplo de código con JSX

```
const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Ejemplo de código sin JSX

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

Y que el código HTML debe estar envuelto en un elemento raiz. Si tenemos la necesidad de código HTML que no depende de un elemento raíz se puede utilizar un tag vacío de HTML como se indica a continuación:

```
const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);
```

Renderización con React

La renderización en React se refiere al proceso en el que los componentes de React se convierten en elementos del DOM (Document Object Model) que se pueden mostrar en el navegador.

Cuando se crea un componente en React, se define su estructura y comportamiento a través de código JavaScript. Luego, cuando se renderiza el componente, React toma esa estructura y la convierte en elementos del DOM que se muestran en la pantalla. Este proceso implica tres pasos:

1. **Construcción del árbol de elementos virtuales**: cuando se renderiza un componente de React, se construye un árbol de elementos virtuales, también conocido como árbol de nodos virtuales o VDOM (Virtual DOM). Este árbol es una representación en memoria del árbol de elementos del DOM que se mostrarán en la pantalla. El VDOM es una estructura de datos liviana que se actualiza con cambios en el estado del componente, lo que significa que no es necesario actualizar todo el DOM cada vez que se realiza un cambio.

2. **Comparación de árboles de elementos virtuales**: una vez que se ha construido el árbol de elementos virtuales, React lo compara con la versión anterior del árbol (si existe) para determinar los cambios que deben aplicarse en el DOM real. React utiliza un algoritmo de diferencias (también conocido como algoritmo de reconciliación) para encontrar y aplicar los cambios más eficientemente.

3. **Actualización del DOM**: finalmente, React actualiza el DOM real con los cambios necesarios que se encontraron durante la comparación de los árboles de elementos virtuales. Esto significa que sólo se actualizan los elementos que han cambiado, lo que mejora el rendimiento de la aplicación.

En resumen, la renderización en React implica la construcción de un árbol de elementos virtuales, la comparación de ese árbol con el árbol anterior y la actualización del DOM real con los cambios necesarios. Este proceso es eficiente y rápido gracias al uso del VDOM y al algoritmo de diferencias de React.

React realiza la renderizacion invocando la función llamada **ReactDOM.Render()** . Dicha función toma dos argumentos, el primero es el código HTML a renderizar y el segundo argumento es en qué elemento del DOM se va a renderizar (contenedor de).

Ejemplo 1

```
ReactDOM.render(<p>Hello</p>, document.getElementById('root'));
```

Ejemplo 2

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<p>Hello</p>);
```

Si vemos el código que generó npx create-react-app, vamos a ver la utilización de este método en index.js:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

En este caso, React va a renderizar el Componente App (que tal como se crea la aplicación viene a ser el componente principal), en el div con id *root*, que si se revisa *index.html* se verá que es un contenedor html estándar.

Formas de aplicar estilos a elementos en un componente en React

Utilización de atributo class

Si se necesita utilizar el atributo *class* de HTML en JSX, como class es una palabra reservada en JavaScript, se necesita utilizar el atributo *className* en vez de *class*. Cuando el JSX se renderice, el mismo convierte el atributo className en class.

```
function MiComponente() {
  return (
    <div className="mi-clase-css">
      Este es un componente con clase CSS
    </div>
  );
}
```

y en el archivo de CSS definir los estilos:

```
.mi-clase-css {
  background-color: red;
  color: white;
}
```

Para enlazar hojas de estilo, se debe añadir una etiqueta <link> al HTML.

También podemos asignar múltiples clases a un elemento utilizando la propiedad *className*. Para hacerlo, simplemente separamos los nombres de las clases con espacios:

```
<div className="mi-clase otra-clase">
  Este es un elemento con dos clases CSS: "mi-clase" y "otra-clase"
</div>
```

En este ejemplo, el elemento tiene dos clases CSS, "mi-clase" y "otra-clase", asignadas a través de la propiedad className.

Debemos tener en cuenta que si deseamos utilizar nombres de clase dinámicos o condicionales, se pueden utilizar expresiones JavaScript dentro de la propiedad className para generar los nombres de clase necesarios en tiempo de ejecución.

Por ejemplo, si la variable *isActive* indica si el elemento está activo o no, se podrían asignar diferentes clases CSS según el valor de la variable utilizando una expresión ternaria:

```
<div className={isActive ? "active-class" : "inactive-class"}>
  Este elemento es {isActive ? "activo" : "inactivo"}
</div>
```

En este ejemplo, la clase CSS "active-class" se asigna si isActive es verdadero, y "inactive-class" se asigna si es falso.

El css del ejemplo anterior sería:

```
.active-class {
  color: green;
  font-weight: bold;
}

.inactive-class {
  color: gray;
  font-weight: normal;
}
```

En React, podemos importar un archivo CSS a un componente utilizando la función import de JavaScript en la parte superior del archivo de componente.

Supongamos que tenemos un archivo estilos.css que contiene las reglas de estilo que deseamos aplicar a nuestro componente. Para importar este archivo al componente, podemos hacer lo siguiente:

```
import React from 'react';
import './estilos.css';

function MiComponente() {
  return (
    <div className="mi-clase">
      Este es mi componente con estilos CSS importados
    </div>
  );
}

export default MiComponente;
```

En este ejemplo, la línea import './estilos.css'; importa el archivo estilos.css al componente. Luego, podemos aplicar las clases CSS definidas en ese archivo a tus elementos en JSX como lo haríamos normalmente.

Hay que tener en cuenta que la ruta al archivo CSS debe ser relativa al archivo de componente en el que estamos importando el archivo. También es importante revisar de que la extensión del archivo CSS sea .css.

Al importar un archivo CSS de esta manera, todas las reglas de estilo definidas en el archivo se aplicarán a todos los elementos en el componente, a menos que se anulen con estilos en línea o clases CSS adicionales en tus elementos.

Utilización de estilos en línea

En React, es posible utilizar la sintaxis de estilo en línea de JavaScript para definir las propiedades CSS en los elementos del componente.

La sintaxis para definir los estilos en línea es muy similar a la sintaxis de estilo en HTML, pero en lugar de utilizar la propiedad *style* como un atributo HTML, se utiliza como una propiedad de objeto en tu JSX.

Por ejemplo, si queremos definir un estilo en línea para un elemento , podemos hacerlo de la siguiente manera:

```
function MiComponente() {
  return (
  <div style={{ backgroundColor: 'red', color: 'white' }}>
    Este es un elemento con estilo en línea
  </div>
  );
}
```

En este ejemplo, la propiedad style del elemento es un objeto de JavaScript que contiene las propiedades CSS que se quieren aplicar. En este caso, la propiedad backgroundColor se establece en 'red' y la propiedad color se establece en 'white'.

Es posible definir cualquier propiedad CSS que se desee utilizando esta sintaxis. Por ejemplo, si queremos establecer la altura y el ancho del elemento, podemos hacer lo siguiente:

```
function MiComponente() {
  return (
    <div style={{ width: '200px', height: '100px' }}>
      Este es un elemento con estilo en línea
    </div>
  );
}
```

La sintaxis de estilo en línea utiliza la convención de nomenclatura camelCase para definir las propiedades CSS en lugar de la convención de nomenclatura kebab-case utilizada en CSS.

Por ejemplo, si en CSS queremos establecer el tamaño de letra de un elemento utilizando la propiedad font-size, en React utilizaríamos la propiedad fontSize en su lugar:

```
function MiComponente() {
  return (
  <div style={{ fontSize: '16px' }}>
    Este es un elemento con un tamaño de letra de 16 píxeles
  </div>
  );
}
```

Aquí, fontSize se escribe en camelCase en lugar de font-size en kebab-case.

Esta convención de nomenclatura camelCase se utiliza para asegurar que la sintaxis de estilo en línea de React sea consistente con la convención de nomenclatura de JavaScript.

Guía Paso a Paso para desarrollar en clase

- Realizar la guía paso a paso de una aplicación en React