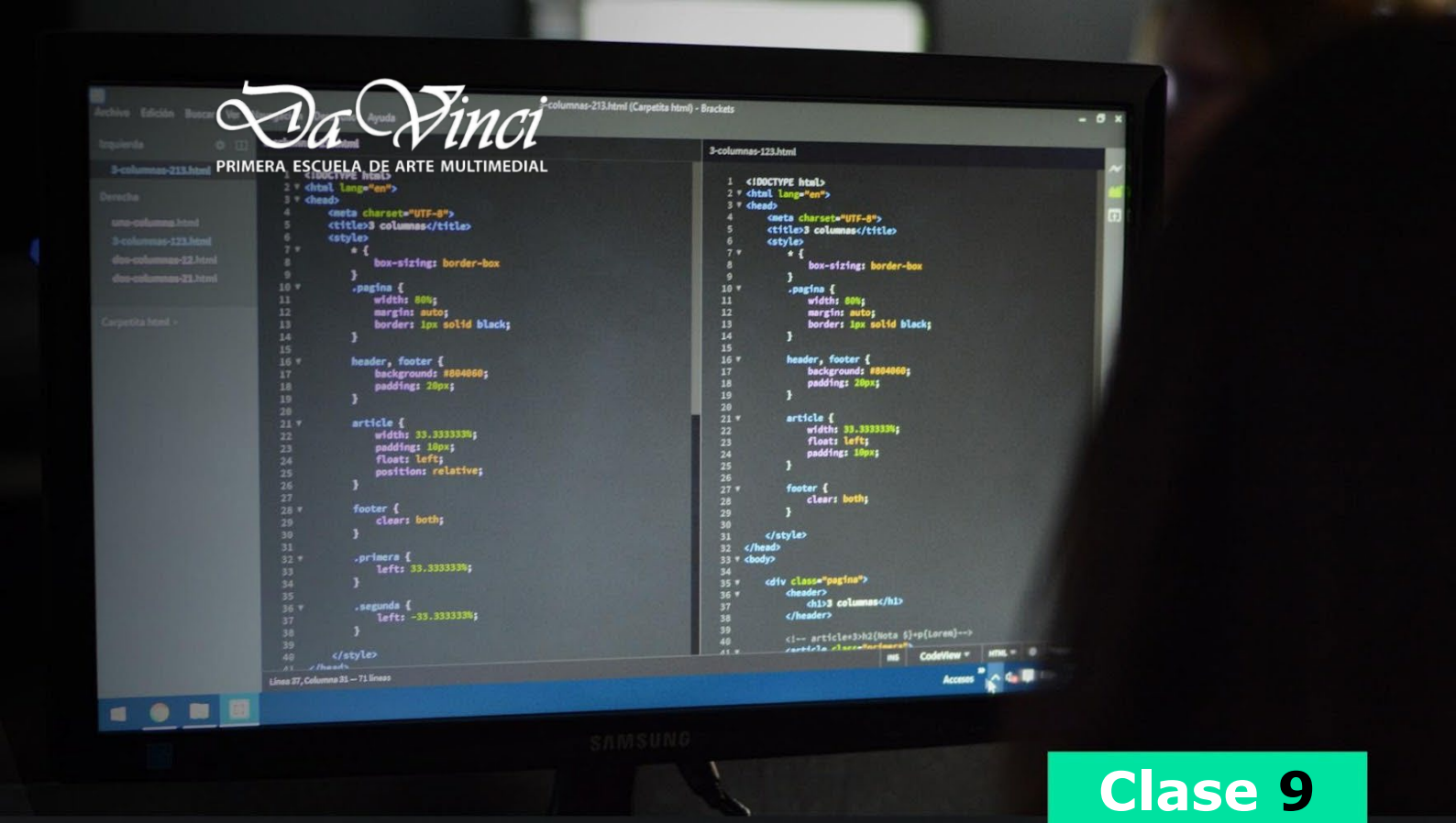

Clase 9

Diseño y Programación Web

Materia:
Aplicaciones Híbridas

Docente contenidista: MARCOS GALBÁN, Camila Belén

React

React es una librería de JavaScript para construir interfaces de usuario, especialmente las interfaces web. Fue desarrollada por Facebook y se utiliza ampliamente en el desarrollo de aplicaciones web debido a su eficiencia y flexibilidad.

Características

Componentes

React permite dividir la interfaz de usuario en componentes reutilizables. Cada componente encapsula su propia lógica y estilo, lo que facilita la gestión y el mantenimiento del código.

Virtual DOM

React utiliza un DOM virtual (Virtual DOM) para optimizar las actualizaciones de la interfaz de usuario. En lugar de actualizar el DOM real directamente, React realiza cambios en el DOM virtual y luego sincroniza las diferencias con el DOM real, lo que mejora significativamente el rendimiento.

JSX

JSX es una extensión de JavaScript que permite escribir código HTML dentro de JavaScript. Esto facilita la creación de componentes React y mejora la legibilidad del código.

Unidirectional Data Flow

React implementa un flujo de datos unidireccional, lo que significa que los datos fluyen en una única dirección a través de la aplicación. Esto simplifica el seguimiento y la gestión del estado de la aplicación.

Hooks

Hooks son una característica avanzada que permite utilizar estado y otras características de React en componentes funcionales, facilitando la reutilización de lógica entre componentes.

Ejemplo básico de un componente React:

```
import React from 'react';

function HelloWorld() {
  return <h1>Hello, World!</h1>;
}

export default HelloWorld;
```

Instalación

Vite y Create React App (CRA) son herramientas populares para configurar y desarrollar aplicaciones React. A continuación, se detallan las principales diferencias y características de cada una:

Create React App (CRA)

Ventajas:

- **Configuración simple**: CRA permite configurar una aplicación React con una configuración mínima, lo que es ideal para principiantes.
- **Documentación y comunidad**: Tiene una amplia documentación y una gran comunidad, lo que facilita encontrar recursos y soporte.
- **Herramientas integradas**: Incluye herramientas como ESLint, Jest para pruebas, y configuraciones de Webpack preestablecidas.

Desventajas:

- **Velocidad de desarrollo**: CRA puede ser más lento en el tiempo de arranque y recarga en caliente, especialmente en aplicaciones grandes.

```
npx create-react-app my-app
cd my-app
npm start
```

Vite

Ventajas:

- **Desempeño**: Vite es significativamente más rápido que CRA. Utiliza un servidor de desarrollo basado en ESBuild, lo que permite arranques rápidos y recarga en caliente extremadamente rápida.
- **Modernización**: Está diseñado para aprovechar las últimas características de ES modules y navegadores modernos.
- **Configurabilidad**: Vite permite una configuración más flexible.

Desventajas:

- **Aprendizaje inicial**: Puede requerir una curva de aprendizaje mayor para desarrolladores acostumbrados a CRA o nuevas herramientas.

```
npm create vite@latest my-app
cd my-app
npm install
npm run dev
```

JSX

JSX (JavaScript XML) es una extensión de la sintaxis de JavaScript utilizada en React para describir cómo debería verse la interfaz de usuario. JSX permite escribir código que se parece a HTML dentro de los archivos de JavaScript, lo que facilita la creación de componentes de UI de manera declarativa.

Características

Sintaxis Similar a HTML

JSX se parece a HTML, lo que hace que el código sea más legible y fácil de escribir.

Permite incluir elementos HTML y componentes React dentro del código JavaScript.

Compilación a JavaScript

JSX no es válido en el navegador por sí solo. Debe ser transformado en llamadas a funciones de JavaScript por un transpilador como Babel.

La transformación convierte las etiquetas JSX en llamadas a React.createElement.

Interpolación de Expresiones JavaScript:

Se puede usar llaves {} para insertar expresiones JavaScript dentro de JSX.

Esto incluye variables, operaciones matemáticas, llamadas a funciones, y más.

Compatibilidad con Componentes:

JSX facilita la composición de componentes React, permitiendo anidar componentes y pasar props de manera sencilla.

DOM y Virtual DOM

El DOM (Document Object Model) y el Virtual DOM son conceptos clave en el desarrollo web, especialmente en el contexto de bibliotecas y frameworks como React.

DOM (Document Object Model)

Qué es:

- El DOM es una representación en memoria de la estructura de un documento HTML o XML.
- Permite a los lenguajes de programación, como JavaScript, interactuar y manipular la estructura, estilo y contenido de la página.

Ventajas:

- **Estándar de la Web**: Es una API estándar compatible con todos los navegadores.
- **Acceso Directo**: Permite manipulación directa de los elementos de la página.

Desventajas:

- **Rendimiento**: Las manipulaciones directas y frecuentes del DOM real pueden ser costosas en términos de rendimiento, especialmente en aplicaciones grandes y complejas.
- **Complejidad**: Manejar el DOM directamente puede volverse complicado y propenso a errores, ya que requiere un manejo cuidadoso de los cambios de estado y actualizaciones de la UI.

Virtual DOM

Qué es:

- El Virtual DOM es una representación en memoria del DOM real.
- Es una abstracción del DOM que permite que las librerías, como React, gestionen actualizaciones de manera más eficiente.

Cómo Funciona:

- **Renderizado Inicial**: Cuando se carga la aplicación, React crea una representación del DOM en memoria (Virtual DOM).
- **Actualización del Estado**: Cuando el estado de un componente cambia, React crea un nuevo Virtual DOM.
- **Diferencias (Diffing)**: React compara el nuevo Virtual DOM con el antiguo para encontrar las diferencias.
- **Parcheo (Patching)**: React aplica solo las diferencias necesarias al DOM real, minimizando las operaciones costosas.

Ventajas:

- **Rendimiento**: Las actualizaciones del Virtual DOM son rápidas y eficientes. React aplica solo las actualizaciones mínimas necesarias al DOM real.
- **Declarativo**: Facilita un enfoque declarativo para construir interfaces de usuario. Los desarrolladores describen cómo debe lucir la UI en función del estado, y React se encarga de las actualizaciones.
- **Simplificación**: Simplifica la gestión de cambios y actualizaciones de la UI, reduciendo el riesgo de errores y problemas de sincronización de estado.

Desventajas:

- **Consumo de Memoria**: El mantenimiento de un Virtual DOM adicional en memoria puede consumir más recursos.
- **Abstracción**: Puede agregar una capa de abstracción adicional que requiere una curva de aprendizaje para los nuevos desarrolladores.

Comparación: DOM vs Virtual DOM

Aspecto	DOM Real	Virtual DOM
Rendimiento	Manipulaciones frecuentes son costosas	Actualizaciones eficientes y rápidas
Manipulación	Directa	Abstraída a través de bibliotecas
Complejidad	Mayor manejo y riesgo de errores	Simplifica la gestión de actualizaciones
Consumo de Recursos	Menor consumo de memoria	Mayor consumo de memoria
Enfoque	Imperativo	Declarativo

Reactividad

En el contexto de React, "reactividad" se refiere a cómo la librería maneja los cambios en los datos y cómo estos cambios se reflejan automáticamente en la interfaz de usuario (UI). Este concepto es fundamental para entender cómo React actualiza y gestiona el estado de los componentes de una manera eficiente y predecible.

Componentes y Estado:

Un componente en React puede tener estado (state). El estado es una estructura que contiene datos que pueden cambiar con el tiempo.

Cuando el estado de un componente cambia, React automáticamente vuelve a renderizar el componente para reflejar los nuevos datos en la UI.

Unidirectional Data Flow:

Los datos en React fluyen en una única dirección, desde los componentes padres hacia los componentes hijos.

Esto significa que un componente padre puede pasar datos a sus hijos a través de props, pero los hijos no pueden modificar directamente el estado del padre. En lugar de eso, los hijos pueden disparar eventos que el padre puede manejar para actualizar su estado.

Re-renderización Automática:

React optimiza las actualizaciones de la UI utilizando el Virtual DOM.

Cuando cambia el estado de un componente, React compara el nuevo estado del Virtual DOM con el estado anterior, encuentra las diferencias (diffing), y solo actualiza las partes del DOM real que han cambiado.

JSX y Reactividad:

Cuando el estado o los props cambian, React vuelve a evaluar el JSX y actualiza la UI en consecuencia.

Componentes

En React, los componentes son las unidades básicas de construcción de una interfaz de usuario. Se pueden clasificar en componentes de clase y componentes funcionales, y el concepto de atomización se refiere a dividir la UI en componentes lo más pequeños y reutilizables posible.

Componentes de Clase:

- Se definen como clases de ES6 que extienden de React.Component.
- Utilizan el método render para devolver el JSX.
- Manejan el estado a través de this.state y los ciclos de vida del componente.

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

class Welcome extends Component {
  constructor(props) {
    super(props);
    this.state = { message: 'Hello, World!' };
  }

  render() {
    return <h1>{this.state.message}</h1>;
  }
}

export default Welcome;
```

Componentes Funcionales:

- Se definen como funciones de JavaScript.
- Inicialmente eran componentes sin estado, pero con la introducción de los Hooks, ahora pueden manejar el estado y otros aspectos del ciclo de vida del componente.

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

function Welcome() {
  const [message, setMessage] = useState('Hello, World!');

  return <h1>{message}</h1>;
}

export default Welcome;
```

Atomización

Atomización es el proceso de dividir una aplicación en componentes pequeños y reutilizables. Este enfoque facilita la gestión, pruebas y mantenimiento del código.

- **Átomos**: Componentes básicos e indivisibles, como botones, inputs, etiquetas.
- **Moléculas**: Combinaciones simples de átomos, como un formulario de login que incluye inputs y botones.
- **Organismos**: Componentes complejos que combinan átomos y moléculas, como un encabezado de página que incluye un menú de navegación y un logo.
- **Plantillas**: Disposición de organismos en una página.
- **Páginas**: Plantillas completas con contenido real.

Ciclo de vida de los componentes

El ciclo de vida de los componentes en React se refiere a las distintas fases que atraviesa un componente desde su creación hasta su eliminación. Estas fases permiten a los desarrolladores ejecutar código en momentos específicos del ciclo de vida de un componente, lo que es útil para tareas como inicialización de datos, actualizaciones y limpieza.

Ciclo de Vida de Componentes de Clase

En componentes de clase, el ciclo de vida se puede dividir en tres fases principales: **Montaje**, **Actualización** y **Desmontaje**.

Montaje (Mounting)

- **constructor()**: Se llama cuando se crea una instancia del componente. Ideal para inicializar el estado.
- **static getDerivedStateFromProps()**: Se llama antes de renderizar, tanto en el montaje inicial como en las actualizaciones posteriores. Permite actualizar el estado en respuesta a cambios en las props.
- **render()**: Renderiza el componente.
- **componentDidMount()**: Se llama inmediatamente después de que el componente se monta (se inserta en el árbol del DOM). Ideal para solicitudes de red, suscripciones, etc.

Actualización (Updating)

- **static getDerivedStateFromProps()**: Igual que en la fase de montaje.
- **shouldComponentUpdate()**: Permite controlar si el componente debe renderizarse o no. Ideal para optimización de rendimiento.
- **render()**: Igual que en la fase de montaje.
- **getSnapshotBeforeUpdate()**: Se llama justo antes de que los cambios del DOM sean aplicados. Permite capturar información del DOM antes de la actualización.
- **componentDidUpdate()**: Se llama inmediatamente después de que el componente se actualiza. Ideal para manejar actualizaciones basadas en cambios previos del DOM.

Desmontaje (Unmounting)

- **componentWillUnmount()**: Se llama justo antes de que el componente sea destruido y removido del DOM. Ideal para limpieza, como cancelar suscripciones o solicitudes de red.

Ciclo de Vida de Componentes Funcionales con Hooks

En componentes funcionales, los Hooks permiten manejar efectos secundarios y el ciclo de vida del componente de manera más clara y concisa.

useState(): Permite manejar el estado en componentes funcionales.

useEffect():

- Similar a componentDidMount, componentDidUpdate y componentWillUnmount en componentes de clase.
- Se llama después de cada renderizado. Si se proporciona un array de dependencias, el efecto solo se ejecuta cuando alguna de esas dependencias cambia.
- Se puede devolver una función de limpieza que se ejecutará antes de desmontar el componente o antes de ejecutar el efecto la próxima vez.

Props y children

En React, **props** y **children** son mecanismos fundamentales para pasar datos y contenido entre componentes. A continuación se explica cada uno en detalle.

Props

Qué son:

- Las "props" (abreviatura de "properties") son argumentos que se pasan a los componentes React desde sus componentes padres.
- Permiten que los componentes sean reutilizables y configurables, ya que pueden recibir datos dinámicos y funciones para manejar eventos.

Cómo se usan:

- Las props se pasan a un componente como atributos en el JSX.
- Dentro del componente, las props se acceden a través del objeto props.

Ejemplo:

```jsx
import React from 'react';

// Definición del componente
function Saludar(props) {
  return <h1>Hello, {props.nombre}!</h1>;
}

// Uso del componente
function App() {
  return (
    <div>
      <Saludar nombre="Alice" />
      <Saludar nombre="Bob" />
    </div>
  );
}

export default App;
```

En este ejemplo:

- El componente Greeting recibe una prop llamada name.
- Al usar el componente Greeting, se pasan diferentes valores para la prop name.

Children

Qué son:

- children es una prop especial en React que permite a los componentes anidar otros componentes o elementos JSX dentro de ellos.
- Facilita la creación de componentes envolventes y estructuras de UI más complejas.

Cómo se usan:

- Los elementos JSX o componentes anidados dentro de un componente se pasan automáticamente como children.
- Dentro del componente, los children se acceden a través de props.children.

Ejemplo:

```
import React from 'react';

// Definición del componente
function Container(props) {
  return <div className="container">{props.children}</div>;
}

// Uso del componente
function App() {
  return (
    <Container>
      <p>Texto a modo de ejemplo.</p>
      <button>Click!</button>
    </Container>
  );
}

export default App;
```

En este ejemplo:

- El componente Container envuelve a sus children, que incluyen un párrafo y un botón.
- Los elementos anidados se pasan automáticamente como props.children y se renderizan dentro del div con la clase "container".

Props vs Children

Aspecto	Props	Children
Propósito	Pasar datos y funciones a componentes	Anidar otros elementos o componentes
Uso	Atributos en el JSX	Elementos anidados entre etiquetas de apertura y cierre
Acceso	props.propName	props.children

Componentes reutilizables

Crear componentes reutilizables en React es esencial para mantener un código limpio, modular y fácil de mantener. A continuación, se presenta un enfoque paso a paso para crear componentes reutilizables, junto un ejemplo.

Principios para Componentes Reutilizables

- **Separación de responsabilidades**: Un componente debe hacer una sola cosa y hacerlo bien.
- **Configurabilidad**: Usar props para permitir la personalización del componente.
- **Composición**: Usar children para permitir la composición de componentes.
- **Estilos desacoplados**: Permitir la personalización de estilos a través de props o clases CSS.

Ejemplo

Botón

```
import React from 'react';

function Button({ label, onClick }) {
  return <button onClick={onClick}>{label}</button>;
}

export default Button;
```