2. Introducción a Javascript

2.1. Introducción y objetivos

El lenguaje de programación por excelencia para la programación en el lado cliente es JavaScript. Prácticamente todos los navegadores modernos y más utilizados tienen soporte para este lenguaje.

En este tema realizaremos una introducción a este lenguaje y veremos los siguientes aspectos:

* Historia. Una breve introducción a la historia de este lenguaje desde su concepción hasta nuestros días.
* IDE (*Integrated Development Environment*) y herramientas. Exploraremos cuáles van a ser las herramientas que usaremos para desarrollar código en JavaScript.
* Conceptos básicos. Se estudiarán la base de este lenguaje, su sintaxis y los tipos de datos que podemos utilizar con él. Como verás, comparte similitudes con otros lenguajes de programación muy conocidos, pero tiene algunas peculiaridades que señalaremos.
* Estructuras de control e iterativas. Estudiaremos cuáles son las sentencias en JavaScript que nos permiten crear lógicas más complejas, en función de ciertas condiciones, incluyendo las iteraciones (bucles).

Si bien todo el contenido que se expondrá aquí es suficiente para conocer las bases de este lenguaje, recomendamos leer la documentación de este lenguaje en el recurso de A fondo, *Documentación de JavaScript,* siempre que se quiera obtener más información sobre cualquiera de los puntos que se mencionarán.

2.2. Historia de JavaScript

Una mañana de mayo de 1995, un directivo de Netscape se acercó a un joven, Brendan Eich, y le dijo algo como esto:

«En menos de dos semanas lanzamos nuestro siguiente navegador: el *Netscape Navigator 2.0.* Queremos que incorpore un intérprete para un lenguaje de programación que pueda interactuar con las páginas HTML que el usuario visualice. Por cierto, le pondremos algún nombre relacionado con Java, ya que este es ahora mismo el lenguaje más famoso y se usa cada vez más en navegadores. Nos vamos a aliar con *Sun Microsystems* (quien detenta la marca *Java*) para intentar frenar a nuestro principal competidor: Microsoft (y su navegador *Internet Explorer*)».

El joven Eich ya sabía lo que tenía que hacer: sacar un nuevo lenguaje de programación de la chistera en un muy corto espacio de tiempo y hacer que se pareciera algo a Java para poder venderlo como tal a la comunidad de desarrolladores.

Inicialmente llamó a su creación Mocha (por el café moca) y luego lo cambió a LiveScript, sin embargo, fue obligado a incluir la palabra Java por motivos exclusivamente comerciales. Sun Microsystems (creador de Java) y Netscape tenían intereses en potenciar el JavaScript y querían que su nombre tuviera gancho. Justo en ese momento, Java era el lenguaje de programación más popular.

A pesar de compartir ciertas similitudes sintácticas, Java y JavaScript son lenguajes completamente distintos y no están relacionados.

JavaScript hizo su debut en la versión 2.0 del navegador Netscape (1995). En 1996, Microsoft, mediante ingeniería inversa, lanzó su propia versión de JavaScript a la que llamó JScript (para evitar infringir la marca ostentada por Netscape) en la versión 3.0 de Internet Explorer.

JavaScript se popularizó inmediatamente, ya que disponía de una baja barrera de aprendizaje. Como contrapartida, pocos desarrolladores se preocuparon por aprender el lenguaje desde el punto de vista técnico y teórico. La mayoría de los programadores web se limitaban (y aún sucede más de lo que a cualquiera le gustaría) a copiar y pegar segmentos de código que veían aquí y allá sin entender cómo funcionaban. Esto llevó a una incomprensión muy grande de un pedazo de tecnología maravillosa. Los programadores profesionales siempre han denostado a JavaScript hasta épocas recientes.

1996. JavaScript pasa a ser un estándar internacional

Sun Microsystems decidió estandarizar el lenguaje junto con la Asociación Europea de Fabricantes de Computadoras (ECMA), quien lo acogió. La versión del lenguaje estandarizado fue llamada ECMAScript para evitar conflictos futuros con la marca Java (ostentada por Sun). A veces, esta doble nomenclatura causa confusión.

A efectos prácticos podemos considerar que ECMAScript es lo mismo que JavaScript, pero típicamente suele emplearse el término JavaScript para ECMAScript en el navegador.

2005. Comienza la dignificación de JavaScript

A comienzos de 2005 emergieron una serie de sitios web que mostraban la versatilidad que podía alcanzarse con aplicaciones web. Estamos hablando de servicios populares, como [Google Maps](https://maps.google.com/) o [Gmail](https://gmail.com/). Estos sitios web prácticamente se comportaban como aplicaciones de escritorio. A estas altas capacidades y versatilidad se las llamó comúnmente web 2.0.

JavaScript resultó esencial en la consecución de la web 2.0, pero, más concretamente, lo que de verdad supuso una catapulta fue la capacidad de modificar ciertas partes de una página web sin necesidad alguna de recarga.

Es lo que se conoce como AJAX, concepto originalmente acuñado por James Garret en un famoso artículo de 2005, *Ajax: A New Approach to Web Applications.*

AJAX permite la búsqueda de información por parte de una página web (mediante código JavaScript albergado en la misma) y el «redibujado» de la misma con base en la información recabada.

2009. Llega JavaScript en el servidor con Node

Un buen día de 2009, un chico sencillo llamado Ryan Dahl (<https://tinyclouds.org/>) tuvo una idea sensacional: usar el motor V8 de Google y sacarlo fuera del navegador. Su sencillo gesto cambió el mundo de la tecnología tal y como lo conocemos. Se convirtió en un software tremendamente popular y, rápidamente, mucha gente comenzó a desarrollar aplicaciones escritas en JavaScript, pero que eran ejecutadas en un ordenador normal y corriente. Es decir, estos programas se ejecutaban como cualquier otro software por parte del sistema operativo.

Hoy en día, Node (o Node.js, <https://nodejs.org/en/>) es un componente fundamental de cualquier desarrollo en el mundo web, ya sea en el cliente (navegador) o servidor.

Junto a Node tenemos al NPM (*Node Package Manager*), que es el gestor de paquetes por defecto que incluye la instalación de Node (a partir de la versión 0.6.3). NPM se ejecuta desde la línea de comandos y se encarga de manejar las dependencias de una aplicación escrita en JavaScript.

2.3. IDE y herramientas

Para trabajar con JavaScript se recomienda hacer uso de las siguientes herramientas, aunque cada alumno puede elegir otras si así lo desea.

A screenshot of a blue and white text

Description automatically generated

Tabla 1. Herramientas usadas para JavaScript. Fuente: elaboración propia.

2.4. Conceptos básicos de JavaScript

En esta sección se estudiará la sintaxis básica de JavaScript. Trataremos los siguientes aspectos:

* Tipos de datos primitivos.
* Declaración de variables y constantes.
* Expresiones y sentencias.

Tipos de datos primitivos

JavaScript es un lenguaje dinámico, esto implica que no es necesario especificar el tipo de los valores (si es un número, una palabra, una lista de cosas, etc.), sino que JavaScript los averigua de manera automática y, además, estos pueden cambiar sin ningún problema. Por ejemplo, es posible especificar que un valor vale inicialmente 1 (un número) y luego «hola» (una cadena de texto).

A black and white rectangle with black lines

Description automatically generated

Figura 1. Variable cuyo tipo de dato cambia dinámicamente. Fuente: elaboración propia.

JavaScript discernirá automáticamente que esa variable a es primero de tipo número (*number*) y luego de tipo cadena de texto (*string*). El ejemplo anterior produciría un error gravísimo en otros lenguajes tipados como Java o C++. En estos lenguajes, una vez que una variable ha sido especificada con un valor (de manera implícita o explicita), ya no hay vuelta atrás: no puede cambiar de tipo.

En los lenguajes tipados, si un valor nace con un tipo, ya no es posible cambiarlo. Esto busca que el código sea más seguro y menos propenso a errores. JavaScript es un lenguaje no tipado, es decir, dinámico.

Hay otros muchos lenguajes dinámicos además de JavaScript, los principales serían Ruby y Python.

Los tipos de datos básicos (también llamados primitivos) en JavaScript son:

A list of text on a paper

Description automatically generated

Tabla 2. Tipos de datos primitivos en JavaScript. Fuente: elaboración propia.

JavaScript cuenta con la instrucción  typeof  para averiguar el tipo de una variable:

A close-up of a white background

Description automatically generated

Figura 2. Instrucción  typeof . Fuente: elaboración propia.

En realidad,  typeof  es lo que se conoce como un operador: es una construcción sintáctica especial que se comporta de manera parecida a una función, pero que se diferencia de estas en la semántica. Los operadores actúan sobre operandos y devuelven un resultado, mientras que las funciones reciben parámetros y también devuelven (o no) un resultado.

Un operador como  typeof  es un operador unario (solo recibe un operando).

Un operador binario requiere dos operandos, por ejemplo, en:



Figura 3. Operador binario de suma. Fuente: elaboración propia.

El operador es el símbolo + y sus operandos son el 4 y el 10.

Cadenas de texto

Una cadena de texto no es más que una secuencia de caracteres. En muchos lenguajes de programación, una cadena de texto no es más que un vector o un *array* de letras. Sin embargo, en JavaScript, una cadena de texto es un tipo de dato básico. Para definir una cadena de texto, simplemente tenemos que poner letras entre comillas (dobles o simples):

A close-up of a sign

Description automatically generated

Figura 4. Cadenas de texto. Fuente: elaboración propia.

A esta forma de definir cadenas de texto se la llama definición literal. Estrictamente, para crear una cadena de texto, deberíamos usar una función constructora junto con la cláusula new. Ejemplo:



Figura 5. Construcción de cadenas con new. Fuente: elaboración propia.

Sin embargo, como este método es algo farragoso, JavaScript permite crear cadenas de texto de manera literal, es decir, simplemente poniéndolas entre comillas.

Las cadenas de texto plantilla o *template literals* son un tipo especial de cadenas de texto. Este tipo de cadenas hace uso de las comillas invertidas. Además, los *template literals* permiten, entre otras cosas, la intercalación de código JavaScript con cadenas de texto. Veamos algunos ejemplos:

A close-up of a sign

Description automatically generated

Figura 6. *Template* literal. Fuente: elaboración propia.

A diferencia de otros lenguajes de programación, los tipos de datos primitivos tienen métodos. Los métodos son conjuntos de instrucciones (funciones) que se aplican sobre el dato.

Otros métodos interesantes son los siguientes (la funcionalidad perseguida por cada uno es bastante directa a partir de su nombre):

A white text with blue text

Description automatically generated with medium confidence

Figura 7. Métodos sobre cadenas. Fuente: elaboración propia.

Números

Los números en JavaScript pueden ser enteros (4, 1, 0, -98, etc.) o en coma flotante (1.21). Al igual que las cadenas de texto, los números también cuentan con métodos para obtener otros resultados derivados.

La diferencia es que, en vez de usar un único punto, tenemos que usar dos para esquivar el punto de la notación decimal:

A black rectangle with blue text

Description automatically generated

Figura 8. Notación exponencial 1. Fuente: elaboración propia.

Aunque también es posible usar paréntesis y recuperar la notación de punto habitual:



Figura 9. Notación exponencial 2. Fuente: elaboración propia.

Cuando un número no es computable, JavaScript, como hacen otros lenguajes, cuenta con la palabra reservada NaN (*not a number*). Por ejemplo, no es posible computar la multiplicación de un número por una cadena de texto y esta operación devolvería NaN. Ejemplo:



Figura 10. NaN. Fuente: elaboración propia.

Es posible convertir cadenas de texto en números con la función Number():

A black and white rectangle with a white background

Description automatically generated

Figura 11. Función Number. Fuente: elaboración propia.

De igual manera podemos llevar a cabo la operación inversa:



Figura 12. Numero a *string*. Fuente: elaboración propia.

*Booleanos*

Los valores *booleanos* solo pueden ser verdaderos o falsos (true o false). Además, las siguientes expresiones se consideran falsas (lo mismo que *false*) en JavaScript:



Figura 13. Valores falsos para un *booleano.* Fuente: elaboración propia.

Como regla general podemos decir que  undefined  suele ser usado por JavaScript de modo automático (cuando no hay valor) y null es utilizado por los programadores de manera manual para dejar bien claro que algo no tiene ningún dato asignado.

Por ejemplo:



Figura 14. Asignación a  null . Fuente: elaboración propia.

Con esto queremos dejar claro explícitamente que x no tiene valor alguno o que x está vacía.

Declaración de variables y constantes

Las variables se usan en programación para referirse a un dato almacenado en memoria. Las variables han de declararse antes de usarse. Con la aparición de la versión 2015 de JavaScript, también conocida como ES6, las variables deben declararse con las palabras clave  let  o  const .

* Con  let , una variable puede cambiar de valor.
* Cuando usamos  const , puede decirse que en este caso se trata de una \*constante\* en lugar de una variable. Su valor no cambiará y será inmutable.

También, al declarar una variable como constante debe quedar «inicializada» (debe recibir un valor) en esa misma línea. Veamos varios ejemplos:

A close-up of a white background

Description automatically generated

Figura 15. Declaración de variables y constantes. Fuente: elaboración propia.

La afirmación anterior tiene una excepción: los objetos. Es decir, aunque una variable se declare como const , si hace referencia a un objeto, sí que es posible mutar ese objeto. Ejemplo:

A close-up of a computer code

Description automatically generated

Figura 16. Modificando objetos aún con  const . Fuente: elaboración propia.

Antiguamente, en lugar de  let  y  const  solo existía la palabra var para identificar algo como una variable. A efectos prácticos, var se parece mucho a  let . Entonces, ¿por qué dos términos para la misma funcionalidad? Muy sencillo. Cuando una variable se declara con var, no respeta el ámbito.

Y, ¿qué es un ámbito? El ámbito es un término muy relevante, es casi cualquier entorno de programación. Se refiere a dónde es posible hacer referencia a una variable (o constante). Empleando un término matemático, se trata del dominio de definición de un valor.

Existen siempre dos tipos de ámbitos principales en cualquier programa JavaScript: global y local.

Ámbito global

Indistintamente de la variable declarada fuera de cualquier bloque se dice que pertenece al ámbito global y es accesible desde cualquier lugar de nuestro programa. El ámbito global es único (para un programa). Parece, de entrada, algo estupendo, pero no siempre es la mejor idea. Lo ideal es solo declarar variables lo más cerca posible (en términos de ámbito) de dónde van a ser utilizadas. Existen muchas razones para esta práctica: desde la limpieza y legibilidad del código a la seguridad.

Ámbito local

Lo primero que tenemos que decir del ámbito local es que, en realidad, existen tantos ámbitos locales como bloques hayamos definido. En cada bloque tenemos un ámbito local. Por descontado, cada bloque define su ámbito local y una variable definida ahí solo es visible dentro del bloque. Si el nombre de una variable se declara fuera y dentro de un bloque, el valor que cuenta para el bloque es el definido en este último.

Si no se vuelve a declarar en un bloque interno, desde ese bloque interno se puede tener acceso a una variable declarada en un bloque más externo.

Un ejemplo de lo anterior:

A computer code with text

Description automatically generated with medium confidence

Figura 17. Ámbitos local y global. Fuente: elaboración propia.

Cuando asignamos un valor a una variable, esta asignación puede ser directa o indirecta, también conocida como por referencia. Por defecto, si asignamos un valor primitivo, la asignación es directa, es decir, al ejecutar la instrucción.



Figura 18. Asignación por valor. Fuente: elaboración propia.

Es como si la variable a «albergara» dentro de sí el valor 10 (lo contiene dentro de sí).

Sin embargo, con objetos esta asignación es por referencia, de manera que varias variables pueden referirse al mismo objeto.

Por ejemplo, en este código:

A black and white rectangle with black lines

Description automatically generated

Figura 19. Asignación por referencia. Fuente: elaboración propia.

Las variables x e y apuntan al mismo objeto: { a: 1, b: 2 }, de manera que es posible cambiarlo (alterar sus contenidos) tanto desde x como desde y.

A black rectangle with a white background

Description automatically generated

Figura 20. Acceso a las propiedades de un objeto. Fuente: elaboración propia.

Y ahora, el objeto original se ha transformado a { a: 4, b: 9 }. En el caso de los tipos de datos básicos, una variable contiene un valor.

Por lo que si hacemos algo parecido al ejemplo anterior:

A black rectangle with a white background

Description automatically generated

Figura 21. Asignación por valor implícito. Fuente: elaboración propia.

Ahora x e y contienen una copia distinta y personal de la cadena de texto «hola», por lo que si ahora modificamos una de ellas (x, por ejemplo), la otra permanece inalterada como si nada hubiera pasado.

A black and white rectangle with a white background

Description automatically generated

Figura 22. Comprobando el efecto de las asignaciones por valor. Fuente: elaboración propia.

Expresiones básicas

Haciendo uso de todos los elementos que se han presentado hasta ahora, hemos construido algunas expresiones o sentencias en lenguaje JavaScript. Un tipo de expresión muy común son las comparaciones.Este tipo de sentencias constan de un operador y de dos operandos.

Para realizar comparaciones de igualdad, en otros lenguajes se suele utilizar el operador de igualdad (==),que en JavaScript devolverá true si los operandos son iguales, pero no se realiza ninguna comprobación de tipo. Para una comparación más exhaustiva que tenga en cuenta el tipo de los operandos, se recomienda utilizar el operador de igualdad estricta (===) que, además de tener en cuenta el valor, tiene en cuenta el tipo. En JavaScript podemos encontrar además de estos, otros operadores muy similares a los que hay en cualquier lenguaje de programación.

Algunos ejemplos:

A close-up of a text

Description automatically generated

Figura 23. Operadores de comparación. Fuente: elaboración propia.

Además, también es posible combinar sentencias como las anteriores mediante operadores lógicos como pueden ser || (OR lógico) y && (AND lógico). El comportamiento de estos operadores es idéntico al que podemos observar en otros lenguajes de programación.

A white rectangular object with blue text

Description automatically generated

Figura 24. Usando varios operadores lógicos. Fuente: elaboración propia.

2.5. Estructuras de control en JavaScript

Muy a menudo, es necesario que una parte del código JavaScript escrito se ejecute cuando una o varias condiciones se cumplan y que no se ejecute en caso contrario. Es decir, tendremos código que se ejecutará de forma condicional. Para poder realizar este tipo de sentencias, JavaScript dispone de diferentes estructuras de control del flujo del código.

La cláusula

If

La sentencia más sencilla y conocida es la que comienza por la cláusula  if , que seguida de una condición lógica (que se evaluará como true o false) define un bloque de código que se ejecutará únicamente si la condición es satisfecha. Un ejemplo sería:

A close-up of a white rectangle

Description automatically generated

Figura 25. Estructura  if . Fuente: elaboración propia.

La consola solo mostrará la frase si la variable edad tiene un valor estrictamente menor que 78.

Junto a la cláusula  if  tenemos otras dos que suelen acompañarla casi siempre. Son  else if  y  else . Si estamos barajando varias opciones en nuestro programa, podemos hacer uso de  else if  para tratar los diferentes escenarios. Por último, si ninguna de las condiciones anteriores se cumplió, podemos hacer uso de else para indicar las acciones a realizar en el caso de que ninguna de las condiciones anteriores se diese. Observemos un ejemplo:

A white background with blue and green text

Description automatically generated

Figura 26. Estructura  if-else . Fuente: elaboración propia.

Al igual que en otros lenguajes de programación, en JavaScript también podemos hacer uso del operador ternario para comprobar una condición y actuar en consecuencia de su cumplimiento (o no) en una única línea.

Su sintaxis, al igual que en otros lenguajes, es la siguiente:

A white rectangle with black text

Description automatically generated

Figura 27. Operador ternario. Fuente: elaboración propia.

Un ejemplo sería:

A white rectangle with blue text

Description automatically generated

Figura 28. Código del operador ternario. Fuente: elaboración propia.

La cláusula switch

La cláusula  switch  nos permite construir disyuntivas más complejas. En realidad, es una copia de la estructura  if-then-else , pero resulta más adecuada cuando el número de casos para tener en cuenta es excesivo.

Te recomendamos que consultes el recurso de A fondo, *Cláusula switch*, para obtener más información sobre esta cláusula.

2.6. Estructuras iterativas en JavaScript

Una estructura iterativa nos permite construir bucles en los cuáles es posible repetir una o varias líneas de código. Estudiaremos por tanto los bucles for y while.

Bucles while

Los bucles while ejecutan las instrucciones que haya dentro de un bloque mientras la condición enunciada se cumpla. Un ejemplo:

A white background with blue text

Description automatically generated

Figura 29. Bucle while. Fuente: elaboración propia.

Si ejecutamos este código en la consola de JavaScript, veremos que el resultado que se imprime por pantalla es el siguiente:

A white rectangle with black text

Description automatically generated

Figura 30. Ejecución del bucle while. Fuente: elaboración propia.

Nótese que se ha hecho uso de *template literals* para intercalar cadenas de texto con el valor de las variables con las que estamos trabajando. Es muy importante que el código que se encuentre dentro del bloque del while pueda dar lugar a un cambio en la condición que se está evaluando. De no ser así, se podría formar un bucle infinito, algo que casi con toda seguridad será un comportamiento no deseado.

Existe una variación del bucle while llamada do while. La diferencia entre ambas es que la versión do while se ejecutará siempre una vez como mínimo, ya que la condición del while se comprobará al final del bloque, mientras que esto no ocurre con la versión que acabamos de ver. Un ejemplo sería el siguiente:

A white background with blue text

Description automatically generated

Figura 31. Bucle do-while. Fuente: elaboración propia.

Si ejecutamos este código en la consola de JavaScript, veremos que el resultado que se imprime por pantalla es el siguiente:

A white rectangle with black text

Description automatically generated

Figura 32. Ejecución del bucle do-while. Fuente: elaboración propia.

Bucles for

Se trata de otra estructura diferente para poder realizar bucles. Su sintaxis es bastante distinta a la de la opción anterior:

A white rectangular sign with black text

Description automatically generated

Figura 33. Bucle for. Fuente: elaboración propia.

Un bucle for tiene cuatro partes muy diferenciadas:

A blue and white text on a blue background

Description automatically generated

Tabla 3. Partes del bucle for. Fuente: elaboración propia.

Al igual que ocurría con los bucles while, es necesario definir bien la condición, el valor inicial de la variable de estado y cómo va a modificarse su valor en cada iteración para evitar dar lugar a bucles infinitos no deseados.

Veamos un ejemplo de este tipo de bucles transcribiendo el ejemplo de la Figura 29:

A white background with blue and black text

Description automatically generated

Figura 34. Bucle for. Fuente: elaboración propia.

Si ejecutamos este código en la consola de JavaScript, veremos que el resultado que se imprime es idéntico al que recibíamos en el caso anterior.

Los bucles for son muy usados para recorrer estructuras de datos, como los *arrays* o las listas. Ahondaremos más sobre este aspecto en la próxima sección.

Existe una variante de este bucle llamado forEach. Realmente es un método que se ejecuta sobre los *arrays*. El método forEach ejecuta una función para cada uno de los elementos del *array*.

Te recomendamos leer la documentación sobre este método en el recurso de A Fondo, *Método forEach.*

2.7. Estructuras de datos en JavaScript

En esta sección estudiaremos diferentes estructuras de datos muy usadas a la hora de trabajar con JavaScript:

* Vectores o *arrays*.
* Conjuntos o *sets.*
* Mapas.

Lo primero que debemos conocer sobre estos tres tipos de estructuras es que son objetos. En JavaScript existen diferentes tipos de datos básicos. Pues bien, todo aquello que no es un dato básico es un objeto. La diferencia principal entre los objetos y los datos básicos es la forma de almacenarlos y manejarlos.

Los datos básicos se manejan como valores independientes (varias variables tienen una copia personal de un mismo dato), pero los objetos se manejan por referencia de manera que distintas variables pueden apuntar a un mismo pedazo de información.

Una vez dicho esto, pasamos a detallar cada una de las estructuras indicadas anteriormente.

Vectores o

*arrays*

Un vector (en adelante, *array*) es un conjunto de valores indexado. Esto quiere decir que cada elemento del *array* tiene una posición concreta, definida por un índice y que, salvo que se produzcan modificaciones, se mantendrá durante toda su vida útil.

Podemos crear un *array* de diversas maneras, dependiendo de si queremos inicializarlo vacío o con datos.

A white rectangular sign with black text

Description automatically generated

Figura 35. Creación de *arrays*. Fuente: elaboración propia.

Todos los elementos de un *array* están indexados y podemos utilizar el índice asociado a cada elemento para obtenerlo. Por ejemplo, en el *array* creado antes, podemos obtener el valor del primer elemento de este, accediendo al elemento cero de la estructura (recordemos que el cero se refiere al primer elemento):

A white rectangular sign with blue text

Description automatically generated

Figura 36. Acceso a elementos de un *array*. Fuente: elaboración propia.

Utilizando la misma notación es posible añadir un nuevo elemento al *array*, haciendo uso de nuevo del operador de asignación (=):



Figura 37. Operador de asignación. Fuente: elaboración propia.

Los *arrays* pueden contener cualquier tipo de datos en su interior, incluso otros *arrays*, lo que daría lugar a *arrays* multidimensionales o matrices. Sin embargo, aquellas variables declaradas con const no podrían ver modificado su valor. Entonces, ¿por qué hemos podido añadir un nuevo elemento al *array*, modificándolo?

La razón es muy sencilla. No hemos modificado la referencia de memoria a la que la variable está apuntando, sino que hemos modificado el contenido de memoria asociado a dicha referencia. No es, por tanto, posible redefinir la referencia a la que la variable apunta, como podemos ver en este ejemplo:

A white background with blue text

Description automatically generated

Figura 38. *Arrays* y referencias. Fuente: elaboración propia.

A pesar de declarar un objeto como constante, es posible modificar sus contenidos. El contenido de un *array* declarado como const sigue siendo mutable.

Conjuntos

Los conjuntos fueron introducidos en la versión 6 de JavaScript. Un conjunto o set es una estructura que representa una colección de valores únicos (no se pueden repetir).

Los sets permiten manejar conjuntos de datos y asegurarnos de manera automática que no contiene duplicados. Para crear un set vacío recurrimos a una función constructora y al operador new:



Figura 39. Declaración de conjunto. Fuente: elaboración propia.

Para añadir un elemento a un set utilizamos el método .add(elemento).



Figura 40. Añadir elementos a un conjunto 1. Fuente: elaboración propia.

Este método devuelve nuevamente el set con el nuevo elemento añadido, por lo que es posible encadenar nuevos.add(). Ejemplo:



Figura 41. Añadir elementos a un conjunto 2. Fuente: elaboración propia.

Fíjate que, a pesar de haber declarado la variable conjunta como constante, podemos modificar el set. Esto es así porque los sets son también objetos y estos sí son modificables a pesar de que hayan sido introducidos mediante la cláusula const.

¿Qué pasa si intentamos añadir un elemento que ya existe en el set? Si intentamos añadir un elemento que ya estaba presente en el set, la orden es ignorada.

Podemos añadir varios elementos de una vez a un set con un *array*:



Figura 42 Creación de un conjunto a partir de un *array*. Fuente: elaboración propia.

Si dicho *array* contuviese elementos ya repetidos, estos son automáticamente filtrados para eliminar duplicidades. Ejemplo:



Figura 43. Los conjuntos no contienen duplicados. Fuente: elaboración propia.

La afirmación anterior tiene una excepción: los objetos. Sí es posible añadir a un set dos objetos que aparentemente son iguales. Esto es así porque los objetos se manejan por referencia y, aunque dos objetos se parezcan, están en espacios de memoria distintos y, por ende, son tratados como entes independientes por parte de JavaScript. Si por ejemplo ejecutamos:



Figura 44. Los conjuntos no contienen referencias duplicadas 1. Fuente: elaboración propia.

Se trataría de un set con dos elementos iguales (dos objetos vacíos, en este caso), pero que son independientes desde el punto de vista lógico. Veamos otro ejemplo:



Figura 45. Los conjuntos no contienen referencias duplicadas 2. Fuente: elaboración propia.

Aunque los dos *arrays* [1,2] y [1,2] contienen la misma información a nuestros ojos, se trata de objetos distintos (a los ojos de JavaScript). Recordemos que los *arrays* en JavaScript también son objetos.

Como otros contenedores de información en JavaScript, los sets también poseen métodos interesantes para obtener información de estos. Algunos ejemplos son:

La propiedad .size devuelve el tamaño del set:



Figura 46. Método size. Fuente: elaboración propia.

El método .has(elemento) comprueba si un elemento existe dentro de un set:



Figura 47. Método has. Fuente: Elaboración propia

El método .delete(elemento) borra valores de un set. Devuelve true o false según el éxito de la operación.



Figura 48. Método delete. Fuente: elaboración propia.

El método .clear() vacía el conjunto completamente y nos devuelve un conjunto vacío:



Figura 49. Método clear. Fuente: elaboración propia.

En el siguiente vídeo, *Trabajando con estructuras de datos en JavaScript*, podrás afianzar algunos de los conceptos que hemos tratado en esta sección con más ejemplos:

Mapas

Los mapas no son más que listas de parejas clave-valor, donde la clave y el valor pueden ser cualquier tipo de dato. En otros lenguajes de programación, son conocidos también como mapas *hash,* tablas o diccionarios. En JavaScript, los mapas también son objetos.

Las claves de un mapa deben ser únicas (algo que ocurre también en los sets). No cuentan con una forma literal para su definición, por lo que para crearlos es necesario usar una función constructora.



Figura 50. Construcción de un mapa. Fuente: elaboración propia.

Al igual que ocurría con los sets, los mapas tienen distintos métodos que nos permiten trabajar con ellos. Los más importantes son:

* Método set (clave, valor), que permite añadir una nueva pareja clave-valor al mapa.
* Método get (clave), a partir de la clave, nos devuelve su valor asociado, si existe.
* Método has (clave), que devolverá true o false en función de si el mapa contiene alguna pareja con la clave indicada.
* Método delete (clave), el cual permitirá borrar una entrada del mapa que tenga la clave especificada.
* Método clear (), que permitirá vaciar completamente el mapa.

Las siguientes líneas de código ilustran el funcionamiento de estos métodos:

A white background with blue text

Description automatically generated

Figura 51. Métodos sobre mapas. Fuente: elaboración propia.