

Guía sobre JAVASCRIPT

Bases sólidas para comenzar a programar en React

15 de agosto de 2021

Índice general

1. Conceptos Esenciales de JavaScript	2
1.1. Entorno de Desarrollo	2
1.2. Introducción	3
1.3. Variables	4
1.4. Funciones	5
1.5. Arreglos	7
1.6. Objetos	10
1.6.1. Un Lenguaje basado en Prototipos	13
1.7. Clases	17
1.8. ¿Cómo se comporta <i>this</i> en JavaScript?	18
1.9. Módulos	20
1.10. Single Thread Language	24
1.11. The Promise Object and the async/await Keywords	25

Capítulo 1

Conceptos Esenciales de JavaScript

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 [1]. Esta sección se basa en conceptos explicados allí.

1.1. 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 sugerimos 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, realmente recomiendo 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.

1.2. Introducción

De la web Developer Mozilla JavaScript Documentation [1], 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 basada 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[2] (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 sección se implementaron en dicha versión.

Comencemos a aprender. Toda sentencia en JavaScript escrita en esta sección se ejecutará usando el intérprete 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 menu en "Terminal" 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 deployadas en producción. No es parte del estándar pero todos los browsers modernos y nodejs lo soportan.

1.3. Variables

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

```
1 | let myFirstVariable;
```

Cuando declaramos una variable sin inicializarla, el intérprete la inicializará con el valor especial *undefined*. Esto se puede observar si intentan imprimir en la consola una variable no inicializada. Pruebenlo. El siguiente ejemplo muestra como declarar una variable y como inicializarla a la vez.

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

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

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

Al declarar la variable con *const* no podremos modificarle su valor, si lo intentan el interprete 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.

```
1 | //my type is string
2 | let changeMyType = "Hello String!";
3 | //now it is number
4 | changeMyType = 100;
```

1.4. Funciones

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

```
1 function saySomething(string) {  
2   console.log(string);  
3 }  
4  
5 saySomething("Hello Function!");
```

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

```
1 (function saySomething(string) {  
2   console.log(string);  
3 })( "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*.

```
1 let x = saySomething("Hello Function!");  
2 //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 ésta variable para invocar la función.

```
1 function returnSomething(string) {  
2   return "This is it: " + string;  
3 }  
4  
5 //assigning a function to a variable  
6 let say = returnSomething;  
7  
8 //calling the function  
9 returnSomething("Hello js!");  
10 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.

```
1  function returnSomething(string) {  
2      return "This is it: " + string;  
3  }  
4  
5  //receives a function as parameter  
6  //invokes it and return the value  
7  function saySomethingMore(fn) {  
8      return fn("Hey !");  
9  }  
10  
11 //passing a function as argument  
12 saySomethingMore(returnSomething); //"This is it: Hey !"
```

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

```
1  //assigning the function  
2  const returnSomething = function (string) {  
3      return "This is it: " + string;  
4  };  
5  
6  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:

```
1  //arrow function with no parameters  
2  const arrowf1 = () => {  
3      return "arrowf1 was invoked!";  
4  };  
5  
6  //arrow function with one parameter  
7  //parenthesis is not necessary here  
8  const arrowf2 = param => {  
9      return "this is the argument: " + param;  
10 };  
11  
12 //arrow functions with one statement  
13 //in the body won't need return  
14 const arrowf3 = (a, b) => a + b;
```

1.5. 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:

```
1 //an empty array
2 let empty = [];
3
4 //an array
5 let family = ["José", "Nicolas", "Lucia", "Enrique"];
```

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

```
1 //an array
2 let family = ["José", "Nicolas", "Lucia", "Enrique"];
3 family[0]; //José
4 family[1]; //Nicolas
5 family[2]; //Lucia
6 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:

```
1 let family = ["José", "Nicolas", "Lucia", "Enrique"];
2
3 //adding an element at the end of an array
4 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 como se utiliza:

```
1 let myParents = ["EnriqueR", "Susana"];
2 let JoseParents = ["Eduardo", "Graciela"];
3 let family = ["José", "Nicolas", "Lucia", "Enrique"];
4 let all = [...myParents, ...JoseParents, ...family];
5 // [
6 //   'EnriqueR', 'Susana', 'Eduardo', 'Graciela',
7 //   'José', 'Nicolas', 'Lucia', 'Enrique'
8 // ]
```


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

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

```
1 let family = ["José", "Nicolas", "Lucia", "Enrique"];  
2 for (let element of family) {  
3   console.log("regular for: ", element);  
4 }
```

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

```
1 let family = ["José", "Nicolas", "Lucia", "Enrique"];  
2  
3 family.forEach(function (value, index, array) {  
4   //value is the element being processed  
5   //index is the index of the current value  
6   //array is the entire array  
7   console.log(value, index, array);  
8 });
```

`.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:

```
1 let family = ["José", "Nicolas", "Lucia", "Enrique"];  
2  
3 family.forEach((value) => {  
4   //do something with the value here  
5 });
```

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 como se utiliza a continuación:

```
1 | let family = ["José", "Nicolas", "Lucia", "Enrique"];
2 |
3 | const members = family.filter((member) => {
4 |   return member.length > 5;
5 | });
6 |
7 | //members = ['Nicolas', 'Enrique']
```

`.filter` recibe una función que evalúa por cada elemento del arreglo si es mayor 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:

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

```
1 | let numbers = [1, 2, 3, 4, 5, 6, 7];
2 | const chain = numbers
3 |   .filter((element) => {
4 |     return element % 2 !== 0;
5 |   }) // [1, 3, 5, 7]
6 |   .map((element) => {
7 |     return element * 2;
8 |   });
9 | //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:

```
1  let [one, two, three] = [1, 2, 3];
2  //one = 1
3  //two = 2
4  //three = 3
5
6  //same as the previous
7  let fewNumbers = [1, 2, 3];
8  [one, two, three] = fewNumbers;
9
10 //and here using spread syntax
11 let [a, b, ...rest] = [1, 2, 3, 4, 5];
12 //a = 1
13 //b = 2
14 //rest = [3, 4, 5]
```

1.6. Objetos

Existen varias formas de crear objetos en JavaScript. Aquí estudiaremos aquellas formas que utilizaremos con frecuencia programando en React. Comencemos estudiando como 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:

```
1  //an object literal
2  let mi = {
3    name: "Enrique",
4    surname: "Molinari",
5    sports: ["football", "tennis"],
6    address: {
7      street: "San Martin",
8      number: 125,
9    },
10   allSports: function () {
11     console.log(this.sports);
12   },
13 };
14 //this is an empty object
15 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:

```
1  let a = 1,
2      b = 2;
3  let obj4 = {
4      a,
5      b,
6  };
7  //obj4 = { a: 1, b: 2 }
```

Hasta el momento hemos visto como crear objetos utilizando la notación literal. Pero para aquellos que venimos estudiando lenguajes estáticos y compilados como Java, nos estaremos preguntando que 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 como funciona una función constructora.

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

```
1  function Book(name, authors, publishedYear) {
2      this.name = name;
3      this.authors = authors;
4      this.publishedYear = publishedYear;
5      this.fullName = function () {
6          return this.name + " by " + this.authors + ". " + publishedYear;
7      };
8  }
9
10 thisBook = new Book("Coding in React",
11                     ["Enrique Molinari"], 2021);
12 thisBook.fullName(); //Coding in React by Enrique Molinari. 2021
13
14 archBook = new Book("Coding an Architecture Style",
15                     ["Enrique Molinari"], 2020);
16 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()`.

1.6.1. 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 como 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
6  //printing thisBook
7  //Book {
8  //   name: 'Coding in React',
9  //   authors: [ 'Enrique Molinari' ],
10 //   publishedYear: 2021,
11 //   fullName: [Function (anonymous)]
12 //}
13 //printing archBook
14 //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 como 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 esta 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 lo 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 estas 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 al la cual le asigno un prototipo y el segundo parámetro es el prototipo a asignar.

1.7. 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 detras 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 + ". " +
11       ↪ this.publishedYear;
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,
2    ↪ Molinari"], 2021);
3  //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` esta 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.

1.8. ¿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 porque 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 éstos 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 si 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 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 explícita, decirle al método que valor de `this` queremos que utilice, algo similar a lo que hicimos con el método `call` en secciones anterior. Veamos como 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, los 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.

1.9. 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 [4]. Ahora esta 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 como 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 como 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
6   ↪   "./module/complex-module.mjs";
7   import { obj, complexThing } from
8   ↪   "./module/complex-module.mjs";
9
10  let o = new AClass("Enrique");
11  o.print();
12
13  //... more code here
```

1.10. Single Thread Language

As per the definition about JavaScript we gave at the beginning of this chapter, we know that JavaScript is a **single threaded** language. This means that the execution of a statement is one at a time. However, as we

know, JavaScript supports *asynchronous* operations, like Ajax calls. So, how does this work? The thing is that the JavaScript interpreter receives some help from the Browser, where somehow, as we will see later, are returned to the JavaScript interpreter as *callbacks*. There are a bunch of operations that the JavaScript interpreter delegates its execution to the Browser. These operations are called **Web APIs**. Among these Web APIs, we can find events (onclick, onmouseover, etc), fetch and XMLHttpRequest (ajax calls), setTimeout, etc.

To handle this, the execution environment of JavaScript has use: a *call stack*, the browser's Web APIs, a *callback queue* and its *event loop*. Lets see the following simple example:

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

Here is the list of how this program is executed in a browser:

1. Statement on line 1 is pushed on to the *call stack* and executed. "starting" is printed on the console.
2. The `setTimeout` on line 3 is delegated to the browser's Web API to be executed. Which basically wait for one second. However execution of the program continue, does not wait because the execution was delegated to the browser's Web API.
3. Statement on line 7 is pushed on to the *call stack* and executed. "finishing" is printed on the console.
4. After the one second elapsed from the `setTimeout` function, the callback arrow function passed as the first argument is then pushed into the *callback queue*. Since there are no more statements to be executed on the *call stack*, the *event loop* get from the top of the *callback queue* the arrow function and push it into the *call stack*. Finally it gets executed. "callback" is printed on the console.

Note that all the callback functions that end up in the *callback queue* gets executed after the call stack is empty and not before. To be very clear with this, look a the example below.

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

Note that on line 5 we are passing `0` seconds to the `setTimeout` function telling the Web API to not wait to push the callback arrow function into the *callback queue*. In any case, the result and the order of the console messages is the same as the example before: "starting", "finishing", "callback".

We can expect exactly the same behaviour from the example below that perform an ajax call:

```
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");
```

`fetch` is delegated to the Web API which performs an ajax call. Once the server respond, the callback arrow functions are pushed into the *callback queue*. The first callback function, on line 3, transform the response obtained from the server to json and the next callback function on line 4 prints that json in the console. Only after printing on the console the text "finishing" on line 5, is when those callbacks functions are pushed into the *call stack* and executed. You can find a more detailed explanation about the JavaScript interpreter and how it works, in the great talk by Philip Roberts[5].

1.11. The Promise Object and the `async/await` Keywords

The Promise Object was introduced in Javascript in ES2015. This object represents an asynchronous operation that might finish successfully or fail. See below how to create an instance of a Promise:

```
1 let p = new Promise(function (resolve, reject) {
2   //function to be executed by the constructor
3 });
4
5 //do something with p
```

As we can see above, the Promise constructor receives a function, called *executor*, that will be invoked by the constructor. The *executor* function receives two additional functions as parameters. The body of the *executor* function perform typically an asynchronous operation and finish by calling the `resolve(value)` function if everything goes well or `reject(reason)` otherwise. See how this is done bellow:

```
1 | let p = new Promise(function (resolve, reject) {  
2 |   //long async operation  
3 |   setTimeout(() => resolve("finished"), 1000);  
4 | });
```

On the example above using `setTimeout` we are simulating an operation that takes one second to finish. After that operation finish it will call the `resolve` function passing as value the string "finished". What can we do with that then?

The Promise object have the `then(handleResolved)` method that receives a function that allows you to work with the parameter passed when you invoke the `resolve` function like we do on line 3 below:

```
1 | let p = new Promise(function (resolve, reject) {  
2 |   //long async operation  
3 |   setTimeout(() => resolve("finished"), 1000);  
4 | });  
5 |  
6 | p.then((value) => console.log(value));
```

As we can see above, on line 6 we are passing the "finished" string as parameter named `value` to the function passed to the `then(handleResolved)`. Then that value is just printed. What is important to note here is that the `handleResolved` function passed to the `then(...)` method is executed only once the promise is resolved.

Suppose now something goes wrong withing the executor, then it can be handled in a different way, see below:

```
1 | let p = new Promise(function (resolve, reject) {  
2 |   //long async operation  
3 |   setTimeout(() => reject("can't be done..."), 1000);  
4 | });  
5 |
```

```
6 | p.then((value) => console.log("success: " + value))
7 |   .catch((value) => console.log(value));
```

Note that on line 3 now we are calling the `reject(reason)` function passing the `reason` value as the string `"can't be done"`. Then, on line 7 note how the function passed to the `catch` method is the one that gets called.

During this book, and usually in React we don't write promises, but we use them frequently. By using the `fetch` method to retrieve data from an external API, we have to deal with a promise. Look at the example below:

```
1 | function fetchPost() {
2 |   fetch("https://jsonplaceholder.typicode.com/posts/1")
3 |     .then((response) => response.json())
4 |     .then((json) => console.log(json));
5 | }
6 |
7 | fetchPost();
```

As you can see on line 2 above, the `fetch` method call returns a promise, which allows us to call the `then` method on it, to work with the fetched data.

Another way to write this is by using the `async` and `await` keywords. These keywords were added to Javascript on ES2017 allowing us to write asynchronous code in a synchronous way. Let's rewrite one of our previous examples to take advantage of these keywords. First we will create a function that return a promise. Functions that return promises are (usually) asynchronous:

```
1 | function thePromise() {
2 |   return new Promise(function (resolve, reject) {
3 |     //long async operation
4 |     setTimeout(() => resolve("finished"), 1000);
5 |   });
6 | }
```

To call these functions the `await` keyword is used provoking to stop the execution until the promise is `resolved` or `rejected`. The call code must be a function declared with the `async` keyword. See below:

```
1 | async function testingKeywords() {
2 |   console.log("before");
3 |   const data = await thePromise();
```

```
4 |     console.log("after");  
5 |     console.log(data);  
6 | }
```

Note that we declared the function `async`, line 1, and we use the `await` keyword before calling the asynchronous function on line 3. It is important to understand that the order of the messages printed on the console is the same as the order of the sentences inside the `testingKeywords()` function, as opposed to what we have discussed before on section 1.10.

To finish this section we are going to rewrite the `fetchPost()` function to use these new keywords. See below:

```
1 | async function fetchPost() {  
2 |     let data = await  
   |     ↪ fetch("https://jsonplaceholder.typicode.com/posts/1");  
3 |     data = await data.json();  
4 |     console.log(data);  
5 | }  
6 |  
7 | fetchPost();
```

As you can see, again, we are declaring the function `async`, and in this case the `fetch` call on line 2, is prepended with the `await` keyword. Prepend the `await` to the `fetch` function means that instead of returning a `Promise` object, it returns (if the promises resolve, you can use a try/catch block to handle errors) a `Response` object. That allows us to call on line 3 directly to the `json()` method of the `Response` object. As that method returns a `Promise`, we also prepend the sentence with the `await` keyword, giving us a Javascript object that is finally printed on the console.

It is important to note that `await` can only be used inside an `async` function.

Bibliografía

- [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/>