3. JavaScript Avanzado

3.1. Introducción y objetivos

Dos términos que constituyen el núcleo de JavaScript son los objetos y las funciones. Por un lado, los objetos engloban las distintas formas de agrupar información (*arrays*, sets, mapas, etc.). Por otro lado, las funciones son las instrucciones que van a operar con esos conjuntos de información y devolvernos información nueva. Esto no es otra cosa, sino el objetivo básico de cualquier lenguaje de programación y, por extensión, cualquier programa.

Dicho de otra manera, los objetos y las funciones son lo que hacen de JavaScript un lenguaje de programación completo, además del más usados en la historia de la informática y la tecnología.

Pero JavaScript nos tiene guardada una sorpresa más: las funciones son también objetos. Esto quedará más resaltado sobre todo cuando estudiemos las funciones de

retrollamada o *callbacks*.

Más concretamente, en este tema hablaremos de:

* Cómo una función puede recibir argumentos.
* La notación nueva para definir funciones de manera rápida en ES6.
* Las funciones asociadas a *arrays*, sets y mapas, también conocidas como funciones de orden superior.
* La definición directa de objetos mediante su notación literal.
* El objeto *this.*

3.2. Definición e invocación de funciones

Existen varias maneras de definir una función, pero la más recomendable es la siguiente:

A close-up of a white background

Description automatically generated

Figura 1. Declaración de una función 1. Fuente: elaboración propia.

Estrictamente hablando, lo que ha ocurrido es que hemos asignado una función anónima (sin nombre) a una variable. No hay problema en darle un nombre a la función, pero no tiene ninguna utilidad especial:

A white card with green and blue text

Description automatically generated

Figura 2. Declaración de función 2. Fuente: elaboración propia.

El comportamiento de una función puede ser modificado mediante los parámetros de entrada que comentaremos más tarde. Fíjate también que ambos ejemplos acaban en un punto y coma (;) en la última línea. Esto es así porque se trata de una instrucción de asignación y, como tal, debe acabar en este símbolo (aunque el cuerpo de la función tenga varias líneas).

Todas las funciones tienen una propiedad, su nombre (name). Es posible acceder a ella con la propiedad .name de la siguiente manera:



Figura 3. Cómo acceder a una propiedad de una función. Fuente: elaboración propia.

Esto debería resultarte curioso con respecto a otros lenguajes de programación. Acabamos de invocar un método/propiedad sobre una función. Esto solo puede significar una cosa, las funciones son objetos. Otra cosa que te resultará curiosa es que, si lo piensas, en los ejemplos anteriores hemos usado una notación literal para definir una función. ¿Acaso no existe una notación para definir funciones que haga uso de constructores? Pues sí, es esta:

A white card with blue and green text

Description automatically generated

Figura 4. Declaración de función 3 . Fuente: elaboración propia.

Sin embargo, no se recomienda su uso, ya que resulta un poco farragoso. Solo lo hemos indicado a nivel académico.

Invocación de funciones

Una vez que hemos definido nuestra función, ya podemos invocarla. Al hacerlo, se ejecutará el conjunto de instrucciones que estaba empaquetado en la misma. Para ello, simplemente tenemos que sufijar un paréntesis de apertura y cierre al nombre de la variable que habíamos asignado antes a la función. Ejemplo:

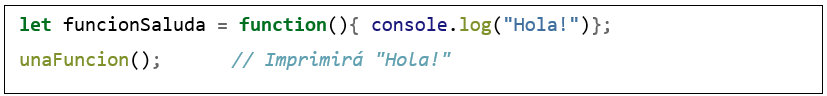


Figura 5. Invocación de funciones. Fuente: elaboración propia.

Si no pusiéramos los paréntesis, JavaScript nos recuerda el hecho de que la variable funcionSaluda apunta, efectivamente, a una función:

A white background with blue text

Description automatically generated

Figura 6.Importancia de los paréntesis para invocar funciones. Fuente: elaboración propia.

Si te has dado cuenta, en muchos ejemplos aparecía la expresión «devuelve tal o cual». Esto es así porque la función de turno, efectivamente, devolvía un valor derivado. Al igual que en matemáticas, las funciones devuelven «cosas» (no solo números). Para indicar que una función retorna algún tipo de información, se usa la cláusula return. Ejemplo:

A black and white rectangle with a black border

Description automatically generated

Figura 7. Palabra reservada return. Fuente: elaboración propia.

Cuando asignamos la invocación de una función a una variable, esta inicializa (o actualiza) con el valor devuelto por la función:

A black rectangle with black lines

Description automatically generated

Figura 8. Guardar el resultado devuelto por una función. Fuente: elaboración propia.

Esquemáticamente, una función suele representarse como una caja negra en la que, por un lado, se ingresan parámetros y, en el otro, se obtiene un resultado tras operar con esos parámetros.

A close-up of a text

Description automatically generated

Figura 9. Representación esquemática de una función. Fuente: elaboración propia.

3.3. Argumentos

Los argumentos o parámetros son los valores que se les pasan a las funciones como entrada.

Desde dentro de la función, las instrucciones que la componen tienen acceso a estos parámetros. En el siguiente ejemplo calculamos la energía asociada a una masa usando la famosa ecuación de Einstein:

La implementación quedaría así:

A white rectangular object with black text

Description automatically generated

Figura 10. Fórmula de la energía en JavaScript. Fuente: elaboración propia.

El ejemplo anterior tiene dos cosas a resaltar.

En primer lugar, una función tiene acceso a valores y variables externos que se encuentran dentro del mismo bloque.

Estos bloques pueden corresponderse a ámbitos locales o al ámbito global. En el ejemplo anterior, la función tiene acceso a la constante c (la velocidad de la luz) sin mayor problema, ya que esta ha sido definida en el mismo bloque que contiene la definición de la función energia (en este caso es el bloque correspondiente al ámbito global).

Por otro lado, en el ejemplo también se muestra la nueva manera introducida en ES6 para expresar la operación matemática elevado a: base\*\*exponente. Antes de ES6 era necesario utilizar la función Math.pow(base, exponente). Luego, veremos que Math es en realidad un objeto que define un espacio de nombres (que a su vez define la función pow()).

A diferencia de otros lenguajes de programación, en JavaScript no es necesario especificar cuántos parámetros son necesarios en una función. Tampoco es necesario avisar del tipo de estos argumentos.

Pero ¿cómo podemos, desde dentro de una función, acceder a un número arbitrario de argumentos? Para ello, JavaScript cuenta con una variable especial llamada arguments. Esta variable no es más que una especie de *pseudoarray*(no es un *array* real) que contiene cada uno de los parámetros que se pasan a la función en orden.

Podemos referirnos a cada uno de ellos usando la misma notación para seleccionar elementos de los *arrays*. En el siguiente ejemplo simplemente devolvemos el primer elemento que se nos pasa:

A white background with green and black text

Description automatically generated

Figura 11. Definición e invocación de una función. Fuente: elaboración propia.

A partir del ES6 existe una mejor forma de trabajar con un número arbitrario de parámetros y esta es con el operador de propagación (...). Este operador transforma todos los argumentos pasados en un array «de verdad»:

A white rectangular object with green text

Description automatically generated

Figura 12. Operador de número variable de argumentos. Fuente: elaboración propia.

Internamente, el operador (...) transforma 4, 5, 1, 3 en [4, 5, 1, 3].

Desde el ES6, las funciones pueden tener parámetros por defecto. Es decir, si no pasamos un parámetro concreto, se tomará uno concreto predefinido. Ejemplo:

A white rectangular object with black text

Description automatically generated

Figura 13. Función con argumentos por defecto. Fuente: elaboración propia.

3.4. Funciones flecha

Con ES6 también se introduce una nueva sintaxis para definir funciones, esta nueva manera es más limpia y directa. Está inspirada en el superset de JavaScript llamado CoffeeScript.

Las funciones flecha (o *arrow functions*) pueden identificarse, precisamente, por la presencia de una flecha (=>). Este símbolo separa los parámetros que recibe una función del cuerpo de la función.

Ejemplo:



Figura 14. Uso de función flecha. Fuente: elaboración propia.

El ámbito de uso de las funciones flecha es, sobre todo, para especificar funciones que sean cortas: pocas líneas de código (normalmente tan solo una) y lógica sencilla. Además, tienen estas características:

* Si solo reciben un parámetro, no es necesario usar paréntesis.
* No es necesario usar la notación de bloque ({}) si la función solo tiene una línea de código.
* La cláusula return tampoco es necesaria si la función solo tiene una línea. En este caso, se devuelve el valor que retorne la instrucción de esa precisa línea.

Veamos un ejemplo:

A black rectangle with black text

Description automatically generated

Figura 15. Declaración de función flecha. Fuente: elaboración propia.

En la función sumarDosNumeros tenemos una función que recibe dos parámetros (a y b), los suma y devuelve el resultado de la suma. Es como si hubiera un return a + b en lugar de solo a + b.

En el siguiente ejemplo, la función flecha es un poco más compleja. Para empezar, consta de dos líneas de código, por lo que ya hay que poner llaves indicadoras de bloque ({}). Por las mismas razones, también es necesario indicar explícitamente el valor devuelto con return.

A white background with black and green text

Description automatically generated

Figura 16. Función flecha más extensa. Fuente: elaboración propia.

3.5. Funciones de retrollamada o callback

Las funciones de retrollamada, o más comúnmente conocidas como *callbacks,* son funciones que se pasan como parámetro a otras funciones. Sí, además de cadena de texto, números, *arrays*, objetos, etc., también podemos pasar funciones. No podía ser de otra manera, ya que las funciones también son objetos. Por ejemplo, una función puede tener propiedades:

A white rectangular object with blue text

Description automatically generated

Figura 17. Función con propiedades. Fuente: elaboración propia.

Puedes comprobar por ti mismo cómo lo que hemos hecho es definir una función (funConProps) desde la cual ella misma accede a una propiedad (funConProps.unaPropiedad). Esta propiedad es inicializada/modificada después. Pero volvamos a las *callbacks*.

El objetivo de una *callback* es que sea invocada en algún momento a lo largo de la función que la recibe.

Las *callbacks* son la base de la programación basada en eventos, la cual es muy usada en aplicaciones basadas en JavaScript.

Puedes consultar más sobre la programación basada en eventos de JavaScript en la sección de A fondo, *Programación basada en eventos.*

Veamos un ejemplo sencillo de definición y uso de una *callback:*

A screenshot of a computer code

Description automatically generated

Fragmento de JavaScript 18. Función con una *callback*. Fuente: elaboración propia.

La función diAlgoYAlgoMas recibe dos parámetros: una cadena de texto y una *callback.* Esta *callback*(yAlgoMas) no es más que una función sencilla de tipo *arrow,* que simplemente devuelve una cadena de texto fija. Con estos dos valores, diAlgoYAlgoMas simplemente los empalma, pero fíjate que el segundo lo extrae de la invocación de la *callback* (funcionCallback()).

El ejemplo anterior también contiene dos cosas reseñables del lenguaje JavaScript.

Por un lado, si una función flecha no acepta parámetros, tenemos que seguir indicando los paréntesis: let yAlgoMas = () => "y algo más!";.

Y la segunda cosa, y más importante, es que, si te has dado cuenta, la función *callback* se define, en este caso, después de la función en la que es utilizada. Esto debería llamarte la atención. Hasta ahora, siempre que usábamos algo (una variable, constante, etc.) debía ser definido con antelación.

Casi todos los lenguajes de programación legendarios (C, C++, etc.) tienen este requerimiento. Entonces, ¿por qué puede definirse la función yAlgoMas después? Pues muy sencillo, por una cosa que se llama *hoisting.*

El *hoisting* implica que JavaScript se ocupa personal y automáticamente de que las variables y funciones estén definidas antes, aunque hayamos escrito su definición después.

3.6. Funciones de orden superior

Las funciones de orden superior no son más que métodos que se aplican sobre *arrays*, sets y mapas. Veamos las más importantes.

Recorrido de arrays

El método forEach(callback) recibe como único parámetro una función *callback* y permite recorrer *arrays* de manera muy sencilla:

A white background with blue text

Description automatically generated

Figura 19. Recorrido de *arrays* con forEach. Fuente: elaboración propia.

En cada llamada de la *callback* (que tiene forma de función flecha en este caso), se pasa el valor de un elemento concreto (matematico) y el índice que ocupa dentro del *array* (indice). Con estos dos parámetros, forEach permite realizar operaciones sobre cada elemento del array (matematicos).

Habrás podido comprobar que la función *callback* no tiene ningún nombre y no ha sido definida previamente. Aparece sin nombre y directamente como parámetro del método forEach(). A este tipo de funciones se las llama funciones anónimas.

Derivación de arrays

El método .map(callback) permite hacer algo parecido al forEach(), solo que, en cada paso, el valor derivado a partir de cada elemento se añade a un nuevo *array.*



Figura 20. Derivación de *arrays*. Fuente: elaboración propia.

Filtrado de arrays

El método filter(callback) permite aceptar o rechazar los elementos de un *array,*los cuales son testeados por la función *callback,* la cual los recibe uno a uno. Cada prueba debe devolver true o false según se pase o no. Veamos un ejemplo que filtra solo los números impares de un *array* que contiene, inicialmente, tanto impares como pares.

A number and numbers in a rectangular frame

Description automatically generated with medium confidence

Figura 21. Filtrado de *arrays*. Fuente: elaboración propia.

En la sección A fondo, *Métodos sobre arrays,* encontrarás un enlace a la documentación de Mozilla que contiene muchos más métodos disponibles para los *arrays*.

3.7. Objetos literales

Todo dato en JavaScript tiene como tipo cualquiera de los siguientes:

* Los tipos básicos, como cadenas de texto, números, booleanos, símbolos,
* undefined y null.
* Los objetos como arrays, sets, mapas y objetos literales.

Los objetos literales no son más que conjuntos de propiedades y sus valores. Los valores de las propiedades pueden ser a su vez cadenas de texto, números, etc. e incluso otros objetos y funciones.

En el caso de que la propiedad tenga como valor una función, a esta se la llama método.

Los objetos son como pequeñas bases de datos que contienen parejas de propiedad y valor. En ese sentido, son parecidos a los mapas que vimos anteriormente. La salvedad es que las propiedades son ahora solo cadenas de texto y sus valores pueden ser cualquier cosa. También incorporan mecanismos interesantes, como el encapsulamiento de código y la herencia, propios de la programación orientada a objetos.

Los objetos literales suelen usarse para mantener información y funcionalidad sobre esa información de manera cohesionada y en un mismo lugar.

Definición de objetos de manera literal

Para definir un objeto usando la notación literal o directa, simplemente tenemos que listar, entre llaves de bloque y separados por comas, parejas de propiedades y su valor.

Ejemplo:

A white rectangular object with blue text

Description automatically generated

Figura 22. Declaración de objeto de forma literal. Fuente: elaboración propia.

En este ejemplo hemos definido un objeto (buzz) que tiene tres propiedades. Una de ellas (nombre) es una cadena texto (tipo básico), otra es un *array* (amigos) y la otra se corresponde con una función, escrita con notación de flecha. De igual manera, los valores de las propiedades pueden estar almacenados externamente en variables o constantes:

A white rectangular object with black text

Description automatically generated

Figura 23. Almacenar propiedades en variables. Fuente: elaboración propia.

Desde ES6, se incorpora una nueva sintaxis resumida para definir objetos literales. Esta consiste en que, si el nombre de la propiedad y el de la variable a la que se refiere tienen el mismo nombre, basta con poner unas de las dos.

Es decir, basándonos en el ejemplo anterior, tendríamos:

A black and white rectangular frame with a black border

Description automatically generated

Figura 24. Sintaxis alternativa de objetos literales. Fuente: elaboración propia.

En este ejemplo, hemos creado un objeto (rex) con una propiedad llamada nombre, que recibe como valor el albergado en una variable con el mismo, valga la redundancia, nombre.

Habrás podido comprobar que hemos estado creando objetos usando una notación literal, de ahí que nos estemos refiriendo todo el tiempo a objetos literales. Evidentemente, existe una notación de función constructora, pero que apenas es utilizada:



Figura 25. Función constructora de objetos. Fuente: elaboración propia.

Métodos y propiedades

Es posible acceder a las propiedades de un objeto mediante la notación de punto (objeto.propiedad). Ejemplo:



Figura 26. Acceso a propiedades con notación de punto. Fuente: elaboración propia.

También es posible usar la notación de corchetes (objeto[propiedad]). Ejemplo:



Figura 27. Acceso a propiedades con notación de corchetes. Fuente: elaboración propia.

De igual manera podemos ejecutar las funciones que se encuentran detrás de algunas propiedades, es decir, podemos invocar métodos.

Ejemplo:



Figura 28. Acceso a métodos con notación de punto. Fuente: elaboración propia.

Como se trata de una función que queremos que se ejecute, tenemos que sufijar los paréntesis de apertura y cierre. Es posible igualmente usar la notación de corchetes que hemos visto antes:



Figura 29. Acceso a métodos con notación de corchetes. Fuente: elaboración propia.

¿Cómo podemos comprobar si un objeto tiene una propiedad? Para eso JavaScript tiene el método hasOwnProperty(nombrePropiedad). Veamos un ejemplo:



Figura 30. Método hasOwnProperty. Fuente: elaboración propia.

O también podemos obtener la lista, en forma de *array*, de todas las propiedades de un objeto con el método Object.keys(objeto). Ejemplo:



Figura 31. Método keys. Fuente: Elaboración propia

La versión ES8 (ojo, no ES6) que vio la luz en 2017 añade el método Object.values(). Este funciona de manera parecida al anterior, pero ahora devuelve un *array* con los valores de estas propiedades:

A close-up of a logo

Description automatically generated

Figura 32. Método values. Fuente: elaboración propia.

Modificación de objetos

No tenemos que conformarnos con el momento de definición de un objeto para dejar claro con qué propiedades cuenta. En cualquier momento podemos añadir una nueva propiedad de manera muy sencilla:



Figura 33. Añadir una propiedad a un objeto. Fuente: elaboración propia.

De igual manera, no hay problema en modificar el valor de una propiedad ya existente:



Figura 34. Modificación de propiedades. Fuente: elaboración propia.

Eso sí, a la hora de borrar propiedades tenemos que usar un nuevo operador llamado delete. Si queremos borrar una propiedad de un objeto, tenemos que indicarlo con la siguiente sintaxis:



Figura 35. Eliminando propiedades. Fuente: elaboración propia.

Este método devuelve true o false según el éxito de la operación. Veamos un ejemplo:

A white rectangular sign with blue text

Description automatically generated

Figura 36. Eliminando propiedades 2 . Fuente: elaboración propia.

Objetos como parámetros

Evidentemente, no hay problema en pasar objetos como parámetros de funciones o métodos. Sin embargo, hay que tener en cuenta que, a diferencia de los tipos básicos, cuando un objeto es pasado como parámetro en una función, este pasa por referencia.

Es decir, dentro de la función, seguimos teniendo acceso al objeto original y no a una copia de este. Esta afirmación no vale para los datos básicos (cadenas de texto y números).

Cuando pasamos un número o cadena de texto como parámetro a una función, su valor se copia y la función maneja dicha copia.

El uso de objetos literales como parámetros de función tiene una utilidad muy concreta en JavaScript, permite pasar a la función información etiquetada, en cualquier orden y de manera muy limpia. Veamos un ejemplo:

A white background with black and red text

Description automatically generated

Figura 37. Paso de argumentos. Fuente: elaboración propia.

3.8. El objeto this

La palabra especial this en JavaScript usada dentro de, por ejemplo, un objeto, hace referencia al propio objeto. Es, en cierta medida, equivalente al this de Java o al self de Objective-C (muy usado hasta 2017 en la programación nativa en macOS e iOS).

Cuando ya tenemos una referencia al propio objeto (a sí mismo) podemos acceder a sus propiedades y hacer cosas con ellas desde las funciones/métodos del propio objeto:

A screen shot of a computer code

Description automatically generated

Figura 38. Usando this. Fuente: elaboración propia.

En este ejemplo, el método tira() puede acceder a la propiedad metal contenida en sí misma. Importante, para hacer referencia a this desde una función definida en un objeto, tenemos que usar la notación habitual de funciones. Dicho de otra manera: las funciones *arrow* no tienen acceso al objeto this.

3.9 Promesas y funciones asíncronas

Las promesas son una nueva manera de llamar a las funciones *callback.* En ese sentido, no añaden ninguna funcionalidad especial, pero simplifican el proceso asíncrono como los de las funciones AJAX. Cuando una promesa se crea, esta invoca una función asíncrona y al objeto promesa (sí, son objetos, como los eventos) se queda en posición de espera hasta que esta termine.

Cuando esto ocurre, puede devolver:

* Resuelta: cuando la operación asíncrona se llevó a cabo satisfactoriamente.
* Rechazada: si la operación asíncrona tuvo algún problema.

El esqueleto en JavaScript de una promesa sería el siguiente:

A screenshot of a computer program

Description automatically generated

Figura 39. Promesa en JavaScript. Fuente: elaboración propia.

La forma de marcar este éxito o no es mediante dos funciones: todoOK() y algoFueMal(). Puedes comprobar por ti mismo como, a pesar de que las promesas solucionan gran parte del código sucio del que eran culpables las *callbacks,* todavía resultan un poco farragosas en términos de sintaxis.

Para simplificar todavía más la sintaxis, en la especificación de JavaScript de 2017 se añadieron las funciones asíncronas:

* Este tipo de funciones están precedidas por la palabra clave async. Dentro de este tipo de funciones podemos ejecutar código asíncrono como si fuera síncrono.
* En cada línea de esta función tenemos que spousar el operador await para marcar una función como asíncrona.
* El valor devuelto se empaqueta como una promesa que puede almacenarse en una variable. Las siguientes líneas de código no se ejecutan hasta que la promesa se resuelve.

Esto se entenderá mejor con un ejemplo:

A white rectangular object with black text

Description automatically generated with medium confidence

Figura 40. Async y await. Fuente: elaboración propia.

En las líneas anteriores, la función cargarUsuario está precedida por la palabra reservada async, lo que implica que la función se ejecuta de manera asíncrona (es decir, al mismo tiempo que el código restante del programa). Dentro de la función, cada línea se ejecuta línea a línea y esto lo garantiza la palabra reservada await.

3.10. Closures

Los *closures* (o cierres, aunque jamás se usa su traducción en español) son una de las características más eficientes de JavaScript y de la programación funcional. Un *closure* es una referencia a una variable que fue creada en el ámbito de otra función, pero esta variable permanece accesible y puede ser usada en otra parte del programa.

La principal característica de un *closure* es que una función interna es declarada dentro de otra función y la primera tiene acceso al ámbito de la segunda.

A computer code with text

Description automatically generated with medium confidence

Figura 41. *Closure*. Fuente: elaboración propia.

La función funcionExterna() solo tiene acceso a la variable variableExterna porque ha sido declarada en su ámbito. La función funciónInterna(), por otro lado, tiene acceso a ambas variables: variableExterna y variableInterna.

Esto implica que siempre que una función sea declarada dentro de otra función, la función interna tendrá acceso a las variables que son declaradas en la función externa.

La potencia de los *closures* radica en que podemos definir una función que devuelve otra función que tiene acceso a las variables de la primera. Esta potencia puede usarse para generar lo que se conoce como fábricas de funciones.

Una fábrica de funciones es una función externa que permite crear funciones hijas que son parecidas o tienen algún tipo de relación.

Veamos un ejemplo. Supongamos que queremos añadir la funcionalidad de convertir entre distintas unidades de longitud. Es decir, queremos poder pasar de metros a centímetros o de millas a kilómetros. Podríamos hacerlo con una fábrica de funciones donde cada función fuese una transformación concreta de una unidad a otra:

A screenshot of a computer code

Description automatically generated

Figura 42. Fábrica de funciones usando *closures*. Fuente: elaboración propia.

En el ejemplo anterior, con una sencilla fábrica de funciones hemos definido dos funciones:

* Con la primera pasamos de metros a centímetros.
* Con la segunda pasamos de centímetros a metros.