Coding Bootcamp

Sprint 4

Temario

Temario

- Ciclo de vida de un componente.
- External Data Access.
- Props.children.

Ciclo de vida
de un componente

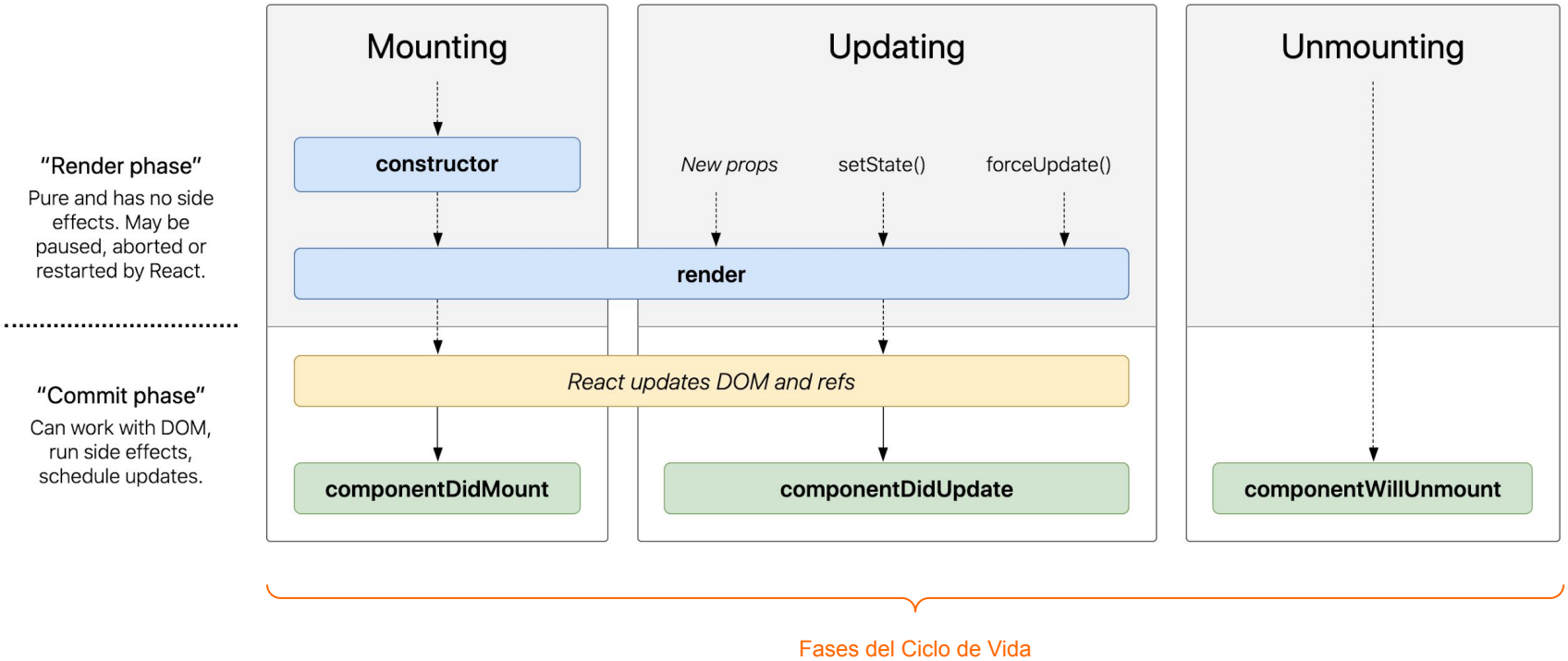
Ciclo de vida de un componente (1/2)

El ciclo de vida de un componente se compone de eventos que ocurren desde el momento en que se crea un componente hasta el momento en que se elimina un componente del DOM.

El ciclo de vida de un componente se divide en tres fases:

1. Cuando un componente se **Monta** en la pantalla (renderiza en el DOM).
2. Cuando un componente se **Actualiza**.
3. Cuando un componente se **Elimina** del DOM.

Ciclo de vida de un componente (2/2)



Métodos de ciclo de vida en un Class Component

```
class MiComponente extends Component {
  constructor(props) {
    super(props);
  }

  componentDidMount() { console.log("Se montó el componente. Se ejecuta luego del método render (inicial)."); }
  componentDidUpdate() { console.log("Se actualizó el componente. Se ejecuta luego del método render (excepto el inicial)."); }
  componentWillUnmount() { console.log("El componente está a punto de desmontarse y destruirse."); }

  render() {
    return (
      <h1>¡Hola alumnos de Hack Academy!</h1>
    );
  }
}
```

Cada Class Component dispone de varios “métodos de ciclo de vida” que se pueden sobreescribir para ejecutar cierto código en determinados momentos del proceso.

Ciclo de vida en un Functional Component (1/6)

Usando el *hook* `useEffect`, dentro de un Functional Component, se puede lograr un comportamiento similar al que se logra con los *Lifecycle Methods* de un Class Component.

De hecho, se podría decir que usar `useEffect` es como usar `componentDidMount`, `componentDidUpdate`, y `componentWillUnmount`, de forma combinada.

Ciclo de vida en un Functional Component (2/6)

Para lograr un funcionamiento similar a `componentDidUpdate` en un Functional Component, se utiliza el `useEffect`, tal como viene por defecto.

```
function MiComponente() {
  useEffect(() => {
    console.log("El componente se acaba de montar (primer render) o actualizar.");
  });
  return <h1>¡Hola alumnos de Hack Academy!</h1>;
}
```

El `console.log`se ejecutará cada vez que se actualiza el componente, ya sea porque cambian sus props o su estado y, por lo tanto, cada vez que se renderiza.

Ciclo de vida en un Functional Component (3/6)

Para lograr un funcionamiento similar a `componentDidMount` en un Functional Component, es necesario pasarle al `useEffect` un segundo parámetro con un array de dependencias vacío.

```
function MiComponente() {
  useEffect(() => {
    console.log("El componente se acaba de montar (primer render).");
  }, []);
  return <h1>¡Hola alumnos de Hack Academy!</h1>;
}
```

Ciclo de vida en un Functional Component (4/6)

Para lograr un funcionamiento similar a `componentWillUnmount` en un Functional Component, es necesario pasarle al `useEffect` un segundo parámetro con un array de dependencias vacío y, además, especificar un `return` en la función “efecto”.

```
function MiComponente() {

  useEffect(() => {

    console.log("El componente se acaba de montar (primer render).");

    return () => console.log("El componente está a punto de desmontarse y destruirse.");

  }, []);

  return <h1>¡Hola alumnos de Hack Academy!</h1>;

}
```

A esta función se le llama “función de **limpieza**” o “función de **cleanup**”.

Ciclo de vida en un Functional Component (5/6)

Si se quiere ejecutar cierto código **antes** de cada actualización del componente y/o luego de que se desmonte, es necesario especificar un `return` en la función “efecto” y no pasar como segundo parámetro del `useEffect` un array de dependencias.

```
function MiComponente() {
  useEffect(() => {
    return () => console.log("El componente se va a actualizar o se acaba de desmontar.");
  });
  return <h1>¡Hola alumnos de Hack Academy!</h1>;
}
```

A esta función se le llama “función de **limpieza**” o “función de **cleanup**”.

Ciclo de vida en un Functional Component (6/6)

⚠️ En caso de utilizar el código de la diapositiva anterior, es necesario tener cuidado con lo siguiente:

Si un componente que ya está montado, tiene especificada tanto una función “efecto” (1er `console.log`) como una función “cleanup” (2º `console.log`), y no se especifica un array de dependencias, cuando se actualice el componente se ejecutará primero la función de “cleanup” y luego el “efecto”. Ver demo.

```
function MiComponente() {

  useEffect(() => {

    console.log("Mensaje 1");

    return () => console.log("Mensaje 2");

  });

  return <h1>¡Hola alumnos de Hack Academy!</h1>;

}
```

⚠️ En este ejemplo, al actualizarse el componente, se mostrará primero “Mensaje 2” y luego “Mensaje 1”.

Al montarse (primer render), se mostrará el “Mensaje 1”.

External Data Access

External Data Access

Existen varias opciones en JavaScript para resolver el acceso a datos externos (llamadas AJAX). Por ejemplo, se podría utilizar Fetch e incluso jQuery.

Se sugiere usar la librería **Axios**, ya que es una de las más populares. Está basada en las Promesas de JavaScript (fundamental para evitar un *callback hell*).

📖 Ver documentación de Axios.

⚙️ Para instalar Axios en el proyecto de React, ejecutar el siguiente comando:

```
npm i axios
```

External Data Access – Ejemplo (1/2)

Para poder modificar el estado del componente, luego de que la llamada AJAX haya finalizado, es necesario utilizar el *hook* `useEffect`. Ejemplo de Axios con promesas:

```
import React from "react";
import axios from "axios";

export default function (props) {
  const [personaje, setPersonaje] = React.useState(null);

  React.useEffect(() => {
    axios.get("https://swapi.dev/api/people/1").then((response) => {
      setPersonaje(response.data);
    });
  }, []);

  return personaje && <p>¡Hola {personaje.name}!</p>;
}
```

La función `useEffect` permite agregar *side effects* (efectos secundarios) a un componente funcional. Por efectos secundarios nos referimos a *data fetching*, *subscriptions*, manipulación de DOM.

`useEffect` recibe como primer parámetro, una función (usualmente denominada "efecto"), la cual se ejecutará cada vez que se actualiza el componente, ya sea porque cambian sus *props* o su estado.

Por defecto, al usar este *hook*, la función "efecto" es llamada luego de cada renderizado del componente. Este comportamiento se puede modificar pasándole un segundo parámetro a `useEffect`, que es un array de dependencias. Si el array está vacío, el "efecto" sólo se ejecutará una vez, luego del primer render.

External Data Access – Ejemplo (2/2)

Para poder modificar el estado del componente, luego de que la llamada AJAX haya finalizado, es necesario utilizar el *hook* `useEffect`. Ejemplo de Axios con async/await:

```
import React from "react";
import axios from "axios";

export default function (props) {
  const [personaje, setPersonaje] = React.useState(null);

React.useEffect(() => {
   const getPersonaje = async () => {
     const response = await axios.get("https://swapi.dev/api/people/1");
     setPersonaje(response.data);
   };
   getPersonaje();
 }, []);

  return personaje && <p>¡Hola {personaje.name}!</p>;
}
```

La función `useEffect` permite agregar *side effects* (efectos secundarios) a un componente funcional. Por efectos secundarios nos referimos a *data fetching*, *subscriptions*, manipulación de DOM.

`useEffect` recibe como primer parámetro, una función (usualmente denominada "efecto"), la cual se ejecutará cada vez que se actualiza el componente, ya sea porque cambian sus *props* o su estado.

Por defecto, al usar este *hook*, la función "efecto" es llamada luego de cada renderizado del componente. Este comportamiento se puede modificar pasándole un segundo parámetro a `useEffect`, que es un array de dependencias. Si el array está vacío, el "efecto" sólo se ejecutará una vez, luego del primer render.

Props.children

Props.children (1/2)

Cuando las expresiones JSX tienen una etiqueta de apertura y de clausura, el contenido entre ellas es enviado al componente como una propiedad especial, que se accede mediante `props.children`.

Estos “hijos” pueden ser de distintos tipos.

En el siguiente ejemplo, `props.children` es un string:

```
<MiComponente>¡Hola alumnos de Hack Academy!</MiComponente>
```

Props.children (2/2)

Esto permite crear componentes que agreguen comportamiento a la aplicación y simplemente envolviendo a otros componentes. Pero para esto el “container” debe explícitamente renderizar a sus hijos, de la siguiente forma:

```
<MiComponente>
    <MiPrimerComponente />
    <MiSegundoComponente />
</MiComponente>
```

```
function MiComponente(props) {
  return <div>{props.children}</div>;
}
```

👉 Ver demo.