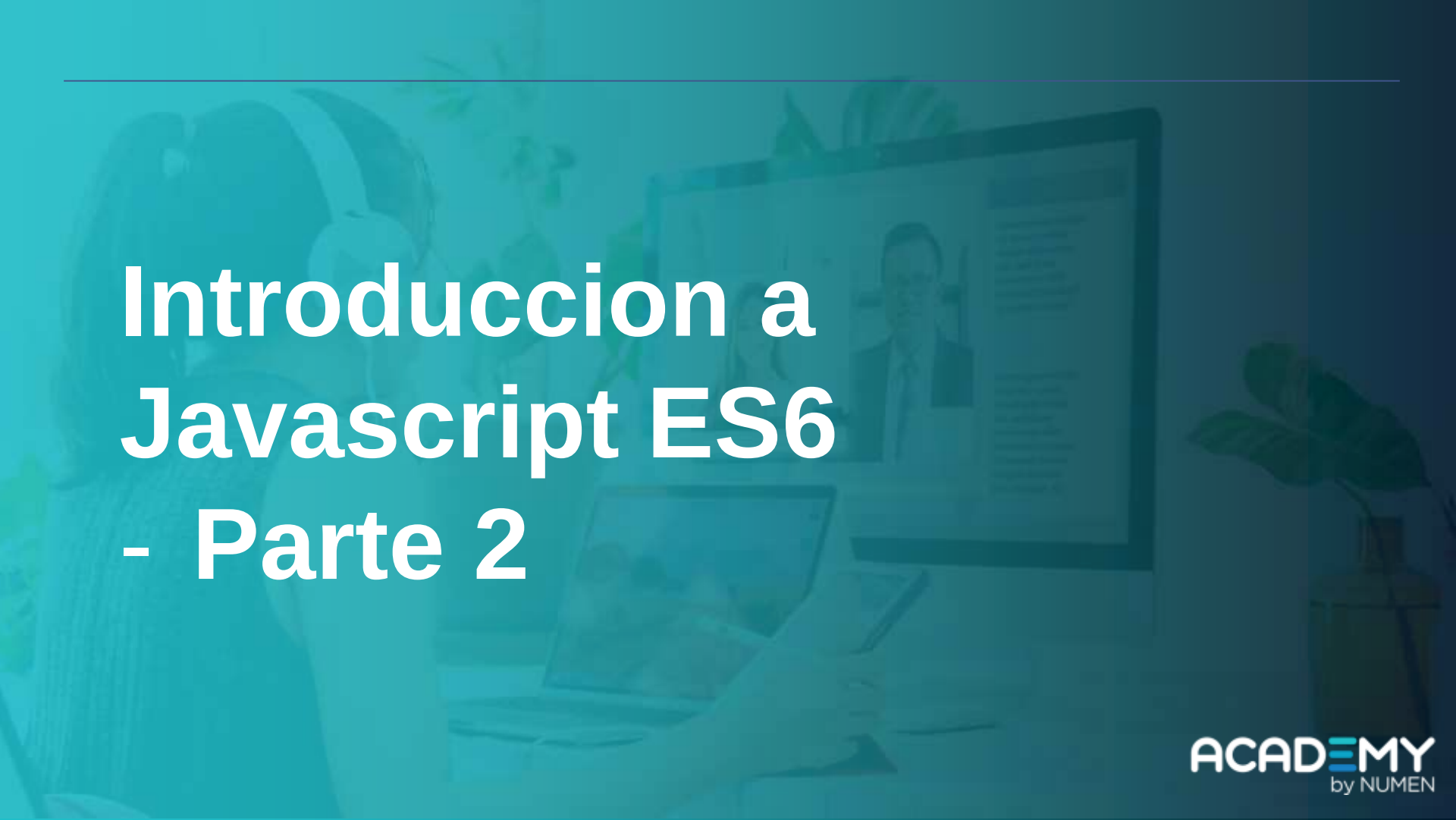
ACADEMY by NUMEN

FULL STACK

Comenzamos en unos minutos


ACADEMY
by NUMEN

Introduccion a Javascript ES6 - Parte 2



ACADEMY by NUMEN

Destructuración

Nos permite desmenuzar un objeto en sus propiedades o un arreglo en sus posiciones a modo de poder acceder a esas propiedades sin tener que acceder a ella por medio de dotwalking (notación de punto).

```
const numeros = [1,2,3]

// Sin destructuración
let uno = numeros[0],
    dos = numeros[1],
    tres = numeros[2]

console.log(uno, dos, tres)

// Con destructuración
const [Uno, Dos, Tres] = numeros
console.log(Uno, Dos, Tres)
```

```
// Objeto destructurado
let jedi = {
    nombre: "Yoda",
    edad: 900,
    raza: "Desconocida"
}

let {nombre, edad, raza} = jedi

console.log(nombre, edad, raza)
```

Parametros REST

Nos permiten representar un número indefinido de argumentos como un array.

```
const sumar = (a,b, ...c) => {
    let resultado = a + b

    c.forEach(numero => resultado = resultado + numero)

    return resultado
}
```

Operador Spread

Nos permite expandir un elemento iterable, como un arreglo, donde normalmente no se podría. En pocas palabras permite conservar una copia de ese elemento con el objetivo de adicionar más elementos en lugar de que estos reemplacen al primero.

```
// Sin operador spread
const array1 = [1,2,3,4,5]
const array2 = [6,7,8,9,10]

const array3 = [array1, array2]

console.log(array3)
```

```
// Con operador spread
const array1 = [1,2,3,4,5]
const array2 = [6,7,8,9,10]

const array3 = [...array1,...array2]

console.log(array3)
```

Estado

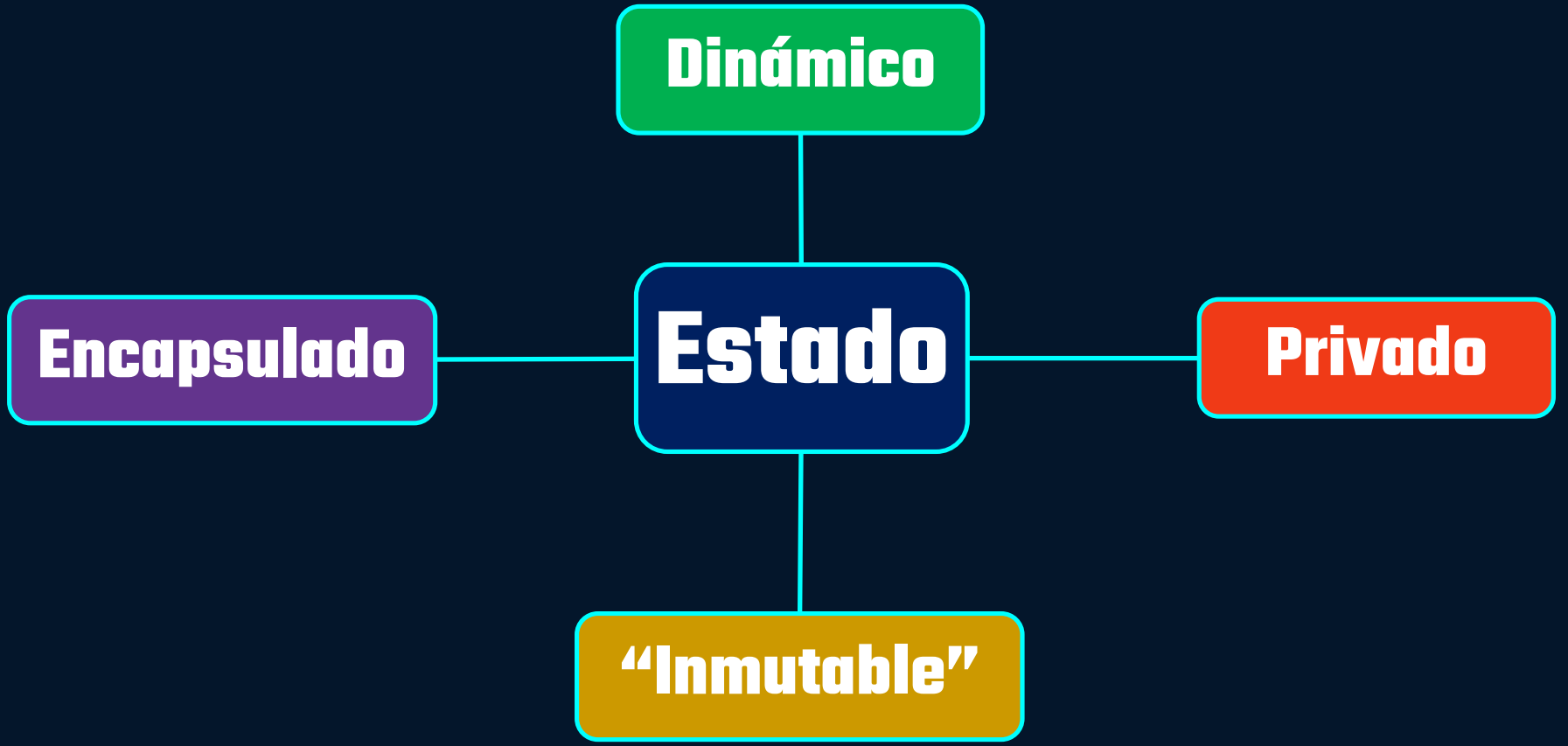
ACADEMY by NUMEN

Programación Reactiva Funcional

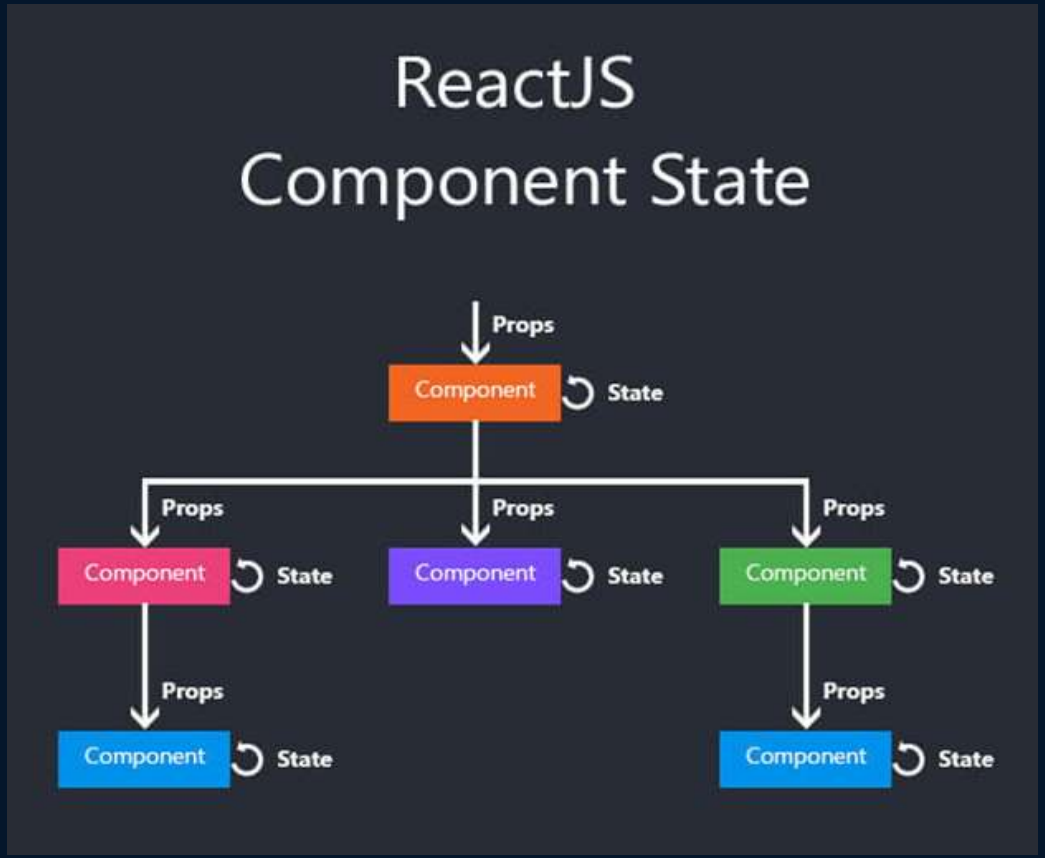
¿Cómo manejar los cambios?

💡 Los cambios de nuestra aplicación se manejan a través del “estado”. ¿Qué contiene el *estado*? Todas aquellas partes de la aplicación que pueden *cambiar*. En otras palabras, es la información o *data* de la misma.

Por ejemplo, en una librería online, el estado podría contener los títulos, autores y categorías de libros, así como también los datos de usuario y más.

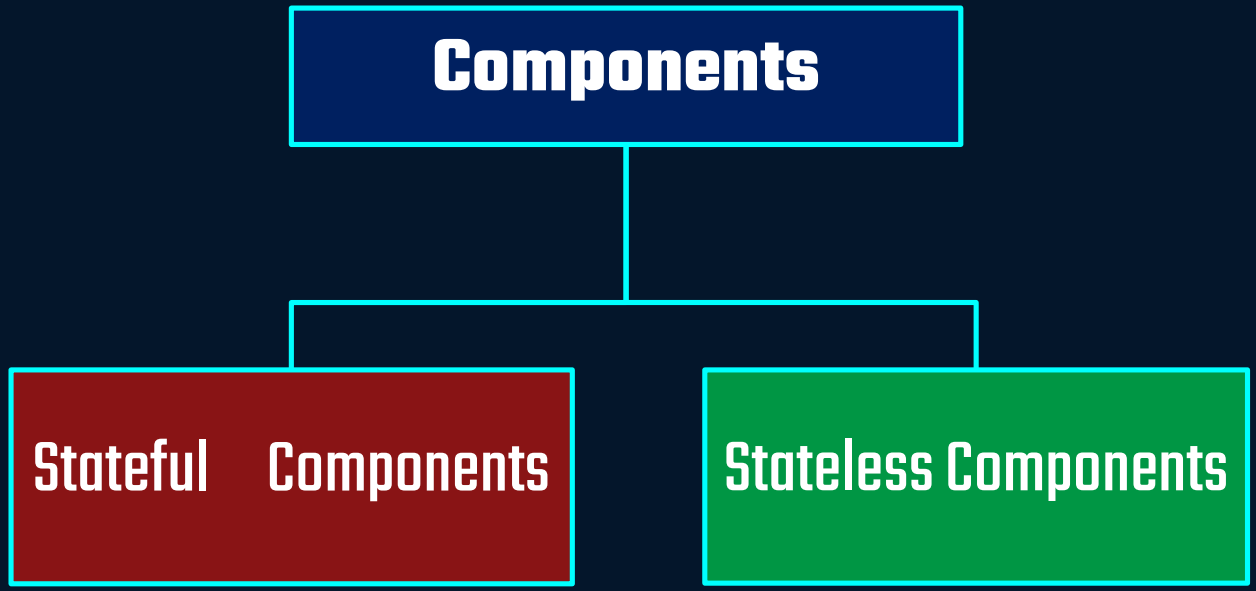

Dinámico
Encapsulado
Estado
Privado
“Inmutable”

ℹ ¿Recuerdan el flujo de datos **unidireccional**? Actualicemos este cuadro, sumándole el estado interno de cada componente.



ACADEMY by NUMEN

¿Cómo se clasifican los componentes en relación al estado?



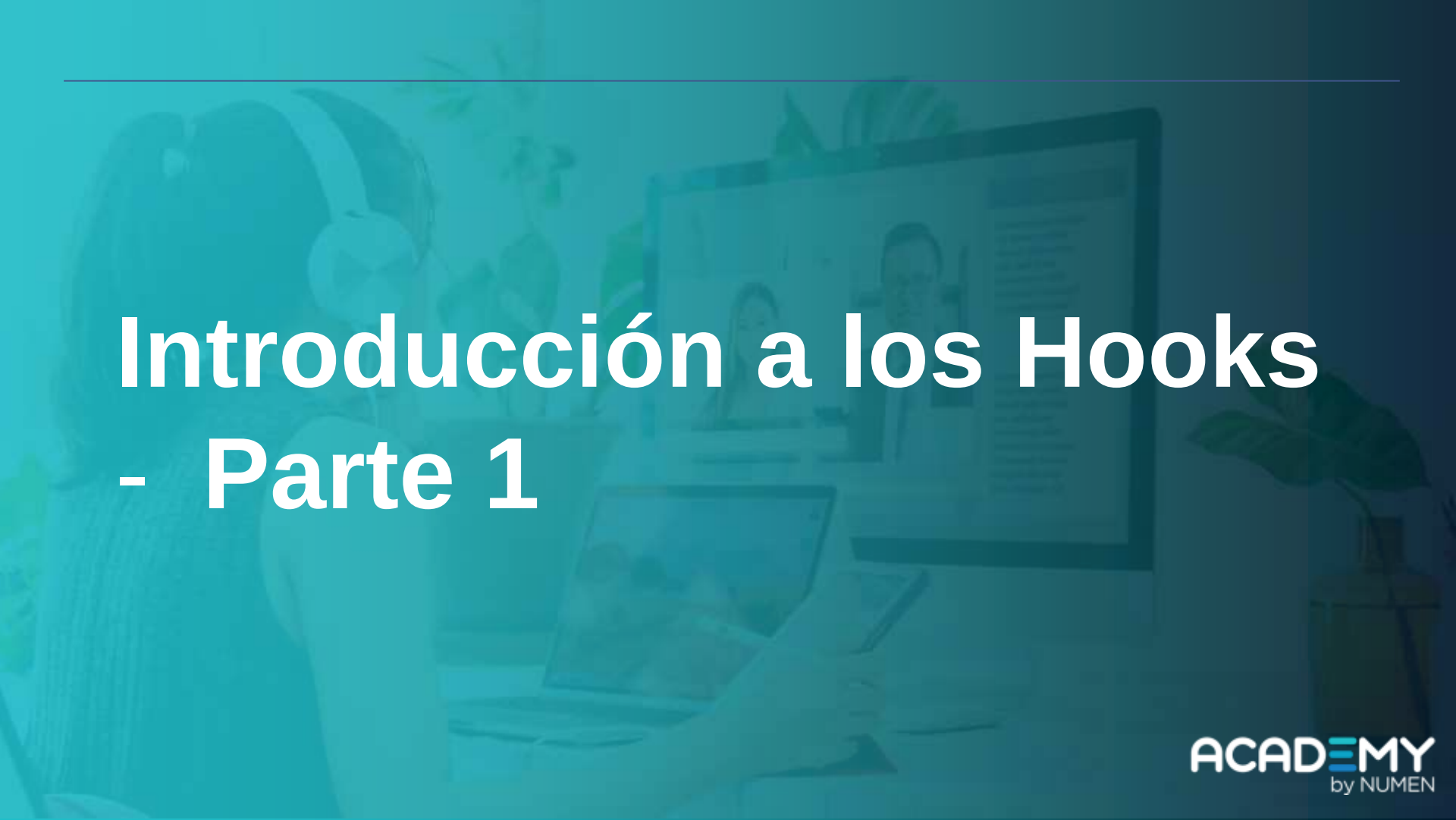
¿Cómo declarar y actualizar el Estado?

💡 En versiones anteriores de React, esto era solo posible en los class components, de manera que eran los únicos stateful components. No obstante, en el año 2019, React lanzó los Hooks - funciones especiales que te permiten trabajar con el estado de un componente, así como también otras características como ciclo de vida, referencias, context, etc.

🔥 Para declarar y actualizar el estado, utilizamos el hook *useState*. ¿Cómo funciona?

😉 Lo veremos en la próxima clase 😉

Introducción a los Hooks - Parte 1



ACADEMY by NUMEN

¿Qué son los Hooks?

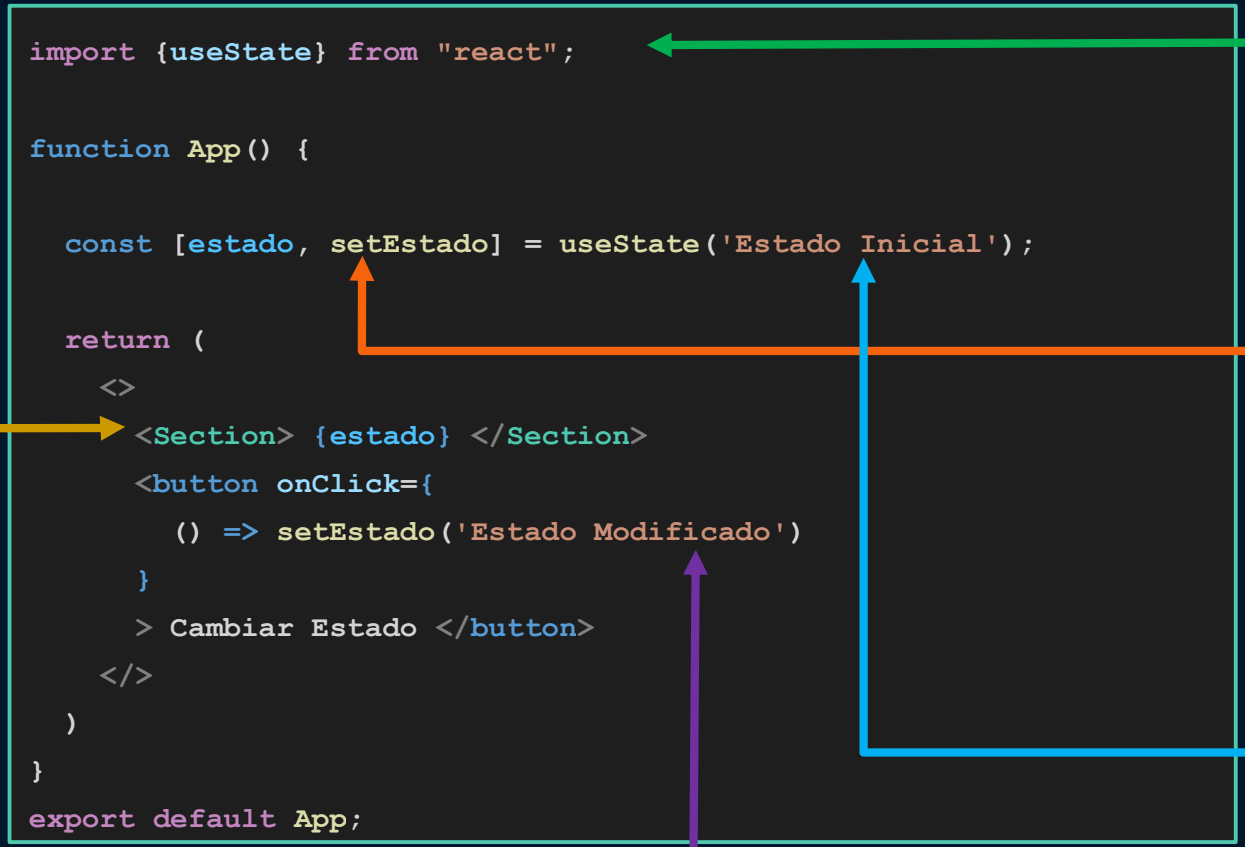
🔥 Los *hooks* son funciones nativas de React que permiten añadirles a los componentes funcionales, características que antes estaban limitadas a los componentes de clase, tales como el manejo de estado, referencias, contexto y ciclo de vida.


useState

useState() ➡ el hook que maneja el Estado

💡 Para declarar y actualizar el estado, utilizamos el hook *useState*. ¿Cómo funciona?

🔥 useState es una función que crea internamente una variable donde podremos almacenar el estado de nuestro componente. Acepta un valor inicial para esa variable y devuelve un array con dos elementos, el valor de la variable y la función para modificarla.



En primer lugar, importamos el hook de react.

Luego, declaramos una variable array a la cual asignamos useState. En dicha variable almacenamos dos valores: el estado y una función manejadora del estado.

Dentro de useState, declaramos el valor inicial de nuestro estado.

Para utilizar el estado, simplemente lo llamamos.

Para modificar el estado, llamamos a la función manejadora y le pasamos como parámetro el nuevo valor del estado.

¡Armemos un Contador con useState!

```jsx
import { useState } from "react";

const Contador = () => {

    const [contador, setContador] = useState(0);

    const sumar = () => setContador(contador + 1);
    const restar = () => setContador(contador - 1);

    return (
        <>
            <div style={{display: "flex"}}>
                <button onClick={sumar}>+</button>
                <h3>{contador}</h3>
                <button onClick={restar}>-</button>
            </div>
        </>
     );
}

export default Contador;
```

¿Cómo mejorar la performance del componente con useState?

```
const [lista, setLista] = useState(() => [
  {id: 1, title: "Lista en Estado 1"},
  {id: 2, title: "Lista en Estado 2"},
  {id: 3, title: "Lista en Estado 3"},
])
```

useState (initialState)

useState (() => initialState)

useState puede recibir una función que retorna el estado inicial. La ventaja de esto por sobre declarar el valor inicial directamente es que el estado inicial solo se carga una sola vez (la primera vez que se renderiza el componente). Caso contrario, se cargará en cada renderizado.

¿Qué tener en cuenta a la hora de hacer varias actualizaciones?

```
const cambiarEstado = () => {
  setLista(prevList => {
    return [
      ...prevList,
      {id: 4, title: "Lista Cambiada"}
    ]
  })
}
```

```
setState ( prevState => {
  return prevState + newState
} )
```

La función actualizadora del estado puede recibir como parámetro una función, que a su vez recibe como parámetro la última actualización del estado (el estado previo) y nos permite retornar el mismo, junto a la nueva actualización. Esto nos permite evitar volver a escribir el estado previo.

Ciclo de vida de un Componente

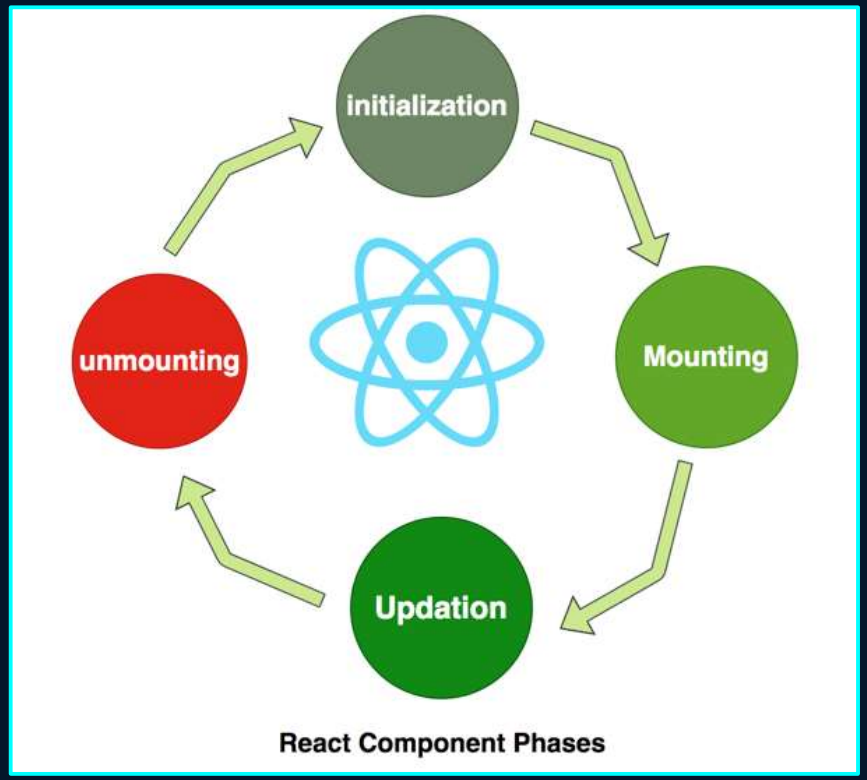
ACADEMY by NUMEN

Ciclo de Vida de un Componente

Son métodos que se ejecutan automáticamente en un Componente de Clase.

Ocurren en 3 fases:

1. Montaje
2. Actualización
3. Desmontaje



Fase de Montaje

Estos métodos se ejecutan cuando se crea un componente y se inserta en el árbol del DOM.

- **constructor()**: Se ejecuta al crear la instancia del componente, en el constructor puedes inicializar el estado y enlazar manejadores de eventos.
- **render()**: Es el único método requerido, cuando se ejecuta, examina el estado y las propiedades y dibuja el componente en el árbol del DOM.
- **componentDidMount()**: Se invoca inmediatamente después de que un componente se ha insertado en el árbol del DOM. Sirve para ejecutar peticiones a APIs, bases de datos, etc.

Fase de Actualización

Estos métodos son ejecutados por cambios en el estado o las propiedades de los componentes.

- **render()**: Redibuja el componente cuando detecta cambios en el estado o las propiedades del componente.
- **componentDidUpdate()**: Se ejecuta inmediatamente después de que la actualización del estado o las propiedades sucede.

Fase de Desmontaje

Estos métodos son ejecutados una vez que el componente ha sido eliminado del árbol del DOM.

- **componentDidUnmount()**: Se ejecuta antes de destruir el componente del árbol del DOM. Útil para limpiar código cuando no es necesario usarlo.


useEffect

useEffect() ➡ el hook que sincroniza los efectos

💡 Simula una, varias o todas las fases del Ciclo de Vida de un componente.

💡 Este hook ayuda a ***sincronizar*** el estado interno de un componente con algún estado externo, por ejemplo, obtener datos desde una API o modificar algo en el DOM.

🔥 *useEffect* ejecuta un efecto "secundario" definido como primer argumento a modo de callback.

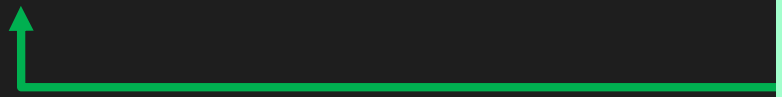
🔥 Este efecto es ejecutado cada vez que uno de los valores del arreglo de dependencias (segundo parámetro) ha cambiado.

```jsx
const Blog = () => {

    const [recurso, setRecurso] = useState('posteos');

    useEffect(() => {
        console.log('Efecto Secundario')
    }, [recurso]);

    return (
        <>
            <div>
                <button onClick={() => setRecurso('posteos')}>Posteos</button>
                <button onClick={() => setRecurso('usuarios')}>Usuarios</button>
                <button onClick={() => setRecurso('comentarios')}>Comentarios</button>
            </div>
            <h2>{recurso}</h2>
        </>
    )
};
export default Blog;
```

El useEffect recibe una función y una variable de dependencia.

Cada vez que haya un cambio en esta dependencia, el efecto secundario se va a ejecutar.

```jsx
const Blog = () => {

    const [windowWidth, setWindowWidth] = useState(window.innerWidth);

    const handleResize = () => {
        setWindowWidth(window.innerWidth)
    }

    useEffect(() => {
        window.addEventListener('resize', handleResize)

        return () => {
            window.removeEventListener('resize', handleResize)
        }
    }, [windowWidth]);

    return (
        <>
            <div>{windowWidth}</div>
        </>
    )
};
export default Blog;
```



Si se le pasa una función como retorno del useEffect, dicha función se ejecuta luego del efecto secundario.

¡Armemos un Reloj! - Parte 1 (Componentes y JSX)

```
function Reloj({ hora }) {
   return <h3>{hora}</h3>;
 }

 const RelojEffect = () => {

          {/*Aqui van los hooks */}

     return (
       <>
         <h2>Reloj con Hooks</h2>
         {  visible ? <Reloj hora={hora} /> : null }
         <button onClick={() => setVisible(true)}>iniciar</button>
         <button onClick={() => setVisible(false)}>detener</button>
       </>
     );
   }
 export default RelojEffect;
```

¡Armemos un Reloj! - Parte 2 (Hooks)

```
const [hora, setHora] = useState(new Date().toLocaleTimeString());
const [visible, setVisible] = useState(false);

useEffect(() => {
  let temporizador;

  if (visible) {
    temporizador = setInterval(() => {
        setHora(new Date().toLocaleTimeString());},
    1000);
  } else {
    clearInterval(temporizador);
  }

  return () => {
    clearInterval(temporizador);
  };
}, [visible]);
```

ACADEMY
by NUMEN