Guía de Iniciación sobre JavaScript

Bases sólidas para comenzar a programar en React

8 de junio de 2023

Índice

Si deseamos dominar React y eso significa comprender por qué y cómo funcionan ciertas cosas, debemos aprender algunos conceptos específicos de JavaScript.

En esta guía rápida explicaremos los conceptos de JavaScript y las construcciones sintácticas necesarias para crear un camino sólido que nos permitan dominar React. Si desea profundizar en más detalles sobre algunos de los temas que se explican aquí u otros sobre JavaScript, recomendamos que visiten el sitio web Mozilla [**?**]. Esta guía se basa en conceptos explicados allí.

1. Entorno de Desarrollo

Para comenzar a programar en JavaScript y React recomendamos Visual Studio Code (VS Code). Y para ser más productivo, especialmente si eres nuevo en React, sugerimos que instales la extensión VS Code ES7 React / Redux / React-Native / JS snippets que proporciona la generación rápida de estructuras sintácticas de JavaScript y React. También sugerímos instalar Prettier, que es un formateador de código JavaScript / React.

Para instalar una extensión, en Visual Studio Code, vaya al menú Archivo], luego Preferencias y luego Extensiones. Verá un cuadro de búsqueda que le permitirá encontrar las extensiones que desea instalar.

Finalmente, recomendamos configurar VS Code para formatear los fuentes al momento de guardar. Puede hacerlo yendo al menú Archivo, luego a Preferencias y luego a Configuración. En el cuadro de búsqueda, escriba Editor: Formato al guardar (Format On Save). Esto formateará su código inmediatamente después de guardarlo.

2. Introducción

De la web Developer Mozilla JavaScript Documentation [**?**], JavaScript se define como:

"JavaScript (JS) es un lenguaje de programación ligero, interpretado, o compilado justo-a-tiempo (just-in-time) con funciones de primera clase. Si bien es más conocido como un lenguaje de scripting (secuencias de comandos) para páginas web, y es usado en muchos entornos fuera del navegador, tal como Node.js, Apache CouchDB y Adobe Acrobat. JavaScript es un lenguaje de programación basado en prototipos, multiparadigma, de un solo hilo, dinámico, con soporte para programación orientada a objetos, imperativa

y declarativa (por ejemplo programación funcional)."

Si Usted es un desarrollador Java, C# o C++, esta definición puede sonar un poco intimidante. Es por ésto que es necesario aprender algunos conceptos, especialmente aquellos que no están disponibles de forma natural en los lenguajes compilados. Para comenzar con esos conceptos, primero explicaremos las construcciones básicas del lenguaje, luego explicaremos lo que significa que un lenguaje tenga **funciones de primera clase** y sea **basado en prototipos**, **multi-paradigma**, **de un solo subproceso** y **dinámico**.

El lenguaje JavaScript se rige por una especificación estándar de ECMA[**?**] (European Computer Manufacturers Association). ECMAScript es el nombre de la especificación. En particular, en 2015 se liberó una versión importante conocida como ES6, ECMAScript 6 o ECMAScript 2015. La mayoría de las construcciones sintácticas que estudiaremos en esta guía se implementaron en dicha versión.

Comencemos a aprender. Toda sentencia en JavaScript escrita en esta guía se ejecutará usando el intérprete de JavaScript de **nodejs**, invocándolo usando la consola de la siguiente manera:

```
$ node yourjsfile.js
```

Utilizando Visual Studio Code, pueden abrir una consola directamente allí, desde el menú en Terminal y luego Nueva Terminal.

Lo primero que en general aprendemos cuando nos iniciamos en un lenguaje de programación es a imprimir texto en la pantalla. Esta guía no es la excepción. Se puede imprimir texto en la pantalla usando el objeto *Console* de la siguiente forma:

```
1 console.log("Coding in React!");
```

El objeto *Console* se creó principalmente como herramienta de debug, no debe utilizarse en aplicaciones desplegadas en producción. No es parte parte del estándar pero todos los Browsers modernos y nodejs lo soportan.

3. Variables

Comencemos a ver como utilizamos **variables** en JavaScript. Declaramos las variables utilizando la palabra reservada *let*, de la siguiente forma:

```
let myFirstVariable;
```

Cuando declaramos una variable sin inicializar, el intérprete la inicializará con el valor especial *undefined*. Esto se puede observar si intentamos imprimir en la consola una variable no inicializada. Pruébenlo.

Ejercicio: Declare una variable sin inicializar e imprímala en la consola

El siguiente ejemplo muestra cómo declarar una variable y como inicializarla a la vez.

```
let myFirstVariable = "Hello value!";
```

También es posible declarar una variable utilizando la palabra reservada *const*:

```
const myFirstConst = "Hello constant value!";
```

Al declarar la variable con *const* no podremos modificar su valor, si lo intentan el intérprete lanzará un error. Pruebenlo.

Dijimos que JavaScript es un **lenguaje dinámico**. Ésto, entre otras cosas, significa que el tipo de una variable puede cambiar en tiempo de ejecución. A diferencia de los lenguajes estáticos donde los tipos de las variables se definen durante la compilación y no pueden cambiar durante la ejecución.

```
//my type is string
let changeMyType = "Hello String!";
//now it is number
changeMyType = 100;
```

4. Funciones

Estudiemos ahora las **functions**. A continuación declaramos una función y luego la invocamos:

```
function saySomething(string) {
  console.log(string);
}

saySomething("Hello Function!");
```

En JavaScript podemos definir funciones que se invocan inmediatamente después de declararlas. Se denominan IIFE (Immediately-invoked Function Expression).

```
  (function saySomething(string) {
    console.log(string);
  })("Hello Function!");
```

Las funciones siempre retornan un valor. Si se omite el uso de *return* para retornar algo explícito, entonces una función retornará el valor *undefined*.

```
let x = saySomething("Hello Function!");
//x is undefined
```

También hemos mencionado que las funciones son objetos de **primera clase**. Ésto significa que al igual que las variables, éstas pueden asignarse, pasarse por parámetro o retornarse de otra función. El ejemplo a continuación, en la línea 6, nos encontramos con una función que se asigna a la variable `say`. Y luego en la línea 10 utilizamos esta variable para invocar la función.

```
function returnSomething(string) {
  return "This is it: " + string;
}

//assigning a function to a variable
let say = returnSomething;

//calling the function
returnSomething("Hello js!");
say("Hello again!");
```

En el siguiente ejemplo, en la línea 12, estamos invocando una función pasando como parámetro a la función `returnSomething`. Y en la línea 8, utilizando el argumento recibido, realizamos la invocación a la función.

```
function returnSomething(string) {
  return "This is it: " + string;
}

//receives a function as parameter
//invokes it and return the value
function saySomethingMore(fn) {
  return fn("Hey !");
}

//passing a function as argument
saySomethingMore(returnSomething); //"This is it: Hey !"
```

También podemos asignar una función a una variable directamente en su declaración, tal como se muestra a continuación:

```
//assigning the function
const returnSomething = function (string) {
  return "This is it: " + string;
};

returnSomething("Hey !"); //"This is it: Hey !"
```

Otra forma de declarar funciones en JavaScript, de una forma algo menos verbosa, se denomina **arrow functions**. Veamos el siguiente ejemplo:

```
//arrow function with no parameters
const arrowf1 = () => {
  return "arrowf1 was invoked!";
};

//arrow function with one parameter
//parenthesis is not necesary here
const arrowf2 = param => {
  return "this is the argument: " + param;
};

//arrow functions with one statement
//in the body won't need return
const arrowf3 = (a, b) => a + b;
```

5. Arreglos

Vamos a utilizar **arreglos** de forma frecuente en React en general y a lo largo de esta guía. Así es como declaramos un arreglo:

```
//an empty array
let empty = [];

//an array
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
```

Los elementos de un arreglo se acceden a través de su índice, donde 0 es el primer elemento.

```
//an array
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
family[0]; //Jóse
family[1]; //Nicolas
family[2]; //Lucia
family[3]; //Enrique
```

Dado que los arreglos en JavaScript son objetos, tenemos varios métodos útiles que podremos utilizar, como el que se muestra a continuación para agregar elementos al final:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

//adding an element at the end of an array
family.push("Pablo");
```

Si quisiéramos incorporar todos los elementos de un arreglo en otro arreglo, tenemos la opción de utilizar la construcción sintáctica denominada *spread syntax*. Veamos cómo se utiliza:

```
let myParents = ["EnriqueR", "Susana"];
let JoseParents = ["Eduardo", "Graciela"];
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
let all = [...myParents, ...JoseParents, ...family];
//[
//  'EnriqueR', 'Susana', 'Eduardo', 'Graciela',
//  'Jóse', 'Nicolas', 'Lucia', 'Enrique'
// ]
```

También es posible utilizar *spread syntax* en los parámetros de las funciones para indicar un número indefinido de argumentos:

```
function restParams(param1, param2, ...params) {
 //params is [3, 4, 5]
}
restParams(1, 2, 3, 4, 5);
```

La forma clásica de iterar sobre un arreglo es utilizando la construcción sintáctica `for`, como se muestra a continuación:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];
for (let element of family) {
  console.log("regular for: ", element);
}
```

Sin embargo, los arreglos proveen de un conjunto de métodos muy convenientes y que utilizaremos con frecuencia. El más simple es `.forEach`:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

family.forEach(function (value, index, array) {
  //value is the element being processed
  //index is the index of the current value
  //array is the entire array
  console.log(value, index, array);
});
```

`.forEach` acepta tres argumentos. `value` que es el elemento a procesar, `index` que es el índice que corresponde al valor que estamos procesando y `array` que es el arreglo que estamos iterando. Si solo nos interesa trabajar con `value`, podemos simplemente escribirlo así:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

family.forEach((value) => {
  //do something with the value here
});
```

Un método más interesante aún es `.filter`, el cual se utiliza para procesar el arreglo y devolver otro con igual o menos elementos. Recibe una función con los mismos parámetros que `.forEach` y devuelve un arreglo para aquellos elementos donde la función evalúa a true. Veamos cómo se utiliza a continuación:

```
let family = ["Jóse", "Nicolas", "Lucia", "Enrique"];

const members = family.filter((member) => {
  return member.length > 5;
});

//members = ['Nicolas', 'Enrique']
```

`.filter` recibe una función que evalúa por cada elemento del arreglo si es mayour a 5. Aquellos elementos que sean mayores a 5 serán parte del nuevo arreglo que se retorna. El arreglo donde se aplica el filtro no es modificado.

Otro de los métodos interesantes, y principalmente útil para React como veremos más adelante, es `.map`. Éste método, al igual que el anterior, recibe una función como parámetro y retorna otro arreglo con el resultado de aplicar la función recibida sobre cada elemento. Veamos el siguiente ejemplo:

```
let numbers = [1, 2, 3, 4, 5, 6, 7];
const doubles = numbers.map((element) => {
  return element * 2;
});
//doubles = [2,  4,  6, 8, 10, 12, 14]
```

Éstos métodos se pueden combinar. Veamos el siguiente ejemplo. Primero aplicamos `.filter` sobre un arreglo de enteros, para quedarnos solo con aquellos que son impares y luego multiplicamos por dos cada uno de los elementos para transformarlos en pares.

```
let numbers = [1, 2, 3, 4, 5, 6, 7];
const chain = numbers
  .filter((element) => {
    return element % 2 !== 0;
  }) //[1, 3, 5, 7]
  .map((element) => {
    return element * 2;
  });
//chain = [2, 6, 10, 14]
```

Existe una forma de asignar cada uno de los elementos de un arreglo a variables que se denomina **destructuring**. Veamos a continuación el siguiente ejemplo:

```
let [one, two, three] = [1, 2, 3];
//one = 1
//two = 2
//three = 3

//same as the previous
let fewNumbers = [1, 2, 3];
[one, two, three] = fewNumbers;

//and here using spread syntax
let [a, b, ...rest] = [1, 2, 3, 4, 5];
//a = 1
//b = 2
//rest = [3, 4, 5]
```

6. Objetos

Existen varias formas de crear objetos en JavaScript. Aquí estudiaremos aquellas formas que utilizaremos con frecuencia programando en React. Comencemos estudiando cómo crear objetos con notación literal: **Object Literal**. Un objeto literal se crea encerrando entre llaves una colección de pares *propiedad*:*valor*, separadas por coma. Veamos el siguiente ejemplo:

```
//an object literal
let mi = {
  name: "Enrique",
  surname: "Molinari",
  sports: ["football", "tennis"],
  address: {
    street: "San Martin",
    number: 125,
  },
  allSports: function () {
    console.log(this.sports);
  },
};
//this is an empty object
let obj = {};
```

Como se puede ver del ejemplo anterior, en un objeto literal, los valores pueden ser arreglos, funciones o incluso otros objetos. Por otro lado, desde

ES6, es posible crear objetos literales utilizando variables como nombre de propiedad. Veamos a continuación como se utiliza en la línea 6:

```
1  let aproperty = "phone";
2  //an object literal with a computed property name
3  let mi = {
4    name: "Enrique",
5    surname: "Molinari",
6    [aproperty]: "+54 2920 259031"
7  };
```

Cada vez que el intérprete de JavaScript evalúa un objeto literal se crea la instancia. Las propiedades de los objetos se acceden utilizando la **notación de punto**. Veamos un ejemplo:

```
1  console.log(mi.name); //Enrique
2  console.log(mi.sports[0]);//football
3  console.log(mi.address.street);//San Martin
4  console.log(mi.phone);//+54 2920 259031
5  mi.allSports(); //invoke the function and prints the sports array
```

En JavaScript es posible agregar propiedades nuevas en tiempo de ejecución a los objetos. Éste concepto es sumamente importante para entender algunas cuestiones más adelante. Observen el siguiente ejemplo. En las líneas 3 y 4, se agregan las propiedades `x` e `y` (inicializadas) al objeto `obj`.

```
1  let obj = {a: 1, b: 2};
2  //add properties to the obj object
3  obj.x = 3;
4  obj.y = 4;
```

Podemos utilizar la expresión denominada spread syntax con objetos también:

```
1  let obj1 = {
2    a: 1,
3    b: 2,
4  };
5  let obj2 = {
6    c: 3,
7    d: 4,
8  };
9  let obj3 = { ...obj1, ...obj2 };
10 //obj3 = { a: 1, b: 2, c: 3, d: 4 }
```

Podemos crear objetos a partir de variables inicializadas, de la siguiente forma:

```
let a = 1,
  b = 2;
let obj4 = {
  a,
  b,
};
//obj4 = { a: 1, b: 2 }
```

Hasta el momento hemos visto cómo crear objetos utilizando la notación literal. Pero para aquellos que venimos estudiando lenguajes estáticos y compilados como Java, nos estaremos preguntando qué pasa si necesitamos una cantidad de instancias desconocidas, y con igual estructura, o sea, creadas a partir de una clase. Para esto, en JavaScript tenemos lo que denominamos **constructor functions**. También tenemos clases como veremos más adelante, pero las clases vienen mucho después, es importante primero entender cómo funciona una función constructora.

Por convención el nombre de una función constructora comienza con mayúscula. Veamos algunos ejemplos a continuación:

```
function Book(name, authors, publishedYear) {
  this.name = name;
  this.authors = authors;
  this.publishedYear = publishedYear;
  this.fullName = function () {
    return this.name + " by " + this.authors + ". " + publishedYear;
  };
}

thisBook = new Book("Coding in React",
                          ["Enrique Molinari"], 2021);
thisBook.fullName(); //Coding in React by Enrique Molinari. 2021

archBook = new Book("Coding an Architecture Style",
                          ["Enrique Molinari"], 2020);
archBook.fullName(); //Coding an Architecture Style by Enrique Molinari. 2020
```

Como podemos observar del ejemplo anterior, la función `Book` se parece mucho a un constructor de los que solemos tener cuando escribimos clases

en lenguajes como Java. En las líneas 2, 3 y 4 definimos propiedades y las inicializamos con los parámetros recibidos. En la línea 5 definimos el método `fullName()`. Luego, en las líneas 10 y 14 creamos dos instancias de `Book` e invocamos al métodos `fullName()`.

6.1. Un Lenguaje basado en Prototipos

Ahora que sabemos como crear objetos con notación literal, como crear instancias a partir de funciones constructoras, podemos continuar explicando qué significa que JavaScript sea un lenguaje **basado en prototipos**. Los lenguajes basados en prototipos son un estilo de lenguajes orientados a objetos en los cuales los objetos son creados sin crear previamente una *clase*. Por este motivo, éstos lenguajes también son conocidos como lenguajes *classless* (sin clases). Por otro lado, lenguajes como Smalltalk, Java, C++ y C#, para nombrar algunos, son conocidos como lenguajes *class-based* (basados en clases).

En los lenguajes basados en prototipos, **no hay clases**, solo objetos. No existe esa diferencia entre clase y objeto. En lenguajes basados en clases, definimos una estructura estática con la posibilidad de definir entre ellas una relación de herencia, la cual no es posible modificar en tiempo de ejecución. En lenguajes basados en prototipos sólo tenemos instancias. No tenes clases, se hace evidente a medida vamos entendiendo cómo funciona JavaScript. Veamos algunos ejemplos.

Tal cual implementamos la función constructora `Book` en el ejemplo anterior, no es la forma ideal. Observemos por qué con el siguiente ejemplo:

```
1  thisBook = new Book("Coding in React",
2                          ["Enrique Molinari"], 2021);
3  archBook = new Book("Coding an Architecture Style",
4                          ["Enrique Molinari"], 2020);
5  //printing thisBook
6  //Book {
7  //  name: 'Coding in React',
8  //  authors: [ 'Enrique Molinari' ],
9  //  publishedYear: 2021,
10 //  fullName: [Function (anonymous)]
11 //}
12 //printing archBook
13 //Book {
```

```
14  //  name: 'Coding an Architecture Style',
15  //  authors: [ 'Enrique Molinari' ],
16  //  publishedYear: 2020,
17  //  fullName: [Function (anonymous)]
18  //}
```

En el ejemplo anterior, creamos dos instancias: `thisBook` y `archBook`, y luego las imprimimos. Como podemos observar, cada una de las instancias no solo cuenta con sus propiedades y valores, sino que también tiene la propiedad `fullName`, con su código. Los fuentes de ese método no es almacenado en una clase y compartido por las instancias de dicha clase como en los lenguajes basados en clases. Si no que esta incluido en cada instancia. Éste es un detalle de implementación que tenemos que es importante entender, pero además devela una de las diferencias que podemos encontrar en lenguajes basados en prototipos.

¿Pero qué sucede con la *herencia*? Al no tener clases, ¿tenemos herencia? La respuesta es Sí. La diferencia es que la relación de herencia en lenguajes basados en prototipos es dinámica, es decir que se puede establecer y cambiar en ejecución. Para entender cómo funciona la herencia en lenguajes sin clases, vamos a presentar el concepto de **prototype**.

Cada objeto en JavaScript (y en general en cualquier lenguaje basado en prototipos) puede estar asociado a lo que se denomina un **objeto prototipo**, del cual hereda todas sus propiedades y métodos. Si sobre una instancia queremos acceder a una propiedad y ésta no está definida allí, se **delega** su búsqueda en el objeto prototipo asociado. Como los prototipos pueden tener también un prototipo, se delegará en ellos hasta que se encuentre la propiedad o termina la ejecución con error al no encontrarla. Ésto se conoce como **prototype chain**.

Las funciones constructoras tienen acceso a una propiedad especial denominada `prototype` y la podemos acceder de la siguiente forma: `Book.prototype`. Con una instancia de un objeto es posible acceder a esta propiedad de la siguiente forma: `thisBook.__proto__` o lo que es lo mismo: `Object.getPrototypeOf(thisBook)`. Sabiendo esto, entonces si queremos mejorar la función constructora `Book` de modo de que cada instancia no cuente con los métodos que se definen, la creamos de la siguiente forma:

```
1  function Book(name, authors, publishedYear) {
2    this.name = name;
```

```
3     this.authors = authors;
4     this.publishedYear = publishedYear;
5   }
6   Book.prototype.fullName = function () {
7     return this.name + " by " + this.authors + ". " + this.publishedYear;
8   };
9
10  thisBook = new Book("Coding in React",
11                          ["Enrique Molinari"], 2021);
12  archBook = new Book("Coding an Architecture Style",
13                          ["Enrique Molinari"], 2020);
14  //printing thisBook
15  //Book {
16  //  name: 'Coding in React',
17  //  authors: [ 'Enrique Molinari' ],
18  //  publishedYear: 2021
19  //}
20  //printing archBook
21  //Book {
22  //  name: 'Coding an Architecture Style',
23  //  authors: [ 'Enrique Molinari' ],
24  //  publishedYear: 2020
25  //}
```

En el ejemplo anterior, definimos el método `fullName()` en el prototipo, en la línea 6. Luego creamos dos instancias y las imprimimos. Ahora podemos ver que el método `fullName()` no está en éstas instancias. Sin embargo, si lo invocamos así:

```
1   thisBook.fullName();
```

JavaScript buscará primero el método en la instancia `thisBook`. Al no encontrarlo, irá por su prototipo y lo ejecutará.

Toda cadena de prototipos finaliza en `Object.prototype`, dado que todo objeto en JavaScript es descendiente de Object. Con lo cual, si ejecutamos la siguiente invocación:

```
1   thisBook.valueOf();
```

JavaScript recorrerá la cadena de prototipos hasta encontrar el método `valueOf()` en `Object.prototype`. Creemos ahora un ejemplo utilizando

herencia. Crearemos el objeto `EBook` que hereda de `Book`, como se muestra a continuación.

```
1  function EBook(filesize, name, authors, publishedYear) {
2    Book.call(this, name, authors, publishedYear);
3    this.filesize = filesize;
4  }
5  let eBook = new EBook(2048, "Coding in React", ["Enrique
   ↪  Molinari"], 2021);
6
7  //printing eBook:
8  //eBook:  EBook {
9  //  name: 'Coding in React',
10 //  authors: [ 'Enrique Molinari' ],
11 //  publishedYear: 2021,
12 //  filesize: 2048
13 //}
```

En el ejemplo anterior creamos la función constructora `EBook`. En la línea 2, invocamos a la función constructora `Book` y para ello utilizamos el método `call` (Function.prototype.call) que permite invocar una función pasándole como parámetro el valor que utilizará para `this`. En éste caso observen que le pasamos aquella instancia de `EBook` que se sea creada vía el operador `new`, como en la línea 5. Éste que estamos haciendo es similar a utilizar `super(...)` dentro de un constructor de una clase, para instanciar e inicializar la superclase.

En la línea 5, como mencionamos, creamos una instancia de `EBook` e imprimimos la instancia. Vemos como resultado de la impresión que contamos con las propiedades de `Book` también. Sin embargo, no tenemos el método `fullName()` definido en `Book.prototype`. Para heredarlo necesitamos setear como prototipo a `Book.prototype`. Lo hacemos de la siguiente manera:

```
1  Object.setPrototypeOf(eBook, Book.prototype);
2
3  //Another way of doing the same as above is this:
4  //thisEBook.__proto__ = Book.prototype;
5  //However __proto__ is deprecated
```

`Object.setPrototypeOf` recibe dos parámetros, el primero es la instancia a la cual le asigno un prototipo y el segundo parámetro es el prototipo a asignar.

7. Clases

Sí! JavaScript tiene clases y su sintaxis es muy similar a la que conocemos de lenguajes basados en clases como Java. Las clases en JavaScript fueron incorporadas al lenguaje en 2015 como parte de EcmaScript 6 (ES6). Lo que es importante entender, es que ésto no transforma a JavaScript en un lenguaje basado en clases. Las clases son solo una **mejora sintáctica** (syntactic sugar) para poder utilizar herencia sin lídear con lo que mostramos en la sección anterior. Por detrás siempre son funciones constructoras y herencia basada en prototipos.

Implementemos entonces la jerarquía `Book` e `EBook` de la sección anterior pero ahora utilizando clases:

```
1  class Book {
2    constructor(name, authors, publishedYear) {
3      this.name = name;
4      this.authors = authors;
5      this.publishedYear = publishedYear;
6    }
7
8    //this method gets added to the Book.prototype
9    fullName() {
10     return this.name + " by " + this.authors + ". " +
       ↪  this.publishedYear;
11   }
12 }
13
14 class EBook extends Book {
15   constructor(filesize, name, authors, publishedYear) {
16     super(name, authors, publishedYear);
17     this.filesize = filesize;
18   }
19 }
```

En el ejemplo anterior construimos la relación de herencia entre `EBook` y `Book`, utilizando clases. Pero en ejecución termina quedando exactamente lo mismo que utilizando funciones constructoras como lo hicimos en la sección anterior. Veamos el siguiente ejemplo que lo demuestra:

```
1  let ebook = new EBook(2048, "Coding in React", ["Enrique,
   ↪  Molinari"], 2021);
2  //Book.prototype is the prototype of the ebook instance
```

```
3  console.log(Book.prototype.isPrototypeOf(ebook)); //true
4  //fullName method is found on the prototype
5  console.log(ebook.fullName()); //Coding in React by Enrique,
   ↪  Molinari. 2021
6  //EBook is a function not a class
7  console.log(typeof EBook); //function
```

Primero hemos creado una instancia de `EBook` y luego en la línea 3 verificamos que `Book.prototype` está seteado como el prototipo de la instancia `ebook`. De esta forma comprobamos que tenemos en ejecución la relación de herencia basada en prototipos como construimos en la sección anterior.

8. ¿Cómo se comporta this en JavaScript?

En las secciones anteriores hemos estado utilizando `this` en aquellas situaciones en las que tu comportamiento es el mismo que en lenguajes basados en clases como Java o C#. Sin embargo, hay ciertas diferencias para algunas construcciones sintácticas que necesitamos entender para poder luego entender el porqué de algunas cuestiones que necesitamos hacer en React.

Si utilizamos `this` en funciones constructoras y creamos las instancias utilizando `new`, `this` apuntará a la instancia creada. Sin embargo, si la función constructora es invocada, `this` tendrá otro valor. Veamos el siguiente ejemplo:

```
1  function Constr(param) {
2    this.param = param;
3  }
4
5  Constr(2); //this is global object window
6  console.log(window.param); //prints 2
```

Invocando a la función `Const` tal como lo hacemos en la línea 5 tiene un efecto bastante extraño. En estos casos, `this` apunta al objeto global `window`. Con lo cual, lo que termina haciendo éste ejemplo es agregando la propiedad `param` al objeto `window` con el valor 2.

El ejemplo anterior no es algo que vayamos a utilizar con frecuencia en React, sin embargo son cuestiones a tener presentes. Otra forma donde el `this` pierde valor es cuando asignamos un método a una variable, y ésto sí lo utilizaremos frecuentemente en React. Veamos el siguiente ejemplo:

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4    }
5    saySomething() {
6      console.log(this.name + " is talking...");
7    }
8  }
9  let enrique = new Person("Enrique");
10 enrique.saySomething(); //Enrique is talking...
11
12 let o = enrique.saySomething; //assigning to a variable
13 o(); //TypeError: Cannot read property 'name' of undefined
```

En el ejemplo anterior creamos en la línea 9 una instancia de la clase `Person` para luego en la línea 10 invocamos al método `saySomething()`. Hasta acá todo perfecto, el `this` en la línea 6 funciona funciona tal como esperamos. Sin embargo, en la línea 12 asignamos el método a una variable y en la siguiente línea utilizamos dicha variable para invocar al método. En esta ocasión, `this` es `undefined`, provocando un error. Para arreglar este comportamiento, necesitamos, en forma explicita, decirle al método que valor de `this` queremos que utilice, algo similar a lo que hicimos con el método `call` en secciones anteriores. Veamos cómo lo resolvemos con el siguiente ejemplo:

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4      this.saySomething = this.saySomething.bind(this);
5    }
6    saySomething() {
7      console.log(this.name + " is talking...");
8    }
9  }
10 let enrique = new Person("Enrique");
11 enrique.saySomething(); //Enrique is talking...
12
13 let o = enrique.saySomething; //assigning to a variable
14 o(); //Enrique is talking...
```

El método bind que utilizamos en la línea 4, devuelve la misma función pero con el valor de `this` seteado con el parámetro recibido. En este caso,

le pasamos `this`, el cual apuntará a la instancia creada. De ésta forma al invocar nuevamente utilizando `o()`, funcionará de la forma esperada.

Otra forma de solucionar este mismo inconveniente es declarando el método dentro de la clase utilizando funciones flecha (arrow functions). Veamos el siguiente ejemplo:

```
1  class Person {
2    constructor(name) {
3      this.name = name;
4    }
5    saySomething = () =>  {
6      console.log(this.name + " is talking...");
7    }
8  }
9  let enrique = new Person("Enrique");
10 enrique.saySomething(); //Enrique is talking...
11
12 let o = enrique.saySomething; //assigning to a variable
13 o(); //Enrique is talking...
```

Definiendo el método `saySomething` de esta forma no tendremos el inconveniente que mostramos antes. Las funciones flecha le asignan al `this` el valor que tenga en el ámbito léxico en el que se crean. El cual en este caso es la clase. Sin embargo, lo métodos definidos de esta forma no son agregados al prototipo del objeto sino que son parte del objeto, provocando lo que ya hemos mencionado que cada nueva instancia tendrá una copia del código del método.

9. Módulos

En ECMAScript 2015 (ES6) incorporaron al lenguaje la posibilidad de definir módulos. En las versiones anteriores de JavaScript si queríamos definir módulos teníamos que utilizar alguna herramienta externa como [**?**]. Ahora está soportado de forma nativa.

Realmente es muy simple de utilizar. En un archivo JavaScript podemos definir funciones, clases, objetos, constantes, variables, etc y exportar aquellos que queremos sea utilizado fuera del módulo. Para exportar utilizando la palabra reservada `export`. Los clientes del módulo deben importar explícitamente

aquello que quiere utilizar. Importan utilizando la palabra reservada `import`. Veamos algunos ejemplos:

```
1  //this is my complex-module.js module
2
3  export function complexThing() {
4    console.log("a complex thing has been executed...");
5  }
6
7  export let obj = {
8    a: 1,
9    b: 2,
10 };
11
12 export class ASimpleClass {
13   constructor(name) {
14     this.name = name;
15   }
16
17   print() {
18     console.log("printing: ", this.name);
19   }
20 }
```

En el módulo *complex-module.js* del ejemplo anterior exportamos una función, un objeto y una clase. También es posible realizar lo mismo de la siguiente forma:

```
1  //this is my complex-module.js module
2
3  function complexThing() {
4    console.log("a complex thing has been executed...");
5  }
6
7  let obj = {
8    a: 1,
9    b: 2,
10 };
11
12 class ASimpleClass {
13   constructor(name) {
```

```
14      this.name = name;
15    }
16
17    print() {
18      console.log("printing: ", this.name);
19    }
20  }
21
22  export { obj, ASimpleClass, complexThing };
```

Por supuesto, en un módulo solo debemos exportar aquellos que queremos exponer hacia afuera del módulo. Veamos ahora cómo podemos utilizar y consumir aquellos que exportamos.

```
1   //this is my main-module.js module
2
3   import { complexThing, obj, ASimpleClass } from
    ↪  "./module/complex-module.mjs";
4
5   //calling the imported function
6   complexThing();
7
8   //printing the imported object
9   console.log(obj);
10
11  //instantiating the imported class
12  let o = new ASimpleClass("Enrique");
13  o.print();
```

En la línea 3 importamos las tres abstracciones que *complex-module.js* exporta. Y simplemente las utilizamos como si estuvieran definidas allí mismo. Es posible definir una abstracción a exportar por defecto. Observen en el siguiente ejemplo cómo exportamos las mismas abstracciones pero la clase la exportamos utilizando las palabras reservadas *export default*:

```
1   //this is my complex-module.js module
2
3   function complexThing() {
4     console.log("a complex thing has been executed...");
5   }
6
```

```
7  let obj = {
8    a: 1,
9    b: 2,
10 };
11
12 class ASimpleClass {
13   constructor(name) {
14     this.name = name;
15   }
16
17   print() {
18     console.log("printing: ", this.name);
19   }
20 }
21
22 export default ASimpleClass;
23 export { obj, complexThing };
```

Esto nos permite, como vemos en el siguiente ejemplo en la línea 3, importar la clase pero con un nombre diferente al que se definió en el módulo donde reside.

```
1  //this is my main-module.js module
2
3  import AClass from "./module/complex-module.mjs";
4  //This line below is the same as the one above
5  //import { default as AClass } from
   ↪  "./module/complex-module.mjs";
6  import { obj, complexThing } from
   ↪  "./module/complex-module.mjs";
7
8  let o = new AClass("Enrique");
9  o.print();
10
11 //... more code here
```

10. Llamadas Ajax usando fetch

A partir de la versión ES6 podemos realizar peticiones Ajax utilizando el método fetch. La firma de éste método es la siguiente:

- `Promise<Response> fetch(url [, init])`

Promise es un objeto que representa valor que puede estar disponible ahora, en el futuro o nunca. Es el resultado de una operación asíncrona. Para quienes utilizan o utilizaron las APIs de concurrencia de Java, es similar al objeto Future. Luego volveremos sobre Promise. Como parámetros el método fetch recibe la URL a la cual invocará y de forma opcional un objeto para configurar el tipo de petición que se realizará. Por ejemplo, la forma más simple de utilizar fetch es la siguiente:

```
1  fetch("https://jsonplaceholder.typicode.com/posts/1")
2    .then((response) => response.json())
3    .then((json) => console.log(json))
4    .catch((error) => console.log(error));
```

Observen que solo utilizamos el primer parámetro que es el único obligatorio. Ésta es una petición HTTP de tipo GET que retorna de un servicio de blog ficticio el post con identificador 1. Las peticiones HTTP de tipo GET solo deben utilizarse para recuperar datos, **nunca** para realizar alguna inserción o modificación. Son peticiones **idempotentes**, es decir que el efecto que provoca en el servicio que consume es el mismo ya sea enviando una única petición o múltiples.

La función fetch retorna un objeto Promise, el cual expone dos métodos que utilizaremos mucho: `then(funcion(response))` y `catch(funcion(error))`. `then(funcion(response))` se llama si hubo una respuesta exitosa por parte del servicio que estamos consumiendo y como parámetros tenemos la respuesta. Entonces con la respuesta en la línea 2 la transformamos a un JSON y luego en la línea 3 lo imprimimos en la consola. Finalmente el método `catch(funcion(error))` es invocado si hubiera algún error con la respuesta obtenida o en su defecto si no es posible obtener respuesta alguna.

Veamos ahora cómo podemos utilizar el parámetro opcional del método fetch para realizar una petición de tipo POST. El método POST no es idempotente, llamarlo varias veces tendrá efectos adicionales sobre el servicio invocado. Veamos un ejemplo:

```
1  fetch("https://jsonplaceholder.typicode.com/posts/", {
2   method: "POST",
3   headers: {
4    "Content-type": "application/json; charset=UTF-8",
5   },
6   body: JSON.stringify({
```

```
7       name: "Enrique",
8       userName: "emolinari",
9       email: "emolinari@unrn.edu.ar",
10     }),
11   })
12     .then((response) => response.json())
13     .then((json) => console.log(json));
```

Observemos que como segundo parámetro del método fetch estamos pasando un objeto literal con las siguiente propiedades:

- `method`: Aquí especifico que es un request de tipo POST.
- `headers`: Aquí le aviso al servicio al cual estoy invocando que los datos enviados como valor de la propiedad `body` tienen una estructura JSON.
- `body`: Aquí van los datos a enviar. Observen que se utiliza el método `JSON.stringify` transformando un objeto literal en un JSON.

11. Lenguaje Single Thread

Dijimos que JavaScript es un lenguaje Single Threaded. Es decir, siempre existe un único hilo de ejecución (con una única pila de ejecución y un heap) que interpreta y ejecuta una sentencia a la vez. La siguiente sentencia no comienza a ejecutarse hasta que la anterior no termina. Esto podría ser muy perjudicial sobre todo con operaciones que lleven algo de tiempo. Por ejemplo, en una petición Ajax mientras esperamos tener una respuesta de algún servicio remoto no podríamos interactuar con alguna otra parte del sitio web. Ésto es así cuando utilizamos el método `window.alert`. Pruebenlo!. Hasta no apretar el botón Aceptar no es posible hacer nada más.

Por éste motivo es que existen en JavaScript operaciones *asincrónicas* como las llamadas Ajax que vimos en la sección anterior. Entonces, ¿cómo es que funciona?

El intérprete de JavaScript recibe ayuda del Browser para ésto. Existen ciertas operaciones que el intérprete de JavaScript reconoce y en lugar de ejecutarlas el mismo, las *delega* en el Browser, quien las ejecuta en su hilo para lugar devolverlas al intérprete como *callbacks*. Entre éstas operaciones encontramos los eventos (onclick, onmouseover, etc), la función Fetch o XMLHttpRequest (ajax calls), setTimeout, etc. Para lograr ésto el ambiente de ejecución de

JavaScript utiliza una *pila de llamadas*, el Browser, una *cola de callbacks* y el *loop de eventos*. Observemos el siguiente ejemplo para entender cómo es su ejecución dentro de éste ambiente:

```
1  console.log("starting");
2
3  setTimeout(() => {
4    console.log("callback");
5  }, 1000);
6
7  console.log("finishing");
```

Veamos entonces como JavaScript resuelve la ejecución de éste programa:

1. La setencia de la línea 1 se inserta en la *pila de llamadas* y es ejecutada. Se imprime en la consola `"starting"`.

2. La función `setTimeout` de la línea 3 se delega al Browser para que éste la ejecute. Y se continúa con la ejecución de la siguiente sentencia.

3. La sentencia de la línea 7 se inserta en la *pila de llamadas* y es ejecutada. Se imprime en la consola `"finishing"`.

4. En el Browser, la ejecución de `setTimeout` hace que se deba esperar un segundo. Cuando se termina, la función que recibe por parámetro `setTimeout`, se inserta en la *cola de callbacks*. Dado que no existen en el programa otras sentencias a ejecutar, el *loop de eventos* comienza a ejecutar aquello que se encuentra en el *cola de callbacks*. Toma la primera y la inserta en la *pila de llamadas* para luego ser ejecutada imprimiendo `"callback"` en la consola.

Es importante entender que todas las funciones callback que se insertan en la *callback queue* son ejecutadas luego de que no hay más sentencias para ejecutar en el programa. Veamos un ejemplo más:

```
1  console.log("starting");
2
3  setTimeout(() => {
4    console.log("callback");
5  }, 0);
6
7  console.log("finishing");
```

En el ejemplo anterior estamos pasando `0` segundos a `setTimeout`, y de igual forma el orden en el que se imprimen los mensajes en consola sigue siendo el mismo que en el ejemplo anterior: `"starting"`, `"finishing"`, `"callback"`. Esto es así porque los pasos de ejecución siguen siendo los mismos. `setTimeout` se envía al Browser, quien agrega a la *cola de callbacks* la función que recibe por parámetro para ser tomada y ejecutada solo después de la última sentencia del programa.

El mismo comportamiento sucede con llamadas Ajax, veamos el siguiente ejemplo:

```
1 console.log("starting");
2 fetch("https://jsonplaceholder.typicode.com/posts/1")
3   .then((response) => response.json())
4   .then((json) => console.log(json));
5 console.log("finishing");
```

La ejecución de `fetch` se delega en el Browser y sigue la ejecución del programa. Cuando el Browser recibe respuesta del servicio externo al que esta invocando, inserta las dos funciones en la *cola de callbacks*. Éstas son ejecutadas luego de imprimir "finishing."[en] la consola.

Recomiendo escuchar la charla de Philip Roberts[?] para más detalles de cómo funciona el intérprete de JavaScript.

Referencias

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript

[2] https://www.ecma-international.org/

[3] https://nodejs.org/

[4] https://requirejs.org/

[5] http://latentflip.com/loupe/