

Esenciales de JavaScript

Ejercicios para el taller

Links a los ejercicios en [codesandbox](#).

Introducción

Esta clase es un repaso a los elementos de JavaScript que son fundamentales para trabajar con React. En este punto se asume que ya has trabajado con JavaScript antes.

Funciones

Las funciones son la base de React. Este, se define a sí mismo como una librería **declarativa**. Donde declarativo hace referencia al paradigma de programación (declarativo), donde describimos el comportamiento en lugar de describir las estructuras de control.

Por lo tanto las funciones son el mecanismo por el cual nosotros describimos los comportamientos de nuestras aplicaciones.

Una función se puede crear de dos formas distintas, a través de una declaración o una expresión.

Declaraciones

La declaración de una función define una función con sus parámetros. Usamos la palabra clave `function`.

Para crear una función a través de una declaración es necesario nombrar nuestra función, opcionalmente puede tener parámetros, a ser usados dentro de la función.

```
function saludar() {  
    console.log("Hola mundo");  
}
```

Expresiones

A diferencia de las declaraciones, aquí las funciones se crean dentro de expresiones y son asignadas a variables.

El nombre de la función es opcional y si no es asignado, la función tomará el nombre de variable a la que se está asignando la expresión.

```
let saludar = function () {  
    return console.log("hola mundo");  
}  
  
// saludar.name // "saludar"
```

Puede verse en la imagen que la creación por una expresión de una función, se compone de:

- La declaración de una variable
- La declaración de una función
- La expresión que asigna la declaración en la variable.

Hoisting

Hay algunas diferencias entre crear una función a través de una expresión y a través de una declaración.

Cuando usamos una expresión, JavaScript hace *hoist* de la declaración de la función, esto quiere decir, que mueve la declaración al inicio del **scope**. *Hablaremos del scope más adelante.*

```
...  
// Invoca antes de declarar  
saludar();  
  
// Declaración  
function saludar() {  
  console.log("hola mundo");  
}  
  
// Console: "hola mundo"
```

```
// Invoca antes de declarar  
saludar();  
  
// Creación por expresión  
let saludar = function saludar() {  
    console.log("hola mundo");  
}  
  
// Console: "Uncaught ReferenceError: saludar is not defined"
```

Datos adicionales sobre expresiones

Algunos datos extra sobre expresiones son:

Como en el ejemplo anterior, podemos darle nombre a una función al momento de la declaración (let saludar = function saludar2), pero esta declaración no es alcanzable fuera de la expresión, por lo que puede ignorarse.

```
● ● ●

let saludar = function saludar2() {
  // Puedo acceder a saludar2 en este Scope.
  console.log(saludar2.name);
}

saludar();

// Console: "saludar2"
```

Podemos hacer funciones por expresiones también con **arrow functions**, pero hablaremos de estas más adelante.

En resumen

Las funciones son la forma de crear declaraciones en JavaScript. Podemos crearlas de dos formas, a través de expresiones o declaraciones. Las expresiones a diferencia de las declaraciones, tienen reglas diferentes, como el hoist sobre la declaración y el nombre que recibe la función.

En React podrán declarar componentes así como sus event handlers siguiendo cualquiera de las dos formas anteriores, teniendo claro las diferencias y conveniencias de cuando usar una u otra forma.

Scope

El scope es el contexto actual de la ejecución. En el contexto podemos encontrar los valores y variables que son visibles o referenciales.

Global

El alcance global es el que ocurre donde nuestra aplicación está siendo ejecutada. Básicamente es la raíz de nuestro script, donde creamos las primeras declaraciones y expresiones.

Local

El alcance local es aquel que ocurre dentro de la ejecución de una función y solo son visibles dentro de su ejecución.



```
// global scope
let nombre = "Jerome";
let curso = "Esenciales de JavaScript";

function impartirCurso() {
    // function scope
    let canal = "Código Facilito"
    console.log(`Se imparte curso ${curso} por ${nombre} a través de ${canal}`);
}

impartirCurso();
```

En nuestra aplicación podemos tener varios grupos de scopes y también scopes anidados, como mencionamos al inicio, el scope global ocurre en la parte más baja, por lo que todos las funciones que ejecutemos podrán ver los valores declarados ahí.

Sin embargo, todo lo que ocurra dentro del alcance de una función, no podrá ser visto por el scope global.

```
// global scope
let nombre = "Jerome";
let curso = "Esenciales de JavaScript";

function impartirCurso() {
    // function scope
    let canal = "Código Facilito"
    console.log(`Se imparte curso ${curso} por ${nombre} a través de ${canal}`);
}

console.log(canal); // Uncaught ReferenceError: canal is not defined

impartirCurso();
```

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 si 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.

```
// global scope
let nombre = "Jerome";
let curso = "Esenciales de JavaScript";

function impartirCurso() {
    // function scope
    curso = "Cocina"
    let canal = "Código Facilito"
    console.log(`Se imparte curso ${curso} por ${nombre} a través de ${canal}`);
}

impartirCurso();
```

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

```
// global scope
let nombre = "Jerome";
let curso = "Esenciales de JavaScript";

function impartirCurso() {
    // function scope
    curso = "Cocina"
    let canal = "Código Facilito"
    console.log(`Se imparte curso ${curso} por ${nombre} a través de ${canal}`);
}

function editarVideo() {
    // function scope
    curso = "Matemáticas"
    console.log(`Se edita video de curso ${curso}`);
}

// ??? Cuál será el valor final y donde será que habrá cambiado?
```

En resumen

En resumen, tenemos el alcance global y el alcance local, el local puede ver y acceder a las declaraciones y expresiones declaradas en el scope global, pero el global no puede acceder al scope local.

Ambos son útiles, pero hay que tener precaución al acceder a las variables del alcance global, ya que, como vimos en el ejemplo, podemos introducir bugs al sistema si no se hace de la forma correcta. En la parte de programación funcional, veremos formas de evitar romper este sistema.

En React cada componente tendrá su propio scope local, ya que cada componente es una función y ahí solo podrán utilizar lo que esté en su mismo alcance o el global. Dentro de los componentes crearán event handlers, los cuales tendrán acceso al scope del componente y nuevamente crearán un scope local anidado exclusivo para ese event handler

Un dato curioso, es que si has desarrollado una aplicación web antes, estás familiarizado con el objeto window. Este es parte del global scope dado que nuestra aplicación se ejecuta EN el navegador y este objeto es declarado en el scope en el que nuestra función también se está ejecutando.

Block scope

Otro tipo de scope que tenemos es aquel que se crea en la declaración de un bloque. El bloque está definido por las llaves {} utilizadas en las sentencias: if, for, while.

En JavaScript tenemos tres formas de declarar variables.

- Let: para declarar variables con un alcance de bloque.
- Const: para declarar valores no reasignables con alcance de bloque.
- Var: para declarar variables con un alcance de función.

Let y const sólo podrán ser “vistas” dentro del bloque donde se declararon mientras que Var podrá ser “vista” por todo el bloque que la función haya creado.

```
● ● ●

function imprimirSecreto(clave) {
  if (clave === "supersecreta") {
    // Var con alcance de la función completa.
    var secreto = "42";
  }

  console.log(`El secreto es ${secreto}`);
  // Console: El secreto es 42
}

imprimirSecreto("supersecreta");
```

```
function imprimirSecreto(clave) {  
    if (clave === "supersecreta") {  
        // let con alcance solo del bloque if.  
        let secreto = "42";  
    }  
  
    console.log(`El secreto es ${secreto}`);  
    // Console: ReferenceError: secreto is not defined  
}  
  
imprimirSecreto("supersecreta");
```

Hoisting de variables

Así como las funciones, las variables también tienen un comportamiento de hoisting, eso causa que algunas declaraciones se creen al inicio del bloque.

La forma más fácil de mirar esto es la siguiente. Si declaramos una variable con var, que tienen alcance completo de la función, esta es declarada e inicializada por efecto de hoist, lo que precede a las sentencias de console, por lo que veremos el siguiente resultado.

```
● ● ●  
console.log(x); // undefined  
var x = 1;  
console.log(x); // 1
```

Otra forma de mirar esto es. El hosting creará la declaración antes de la asignación con un valor default `undefined`, por lo que primero vemos en la consola `undefined` y después de que el valor es asignado, veremos el valor en pantalla.

```
● ● ●  
var x = undefined;  
console.log(x); // undefined  
x = 1;  
console.log(x); // 1
```

A diferencia de let y const, estas también tiene el efecto de hoist, pero no son inicializadas con un valor “default”, por lo que si intentamos hacer referencia a ellas estas tendremos un error.

```
console.log(x); // ReferenceError
let x = 1;
console.log(x);
```

En resumen

En javascript tenemos variables que, como las funciones, atienden a un scope, sea que fueron declaradas en un bloque, para el caso de let y const o que fueron declaradas en el alcance de una función, para el caso de var.

Así como advertimos del caso del local scope y global scope para evitar tener bugs raros en la aplicación, tenemos que tener esto en cuenta para evitar tener efectos raros al declarar variables con el uso de let, const y var.

Contexto

Es el valor contenido en la palabra clave `this`. En JavaScript podemos pensar en las aplicaciones como una serie de objetos muy grande y el contexto es el objeto sobre el que nuestras funciones se están ejecutando.

```
● ● ●

// Nosotros no vemos esto, pero esto es una super
// simplificación de Window.
let applicationRuntime = {
  run() {
    console.log("Dentro de applicationRuntime");
    // El contexto es applicationRuntime.
    // Por que estamos invocandolo desde applicationRuntime.run();
    console.log(`El contexto es: `, this);
    application.execute();
  }
};

// El browser hace invoca a Window.
applicationRuntime.run();
```

```
let application = {
  execute() {
    console.log("Dentro de application");
    // El contexto es application.
    // Por que estamos invocandolo desde application.execute();
    console.log(`El contexto es: `, this);

    function obtenerNombre() {
      console.log("Dentro de obtenerNombre");
      console.log(`El contexto global es: `, globalThis);
      // No hay contexto definido.
      // Por que se esta invocando como funcion directa imprimirSaludo().
      console.log(`El contexto es: `, this);
      return "Jerome";
    }

    function imprimirSaludo() {
      let nombre = obtenerNombre();
      console.log(`Hola ${nombre}`);
    }
    imprimirSaludo();
  }
};
```

En aplicaciones web casi siempre veremos que `this` hace referencia a Window, ya que este es el objeto que está invocando nuestras funciones dentro de JavaScript. Por otro lado, cuando se crea un nuevo contexto como podemos ver en la función obtenerNombre no tiene ningún contexto asociado, dado que se está invocado como una función y no desde un objeto.

Importante: este comportamiento está definido dentro del modo estricto de JavaScript, que es el que usamos usualmente al trabajar con React. Este modo ayuda a reducir los errores silenciosos en nuestras aplicaciones y estandariza los comportamientos.

De otra forma en el ejemplo anterior, imprimirSaludo tendría como referencia a Window. Dado que en el modo no estricto todas las funciones asignan el contexto de ejecución aunque no se esté llamando desde un objeto.

Siempre podemos acceder a un globalThis que siempre está asociado al objeto Global que ejecutar nuestros scripts, en el navegador es Window, en Node es global y uno mas en los Workers llamado WorkerGlobalScope.

Funciones flecha

A diferencia del ejemplo anterior, si todas las funciones estuvieran creadas usando funciones fecha, nunca tendríamos un contexto asociado. Por lo que no se recomienda usar funciones arrow nunca como métodos de los objetos.

Importante: Nuevamente en el ejemplo se está ejecutando en modo estricto, de otra forma this, haría referencia a Window.

```
let objA = {
    numbers: [1, 2, 3],
    print() {
        this.numbers.forEach(function () {
            // Usar function creo su nuevo contexto.
            // ya no sera posible acceder a `this`.
            console.log(this); // undefined
        });
    }
};
```

```
let objB = {
  numbers: [1, 2, 3],
  print() {
    this.numbers.forEach(() => {
      // Usar no creo su nuevo contexto.
      // adquirio el contexto de print al ser ejecutado como metodo.
      console.log(this); // {numbers: Array(3), print: f print()}
    });
  }
};
```

En resumen

El contexto es valor que hace referencia al objeto que está llamando nuestra función. El comportamiento de este cambia entre modo estricto y no estricto. Las funciones flecha ante las funciones clásicas tienen la limitante de no asociar un contexto de ejecución, por lo que no son buenas para usar como métodos, sin embargo heredan el contexto padre de la función que las está llamando.

Referencias:

- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Strict_mode

Referencia y Valor

En JavaScript tenemos dos grupos de tipos de datos, los tipos de datos que conocemos como boolean, string, etc. Estos grupos son Objetos y Primitivos, todos los objetos primitivos son de bajo nivel, inmutables y sus valores son pasados entre funciones como valores.

Por otro lado, los Objetos son grupos de valores referenciados y como su definición lo indica, sus valores se pasan entre funciones como referencias indirectas, lo cual las hace mutables.

Si regresamos al ejemplo del inicio, en lugar de acceder a las variables globales para construir nuestro mensaje, las pasamos como argumentos, podemos notar cómo se mantiene el valor de las variables globales a pesar de que reasignamos el valor de curso.

```
let nombre = "Jerome";
let curso = "Escenciales de JavaScript";

function impartirCurso(nombre, curso) {
  curso = "Cocina"
  console.log(`Se imparte curso ${curso} por ${nombre}`);
  // Console: Se imparte curso Cocina por Jerome
}

impartirCurso(nombre, curso);

console.log(curso);
// Console: Escenciales de JavaScript
```

Este comportamiento funciona porque curso es de tipo string, por lo tanto su valor se pasa solo por valor, ignorando que hay una referencia a la variable global curso.

```
let propiedades = {
  nombre: "Jerome",
  curso: "Escenciales de JavaScript",
}

function impartirCurso(propiedades) {
  propiedades.curso = "Cocina"
  console.log(`Se imparte curso ${propiedades.curso} por ${propiedades.nombre}`);
  // Console: Se imparte curso Cocina por Jerome
}

impartirCurso(propiedades);

console.log(propiedades.curso);
// Console: Cocina
```

Por otro lado, si cambiamos el tipo de dato a un Objeto y modificamos el valor de una de sus propiedades, esta propiedad se habrá visto modificada igual que si hubiéramos reasignado el valor global y tendríamos los mismo problemas que si estuviéramos modificamos variables globales sin cuidado.

En resumen

JavaScript tiene dos grupos de categorías de tipos. Los primitivos y los objetos, todos los primitivos pasan son pasados por valor entre nuestras funciones, por lo que es más seguro trabajar con ellos y no mutar accidentalmente algún valor durante la ejecución. Los tipos que se pasan por referencia son mutables y podemos alterar sus propiedades si modificamos sus valores directamente.

Como pudimos ver en este ejemplo, a diferencia de la primera versión en el segmento de funciones, usar los valores pasados en los argumentos de la función, nos ayudó a preservar el valor de las variables globales y prevenir algún bug raro en nuestra aplicación. Más adelante puliremos algunos detalles adicionales para mantener la “pureza” de nuestras funciones y construir aplicaciones menos propensas a bugs.

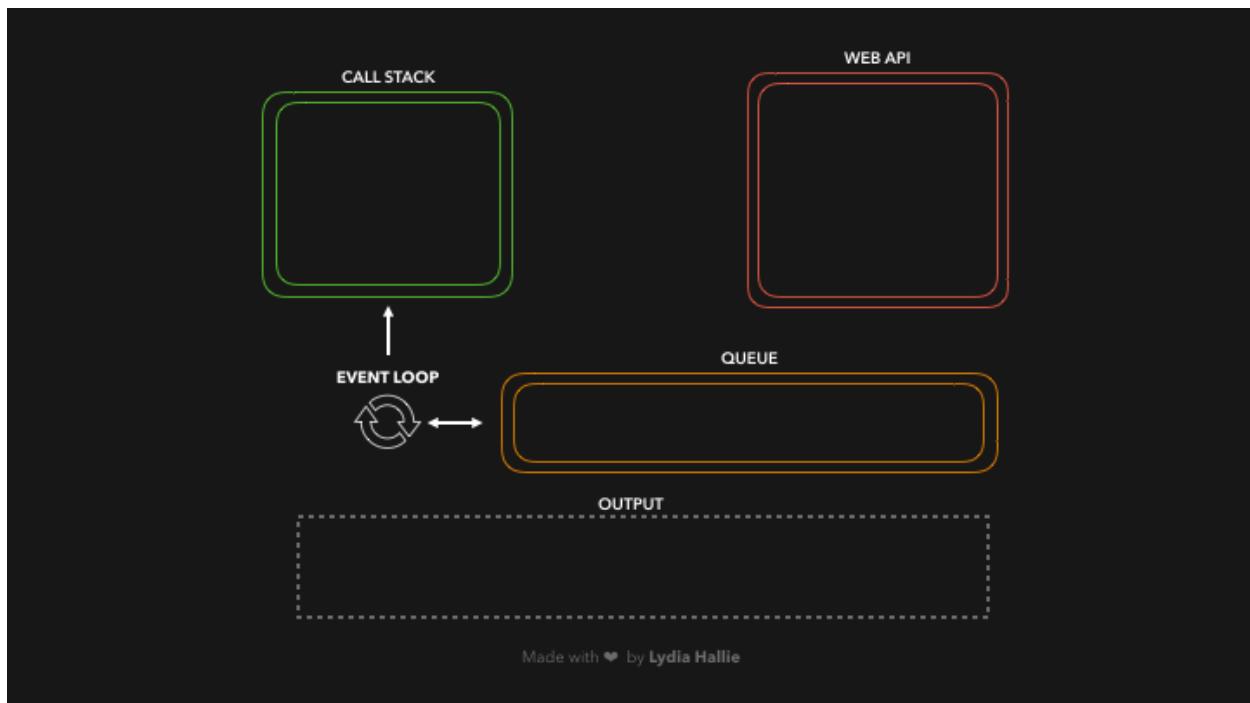
EventLoop

JavaScript es un lenguaje single thread, lo que quiere decir que solo puede hacer una sola tarea a la vez dado un tiempo determinado. Por lo tanto si nosotros ejecutamos tareas que tomen mucho tiempo resolverse nuestra aplicación se quedará detenida tanto tiempo dure esta tarea.

Para permitirle a JavaScript hacer tareas de larga duración, que hagan llamadas HTTP, setear timers, etc. Estas tareas son diferidas a un proceso externo que pueda ejecutarlas en background sin bloquear el runtime de JavaScript.

En el caso de Node como en el Browser, estas API corren en un proceso diferente del navegador o del kernel y solo notifican a JavaScript cuando el proceso haya terminado para continuar con el flujo de la aplicación.

Esta comunicación ocurre a través del EventLoop o el ciclo de eventos. Este es un modelo de concurrencia que permite a JavaScript, ejecutar sus propias tareas y aquellas que tienen el soporte para procesarlas en otro proceso diferirse hasta que se hayan resuelto.



Podemos ver en el ejemplo, como una tarea es ejecutada en el stack de funciones de JavaScript, el proceso de timeout es diferido y ejecutado por el otro proceso en el Browser y cuando este finaliza, nos notifica a través de una cola de eventos.

Visualización de <https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif>

Promesas

Para acceder a esta programación asíncrona, en JavaScript tenemos una la API de Promises, este es un objeto que nos permite de forma simple, ejecutar el código y organizar los callbacks de estos proceso asíncronos de forma simple.

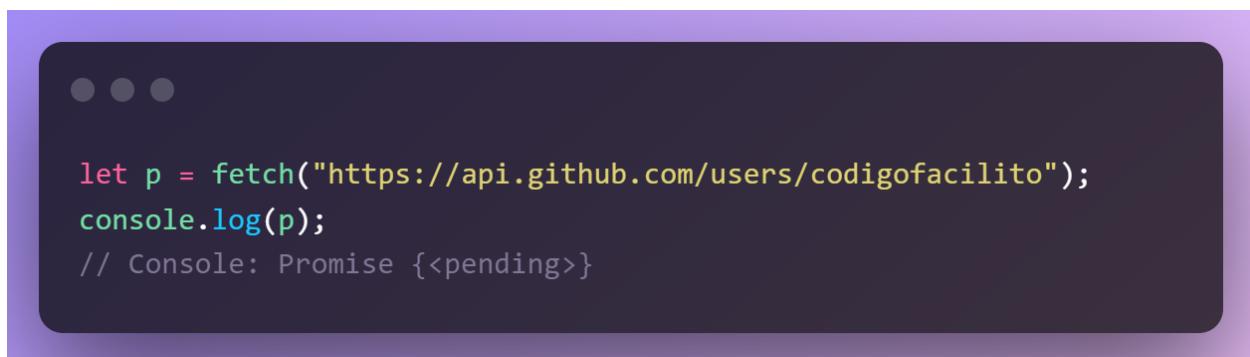
Una promesa puede encontrarse en alguno de los siguientes estados:

fulfilled: 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.



A screenshot of a browser's developer tools console. It shows a single line of code: `let p = fetch("https://api.github.com/users/codigofacilito"); console.log(p); // Console: Promise {<pending>}`. Above the code, there are three small circular dots indicating the code has been executed. The console output below shows the promise object with its state as pending.

Si miramos el ejemplo de `fetch`, esta es una de esas APIs que nos permite ejecutar tareas en otro lado, sin bloquear las tareas que tiene que ejecutar JavaScript. Si miramos el valor que regresa una llamada `fetch`, esta nos indica que está en estado pendiente y el valor aún no se ha resuelto.

La petición fue enviada pero como vemos en el diagrama anterior, tenemos que tener una forma de obtener el valor una vez que este se haya resuelto del Event Queue.

```
...  
  
fetch("https://api.github.com/users/codigofacilito")  
  .then(result => console.log(result))  
  .catch(error => console.error(error))  
  .finally(() => console.log('Cleanup...'));  
  
// Console: Response {type: "cors", url: "...."  
// Console: Cleanup...
```

Para poder obtener el valor de la respuesta una vez que esta se haya resuelto, tenemos que usar el método `then`, que se ejecutará hasta que la promesa se haya resuelto de forma **satisfactoria**.

Al introducir programación asíncrona, también introducimos una nueva categoría de errores que pueden ocurrir en nuestra aplicación, en el caso de `fetch`, nos exponemos a los problemas que las conexiones a internet enfrentan o la disponibilidad del servicio.

Para poder escuchar y capturar los errores de estas APIs, tenemos que usar el método `catch`, que como su nombre lo indica, similar a `try/catch`, podemos capturar las excepciones resultantes de estas operaciones.

`Promise` también expone un método `finally` que se ejecutará cuando la promesa se resuelva, no importa si es de forma satisfactoria o no. Te puede ayudar a hacer `cleanup` de algunos otros valores, reiniciar algún estado, etc.

En resumen

JavaScript es un lenguaje single thread que solo puede hacer una cosa a la vez. Para poder hacer trabajo más pesado, este es delegado a otros procesos a través de APIs. El `eventloop` se encarga de ordenar la ejecución de estas tareas diferidas y notificarnos cuando estén resueltas. Finalmente JavaScript nos ofrece una API de Promises para poder interactuar con estas API asíncronas de forma simple y ordenada

Chaining

La API de Promises es bastante poderosa, ya que a través de los métodos `then`, `catch` y `finally`, podemos crear procesos complejos a través de un modelo similar a la composición de funciones.

```
const proceso = () =>
  validarPeticion()
    .then(formatearEmail)
    .then(actualizarDatosDeUsuario)
    .then(() =>
      enviarCorreo()
        .catch(() => console.warn("Fallo correo")))
    .then(formatearResultado)
    .then(() => console.log("Fin de proceso"))
    .catch(() => console.error("Fallo"))
    .finally(() => console.log("Limpiar proceso"));
```

Este modelo de composición nos ayuda a tener aplicaciones mucho más resilientes, ya que nos aseguramos de que la siguiente operación solo se ejecutara hasta que la anterior se haya resuelto de forma satisfactoria.

Finalmente podemos capturar y controlar los errores que ocurran durante toda la operación, por ejemplo, si queremos que el fallo de una operación no interrumpa todo el proceso, ya que dicha operación no es sumamente vital, podemos capturar esos errores intermedios y permitir al proceso terminar.

Async y Await

En JavaScript también tenemos el soporte de `async/await`, la cual nos permite trabajar en con Promises como si fuera código asíncrono, esto es conveniente ya que permite combinar el estilo de programación secuencial usando promesas.

Algo que remarcar es que el API de Promesas es declarativa, mientras que `async/await` nos hará escribir código imperativo. Por lo que es importante evaluar cuándo es conveniente usar uno u otro.

```
const proceso = async () => {
    try {
        await validarPeticion();
        await formatearEmail();
        await actualizarDatosDeUsuario();

        try {
            await enviarCorreo();
        } catch (err) {
            console.warn("Fallo correo");
        }

        await formatearResultado();
        console.log("Fin de proceso");
    } catch (err) {
        console.error("Fallo");
    } finally {
        console.log("Limpiar proceso");
    }
};
```

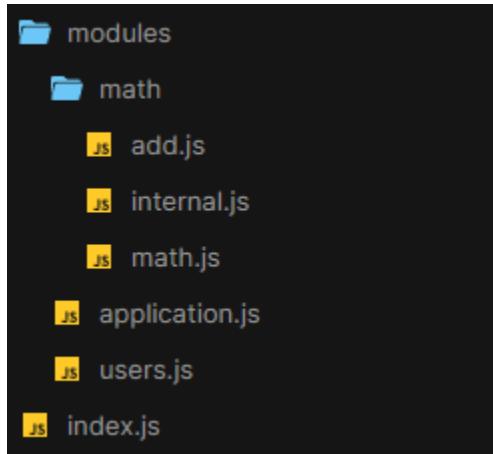
En resumen

El API de Promesas nos permite crear procesos complejos a partir de la composición de funciones. Podemos usar `async/await` para tener la flexibilidad del código tradicional mientras trabajamos con operaciones asíncronas.

ES Modules

Los módulos en JavaScript es la forma de dividir nuestra aplicación en piezas más pequeñas, fáciles de usar y fáciles de mantener.

Los módulos introducen su propio Scope, el llamado Module Scope, por lo que no debes preocuparte por si las variables que usas en tu módulo fueron utilizadas en otro. También nos permiten declarar explícitamente las funciones y objetos de forma explícita, por lo que reduce la ambigüedad de saber cuáles funciones y expresiones existen.



```
import users from "./users";
import math from "./math/math";

export default async function application() {
  console.log("Running application...");
  const user = await users.getUsers();
  console.log(`User ${JSON.stringify(user)}`);
  const newBalance = math.add(user.balance, 100);
  console.log(`User's new balance ${newBalance}`);
}
```

Nuestra aplicación puede dividir la complejidad de los procesos que tiene que ejecutar y solo hacer composición sobre los diferentes módulos que se ofrecen.

Además de la separación, los módulos también permiten la encapsulación de proceso internos y exclusivos de cada módulo.

```
• • •  
import { protectedStuff } from "./internal";  
  
export default function add(a, b) {  
  protectedStuff();  
  return a + b;  
}
```

```
• • •  
export function protectedStuff() {  
  console.log("Protected stuff...");  
}
```

Podemos usar los named export o imports, como los defaults. Como vemos, la palabra export declara lo que se está exportando y podrá ser usado desde otro módulo.

Cuando usamos la palabra clave default, estamos declarando que un módulo sólo exporta un solo elemento. Puedes decidir entre usar named o default dependiendo de lo que mejor convenga para mantener la encapsulación del mismo.

También al usar defaults, evita la colisión entre dos módulos que exportan un método con el mismo nombre, dándole una especie de Contexto o nombre de espacio.

Características “Modernas”

En esta última sección daremos un repaso a las características que puntualmente nos serán muy útiles mientras desarrollamos aplicaciones en React

Template literals

Nos permiten en lo simple, hacer interpolación de string, como hemos visto en los ejemplos anteriores, podemos combinar variables con string para poder formar mensajes complejos sin tanto problema.

```
let nombre = "Jerome";
let mensaje = `

Hola, los saluda ${nombre},
Gracias por estar en este curso.`;

console.log(mensaje);
```

Como podemos ver, también nos permite construir string multi-línea.

```
const nombre = "Jerome";
const edad = 26;

function myTag(strings, nombre, edad) {
  const mensaje = edad > 99 ? "no tan joven" : "joven";
  return `${nombre}${strings[1]}${mensaje}`;
}

myTag`${nombre} es ${edad}.`;
// Console: 'Jerome es joven'
```

Un uso más avanzado de los templates, son los tagged templates, con estos podemos interpretar y reconstruir mensajes, así como aplicar operaciones más complejas como si fueran argumentos de una función.

Esta funcionalidad de modificar la función de procesamiento para el string habilita un abanico de posibilidades, puedes darte una idea en [este repo](#), que agrupa tagged templates de utilidad.

Object destructuring

Destructura de objetos, es una propiedad que podemos usar para declarar variables a partir de los valores en los objetos o arreglos. Es como desempaquetar su valor y asignarlo en una variable.

```
const array = [1, 2, 3];
const [uno, dos, tres] = array;
```

Para el caso de los arreglos, la posición de los valores, está estrictamente relacionada al valor que va a adquirir la variable.

```
const obj = { nombre: "Jerome", edad: 26 }
const { nombre: primerNombre, edad } = obj;
```

A diferencia de los arreglos, podemos tomar las propiedades que ya están definidas en los objetos por sus nombres o reasignar las propiedades a otro a un nombre de variable diferente.

Rest Spread

Con el operador spread, podemos expandir cómo empaquetar valores.

Para hacer spread (expandir), tenemos que hacerlo solamente sobre objetos iterables, como string, objetos, arreglos, etc.

```
function suma(a, b) {  
    return a + b;  
}  
  
let numeros = [10, 20];  
suma(...numeros);
```

Esta sintaxis la podemos usar tanto para crear nuevos objetos o pasar valores a una función. Un dato curioso, si recuerdan el caso de los objetos que se pasan por valor como los que se pasan por referencia, el spread nos puede ayudar a pasar un objeto por sus valores sin pasar la referencia del objeto y preservar su valor.

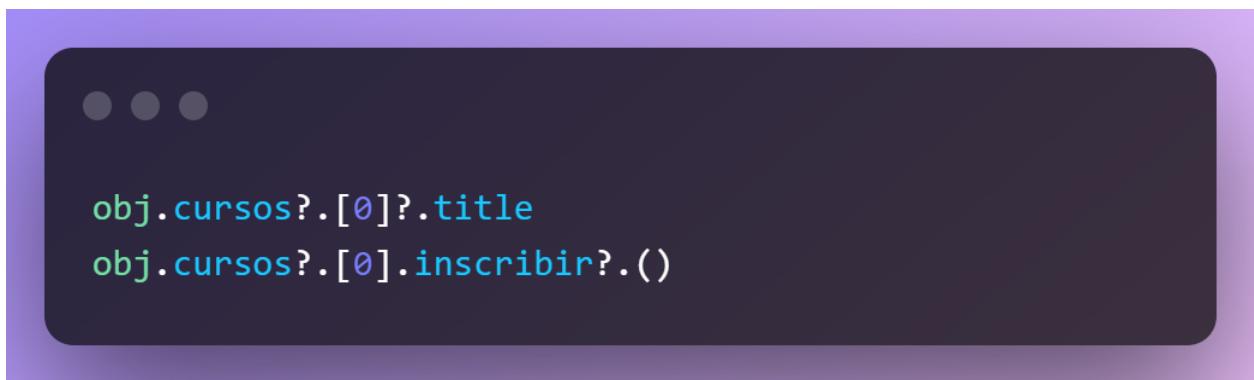
```
let numeros = [10, 20];  
let obj = { nombre: "Jerome", edad: 26 };  
  
let copiaDeNumeros = [...numeros, 30, 40];  
let copiaDeObjeto = {...obj, edad: 20};
```

En el caso de los arreglos, este también te permite agregar nuevos valores al arreglo sin mutar la referencia original, creando una nueva copia. Pero en el caso de los objetos, si las propiedades que están agregando, son las mismas que el objeto ya tiene, este modificará los valores en la copia sin agregar nuevas propiedades.

Optional chaining

Probablemente este es una de las features más modernas del lenguaje, esta nos permite acceder de forma condicional a la propiedad de un objeto sin el temor de que este venga null o indefinido.

JavaScript al ser un lenguaje dinámico siempre corremos el riesgo de este tipo de errores en runtime, por lo que este operador es bastante conveniente, pero también hay que usarlo con responsabilidad, ya que puede crear código no tan legible. Este operador se puede usar en objetos, arreglos y funciones.



Algunos casos donde ayuda este operador, son para validaciones de propiedades o asignar valores por defecto.

```
let user = { name: "Jerome" };
// user.address != null && user.address.street != null
if (user.address?.street != null)
  throw new Error("invalid address")

let userPreference = user?.newApiSetting ?? fallbackApiSetting;
```

Cuando una aplicación introduce una nueva funcionalidad a sus usuarios, hay veces en la que todos los usuarios no tienen compatibilidad con esta feature y los bugs pueden ser comunes en estas migraciones, por lo que usando también el operador nullish, podremos asignar valores por defecto reduciendo los casos de error.

En resumen

El optional chaining es conveniente en React para la validación de props o los valores default cuando usas custom hooks, ya que por la forma en la que funciona React, a veces hay que acceder a las propiedades de un estado antes de que estas estén definidas.

En Conclucion

Aquí hemos visto un repaso general por las funcionalidades que ofrece JavaScript al desarrollar. Una vez que entremos en materia de React tener estos conceptos frescos en mente nos ayudarán a conducir un mejor diseño y control sobre nuestras aplicaciones

Referencias

- Eloquent JavaScript - <https://eloquentjavascript.net/>
- You Don't Know JS - <https://github.com/getify/You-Dont-Know-JS>
- Iterable -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols
-