Índice

Introducción de la clase

En las próximas dos clases vamos a dar un repaso de algunas características de JavaScript que van a ser útiles para seguir las clases de React. Se asume para estas clases que ya has trabajado antes con JavaScript, por lo que en estas clases no nos centraremos en los fundamentos, si no en los detalles de ciertas funcionalidades que son útiles en React.

Funciones

Las funciones en JavaScript son importantísimas, y esto se refleja en la evolución que ha tenido React, donde principalmente trabajamos con funciones y con conceptos de programación funcional, un paradigma de programación que se centra precisamente en el uso de funciones.

Comencemos por un repaso de cómo crear una función, para lo que existen dos formas distintas, declaración de función, y una expresión de función.

Cuando iniciamos con la palabra reservada function una nueva línea de código,se trata de una declaración de función, si no, es una expresión de función.

Una de las diferencias clave es que cuando hacemos declaración de función, la función debe de tener un nombre, esto lo confirmamos si escribimos:

```
function(){
  console.log(“Hola mundo”);
}
```

Mientras que en una expresión de función, el nombre que se coloca después de function es irrelevante, por lo que no hay razón para colocarlo

```
let f = function(){}
let f2 = function saludar(){}; // saludar no está definido
```

JavaScript mueve la declaración de funciones y variables declaradas con var, a lo más alto del alcance, a esto lo llamamos hoisting. En términos prácticos esto significa que podemos usar una función incluso antes de escribirla, porque JavaScript moverá la declaración al inicio:

```
f();

function f(){ console.log(“Hola mundo”); }
```

Esto solo sucede cuando usamos declaración de funciones, en expresión de funciones el proceso de hoisting no aplica:

```
f(); // Error

let f = function (){ console.log(“Hola mundo”); }
```

Existe una sintaxis alternativa, cuando usamos expresión de funciones, las funciones de flecha o arrow functions:

```
()=>{}
```

Ahondaremos más en el tema de funciones de flecha cuando hablemos del contexto, por ahora cabe mencionar que las funciones de flecha cumplen las características de una expresión de función.

Es importante mencionar que, aunque en expresión de función el nombre es opcional, la forma en que podemos asignar un nombre a este tipo de funciones es asignándolas a una variable:

```
let f = ()={} // Puede entenderse como que el nombre de esta función es f
```

En resumen:

- Una función puede ser “creada” con una declaración de función o una expresión de función
- Las declaraciones de función se mueven al principio del alcance, por lo que pueden ser usadas antes de aparecer en el código, a esto se le llama hoisting
- Las expresiones de función pueden o no tener nombre.
- Las expresiones de función sin nombre, se conocen como funciones anónimas.
- Las expresiones de función pueden usar la sintaxis de flecha

El alcance o scope

El scope es una colección de variables, funciones y objetos, que están a tu alcance en algún punto de tu código. También podemos entenderlo como las reglas que definen en qué partes del código una variable está disponible y en cuáles no.

En javaScript existen dos tipos de scope: el scope global y el scope local.

El alcance global es todo aquello que no ha sido declarado dentro de un bloque o una función.

El alcance local hace referencia a todos los elementos disponibles solo para una función. Cada vez que llamas una nueva función, se crea un alcance local para esta función.

La diferencia es que las variables globales se pueden usar tanto en el alcance global como en el local, esto quiere decir que las variables globales pueden ser usadas dentro de una función, por su parte, las variables locales, solo pueden usarse dentro de la función declarada.

```
let nombre = "Uriel";

function saludar(){
  let edad = 22;
  console.log(`${nombre} tiene ${edad} años de edad `);
}

saludar();
console.log(nombre)
console.log(edad);
```

Esto puede ser explicado con esta analogía de un gimnasio: Los aparatos para hacer ejercicio se colocan en el alcance global y cualquier persona puede usarlos.

Si tu llevas tu mochila con tu agua y quizás una toalla, estos elementos no son del alcance global, son del alcance local porque otras personas no pueden tomarlas y usarlas, solo tú.

Este ejemplo es interesante porque destaca uno de los problemas de usar el alcance global, todos dependen de los mismos elementos, tal que sí alguien hace un mal uso de uno de estos elementos, todos se ven afectados. En la analogía, si alguien rompe un aparato o lo ensucia, otros no podrán usarlo.

A diferencia, el alcance local, aunque igual es propenso a bugs, su impacto puede ser más limitado, pero sobre todo, más fácil de identificar.

Esto es similar a decir, si tengo un cuarto donde guardo dinero, y 10 personas tienen la llave de ese cuarto, ¿quién es responsable si el dinero desaparece?

En lo que respecta a las funciones, y esto es algo de lo que hablaremos más a profunidad en la clase de programación funcional, una función debe operar solo con la información del alcance local, y todo aquello que necesite del exterior debe comunicarse por argumentos, y todo lo que necesita comunicar hacia el exterior debe ser vía el retorno.

En resumen, tenemos dos tipos de scope
* El local
* El global

El global recuerda que está disponible en todas las partes del código, pero no es recomendable abusar de uso porque puede añadir complejidad al código.

El local es el que se crea para cada función cuando esta se ejecuta, todo lo que reside ahí, sólo vive mientras se ejecuta la función que estamos usando, de ser posible,se recomienda que prefieras colocar tus elementos en el scope local de una función y no en el global.

El alcance de bloque

Con la introducción de let y const, también se introduce el alcance de bloque, para el que los recursos definidos dentro de un bloque de ejecución, sólo están disponibles para ese bloque, y no para toda la función. Veamos.

Considera el siguiente ejemplo:

```
function Presentacion(nombre, admin){

  if(admin) {var rol = 'Administrador';}

  console.log(`Soy ${nombre} y mi rol es: ${rol}`);
}

Presentacion('Uriel', true);
```

Qué pasa si en lugar de var, usamos let:

```
  if(admin) { let rol = 'Administrador'; } // Error rol no está definido
```

Esto pasa porque las variables y constantes que se definen con let y const, son de alcance de bloque, y sólo son visibles dentro del bloque en el que han sido declaradas, en este caso solo dentro del cuerpo de la condición.

Let y const tienen alcance dentro del bloque más próximo, mientras que var dentro de la función más próxima. En palabras prácticas, si declaras una variable con let o const, dentro de un ciclo o una condición, esta estará disponible sólo dentro de estas estructuras, además

claro puedes declararlas dentro de una función y colocarlas en el scope local, siempre y cuando, como dije antes, no estén dentro de otro bloque.

Hoisting de variables

Para complementar el tema, y una vez que definimos que una de las principales diferencias entre let y var, es que let es de alcance de bloque y var es de alcance global, hablemos del concepto de hoisting para variables.

Como definimos antes, hoisting es el concepto por el cuál, la declaración de funciones y la declaración de variables, se mueven al inicio del alcance, lo cuál nos permite usar estos elementos antes de ser “declarados”.

Veamos un ejemplo:

```
console.log(x);
var x = 20;
```

¿Por qué imprime undefined?
Esto sucede porque la declaración de la variable se movió al principio del scope, pero no la asignación, por lo que la variable existe pero aún no tiene valor al momento de la impresión.

Ahora, qué pasará si usamos let:

```
console.log(x);
let x = 20;
```

En este caso la ejecución del código termina en uno error dado que las variables que usan let, antes de una asignación, tienen como valor “uninitialized”.

En resumen, diferenciamos var y let porque:

- var siempre se aloja en el scope local dentro de una función, o en el global fuera de una función
- let puede alojarse en el scope, local, de bloque, y global, según sea declarada en una función, en un bloque, o fuera de ambos.
- El valor para una variable con var, que no ha sido asignada, es undefined.
- El valor para una variable con let, que no ha sido asignada, es unitialized.
- Las variables también son alojadas al tope del scope, sin embargo, si intentas usar una variable no inicializada, recibirás un error.

El contexto

El contexto es el valor que tiene **this** y usualmente hace referencia al objeto sobre el que se está ejecutando el código. En adelante puedes asumir que para esta sesión los conceptos de this y el contexto son intercambiables.

Cuando una función no pertenece a un objeto, el contexto es el objeto global, en el caso del navegador sería el objeto window:

```
function contexto(){
  return this;
}

console.log( contexto() );
```

Esto se explica ya que una función de alcance global se llama a través del objeto global, por eso el contexto es dicho objeto.

El valor del contexto cambia cuando llamamos nuestra función a través de un objeto:

```
let objeto = {
 f: contexto
}

objeto.f();
```

Esto nos lleva a recordar algo muy importante:

El valor del contexto es el objeto que ejecuta la función.

La clave para entender qué valor adopta el contexto es que el valor depende de dónde se ejecuta la función, y no depende de dónde se declara.

```
let obj = {
  x: function(){ return this; }
}

let y = obj.x;

console.log( obj.x() );
console.log( y() );
```

El contexto y las funciones de flecha

Las funciones de flecha, se diferencían, además de por su sintaxis alternativa, en que no asignan un valor a this, lo heredan del contexto en que fueron ejecutadas.

Como se decribe aquí, si usas this dentro de una función de flecha, este se toma del exterior.

Esta característica de no hacer binding de this, delimita los usos que debemos dar a las funciones de flecha, principalmente cuando introducimos funciones anónimas, por ejemplo:

```
let obj = {
  numbers: [1,2,3],
  print: function(){
    console.log(this);
    this.numbers.forEach(function(){ console.log(this) // Tal vez necesitemos el obj })
  }
}

obj.print();
```

En este ejemplo vemos cómo, la función anónima que se ejecuta en cada iteración de forEach asigna su propio valor de this, tal como todas las funciones declaradas con la palabra function hacen.

En estos escenarios donde muy probablemente no deseamos cambiar el valor del contexto en la función anónima, podemos usar funciones de flecha:

```
let obj = {
  numbers: [1,2,3],
  print: function(){
    console.log(this);
    this.numbers.forEach(() => { console.log(this) // Tal vez necesitemos el obj })
  }
}

obj.print();
```

En resumen:

- Las funciones de flecha no cambian el valor de this, lo heredan o lo toman de “afuera”.
- Es por esto que las funciones de flecha no deben usarse como métodos
- Esto abre la puerta a usar las funciones de flecha cuando no queremos que el contexto en el cuerpo de la función, cambie

Paso por valor y paso por referencia

En JavaScript, un argumento puede ser enviado por valor o por referencia. Esta diferencia influye en qué sucede con el argumento original después de la ejecución de la función.

Decimos que un valor ha sido enviado como referencia, cuando el parámetro apunta a la misma dirección que el argumento original, cualquier modificación sobre este valor afecta al argumento original.

Por su parte, decimos que pasamos por valor cuando el argumento enviado es copiado en una dirección distinta para el parámetro, cualquier modificación al parámetro no afecta el valor original.

Cualquier valor que no sea un objeto o un arreglo es pasado por referencia, esto incluye números, cadenas, booleanos, entre otros. Cuando el argumento es enviado, se genera una copia propia que se usará dentro de la función:

```
let edad = 20;

function modificador(edad){ edad = 25; }

console.log(edad);
modificador(edad);
console.log(edad);
```

Con este ejemplo vemos como el valor al que apunta la variable edad se mantiene intacto, a pesar de que el parámetro fue modificado en la ejecución de la función.

Por su parte, los objetos y los arreglos son enviados por referencia. Esto quiere decir que al enviar estas estructuras como argumento, se envía la dirección en memoria en la que están guardados, el parámetro apunta a la misma dirección, por lo que si modificamos el valor, el cambio se refleja fuera de la función:

```
let edades = [20]; // Aún no vemos arreglos

function modificador(edades){ edades[0] = 25; }

console.log(edades);
modificador(edades);
console.log(edades);
```

Para este ejemplo, la variable edades refleja los cambios hechos en la ejecución de la función porque apunta a la misma dirección en que la modificación sucedió.

¿Por qué esto es importante para ti? A lo largo de la clase de programación funcional, y durante tu trabajo con React aprenderás del concepto de “efectos secundarios”, y los valores pasados por referencia son una puerta a que una función produzca efectos secundarios, pero hablaremos más de eso en la siguiente clase.

Promesas

JavaScript es un lenguaje de programación asíncrono que se ejecuta sobre un solo hilo. Esto significa que un proceso lento y tardado que no es procesado de manera asíncrona puede bloquear la ejecución del resto de nuestro código.

Para solucionar esto, JavaScript introduce el event loop, o ciclo de eventos. El event loop se compone de dos componentes principales, una cola de mensajes y un ciclo que se encuentra iterando esta cola de mensajes. La programación asíncrona en JavaScript funciona empujando ciertas operaciones a esta cola de actividades, para que no bloqueen la ejecución de código mientras terminan, el trabajo del event loop es estar preguntando las operaciones de la cola de actividades si ya han finalizado, y cuando lo hacen, reanuda la ejecución de dicha operación.

Para poder trabajar con el event loop JavaScript introduce el concepto de promesas, que nos permiten definir código a ejecutarse cuando una tarea fue finalizada.

Una promesa es un tipo de objeto retornado por una operación asíncrona, utilizando este objeto podemos obtener el valor final de la operación asíncrona, o los posibles errores que encuentre en su ejecución. Es por esto que comúnmente decimos que, una promesa es un objeto que produce un valor en el futuro.

Una promesa puede encontrarse en alguno de los siguientes estados:

fullfiled: O Completada, significa que la promesa se completó con éxito
rejected: O rechazada, significa que la promesa no se completó con éxito
pending: O pendiente que es el estado de la promesa cuando la operación no ha terminado, aquí decimos que la promesa no se ha cumplido.
settled: O finalizada, cuando la promesa terminó ya sea con éxito o con algún error.

Veamos un ejemplo, para eso usaré fetch, una función del navegador que nos permite hacer operaciones asíncronas y que como resultado entrega una promesa:

let p = fetch("https://api.github.com/users/codigofacilito");

Si inspeccionamos el objeto p, podemos ver que se trata de una promesa:

```
console.log(p);
console.log(typeof p);
```

La función fetch, al ser una operación asícrona retorna un objeto promesa, y no el resultado de la operación realizada (consultar una página), es a través de la promesa que podemos acceder al resultado de la operación asíncrona:

La forma en que obtenemos el resultado de una promesa, y podemos ejecutar código cuando esta ha finalizado, es enviando funciones a métodos específicos de las promesas que ejecutarán dichas funciones cuando la operación asíncrona finalice.

Existen 3 métodos en una promesa para los que podemos añadir código que se evalúe en distintos estados de la promesa

- then: Ejecuta el callback cuando la promesa se cumplió con éxito
- catch: Ejecuta el callback cuando la promesa no se cumplió
- finally: Independientemente de si se cumplió o no, ejecutó el callback.

Por ejemplo:

```
p.then( ()=> console.log(d) ).catch( err => console.log(err) );
```

Encadenar promesas

Cuando el callback de una promesa, retorna otra promesa, podemos encadenar las operaciones then:

```
// Esto no
p.then( d => d.json().then( r => console.log(r)) )

// Mucho mejor
p.then( d => d.json() ).then( r => console.log(r));
```

Si una de las promesas falla, la promesa se marca como “rejected” y se ejecuta el callback de catch asociado:

```
p.then( d => d.json() ).then( r => console.log(r)).catch(err => console.log(err));
```

Cabe mencionar que cuando encadenamos promesas, se ejecuta la siguiente operación asíncrona, hasta que la enterior ha terminado. Es por eso que solo encadenamos promesas cuando una tarea asíncrona depende o necesita que otra haya finalizado.

Si, por otro lado, deseas lanzar varias operaciones asíncronas que no dependen una de la otra, es mejor usar Promise.all

Async/Await

Las funciones asíncronas permiten que usemos expresiones “await” dentro del cuerpo de la función. Estas expresiones nos permiten trabajar con promesas como si fueran código síncrono.
Async y await son una mejora sintáctica para el trabajo con código asíncrono, veamos cómo funcionan.
Una función asíncrona es aquella cuya declaración inicia con la palabra async:

```
async function leerRepos(){
}
```

Dentro del cuerpo de una función asíncrono puedes usar expresiones await para trabajar con promesas, por ejemplo:

```
async function leerRepos(){
  let data = await fetch("https://api.github.com/users/codigofacilito");
}
```

Esto es igual a escribir:

```
function leerRepos(){
  fetch("https://api.github.com/users/codigofacilito").then(data => {});
}
```

Como puedes ver, es una diferencia sintáctica para el que argumentablemente el códig ode async/await es más legible.

Esto nos permite trabajar con código asíncrono como si fuera código síncrono, sin la complejidad de la sintaxis para encadenar promesas:

```
async function leerRepos(){
  let data = await fetch("https://api.github.com/users/codigofacilito");
  let jsonData = await data.json();
  console.log(jsonData);
}
```

Cualquier función asíncrona retorma explicita o implícitamente una promesa, es por eso que, aunque jsonData es un objeto, si lo retornamos, irá “envuelto” en una promesa:

```
async function leerRepos(){
  let data = await fetch("https://api.github.com/users/codigofacilito");
  let jsonData = await data.json();
  return jsonData;
}

console.log(typeof jsonData);
```

Los errores asíncronos de una función pueden ser manejados usando try y catch:

```
async function leerRepos(){
  try{
    let data = await fetch("https://codigofacilito.com/articulos/");
    let jsonData = await data.json();
    console.log(jsonData);
  }catch(err){
    console.log(err);
  }
}
```

Si una función asíncrona arroja un error que no fue manejado por un try/catch, este puede ser capturado fuera por la promesa que retorna:

```
async function leerRepos(){
  let data = await fetch("https://codigofacilito.com/articulos/");
  let jsonData = await data.json();
  return jsonData;
```

```
}

leerRepos().catch(err => console.log(err));
```