8

INTRODUCCIÓN A JAVASCRIPT

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 ybasado 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 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 objetos, 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.

8.1 VARIABLES Y ÁMBITOS

Cuando se trabaja con JavaScript pueden surgir muchas dudas y, por ello, hay que tener claro lo que es una variable y cuál es su ámbito.

8.1.1 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.

Hasta no hace tanto, la más frecuentemente utilizada es la palabra reservada VAR, ya que era la más compatible con todas las versiones de ECMAScript y la menos restrictiva y más compatible entre navegadores, incluyendo Internet Explorer 11.

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

Sin embargo, hoy en día, la forma más extendida para realizar la declaración de variables es a través de la palabra reservada LET.

```
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.

Existe forma disponible para realizar la declaración de variables y es a través de la palabra reservada CONST.

```
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.

8.1.2 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, cuando se accede o utiliza una variable, primeramente, se busca en la parte del código o bloque que está delimitado por las llaves (el ámbito local). Más tarde, si 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.

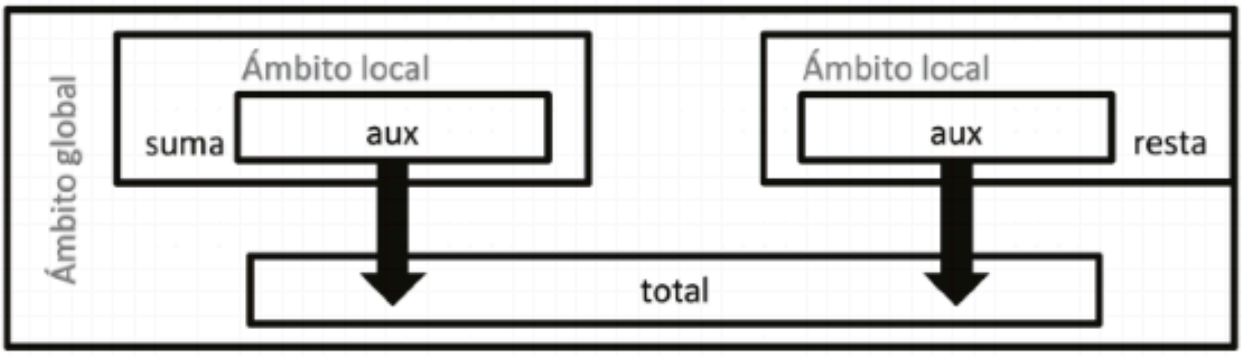
```
let total = 0;

function suma(a, b) {
let aux = a + b;
total = aux;
}

function resta(a, b) {
let 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 los 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.

8.2 TIPOS DE DATOS

JavaScript dispone de dos tipos de datos que son identificados como primitivos y objetos. Sin embargo, en JavaScript, como se verá más adelante, todo puede ser considerado como un objeto, incluyendo los valores primitivos.

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.

Como norma se puede afirmar que todo tipo de datos tiene, entre sus propiedades, la propiedad **CONSTRUCTOR** que devuelve la función constructora nativa y, dependiendo de cada caso, la propiedad **LENGTH**, que devuelve la longitud de la cadena o el objeto y la propiedad **PROTOTYPE**, que permite u ofrece la posibilidad de añadir nuevas propiedades y métodos a los objetos.

A su vez, también es importante destacar que JavaScript presenta una característica denominada autoconversión de tipos que hace que, si los operandos no son del mismo tipo (esto es, no son todos numéricos, booleanos, de cadena, ...), el sistema realizará una conversión de tipos automática antes de realizar la operación.

Como ejemplo de autoconversión de tipos, si se suma un número con una cadena, el resultado será una cadena. Si se suma un booleano con una cadena, el resultado será numérico, pero, si se suma un booleano con un valor numérico, el resultado será de tipo numérico (esto es porque, en JavaScript, **true** equivale a 1 y **false** equivale a 0).

8.2.1 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(true); // Devuelve "true"
String(String(5)); // Devuelve "5"
String(var); // Devuelve error de sintaxis
```

8.2.1.1 PROPIEDADES

Además de las propiedades **CONSTRUCTOR** y **PROTOTYPE**, el objeto STRING presenta, esencialmente, las siguientes propiedades:

Propiedad	Descripción y ejemplo
length	Devuelve la longitud de la cadena, en unidades de código UTF-16. `"Hola".length; // devuelve 4`

8.2.1.2 MÉTODOS

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

Método	Descripción y ejemplo
charAt	Devuelve el carácter correspondiente a la posición proporcionada por parámetro. Por defecto, la posición es 0. `"Hola".charAt(0); // devuelve "H"`
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`
endsWith	Devuelve un booleano que indica si la cadena termina con la subcadena proporcionada por parámetro. `"Hola".endsWith("do"); // devuelve false`
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. Es sensible a mayúsculas y minúsculas. `"aa".indexOf("A",0); // devuelve -1`
lastIndexOf	Este método presenta los mismos argumentos al anterior, con la salvedad de que busca desde el final hasta el principio. `"aa".lastIndexOf("a"); // devuelve 1`

match	Permite encontrar coincidencias en una cadena mediante expresiones regulares, las cuales se verán más adelante.
	`"Hola m".match(/m/i);`
	`// devuelve un objeto array con:`
	`['m', index: 5, input: 'Hola m', groups: undefined]`

replace	Permite realizar reemplazos en una cadena a través de otra cadena o una expresión regular, las cuales se verán más adelante.
	`"Palabra".replace("a", ""); // devuelve "Plabra"`
	`"Palabra".replace(/a/ig, "4"); // devuelve "P4l4br4"`

search	Devuelve la posición de la primera aparición de la cadena proporcionada por parámetro. Aunque este método acepta cadenas como parámetro, si esto se produce, será transformado de forma automática a una expresión regular, las cuales se verán más adelante. `"Hola mundo".search("mundo"); // devuelve 5`
slice	Devuelve el fragmento de la cadena que esté comprendido entre las posiciones proporcionadas por parámetro. Aunque puede resultar similar al método `SUBSTRING`, sus resultados pueden ser muy diferentes. `"Hola mundo".slice(0, 4); // devuelve "Hola"`
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"]`
starsWith	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", 5); // devuelve true`
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"`
substring	Devuelve el fragmento de cadena que se encuentre entre las posiciones proporcionadas por los parámetros. Aunque puede resultar similar al método `SLICE`, sus resultados pueden ser muy diferentes. `"Hola mundo".substring(1,6); // devuelve "ola m"`
toLowerCase	Devuelve la cadena convertida a minúsculas. `"hOla".toLowerCase(""); // devuelve "hola"`

toUpperCase	Devuelve la cadena convertida a mayúsculas. `"mUndo".toUpperCase(""); // devuelve "MUNDO"`

trim	Devuelve la cadena sin los espacios en blanco que puedan existir en los extremos.
	`"Hola ".trim(); // devuelve "Hola"`
	`" mundo ".trim(); // devuelve "mundo"`

8.2.1.3 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
```

8.2.1.4 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, no obstante, 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 */
console.log('Nombre: Pablo\n\
Apellidos: Fernández');

/* Literales de plantilla (sólo con ES6 y superiores) */
console.log(`Nombre: Pablo
Apellidos: Fernández`);
```

Si ejecutásemos estos fragmentos de código, comprobaremos que imprimen exactamente lo mismo, sin embargo, si ahora quisiéramos insertar una variable como parte de la expresión de cadena, en la primera forma tendríamos que "cortar" por el medio y establecer el nombre de la variable.

```
let nombre = 'Pablo';

console.log('Nombre:\t\t' + nombre + '\n\
Apellidos\t: Fernández');
```

Pero, en la segunda forma, es posible hacerlo sin tener que "cortar" por medio. Esto es gracias a lo que denominan la anotación "Syntactic Sugar", la cual se caracteriza porque el nombre de la variable va asignado entre llaves dentro del mismo literal, lo que facilita su lectura o proporciona algo más de limpieza.

```
let nombre = 'Pablo';

console.log(`Nombre:\t\t${nombre}
Apellidos:\tFernández`);

// Ambos fragmentos de código deberían mostrar algo como:
Nombre: Pablo
Apellidos: Fernández
```

8.2.2 Tipo Number

El tipo **Number** se utiliza para el tratamiento de números enteros, decimales o exponenciales. Este tipo, además, provee de un constructor asociado que puede ser utilizado para realizar una conversión explícita.

```
Number("4"); // Devolverá 4
Number("Hola") // Devolverá NaN porque no es un número
Number("21 calles") // Devolverá NaN porque no es un número
Number(1e3) // Devolverá 1000
```

```
Number(true); // Devolverá 1
Number(false); // Devolverá 0
```

8.2.2.1 PROPIEDADES

Además de las propiedades **CONSTRUCTOR** y **PROTOTYPE**, el objeto NUMBER presenta, esencialmente, las siguientes propiedades:

Propiedad	Descripción y ejemplo
EPSILON	Devuelve la diferencia entre el valor 1 y el número de punto flotante más pequeño mayor que 1. `0.2 > Number.EPSILON; // devuelve true`

MAX_SAFE_INTEGER	Devuelve el entero seguro máximo en JavaScript, que en este caso es $(2^{53} - 1)$.
	`Number.MAX_SAFE_INTEGER`
	`// Devuelve 9007199254740991`

MAX_VALUE	Devuelve el mayor valor numérico representable en JavaScript.
	`Number.MAX_VALUE`
	`// Devuelve 1.7976931348623157e+308`

MIN_SAFE_INTEGER	Devuelve el entero seguro máximo en JavaScript, que en este caso es $-(2^{53} - 1)$.
	`Number.MAX_SAFE_INTEGER`
	`// Devuelve -9007199254740991`

MIN_VALUE	Devuelve el menor valor numérico representable en JavaScript. `Number.MAX_VALUE // Devuelve 5e-324`
NaN	Devuelve una representación comparable de un valor que se identifica como Not-A-Number. `"a+2" == Number.NaN // Devuelve false`
POSITIVE_INFINITY	Devuelve una representación comparable del valor infinito positivo. `Number.NEGATIVE_INFINITY // Devuelve Infinity`
NEGATIVE_INFINITY	Devuelve una representación comparable del valor infinito positivo. `Number.NEGATIVE_INFINITY // Devuelve -Infinity`

8.2.2.2 MÉTODOS

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

Propiedad	Descripción y ejemplo

isFinite	Devuelve un booleano que indica si el valor proporcionado por parámetro es o no un valor finito. Sólo es efectivo cuando se utiliza en conversiones.
	`isFinite(200); // devuelve true`
	`isFinite("Hola"); // devuelve false`

isNaN	Devuelve un booleano que indica si el valor proporcionado por parámetro es o no un valor numérico. Sólo es efectivo cuando se utiliza en conversiones.
	`isNaN("200"); // devuelve false`
	`isNaN("hola"); // devuelve true`

toExponential	Devuelve el número proporcionado por parámetro en notación exponencial. `(2.1).toExponential(3); // devuelve "2.100e+0"`
toFixed	Devuelve el número en notación decimal con el número de decimales indicado por el parámetro. `(2.1).toFixed(3); // devuelve "2.100"`

toPrecision	Devuelve el número en notación decimal para que coincida con la longitud proporcionada por el parámetro. Si la parte entera del número es cero, será redondeado con el número de decimales que indica el parámetro. De lo contrario, será redondeado al número de decimales que resulten de restar el valor pasado y el número de dígitos de la parte entera.
	`(2.1).toPrecision(4); // devuelve "2.100"`
	`(21.1).toPrecision(4); // devuelve "21.10"`
	`(0.21).toPrecision(4); // devuelve "0.2100"`

toString	Devuelve el número en formato String. `(2.1).toString(); // devuelve "2.1"`

8.2.2.2.1 Conversión de números

Además de poder realizar conversiones a través de su constructor, el tipo **Number** permite hacer conversiones a través de métodos como **toString** para hacer, por ejemplo, transformaciones de notación numérica a Strings.

```
(2.0).toString(); // Devuelve "2"
(2).toString(); // Devuelve "2"
2.toString(); // Error de sintaxis
Number(new Date()) // Devuelve un timestamp como 1567845361045
```

8.2.2.3 FORMATEADO DE NÚMEROS

JavaScript dispone de varias opciones para formatear números, sin embargo, lo más frecuente es encontrar desarrollos a medida en vez de utilizar la potencia del lenguaje. De hecho, gracias al método toLocaleString, podemos formatear números y monedas de forma sencilla. Por ejemplo, para formatear un número es posible hacer:

```
(123456.123).toLocaleString(); // Devuelve "123.456,123"
```

Como se puede apreciar, para mostrar el valor en notación decimal no se ha proporcionado ningún parámetro, sin embargo, si queremos establecer una notación distinta, debemos configurar algunas propiedades separadas en dos argumentos.

El primer argumento del objeto toLocaleString es el código de idioma que define el idioma según el estándar BCP 47 y que, actualmente, están contemplados por la normativa RFC 5646. El segundo argumento es un JSON de opciones que especifica las diferentes propiedades que definen el formato, desde su tipo (número o moneda), hasta el número mínimo de dígitos significativos.

Propiedad	Descripción y ejemplo

`style`	Es un String que indica el formato a presentar los datos. Sus posibles valores son **`decimal`**, que es el valor por defecto e indica que se debe tratar como un número decimal, **`currency`**, que indica que se debe tratar como una divisa y **`percent`**, que indica que se debe tratar como un valor porcentual, establecido en tanto por uno.
	`let options = { style: "percent", minimumFractionDigits: 2 };`
	`(0.52).toLocaleString("es-ES", options);`
	`// Devuelve "52,00 %"`

`currency`	Esta propiedad nos permitirá configurar qué tipo de moneda deseamos utilizar en modo abreviado de texto. Esto es, **EUR**, **USD**,... Todos sus posibles valores están disponibles en dirección *https://es.iban.com/currency-codes.*

`currencyDisplay`	Si se establece el formato "currency", esta propiedad nos permitirá configurar cómo se desea presentar la notación. Sus posibles valores son **`symbol`**, para indicar que se muestre el símbolo asociado a la moneda **`code`**, para indicar que se muestre la abreviatura asociada a la moneda y **`name`**, para indicar que se muestre el texto asociado y traducido.
	`let options = { style: "currency", currency: "USD", currencyDisplay: "symbol", minimumFractionDigits: 2 };`
	`(123.12).toLocaleString("es-ES", options);`
	`// Devuelve "123,12 US$"`

useGrouping	Esta propiedad es un booleano que indica si se desea utilizar el separador de miles o no. Su valor por defecto es `true`.
	`let options = { style: "decimal",`
	`useGrouping: true,`
	`minimumFractionDigits: 2 };`
	`(12345.52).toLocaleString("es-ES", options);`
	`// Devuelve "12.345,52"`

minimumIntegerDigits	Esta propiedad es un valor entero que indica el número mínimo de dígitos que debe utilizarse en la parte entera. El rango de valores utilizable es de 1 a 21 y su valor por defecto es 1.
	`let options = { style: "decimal",`
	`useGrouping: false,`
	`minimumIntegerDigits: 5,`
	`minimumFractionDigits: 2 };`
	`(5.25).toLocaleString("es-ES", options);`
	`// Devuelve "00005,25`

minimumFractionDigits	Esta propiedad es un valor entero que indica el número mínimo de dígitos que debe utilizarse en la parte decimal. El rango de valores utilizable es de 0 a 20 siendo, su valor por defecto 0 si el formato utilizado es "decimal" o "porcentual" y 2 si el formato utilizado es "divisa", aunque este último valor puede ser diferente según la lista de códigos de moneda ISO 4217.
	`let options = { style: "decimal",`
	`useGrouping: false,`
	`minimumIntegerDigits: 3,`
	`minimumFractionDigits: 2 };`
	`(5.0).toLocaleString("es-ES", options);`
	`// Devuelve "005,00`

maximumFractionDigits	Esta propiedad es un valor entero que indica el número máximo de dígitos que debe utilizarse en la parte decimal. El rango de valores utilizable es de 0 a 20 y su valor por defecto es 3.
	`let options = { style: "decimal",`
	`useGrouping: false,`
	`minimumIntegerDigits: 3,`
	`minimumFractionDigits: 2,`
	`maximumFractionDigits: 2 };`
	
	`(5.009).toLocaleString("es-ES", options);`
	`// Devuelve "005,01"`

8.2.2.4 OPERACIONES CON NÚMEROS

Como se verá pronto, existen ciertas operaciones básicas (suma, resta, multiplicación, división o exponenciación) que se pueden realizar sin tener que recurrir a métodos externos. Sin embargo, hay otras operaciones en las que es mejor tener una ayuda. Esta ayuda es el objeto **`Math`**.

8.2.2.4.1 El objeto Math

Este objeto proporciona una serie de constantes y métodos para facilitar la realización de operaciones matemáticas.

Como constantes disponemos de:

Constante	Descripción
`E`	Devuelve un valor aproximado de la constante de Euler. `Math.E // Devuelve 2.718281828459045`
`LN2`	Devuelve un valor aproximado al logaritmo neperiano de 2. `Math.LN2 // Devuelve 0.6931471805599453`
`LN10`	Devuelve un valor aproximado del logaritmo neperiano de 10. `Math.LN10 // Devuelve 2.302585092994046`
`LOG2E`	Devuelve un valor aproximado del logaritmo en base 2 de la constante de Euler. `Math.LOG2E // Devuelve 1.4426950408889634`
`LOG10E`	Devuelve un valor aproximado del logaritmo en base 10 de la constante de Euler. `Math.LOG10E // Devuelve 0.4342944819032518`
`PI`	Devuelve un valor aproximado de la constante de PI, relación existente entre la circunferencia de un círculo y su diámetro. `Math.PI // Devuelve 3.141592653589793`
`SQRT1_2`	Devuelve un valor aproximado de la raíz cuadrada de um medio, es decir, de 1 sobre la raíz cuadrada de 2. `Math.SQRT1_2 // Devuelve 0.7071067811865476`
`SQRT2`	Devuelve un valor aproximado de la raíz cuadrada de 2. `Math.SQRT2 // Devuelve 1.4142135623730951`

Y como métodos disponemos de:

Método	Descripción
`abs`	Devuelve el valor absoluto del valor indicado. `Math.abs('-1'); // Devuelve 1`
`acos`	Devuelve el arcocoseno del valor indicado. `Math.acos('0.999'); // Devuelve 0.044725087168733454`

acosh	Devuelve el arcocoseno hiperbólico del valor indicado. `Math.acosh('2'); // Devuelve 1.3169578969248166`
asin	Devuelve el arcoseno del valor indicado. `Math.asin('0.999'); // Devuelve 1.526071239626163`
asinh	Devuelve el arcoseno hiperbólico del valor indicado. `Math.asinh('2'); // Devuelve 1.4436354751788103`
atan	Devuelve la arcotangente del valor indicado. `Math.asin('0.999'); // Devuelve 0.784897913314115`
atanh	Devuelve la arcotangente hiperbólica del valor indicado. `Math.asinh('0.999'); // Devuelve 3.8002011672501994`
cbrt	Devuelve la raíz cúbica del valor indicado. `Math.cbrt('27'); // Devuelve 3`

ceil	Devuelve el valor entero más pequeño redondeando hacia arriba.
	`Math.ceil('-1.99'); // Devuelve -1`
	`Math.ceil('1.99'); // Devuelve 2`

cos	Devuelve el coseno del valor indicado. `Math.cos('0'); // Devuelve 1`
cosh	Devuelve el coseno hiperbólico del valor indicado. `Math.cosh('1'); // Devuelve 1`
exp	Devuelve la potencia de la constante de Euler elevado al valor indicado. `Math.exp('1'); // Devuelve 2.718281828459045`

floor	Devuelve el valor entero más grande redondeando hacia abajo.
	`Math.floor('-1.99'); // Devuelve -2`
	`Math.floor('1.99'); // Devuelve 1`

log	Devuelve el logaritmo neperiano del valor indicado. `Math.log('0'); // Devuelve 1`
log10	Devuelve el logaritmo en base 10 del valor indicado. `Math.log10('2'); // Devuelve 0.3010299956639812`
log2	Devuelve el logaritmo en base 2 del valor indicado. `Math.log10('2'); // Devuelve 1`
max	Devuelve el mayor valor de los valores indicados `Math.max(2, 3, 5, 8, -1, 6); // Devuelve 8`

min	Devuelve el menor valor de los valores indicados `Math.min(2, 3, 5, 8, -1, 6); // Devuelve -1`
pow	Devuelve la potencia del primer valor elevado al segundo valor o parámetro. `Math.pow(2, 3); // Devuelve 8`

random	Devuelve un valor pseudoaleatorio entre 0 y 1. Para conseguir números entre un máximo y mínimo, se pueden multiplicar este resultado por la diferencia entre ambos valores y sumarle el mínimo.
	`// Devuelve un valor decimal entre 0 y 1`
	`Math.random();`
	`// Devuelve un valor decimal entre max y min`
	`Math.random() * (max - min) + min;`
	`// Devuelve un valor entero entre max y min`
	`Math.trunc(Math.random() * (max - min) + min;`

round	Devuelve el valor redondeado al entero más cercano.
	`Math.round('-1.09'); // Devuelve -1`
	`Math.round('1.09'); // Devuelve 1`
	`Math.round('1.59'); // Devuelve 2`

sign	Devuelve el signo del valor indicado expresado en forma de -1 a 1.
	`Math.sign('-99'); // Devuelve -1`
	`Math.sign('0'); // Devuelve 0`
	`Math.sign('99'); // Devuelve 1`

sin	Devuelve el seno del valor indicado. `Math.sin('1'); // Devuelve 0.8414709848078965`
sinh	Devuelve el seno hiperbólico del valor indicado. `Math.sinh('1'); // Devuelve 1.1752011936438014`
sqrt	Devuelve la raíz cuadrada del valor indicado. `Math.sqrt('2'); // Devuelve 1.4142135623730951`
tan	Devuelve la tangente del valor indicado. `Math.tan('1'); // Devuelve 1.5574077246549023`
tanh	Devuelve la tangente hiperbólica del valor indicado. `Math.tanh('1'); // Devuelve 0.7615941559557649`

trunc	Devuelve la parte entera eliminando todos los decimales.
	`Math.trunc('-1.09'); // Devuelve -1`
	`Math.trunc('-1.59'); // Devuelve -1`
	`Math.trunc('1.09') // Devuelve 1`
	`Math.trunc('1.59'); // Devuelve 1`

8.2.3 Tipo BigInt

El tipo **BigInt** se utiliza para el tratamiento de números enteros que no admite decimales. Se caracteriza porque añade "n" al final y porque, mientras que **Number** puede manejar valores de 64 bits, **BigInt** puede manejar enteros de precisión arbitraria en donde la limitación es la memoria disponible del sistema host.

Otras diferencias que podemos encontrar en **Number** y **BigInt** es que, los valores **BigInt** no son estrictamente números, son más precisos ya que no sufren problemas de precisión de coma flotante, no admiten mezclarse con otros tipos (es decir, si se intenta operar un **Number** con un **BigInt** se producirá un error) y no puede utilizar el objeto **Math**.

Los valores BigInt sólo deben usarse se necesiten números mayores que el valor de la propiedad o constante Number.MAX_SAFE_INTEGER.

```
BigInt(9007199254740991); // Devuelve 9007199254740991n
BigInt("0x1fffffffffffff") // Devuelve 9007199254740991n
BigInt("21 calles") // Devuelve Error de sintaxis
BigInt(true); // Devuelve 1n
BigInt(false); // Devuelve 0n
```

8.2.3.1 PROPIEDADES

El tipo `BOOLEAN` no presenta propiedades, a excepción de **`CONSTRUCTOR`** y **`PROTOTYPE`**.

8.2.3.2 MÉTODOS

El número de métodos disponibles para este objeto son, básicamente, dos:

Método	Descripción y ejemplo

`asIntN`	Permite truncar un valor BigInt al número dado de bits menos significativos devolviéndolo como un entero con signo.
	`BigInt.asIntN(8, 64n); // devuelve 64n`
	`BigInt.asIntN(7, 64n); // devuelve -64n`

`asIntN`	Permite truncar un valor BigInt al número dado de bits menos significativos devolviéndolo como un entero sin signo.
	`BigInt.asUintN(8, 64n); // devuelve 64n`
	`BigInt.asUintN(7, 64n); // devuelve 64n`

`toString`	Permite convertir el valor BigInt a un valor de cadena sin la "n" final. `BigInt.asUintN(8, 64n).toString() // devuelve "64"`

8.2.4 Tipo Boolean

El objeto **Boolean** se utiliza para la gestión de valores de tipo verdadero o falso.

En JavaScript, los valores booleanos pueden ser utilizados para realizar determinadas operaciones (como son las aritméticas). Esto es posible porque, la constante o literal **true** equivale al valor numérico 1 y, la constante o literal **false** equivale al valor numérico 0.

Provee de un constructor asociado que puede ser utilizado para realizar una conversión explícita, sin embargo, hay que tener en cuenta que, para el objeto Boolean, todo lo que no sea 0, vacío o null será **true**.

```
Boolean(0); // Devuelve false
Boolean(“”); // Devuelve false
Boolean(null); // Devuelve false
Boolean(“,”); // Devuelve true
Boolean(4); // Devuelve true
Boolean(1) - 1 == false // Devuelve true
```

8.2.4.1 PROPIEDADES

El tipo `BOOLEAN` no presenta propiedades, a excepción de **`CONSTRUCTOR`** y **`PROTOTYPE`**.

8.2.4.2 MÉTODOS

El tipo BOOLEAN sólo presenta un único método.

Método	Descripción
`toString`	Devuelve el valor booleano en formato String. `false.toString(); // devuelve “false”`

8.2.5 Tipo Symbol

Existe un tipo primitivo especial de datos denominado **Symbol** que posee la característica de que sus valores son únicos e inmutables. Es un tipo de datos que casi no se utiliza, sin embargo, puede ser útil cuando se desean añadir claves de propiedades únicas a un objeto de forma que no sean iguales a las claves que cualquier otro objeto.

Para crear un símbolo sólo hay que hacer:

```
let s1 = Symbol(“Hola Pablo”);
let s2 = Symbol(“Hola Pablo”);
console.log(s1 == s2); // Devuelve false
```

Como se puede apreciar en el ejemplo anterior, si creamos dos símbolos idénticos en variables diferentes, su comparación dará como resultado falso. Esto es así porque, en realidad, no se está convirtiendo un dato en símbolo, sino que se está creando un símbolo que tiene como descripción ese dato.

8.2.6 Literal null

El tipo **null** es un tipo especial de objeto que indica que no tiene valor, está vacío o no está referenciado. Es realidad, es como si se tratase de una constante pero tratado de otra manera.

Para comprobar si un objeto o elemento es de este tipo se puede recurrir a utilizar la función condicional binaria típica que devolverá si el tipo de dato actual es o no nulo.

```
this == null // Devolverá false
```

Si la condición de la izquierda es una variable, en vez de un objeto, y ésta no se encuentra definida, probablemente mostrará un error de tipo "**Uncaught ReferenceError: ___ is not defined**".

Las situaciones más frecuentes en las que podemos utilizar esta metodología son cuando se buscan elementos inexistentes en la página porque su resultado es null.

```
document.getElementById("elementoNoexistente") == null
```

8.2.7 Literales undefined y typeof

El tipo **undefined** es un tipo especial que indica que no tiene valor alguno, pero está referenciado. Por ejemplo, si una variable no tiene un valor asignado, pero está declarada, su "valor" será considerado como **undefined**.

Para comprobar si un objeto o elemento es de este tipo se puede recurrir a la utilización de la función **typeof**, que devolverá el tipo de dato o, en caso contrario, devolverá undefined.

```
typeof x == "undefined"
```

Los métodos o funciones también pueden devolver este tipo valor. De hecho, es muy frecuente encontrase con una función que devuelva el valor **undefined** y eso, puede ser, o bien porque el valor que se devuelve no tiene nada asignado, o bien porque no se devuelve nada.

8.3 OPERADORES Y EXPRESIONES

El número de operadores que presenta son muchos, aunque, en esencia, son los mismos que cualquier otro lenguaje.

En lo referente al orden de los operadores y sus pesos, en general, se sigue el estándar de todos los lenguajes de programación. Así, los operadores de adición y sustracción tienen menos peso que los de multiplicación, división, resto y exponenciación. Por ello, es necesario tener que recurrir a agrupaciones forzadas a través de paréntesis. Por ejemplo:

```
3 + 5 * 2; // Devolverá 13
```

En lo referente a la evaluación, ésta, se realiza de izquierda a derecha, es decir, que primero se evaluará la parte izquierda y, si se cumple, se seguirá evaluando hacia la derecha.

8.3.1 Operadores generales

Operador		Descripción y ejemplos
Tipo	**ID**	

Asignación	=	También conocido como operador de asignación, permite asignar el valor de la expresión declarada en la parte derecha al identificador declarado en la parte izquierda. El operador de asignación puede combinarse con la mayoría de los operadores que se van a ver poniéndolos como prefijo.
		let aux = 0;
		`aux += 10; // Devuelve 1`
		`aux *= 2 // Devuelve 20`

Concatenación	+	También conocido como operador de concatenación, permite unir los operandos uno detrás de otro. `"2" + 1; // Devuelve "21"`
Aritmético	+	También conocido como operador de adición, permite que el operador izquierdo sea sumado al derecho y devuelva su resultado. `2.01 + 1.02; // Devuelve 3.03`
Aritmético	-	También conocido como operador de sustracción, permite que el operador derecho sea restado al izquierdo y devuelva su resultado. `true - 1 // Devuelve 0`
Aritmético	*	También conocido como operador de multiplicación, permite que el operador izquierdo sea multiplicado por el derecho y devuelva su resultado. `"4" * "2" // Devuelve 8`
Aritmético	/	También conocido como el operador de división, permite que el operador izquierdo sea dividido por el derecho y devuelva su resultado. `true / "2" // Devuelve 0.5`
Aritmético	%	Permite que el operador izquierdo sea dividido por el derecho y devuelva el resto de su división entera. `"20" % "3"; // Devolverá 2`

Aritmético	**	Permite que el operador izquierdo sea elevado al valor indicado por el derecho y devuelva su resultado. `2 ** 3; // Devolverá 8`

Aritmético	++	Permite que el valor del operando se vea incrementado en 1 a través de una expresión reducida. Si el operador de incremento está antes del identificador de la variable, se devolverá el valor después de ser incrementado, pero, si el operador de incremento está después del identificador de la variable, se devolverá el valor antes de ser incrementado.
		`let a = 0, b = ++a; // Devuelve a = 1, b = 1`
		`let a = 0, b = a++; // Devuelve a = 1, b = 0`

Aritmético	—	Es el mismo tipo de operación que la anterior, sólo que, en vez de incrementar, decrementa. La casuística es la misma que en el caso anterior, pero con decrementos.
		`let a = 0, b = --a; // Devuelve a = 0, b = 0`
		`let a = 0, b = a--; // Devuelve a = 0, b = 1`

Lógico	&&	Permite realizar una conjunción lógica. Esto significa que, si ambas expresiones son de tipo booleano y ambas son verdaderas, el resultado devuelto será true, en cualquier otro caso, devolverá false. Cabe destacar que, si la expresión declarada en la parte izquierda puede ser convertida a false, devolverá su resultado, de lo contrario, devolverá el resultado de la expresión declarada en la parte derecha del operador
		`false && false; // Devuelve true`
		`false && (3 == 4); // Devuelve false`
		`“Hola” && “Ana”; // Devuelve “Ana”`

Lógico	||	Permite realizar una disyunción lógica. Esto significa que, si ambas expresiones son de tipo booleano y alguna es verdadera, el resultado devuelto será true, en cualquier otro caso, devolverá false. Cabe destacar que, si la expresión declarada en la parte izquierda puede ser convertida a true, devolverá su resultado, de lo contrario, devolverá el resultado de la expresión declarada en la parte derecha del operador.
		`true || false; // Devuelve true`
		`false || (3 == 4); // Devuelve false`
		`“Hola” || “Ana”; // Devuelve “Hola”`

Lógico	!	Permite realizar una negación lógica. Esto significa que, si la expresión declarada puede ser transformada a true devolverá false, de lo contrario, devolverá true.
		`!true; // Devuelve false`
		`!(3 == 4); // Devuelve true`
		`!”Hola”; // Devuelve false`

Lógico	**??**	Permite realizar una disyunción lógica basándose en valores nulos o no definidos. Esto significa que devolverá el primer valor que encuentre no nulo y no indefinido.
		`0 ?? 5 // Devuelve 0`
		`0 || 5 // Devuelve 5`
		`undefined ?? 0 // Devuelve 0`

Condicional	==	Permite comparar si dos operandos son iguales en cuanto a su valor.
		`false == false; // Devuelve true`
		`1 == "1" // Devuelve true`

Condicional	===	Permite comparar si dos operandos son iguales en cuanto a su tipo y valor.
		`false === false; // Devuelve true`
		`1 === "1" // Devuelve false`

Condicional	!=	Permite comparar si dos operandos son distintos en cuanto a su valor.
		`false != false; // Devuelve false`
		`1 != "1" // Devuelve false`

Condicional	!==	Permite comparar si dos operandos son distintos en cuanto a su tipo y valor.
		`false !== false; // Devuelve false`
		`1 !== "1" // Devuelve true`

Condicional	>	Permite comparar si el valor de la izquierda es mayor que el valor de la derecha. Si se comparan cadenas, los caracteres se transforman a ASCII.
		`"b" > "a"; // Devuelve true`
		`2 > 3 // Devuelve false`

Condicional	>=	Permite comparar si el valor de la izquierda es mayor o igual que el valor de la derecha. Si se comparan cadenas, los caracteres se transforman a ASCII.
		`"b" >= "a"; // Devuelve true`
		`2 >= 3 // Devuelve false`

Condicional	<	Permite comparar si el valor de la izquierda es menor que el valor de la derecha. Si se comparan cadenas, los caracteres se transforman a ASCII.
		`"b" < "a"; // Devuelve false`
		`2 < 3 // Devuelve true`

Condicional	<=	Permite comparar si el valor de la izquierda es menor o igual que el valor de la derecha. Si se comparan cadenas, los caracteres se transforman a ASCII.
		`"b" < "a"; // Devuelve false`
		`2 < 3 // Devuelve true`

Condicional	?	También conocido como operador condicional ternario es el único operador condicional que requiere de tres operandos. La parte izquierda de la expresión será evaluada y, en caso de ser verdadero, devolverá el resultado de la expresión detrás del símbolo de interrogación. De lo contrario, devolverá el resultado de la expresión detrás del símbolo de dos puntos. `const aux = value == 'on' ? 'class="checked"' : '';`

Miembro	.	Permite asignar o acceder a una propiedad de un objeto, no obstante, esto sólo funcionará si conoce la clave de la propiedad. Si el nombre de esa propiedad está declarado en una variable, deberá usarse la notación de corchetes en su lugar.
		`var aux = {name: "Ana", age: 18}`
		`aux.name // Devuelve "Ana"`

Miembro	?.	Permite asignar o acceder a una propiedad de un objeto si existe. Puede ser útil para comprobar si la propiedad de un objeto existe.
		`var aux = {name: "Ana", age: 18}`
		`aux?.edad ? aux.edad : aux.age // Devuelve 18`

Especial	,	El operador coma puede servir para separar expresiones cuando se realizan declaraciones, aunque también se utiliza para evaluar todas las expresiones de izquierda a derecha y devolver la última. Esta casuística es frecuente verla en bucles iterativos de tipo **for** para actualizar varias variables en cada ciclo, pero eso lo veremos en el siguiente capítulo.
		`let i = 0, j = 10;`
		`i = j + 2;`

8.3.2 Operadores bit a bit

Los operadores bit a bit manejan los operandos como si fuesen un conjunto de 32 bits, es decir, como un número formado por ceros y unos y cuya longitud es igual a 32. Para un operando o numérico como pueda ser 7, primero se transforma a su equivalente en base 2, es decir 00000000000000000000000000000111, y luego se realizan las operaciones bajo esta representación binaria.

8.3.2.1 OPERADOR &

El operador & (AND) binario da como resultado 1 sólo si ambos bits son 1.

```
let b1 = 12; // Equivale a 1100
let b2 = 4 // Equivale a 0100
_____________________________________

let b3 = b1 & b2; // Resultado: 0100 En decimal: 4
```

8.3.2.2 OPERADOR |

El operador | (OR) binario da como resultado 0 sólo si ambos bits son 0.

```
let b1 = 12; // Equivale a 1100
let b2 = 4; // Equivale a 0100
_____________________________________

let b3 = b1 | b2; // Resultado: 1100 En decimal: 12
```

8.3.2.3 OPERADOR ^

El operador ^ (XOR) binario da como resultado 1 si uno de los bits es 1.

```
let b1 = 12; // Equivale a 1100
let b2 = 4; // Equivale a 0100
_____________________________________

let b3 = b1 ^ b2; // Resultado: 1000 En decimal: 8
```

8.3.2.4 OPERADOR ~

El operador ~ (de complementación) se representa con el símbolo de la virgulilla y da como resultado la negación de todos y cada uno de los bits que conforman el operando.

```
let b1 = 38; // Equivale a 00100110
_____________________________________

let b2 = ~ b1; // Resultado: 11011001 En decimal: -39
```

8.3.2.5 OPERADORES DE DESPLAZAMIENTO

Los operadores de desplazamiento sirven para aumentar o disminuir el resultado en forma de potencia las veces que indique el operando de la derecha. Es decir, el operando situado a la izquierda se convertirá a binario y será multiplicado o dividido por 2 tantas veces como indique el operando situado a la derecha.

Para realizar un desplazamiento a la izquierda se debe utilizar el símbolo menor que repetido dos veces. El resultado de realizar este desplazamiento a la izquierda equivaldrá a convertirlo en binario y añadir un cero por la derecha. También equivaldrá a multiplicar por dos.

Para realizar un desplazamiento a la derecha se debe utilizar el símbolo mayor que repetido dos veces. El resultado de realizar este desplazamiento a la derecha equivaldrá a convertirlo en binario y añadir un cero por la izquierda y eliminar el último dígito de la derecha. También equivaldrá a dividir por dos sin resto.

```
4 >> 2 // Devuelve 1
4 << 2 // Devuelve 16
```

8.4 CONTROL DE FLUJO Y GESTIÓN DE ERRORES

8.4.1 Estructura if

La estructura condicional **if** se compone de una expresión que será evaluada y, en función de su respuesta, provocará la ejecución de su contenido o no.

```
if(fecha == '20-02-2002' ){
console.log('Es 20 de febrero de 2002');
}
```

8.4.2 Estructura if...else

La estructura condicional **if...else** se compone de una expresión que será evaluada y, si la respuesta es afirmativa, provocará que la ejecución se desvíe por el bloque delimitado por la sentencia **if**. En cualquier otro caso, se desviará por el bloque delimitado por la sentencia **else**.

```
if(fecha == '20-02-2002' ){
console.log('Es 20 de febrero de 2002')
} else {
console.log('No es 20 de febrero de 2002')
}
```

8.4.3 Estructura switch

Si la condición puede tomar un número elevado de casuísticas, es preferible utilizar una estructura **switch**. Su uso, permite realizar una implementación igual de rápida, pero más clara.

La estructura **switch** se compone de una condición que será evaluada en la cabecera de la estructura y unos posibles valores que se irán contemplando en cada caso a través de la sentencia **case**. Si se cumple alguno de los casos contemplados en las sentencias **case**, se ejecutará el conjunto de instrucciones ubicadas dentro de su bloque, en cualquier otro caso, se ejecutarán las instrucciones contenidas dentro del bloque delimitado por la sentencia **default**.

```
switch(marca) {
case 'ford':
console.log('El coche es de la marca Ford');
break;
case 'seat':
console.log('El coche es de la marca Seat');
```

```
break;
default:
console.log('El coche es de otra marca');
}
```

El valor de la sentencia **case** no tiene por qué ser una constante, puede ser una expresión o función que devuelva un valor que sea utilizado para realizar la selección de la secuencia.

Al igual que pasa con la estructura for, switch utiliza la instrucción **break** para romper la secuencia y salir de la estructura, con la diferencia de que, en la estructura switch, el uso de break es obligatorio.

8.4.4 Control de errores por tipo de dato

Existen varias formas de controlar los errores en JavaScript. Algunos, como se ha visto anteriormente pueden realizarse a través de estructuras de control de flujo como es el caso de la sentencia **if...else**, pero hay más posibles casuísticas.

En ocasiones, se requiere del uso de instrucciones específicas que nos permitan predecir su valor para evitar errores de conversión. Este es el caso de **typeof**.

```
let fecha = '';
if(typeof arguments[0] == 'object' ){
fecha = new Date(aux.anio + "-" + aux.mes + "-" + aux.dia);
} else {
fecha = new Date(aux);
}
```

En el ejemplo, podemos observar que se utiliza el objeto **arguments**. Este objeto es se corresponde con una especie de array que contiene todos los parámetros que reciben las funciones en JavaScript. El elemento 0 se corresponde con el primer parámetro.

También podemos ver que se utiliza la sentencia typeof que permite averiguar el tipo de dato que viene. Si el parámetro enviado es de tipo Object (suponemos que es un JSON predefinido), la variable fecha se construirá a partir de cada una de las propiedades de ese objeto. Si el parámetro enviado es de tipo String se utilizará como valor textual para definir la fecha.

8.4.5 Control de errores por presencia

Si queremos averiguar si un objeto tiene una propiedad o método en su definición, podemos realizarlo a través del operador **in** de JavaScript.

```
console.log('insertRule' in document.styleSheets[0]);
```

En el ejemplo, comprobamos que el DOM tiene definido el método insertRule y lo mostramos por consola.

Este tipo de comprobaciones se suele hacer para detectar si el navegador tiene disponible una determinada característica y, en función de ello, realizar una operación u otra.

Si lo que queremos saber es si un JSON contiene una propiedad concreta podemos hacerlo a través del método **hasOwnProperty**. Este método nos devolverá **true** o **false** en función de si existe o no su declaración en el JSON.

```
let json = { nombre: 'Pablo', edad: 18 };
console.log(json.hasOwnProperty("apellidos"));       // Devolverá false
console.log(json.hasOwnProperty("nombre")); // Devolverá true
```

8.4.6 Manejo de excepciones

8.4.6.1 SENTENCIA TRY...CATCH

Las formas anteriormente descritas podrían ser una manera tan buena como cualquier otra para gestionar los errores predecibles, sin embargo, hay casos en los que no podemos tratarlos así y tenemos que recurrir al manejo de excepciones.

En JavaScript, el manejo de excepciones es casi un requerimiento porque su ejecución es secuencial y continua. Si se produce un error en una línea del código, JavaScript ya no ejecutará nada de lo que esté declarado por debajo de ella.

Para evitar esta casuística podemos recurrir a la sentencia **try...catch**. Este tipo de estructura es muy útil para controlar errores de conversión, sintaxis, referencia o, incluso, de ejecuciones internas, entre otros casos.

Imaginemos el siguiente caso:

```
allert("Hola mundo!");
console.log("Todo OK!");
```

Si intentamos ejecutar el código anterior, se producirá una excepción en la aplicación que provocará una interrupción del código y mostrará un mensaje de error que dirá algo como "Uncaught ReferenceError: allert is not defined".

Ahora probemos con el siguiente código:

```
try {
allert("Hola mundo!");
} catch(err) {
console.log(err.message);
}

console.log("¡Todo OK!");
```

Si ahora intentamos ejecutar este último código, no se producirá ninguna interrupción. Sólo se nos mostrará por consola un mensaje de error que dice "allert is not defined" y, a continuación, mostrará el mensaje de "¡Todo OK!".

8.4.6.2 SENTENCIA FINALLY

La sentencia **finally** permite ejecutar una secuencia de instrucciones se produzca o no un error en la estructura **try...catch**, sin embargo, no suele ser utilizada porque no es compatible con muchos navegadores, incluyendo Internet Explorer.

El conjunto de instrucciones dentro del bloque de esta sentencia se ejecuta, incluso aunque no exista el bloque catch.

```
try {
allert("Hola mundo!");
} catch(err) {
console.log(err.message);
} finally {
console.log("Todo OK!");
}
```

8.4.6.3 SENTENCIA THROW

Si por alguna razón quisiéramos lanzar una excepción, ya sea predefinida o personalizada, JavaScript nos provee de una sentencia que permite hacerlo en cualquier momento de la ejecución del código.

Lo normal es que esta instrucción se utilice con objetos complejos que manipulan los errores producidos, no obstante, puede usarse, incluso, con tipos de datos primitivos.

```
throw "Error"; // Devuelve "Uncaught Error"
throw 18; // Devuelve "Uncaught 18"
throw false; // Devuelve "Uncaught false"
```

Sin embargo, como decía antes, lo normal es utilizar con objetos personalizados en combinación de otras sentencias y objetos.

Uno los recursos más utilizados para esta tarea quizás sea el objeto global **Error**. El objeto Error permite representar un error en tiempo de ejecución que tiene, como único argumento, un String.

```
function excepcionPersonalizada(mensaje) {
let error = new Error(mensaje);

error.code = "Error cero";
return error;
}

excepcionPersonalizada.prototype = Object.create(Error.prototype);

throw excepcionPersonalizada("Error de entrada")
```

La función excepcionPersonalizada define el nuevo tipo de excepción y, más tarde, con prototype, le asignamos el prototipo del objeto Error. De esta manera, cuando se lance la excepción con throw, se mostrará el mensaje requerido indicando en qué objeto se produjo y la línea donde se produjo la excepción.

A continuación, se muestra un ejemplo de lo que se produciría si lanzamos la excepción descrita.

```
Uncaught Error: Error de entrada
at excepcionPersonalizada (main.js:134)
at main.js:142
```

8.5 BUCLES Y LA ITERACIÓN

Un bucle suele identificarse con una acción que se repite un número finito de veces. Los bucles son un recurso muy útil para eso, sin embargo, si no se establecen bien los límites pueden generar efectos no deseados o incluso bloquear el sistema.

JavaScript, como casi todos los lenguajes de programación, dispone de cuatro estructuras para realizar procesos iterativos, aunque, una de ellas (for), tiene dos variaciones que pueden resultar muy versátiles.

8.5.1 Estructura for

Los bucles formados por esta estructura o sentencia se caracterizan porque todo su contenido se ejecuta un número predefinido de veces hasta que la condición de finalización se cumpla.

Los procesos iterativos definidos a través de la sentencia **for** se componen de tres expresiones y tienen un rango de número de iteraciones de CERO a N, es decir, que puede que no se ejecute ni una sola vez si la condición de entrada no se cumple.

La primera expresión, habitualmente, inicia la variable que será utilizada como contador. Sin embargo, su sintaxis permite establecer varias variables separadas por comas, como se verá un poco más adelante.

La segunda expresión, se utiliza para indicar cuándo o en qué momento debe detenerse el proceso iterativo.

La tercera y última expresión es la que produce una acción de incremento o decremento cada vez que se inicie un nuevo ciclo. No obstante, su sintaxis también permite establecer varios incrementos o decrementos al mismo tiempo.

Un ejemplo de uso básico podría ser:

```
let chars = new Array();
for(let x = 0; x < 100000; x++){
chars[x] = {code: x, char: String.fromCharCode(x)};
}
```

Si ejecutásemos el código anterior podríamos ver que, en la consola del navegador, se muestra el valor de **x** que, en este caso, irá desde “X: 0” hasta “X: 9”.

Un ejemplo de uso un poco más complejo podría ser el siguiente:

```
for (let i = 0, j = 10; i <= j; i++, j--){
console.log(i, j)
}
```

En este ejemplo, podemos observar que se definen varias variables y que, en la consola del navegador, se muestra el valor de **i** y **j** que, en este caso, irán desde 0 a 5 y desde 10 hasta 5 respectivamente. Por lo tanto, se ejecutará mientras **i** y **j** sean distintos, es decir, se ejecutará 6 veces.

8.5.2 Estructura for...in

Esta variación del bucle **for** estándar permite iterar un objeto a través de sus nombres de propiedades enumerables. Los objetos iterables por esta sentencia pueden ser matrices, mapas, argumentos, conjuntos de datos, ... Por cada propiedad que se captura, JavaScript ejecuta su contenido hasta que ya no haya más propiedades que capturar.

```
let arr = [1, 1, 2, 3, 5, 8];
for (let x in arr){
console.log("X: ", x);   // Devuelve "X: 0" hasta "X: 9"
}
```

Aunque esta forma de recorrer los objetos pueda resultar muy cómoda, no es recomendable utilizarla cuando el número de elementos es muy elevado o cuando el objeto a recorrer es muy grande en tamaño porque el rendimiento puede bajar exponencialmente.

Para ver mejor hasta qué punto puede afectar el bucle veámoslo con un ejemplo. Vamos a averiguar cuánto tarda un bucle for en copiar el array "chars" utilizado en el primer ejemplo.

```
console.time();
let arrFor = new Array();
for(let x = 0; x < chars.length; x++){
arrFor[x] = chars[x];
}
console.timeEnd();
```

Si ejecutamos este código varias veces podremos observar que, de media, tarda **unos 11ms**.

Ahora averigüemos lo que tarda la misma operación, pero realizada mediante una estructura for...in:

```
console.time();
let arrForIn = new Array();
for(let key in chars){
arrForIn[key] = chars[key]
}
console.timeEnd();
```

Si ejecutamos este código varias veces podremos observar que, en general, tarda bastante más del doble, en nuestro test, **una media 23ms**.

8.6 ESTRUCTURA FOR...OF

Esta variación del bucle **for** estándar permite iterar un objeto a través de sus valores enumerables. Los objetos iterables por esta sentencia pueden ser matrices, mapas, argumentos, conjuntos de datos, ... Por cada propiedad que se captura, JavaScript ejecuta su contenido hasta que ya no haya más valores que capturar.

```
let arr = [1, 1, 2, 3, 5, 8];
for (let x of arr){
console.log("X: ", x);   // Devuelve 1,1,2,3,5,8
}
```

Aunque esta forma de recorrer los objetos pueda resultar muy cómoda, no es recomendable utilizarla cuando el número de elementos es muy elevado o cuando el objeto a recorrer es muy grande en tamaño porque el rendimiento puede bajar de forma abrumadora.

Para ver mejor hasta qué punto puede afectar esta estructura al rendimiento, veámoslo con un ejemplo. Ya sabemos lo que tarda el bucle for en copiar el array "chars" utilizado en el primer ejemplo (unos 11ms) y también, lo que tarda en copiar ese array mediante la estructura for...in (unos 23ms). Por ello, vamos a conocer cuánto tarda en realizar la copia a través del bucle for...of:

```
console.time();
let arrForOf = new Array();
for(let [key, value] of chars.entries()){
arrForOf[key] = value
}
console.timeEnd();
```

Si ejecutamos este código varias veces podremos observar que, en general, tarda casi cuatro veces más que el bucle for, en nuestro test, **una media 39ms**.

8.7 ESTRUCTURA FOREACH

Los bucles formados por esta estructura se caracterizan porque ejecutan una función de callback en cada iteración.

En JavaScript, este tipo de bucle tiene un rendimiento más alto que la iteración a través del bucle for. De hecho, en situaciones normales, esta estructura **es hasta un 66 por ciento más rápida de media**, sin embargo, si se trabaja con objetos muy grandes la distancia de tiempos entre el forEach y for puede no ser relevante.

Para ver mejor hasta qué punto puede afectar esta estructura al rendimiento, veámoslo con un ejemplo. Si recordamos, la copia del array chars a través de la estructura for tardaba unos 11ms. Vamos a averiguar cuánto tarda un bucle forEach en copiar el array "chars" utilizado en el primer ejemplo.

```
console.time();
let arrForEach = new Array();
source.forEach(function(v, i){
arrForEach[i] = v
});
console.timeEnd();
```

Si ejecutamos este código varias veces podremos observar que, en general, tarda un 20 por cierto menos, en nuestro test, **una media 9ms**.

Cabe destacar que esto no es una característica general para todos los lenguajes, sino más bien lo contrario. En muchos lenguajes y, sobre todo, en los compilados como Java, el bucle forEach baja el rendimiento porque, por norma general, la evaluación y asignación de variables es un trabajo más costoso para la máquina que acceder a un índice de forma directa.

La función de **callback** puede recibir tres parámetros, aunque lo normal es que reciba sólo dos.

```
["A", "B", "C", "D"].forEach(function(valor, indice){
console.log("Índice:", indice, "Valor:", valor);
});
// Devuelve Índice: 0 Valor: A
// Devuelve Índice: 1 Valor: B,...
```

El parámetro **valor** es el elemento actual en el momento de la iteración y el parámetro **índice** es la posición actual en el momento de la iteración, que es opcional.

Como decía, hay un tercer parámetro que llamaremos **array** y que es la matriz, mapa o conjunto que está siendo usado. Este parámetro también es opcional.

```
let arr = ["A", "B", "C", "D"];
arr.forEach(function(val, idx, arr){
arr[idx] = val.charCodeAt();
});

console.log(arr); // Devuelve [65, 66, 67, 68]
```

Como se puede apreciar, este tercer parámetro se suele utilizar cuando se desea manipular el origen, como es este caso, que convierte el carácter enviado a su código Unicode.

La estructura forEach no admite más parámetros, no obstante, existe una posibilidad más de configuración, el objeto que actuará como **this**. Veámoslo con un ejemplo:

```
function calculadora() {
this.suma = 0;
```

```
this.producto = 1;
}

calculadora.prototype.sumar = function(array) {
array.forEach(function(valor) {
this.suma += valor;
}, this);
};

calculadora.prototype.multiplicar = function(array) {
array.forEach(function(valor) {
this.producto *= valor;
}, this);
};

let arr = [1,1,2,3,5,8];
let s = new calculadora();
s.sumar(arr);
s.multiplicar(arr);

console.log(s.suma, s.producto) // Devuelve 20 240
```

Si ejecutamos este código podremos ver que, el parámetro this, hace que el objeto this sea reemplazado por el objeto que representa a calculadora. De no haber establecido el objeto this en la función, lo que se recibiría no sería el objeto "calculadora", sino el objeto global (window).

La estructura forEach sólo se puede utilizar en arrays, mapas y conjuntos y siempre devuelve **undefined**.

8.8 ESTRUCTURA DO...WHILE

Los bucles formados por esta estructura o sentencia se caracterizan porque todo su contenido se ejecuta un número impredecible de veces hasta que la condición de finalización se cumpla.

Los procesos iterativos definidos a través de la sentencia **do...while** se componen únicamente de una expresión condicional y tienen un rango de número de iteraciones de UNO a N, es decir, que siempre se ejecutará su contenido, al menos, una vez.

La expresión suministrada en la condición puede ser tan compleja como se desee, sin embargo, si no se define bien puede provocar bucles infinitos y bloquear la aplicación o el sistema.

```
let x = 0;
do {
x += 1; // Forma abreviada de hacer x = x + 1;
console.log("X: ", x);
} while (x < 10);
```

El ejemplo anterior mostrará en la consola del navegador el valor de x que, en este caso, irá desde 1 hasta 10.

> **NOTA**
>
> En lo referente al rendimiento, la estructura while es un poco más rápido y eficiente al realizar la operación de copiado del array chars (del bucle for).

8.9 ESTRUCTURA WHILE

Los bucles formados por esta estructura o sentencia se caracterizan porque todo su contenido se ejecuta un número impredecible de veces hasta que la condición de finalización se cumpla.

A diferencia con la estructura **do...while**, la condición se evalúa antes de entrar y, precisamente por esta razón, el rango de número de iteraciones es de CERO a N, ya que puede que no se ejecute ni una sola vez si la condición de entrada no se cumple.

Los procesos iterativos definidos a través de la sentencia while se componen, únicamente, de una expresión condicional. Dicha expresión puede ser tan compleja como se desee, sin embargo, si no se define bien puede provocar bucles infinitos y bloquear la aplicación o el sistema.

```
let x = 0;
while (x < 10){
x += 1; // Forma abreviada de hacer x = x + 1;
console.log("X: ", x)
};
```

El ejemplo anterior mostrará en la consola del navegador el valor de x que, en este caso, irá desde 1 hasta 10.

> **NOTA**
>
> En lo referente al rendimiento, la estructura while es un poco más rápido y eficiente al realizar la operación de copiado del array chars (del bucle for).

8.10 SENTENCIA BREAK

Los procesos iterativos pueden ser interrumpidos de forma forzada a través de la sentencia **break**. Esta instrucción puede ejecutarse en el momento que se desee, sin embargo, no suele ser considerada como una buena práctica ya que rompe la secuencialidad del código. No obstante, puede ahorrar mucho tiempo de ejecución.

```
for (let x = 0; x < 10; x++){
if (x == 5) break;
console.log("X: ", x);
}
```

El ejemplo anterior mostrará en la consola del navegador el valor de x que, en este caso, irá desde 0 hasta 5, ya que los demás valores (del 6 al 9) se ignorarán.

8.11 SENTENCIA CONTINUE

Los procesos iterativos pueden ser alterados de forma forzada a través de la sentencia **continue**. Esta instrucción puede ejecutarse en el momento que se desee, sin embargo, al igual que pasa con la sentencia **break**, no se considera una buena práctica porque se suele pensar que, si el objeto a iterar tiene elementos que hay que omitir, es que no está bien construido dicho objeto, pero hay veces que no hay más remedio y, por eso, tenemos esta instrucción.

```
for (let x = 0; x < 10; x++){
if (x == 5) continue;
console.log("X: ", x);
}
```

El ejemplo anterior mostrará en la consola del navegador el valor de **x** que, en este caso, irá desde 0 hasta 4 y desde 6 hasta 9.

8.12 PRACTICA Y JUEGA

Test de JavaScript: Intro	Código QR
Juega a averiguar todas las respuestas correctas con el mínimo número de errores y en el menor tiempo posible. **https://codepen.io/pefc/full/WNgEvBo**	