Domine JavaScript

4ª edición

Pablo E. Fernández Casado

Ra-Ma®

edU

Domine JavaScript

4a Edición

Pablo E. Fernández Casado

Ra-Ma®

edU®
Conocimiento a su alcance

Bogotá - México, D.F.

Fernández Casado, Pablo E.

Domine JavaScript/ Pablo E. Fernández Casado / 4a. Edición--. Bogotá: Ediciones de la U, 2021
438 p. ; 24 cm
ISBN 978-958-792-301-8 e-ISBN 978-958-792-302-5
1. Programación 2.Variables 3. Tipos de datos 4. Operadores 5. Objetos 6. Funciones Tít.
621.39 ed.

Edición original publicada por © Editorial Ra-ma (España)
Edición autorizada a Ediciones de la U para Colombia

Área: Informática
Cuarta edición: Bogotá, Colombia, septiembre de 2021
ISBN. 978-958-792-301-8

Ediciones de la U es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.
Carátula: Ediciones de la U
Impresión: DGP Editores SAS
Calle 63 #70D-34, Pbx (57+1) 3203510

Impreso y hecho en Colombia
Printed and made in Colombia

A mi familia

ÍNDICE

PRÓLOGO

JavaScript es un lenguaje de programación interpretado, basado en el estándar ECMAScript (European Computer Manufacturer's Association Script). Se caracteriza por ser un lenguaje de programación orientado a eventos y basado en prototipos, dinámico y no demasiado tipado.

Sus orígenes se sitúan en 1995 y su nombre original era Mocha. Sin embargo, no tardó mucho en ser renombrado a LiveScript hasta que, finalmente, fue bautizado como JavaScript. La razón de este último cambio fue porque Sun Microsystems (propietaria de Java) compró Netscape y, como estrategia de marketing, decidió llamarlo como su "perla" más preciada. En resumen, que JavaScript no es el lenguaje script de Java.

Cabe destacar que ya, en el año 2012, todos los navegadores soportaban el estándar ECMAScript 5.1, con alguna excepción. No obstante, fue en el año 2015 cuando JavaScript alcanzó casi todo su potencial, con la llegada de ECMAScript 6.

El uso que se le da a JavaScript está, básicamente, en el lado del cliente y son los navegadores quienes lo implementan como parte de su potencial. Es por esta razón que muchas sentencias, métodos y eventos no funcionan igual, dependiendo de en qué navegador estemos trabajando y puede que, incluso, algunas funcionalidades ni si siquiera, funcionen. Por suerte parece que, no tardando mucho, esto va a cambiar.

También existe, como muchos sabrán, un JavaScript que trabaja en el lado del servidor, aunque su uso está más encaminado a la programación orientada a eventos, desarrollo de microservicios y diseño de aplicaciones con alta carga de computación.

En lo referente a su sintaxis, JavaScript resulta tener un cierto parecido con Java, sin embargo, fue construido basándose en la sintaxis de C.

Por último, sólo aclarar que, este libro está pensado para trabajar de la manera más compatible posible, es decir, que la mayoría de las cosas que aquí se explican, deberían funcionar en todos los navegadores, incluyendo Internet Explorer 11.

1

VARIABLES Y AMBITOS

A continuación, se realiza una visión inicial sobre las características básicas de JavaScript.

1.1 DECLARACIÓN DE VARIABLES

Como en casi todos los lenguajes de programación, los identificadores de variables sólo pueden empezar por una letra mayúscula, minúscula, guion bajo o símbolo dólar. No se permiten nombres de variables que empiecen por otros símbolos o dígitos y no admiten ningún tipo de operador lógico o matemático.

Para declarar una variable podemos recurrir a tres palabras reservadas, dependiendo de la versión de ECMAScript que tengamos disponible en el navegador.

La más frecuentemente utilizada es la palabra reservada **var,** ya que es compatible con todas las versiones de ECMAScript y es la menos restrictiva y más compatible entre navegadores, incluyendo Internet Explorer 11.

```
var fechaActual = new Date();
```

Otra de las formas de realizar la declaración de una variable podría ser a través de la palabra reservada **let**, sin embargo, este tipo de declaración sólo es compatible con los navegadores más modernos que contemplan ECMAScript 2015 (ES6), lo que no incluye a Internet Explorer 11.

```
let fechaActual = new Date();
```

Mientras que el uso de **var** permite la redefinición o sobreescritura de variables, este tipo de declaración no. Una vez que se haya realizado la primera

definición, no se permitirá que el nombre de la variable pueda volver a ser definida dentro del mismo contexto o bloque, no obstante, esta limitación puede ayudar a evitar errores debidos a la sobreescritura accidental.

La tercera forma que disponemos para realizar la declaración de variables es a través de la palabra reservada **const**, sin embargo, al igual que pasa con la palabra reservada **let**, este tipo de declaración sólo es compatible con los navegadores más modernos que contemplan ECMAScript 2015 (ES6), lo que no incluye a Internet Explorer 11.

```
const fechaActual = new Date();
```

En este caso, la principal diferencia es que, mientras que **var** y **let** permiten la reasignación de valores, **const** define el identificador como una declaración de constante y prohíbe su reasignación.

1.2 ÁMBITO DE LAS VARIABLES

El ámbito de las variables es un tema, a veces, complicado. Sea cual sea el lenguaje de programación siempre se producen confusiones sobre su origen y cómo afectan las variables, por ello, empezaremos por lo básico.

El ámbito de una variable, también conocido como scope, es el bloque o parte del código donde, esa variable, se define y está accesible. En JavaScript, los ámbitos sólo pueden ser dos: global y local.

Aunque es un poco más complejo, podríamos decir que, el ámbito local es la parte que está definida dentro de un bloque delimitado por llaves y, el ámbito global es la parte que está fuera de dichas llaves.

Dicho de otra manera. Cuando se accede o utiliza un identificador de variable, primeramente, se busca en la parte del código o bloque que está delimitado por las llaves (el ámbito local). Más tarde, si se no encuentra la declaración de esa variable en ese ámbito, se busca en los ámbitos locales de sus bloques padre hasta llegar al ámbito global, que, como ya veremos, es el objeto global (window).

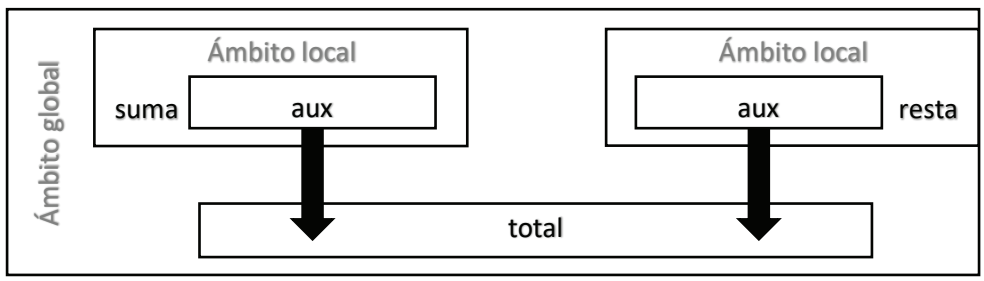
Imaginemos una situación sencilla en la que tenemos unas funciones que operan con una variable externa a las funciones.

```
var total = 0;

function suma(a, b) {
    var aux = a + b;
    total = aux;
```

```
}

function resta(a, b) {
    var aux = a - b;
    total = aux;
}
```

Si observamos el código anterior, podremos ver que, la variable **aux**, se ha definido dentro de los ámbitos locales (los delimitados por las llaves de las funciones) y que, la variable **total** se ha definido en el ámbito global, lo que permite que pueda ser accedida y actualizada desde las funciones suma y resta (las cuales generan un ámbito local dependiente del ámbito global, que es el ámbito padre).

Gráficamente, podríamos decir que es como una pila que va añadiendo elementos y que se caracteriza porque, los elementos que están definidos dentro de un cuadro o bloque, pueden acceder a los elementos que los engloban. Es decir, algo como:



Como la variable **total** ha sido definida en el ámbito global, las funciones suma y resta pueden acceder a la variable y actualizarla. Por tanto, una forma simple de definir el ámbito global es "aquel que puede ser accedido desde cualquier punto del script o programa".

Como las variables definidas como **aux** están declaradas en bloques delimitados por llaves, su ámbito será local y no podrán ser accedidas o utilizadas desde fuera de su propio ámbito.

En resumen, la declaración y uso de variables se establece de forma jerarquizada en dirección ascendente, es decir, lo que no esté en el nivel actual, será buscado en los niveles superiores y, si no lo encuentra, es cuando se producirá un error de referenciación.

2

TIPOS DE DATOS

JavaScript dispone de dos tipos de datos, primitivos y objeto.

Los tipos de datos primitivos son los que representan un único dato, son inmutables y no tienen métodos.

Los tipos de datos objeto son los que representan una o varias colecciones de datos primitivos y permiten su manipulación a través de propiedades y/o métodos.

En JavaScript, como se verá más adelante, todo son objetos.

2.1 TIPO STRING

El objeto **String** se utiliza para el tratamiento de cadenas de texto. Este tipo, además, provee de un constructor asociado que permite realizar conversiones explícitas.

```
String(4);      // Devuelve “4”
String(0.7);    // Devuelve “0.7”
String(2,4);    // Devuelve “2”
String(true);   // Devuelve “true”
String(String(5)); // Devuelve “5”
String(var);    // Devuelve error de sintaxis
```

2.1.1 Propiedades

El objeto String tiene, esencialmente, tres propiedades:

Propiedad	Descripción
constructor	Devuelve la función constructora nativa.
length	Devuelve la longitud de la cadena
prototype	Permite añadir nuevas propiedades y métodos al objeto.

2.1.2 Métodos

Los valores primitivos también son considerados objetos. Por esta razón, cualquier literal como "Esto es una cadena" puede ejecutar un método o una propiedad.

El número de métodos disponibles para este objeto es elevado, por lo que, a continuación, se muestran los más utilizados:

Método charAt

Devuelve el carácter correspondiente a la posición proporcionada por parámetro. Por defecto, la posición es 0.

```
"Hola".charAt(0);            // devuelve "H"
```

Método charCodeAt

Devuelve el código Unicode del carácter que corresponda a la posición proporcionada por parámetro. Por defecto, la posición es 0.

```
"Hola".charCodeAt(0);        // devuelve 72
```

Método concat

Devuelve otro String que tiene, como resultado, la unión entre de todas las cadenas proporcionadas por parámetro y la actual. El separador de parámetros es el símbolo coma.

```
"Hola".concat(" ", "mundo");        // devuelve "Hola mundo"
```

Método endsWith

Devuelve un booleano que indica si la cadena termina con la subcadena proporcionada por parámetro.

```
"Hola mundo".endsWith("do");        // devuelve true
```

Método indexOf

Devuelve la primera posición en la que aparezca la subcadena proporcionada por parámetro. Si el resultado de la búsqueda fue infructuoso, el resultado será -1.

Tiene un segundo parámetro opcional que indica desde qué posición se debe empezar a buscar y que, por defecto, es 0.

NOTA

Este método es sensible a mayúsculas y minúsculas.

```
"Hola mundo".indexOf("o", 0);       // devuelve 1
```

Método lastIndexOf

Devuelve la última posición en la que aparezca la subcadena proporcionada por parámetro. Si el resultado de la búsqueda fue infructuoso, el resultado será -1.

El método de búsqueda es al revés que el método **indexOf**, es decir, que busca desde el final hasta el principio.

Tiene un segundo parámetro opcional que indica desde qué posición se debe empezar a buscar y que, por defecto, es la longitud de la cadena.

NOTA

Este método es sensible a mayúsculas y minúsculas.

```
"Hola mundo".lastIndexOf("o");      // devuelve 9
```

Método match

Permite encontrar coincidencias en una cadena mediante expresiones regulares.

> **NOTA**
>
> Las expresiones regulares se verán en otro capítulo más adelante.

```
"Hola mundo".match(/ho/i);
// devuelve un array con:
["Ho", index: 0, input: "Hola mundo", groups: undefined]
```

Método normalize

Permite convertir a la forma normal Unicode una cadena pasada como argumento.

La forma normal a elegir tiene cuatro posibles valores:

- NFC: Forma de Normalización de Composición Canónica.
- NFD: Forma de Normalización de Descomposición Canónica.
- NFKC: Forma de Normalización de Composición de Compatibilidad.
- NFKD: Forma de Normalización de Descomposición de Compatibilidad.

A parte de lo evidente, este método es útil, por ejemplo, para reemplazar acentos.

Por defecto, su valor es NFC.

```
"Ambigüedad inherente".normalize('NFD').replace(/[\u0300-\u036f]/g, "");
// devuelve "Ambiguedad inherente"
```

> **NOTA**
>
> Este método no funciona en Internet Explorer 11.

Método repeat

Devuelve la concatenación de la cadena repetida las veces que se indique por parámetro.

```
"Hola. ".repeat(2);             // devuelve "Hola. Hola.
```

Método replace

Permite realizar reemplazos en una cadena a través de otra cadena o una expresión regular.

> **NOTA**
>
> Las expresiones regulares se verán en otro capítulo más adelante.

```
"Hola mundo".replace(" mundo", "");    // devuelve "Hola"
"Palabra".replace(/a/ig, "0");     // devuelve "P010br0"
```

Método search

Devuelve la posición de la primera aparición de la cadena proporcionada por parámetro.

Aunque este método acepta Strings como parámetro. Lo que utiliza son expresiones regulares. Por esta razón, si se introduce un String, será transformado de forma automática a una expresión regular.

> **NOTA**
>
> Las expresiones regulares se verán en otro capítulo más adelante.

```
"Hola mundo".search("mundo");       // devuelve 5
```

Método slice

Devuelve el fragmento de la cadena que esté comprendido entre las posiciones proporcionadas por parámetro.

ⓘ NOTA

Este método es muy similar al método substring, sin embargo, los resultados pueden ser muy diferentes.

```
"Hola mundo".slice(0, 4);          // devuelve "Hola"
```

Método split

Devuelve un array con todos los fragmentos de cadena que resulten de dividir la cadena origen a través otra cadena o expresión regular proporcionada por parámetro.

```
"Hola mundo".split(" ");        // devuelve ["Hola", "mundo"]
```

Método startsWith

Devuelve un booleano que indica si la cadena empieza por el valor proporcionado por parámetro.

Acepta un segundo parámetro que indica dónde se debe empezar a realizar la búsqueda. Por defecto es 0.

```
"Hola mundo".startsWith("mundo");      // devuelve false
"Hola mundo".startsWith("mundo", 5);   // devuelve true
```

Método substr

Devuelve el fragmento de cadena que empieza por la posición indicada en el primer parámetro y cuya longitud es el valor proporcionado por el segundo.

```
"Hola mundo".substr(1,6);       // devuelve "ola mu"
```

Método substring

Devuelve el fragmento de cadena que se encuentre entre las posiciones proporcionadas por los parámetros.

> **NOTA**
>
> Este método es muy similar al método substring, sin embargo, los resultados pueden ser muy diferentes.

```
"Hola mundo".substring(1,6);    // devuelve "ola m"
```

Método toLowerCase

Devuelve la cadena convertida a minúsculas.

```
"Hola mundo".toLowerCase("");   // devuelve "hola mundo"
```

Método toUpperCase

Devuelve la cadena convertida a mayúsculas.

```
"Hola mundo". toUpperCase("");  // devuelve "HOLA MUNDO"
```

Método trim

Devuelve la cadena sin los espacios en blanco que puedan existir en los extremos.

```
"Hola mundo      ".trim();      // devuelve "Hola mundo"
"     Hola   mundo  ".trim();   // devuelve "Hola   mundo"
```

2.1.3 Conversión de Strings

Además de poder realizar conversiones a través de su constructor, el tipo **String** también permite hacer conversiones mediante otras funciones como, por ejemplo, **parseInt** y **parseFloat**, las cuales permiten hacer transformaciones de tipo Strings a tipo número.

```
parseInt("4")       // Devuelve 4
parseInt("hola")    // Devuelve NaN (no es un número)
parseInt("21 calles")  // Devuelve 21
parseInt("1e3")     // Devuelve 1
parseFloat("1.5")      // Devuelve 1.5
parseFloat("1,5")      // Devuelve 1

String(new Date())      // Devuelve la fecha actual en formato GMT
```

2.1.4 Formateado de Strings

JavaScript dispone de varias opciones para formatear texto, desde construcciones a través de literales de cadena, hasta secuencias escapadas en hexadecimal o Unicode.

```
/* Literales de cadena */
‘Esto es un literal de cadena’
“Esto es otro literal de cadena”

/* Secuencia escapada en hexadecimal */
“\x41”      // Devuelve “A”

/* Secuencia escapada en Unicode */
“\u0041”    // Devuelve “A”
```

Como se puede apreciar, los literales de cadena no tienen nada de especial, no obstante, el escapado puede ser interesante en varios ámbitos como, por ejemplo, en situaciones dónde se necesita mostrar símbolos especiales o iconos.

En ECMAScript 6 existe una forma adicional de escapar texto, mediante el uso de puntos de escape. Esta anotación permite que, cualquier carácter, pueda ser escapado utilizando valores hexadecimales comprendidos entre 0x000000 y 0x10FFFF, o lo que es lo mismo, entre 0 y 1048576. Además, resulta interesante porque evita tener que escribir códigos Unicode dobles.

```
console.log(‘\u{1F440}’, “\uD83D\uDC40”);
```

La línea de código anterior, muestra el icono de ojos Emoji de Unicode (□). La anotación de la izquierda está representada con codificación HTML Entity hexadecimal. La anotación de la derecha está representada con codificación C/C++/Java.

Todos los ejemplos anteriores representan valores en una única línea, sin embargo, también existe la posibilidad de trabajar en modo multilínea.

El modo multilínea se puede realizar de dos formas, con ayuda del símbolo de barra invertida, o a través de literales de plantilla.

```
/* Literales de cadena multilínea (sólo con ES5.1 e inferiores) */
console.log(‘Nombre: Pablo\n\
Apellidos: Fernández’);

/* Literales de plantilla (sólo con ES6 y superiores) */
```