Introducción a JavaScript

Fuente lenguajejs.com

Cualquier página web de Internet está construida, como mínimo, por **HTML** (*un lenguaje de marcas*) y **CSS** (*un lenguaje de estilos*). El primero de ellos permite construir todo el marcado de la página (*contenido e información*) mediante etiquetas HTML y dotando de semántica a la información mediante la naturaleza de dichas etiquetas. Posteriormente, el segundo de ellos permite darle estilo a la página y construir una interfaz visual más agradable para el usuario.

> La separación bien marcada de estos dos pilares permite que si en algún momento necesitamos modificar la información (*o el diseño*) de la página, no tengamos también que lidiar con modificaciones en el otro.

Sin embargo, utilizar sólo y exclusivamente **HTML** y **CSS** en una página nos limita considerablemente. Si bien es cierto que con estos dos lenguajes podemos hacer un gran abanico de cosas, hay otras que serían totalmente imposibles, o al menos, mucho más fáciles de realizar si tuviéramos un **lenguaje de programación** a nuestra disposición. Y en este punto es donde aparece **Javascript**.

¿Qué es Javascript?

Javascript es un **lenguaje de programación**, o lo que es lo mismo, un mecanismo con el que podemos decirle a nuestro navegador que tareas debe realizar, en que orden y cuantas veces (*por ejemplo*).

Muchas de las tareas que realizamos con HTML y CSS se podrían realizar con Javascript. De hecho, es muy probable que al principio nos parezca que es mucho más complicado hacerlo con Javascript, y que por lo tanto no merece la pena. Sin embargo, con el tiempo veremos que Javascript nos ofrece una mayor flexibilidad y un abanico de posibilidades más grande, y que bien usadas, pueden ahorrarnos bastante tiempo.

Para comprenderlo, un ejemplo muy sencillo sería el siguiente:

```
<div class="item">
  <p>Número: <span class="numero">1</span></p>
```

```
  <p>Número: <span class="numero">2</span></p>
  <p>Número: <span class="numero">3</span></p>
  <p>Número: <span class="numero">4</span></p>
  <p>Número: <span class="numero">5</span></p>
</div>
```

Imaginemos que tenemos que crear una lista de números desde el **1** hasta el **500**. Hacerlo solamente con HTML sería muy tedioso, ya que tendríamos que copiar y pegar esas filas varias veces hasta llegar a 500. Sin embargo, mediante Javascript, podemos decirle al navegador que escriba el primer párrafo **<p>**, que luego escriba el mismo pero sumándole uno al número. Y que esto lo repita hasta llegar a 500.

De esta forma y con este sencillo ejemplo, con HTML habría que escribir **500 líneas** mientras que con Javascript no serían más de **10 líneas**.

Dificultad

Aunque Javascript es ideal para muchos casos, es mucho más complicado aprender Javascript (*o un lenguaje de programación en general*) que aprender HTML o CSS, los cuales son mucho más sencillos de comprender. Antes debemos conocer varias cosas:

- Para **aprender Javascript** debemos conocer el lenguaje **Javascript**, pero no podremos hacerlo si no sabemos programar. Se puede aprender a programar con Javascript, pero es recomendable tener una serie de fundamentos básicos de programación antes para que no nos resulte muy duro.
- Para **aprender a programar** antes debemos saber como «trabaja un ordenador». Programar no es más que decirle a una máquina que cosas debe hacer y como debe hacerlas. Eso significa que no podemos pasar por alto nada.
- Para **darle órdenes a una máquina** debemos tener claro que esas órdenes son correctas y harán lo que se supone que deben hacer. Si le indicamos a una máquina los pasos para resolver un problema, pero dichos pasos son erróneos, la máquina también hará mal el trabajo.

ECMAScript

ECMAScript es la especificación donde se mencionan todos los detalles de **cómo debe funcionar y comportarse Javascript** en un navegador. De esta forma, los diferentes navegadores (*Chrome, Firefox, Opera, Edge, Safari...*) saben cómo deben desarrollar los motores de Javascript para que cualquier código o programa funcione exactamente igual, independientemente del navegador que se utilice.

> ECMAScript suele venir acompañado de un número que indica la **versión o revisión** de la que hablamos (*algo similar a las versiones de un programa*). En cada nueva versión de ECMAScript, se modifican detalles sobre Javascript y/o se añaden nuevas funcionalidades, manteniendo Javascript vivo y con novedades que lo hacen un lenguaje de programación moderno y cada vez mejor preparado para utilizar en el día a día.

Teniendo esto en cuenta, debemos saber que los navegadores web intentan cumplir la especificación ECMAScript al máximo nivel, pero no todos ellos lo consiguen. Por lo tanto, pueden existir ciertas discrepancias. Por ejemplo, pueden existir navegadores que cumplan la especificación ECMAScript 6 al 80% y otros que sólo la cumplan al 60%. Esto significa que pueden haber características que no funcionen en un navegador específico (*y en otros sí*).

Además, todo esto va cambiando a medida que se van lanzando nuevas versiones de los navegadores web, donde su compatibilidad ECMAScript suele aumentar.

Versiones de ECMAScript

A lo largo de los años, Javascript ha ido sufriendo modificaciones que los navegadores han ido implementando para acomodarse a la última versión de **ECMAScript** cuanto antes. La lista de versiones de ECMAScript aparecidas hasta el momento son las siguientes, donde encontramos las versiones enmarcadas en lo que podemos considerar **el pasado de Javascript**:

Ed.	Fecha	Nombre formal / informal	Cambios significativos
1	JUN/1997	ECMAScript 1997 (ES1)	Primera edición
2	JUN/1998	ECMAScript 1998 (ES2)	Cambios leves

3	DIC/1999	ECMAScript 1999 (ES3)	RegExp, try/catch, etc...
4	AGO/2008	ECMAScript 2008 (ES4)	Versión abandonada.
5	DIC/2009	ECMAScript 2009 (ES5)	Strict mode, JSON, etc...
5.1	DIC/2011	ECMAScript 2011 (ES5.1)	Cambios leves

A partir del año 2015, se marcó un antes y un después en el mundo de Javascript, estableciendo una serie de cambios que lo transformarían en un lenguaje moderno, partiendo desde la especificación de dicho año, hasta la actualidad:

Ed.	Fecha	Nombre formal / informal	Cambios significativos
6	JUN/2015	ECMAScript 2015 (ES6)	Clases, módulos, generadores, hashmaps, sets, for of, proxies...
7	JUN/2016	ECMAScript 2016	Array includes(), Exponenciación **
8	JUN/2017	ECMAScript 2017	Async/await

9	JUN/2018	ECMAScript 2018	Rest/Spread operator, Promise.finally()...
10	JUN/2019	ECMAScript 2019	Flat functions, trimStart(), errores opcionales en catch...
11	JUN/2020	ECMAScript 2020	Dynamic imports, BigInt, Promise.allSettled

En ocasiones, algunos navegadores deciden implementar pequeñas funcionalidades de versiones posteriores de ECMAScript antes que otras, para ir testeando y probando características, por lo que no es raro que algunas características de futuras especificaciones puedan estar implementadas en algunos navegadores.

Una buena forma de conocer en qué estado se encuentra un navegador concreto en su especificación de ECMAScript es consultando la tabla de compatibilidad Kangax. En dicha tabla, encontramos una columna «Desktop browsers» donde podemos ver el porcentaje de compatibilidad con las diferentes características de determinadas especificaciones de ECMAScript.

> Nota que de ECMAScript 6 en adelante, se toma como regla nombrar a las diferentes especificaciones por su año, en lugar de por su número de edición. Aunque en los primeros temas los mencionaremos indiferentemente, ten en cuenta que se recomienda utilizar **ECMAScript 2015** en lugar de **ECMAScript 6**.

Estrategia «crossbrowser»

Dicho esto, y teniendo en cuenta todos estos detalles, es muy habitual que el programador esté confuso en cómo empezar a programar y que versión ECMAScript adoptar como preferencia.

Generalmente, el programador suele tomar una de las siguientes estrategias «crossbrowser» para asegurarse que el código funcionará en todos los navegadores:

Enfoque	Código escrito	Descripción
Conservador	ECMAScript 5	Incómodo de escribir. Anticuado. Compatible con navegadores nativamente.
Delegador	*Depende*	Cómodo. Rápido. Genera dependencia al framework/librería.
Evergreen	ECMAScript 6+	Cómodo. Moderno. No garantiza la compatibilidad con navegadores antiguos.
Transpilador	ECMAScript 6+	Cómodo. Moderno. Preparado para el futuro. Requiere preprocesado.

Vamos a explicar cada una de estas estrategias para intentar comprenderlas mejor.

Enfoque conservador

El programador decide crear código **ECMAScript 5**, una versión «segura» que actualmente una gran mayoría de navegadores (*incluido Internet Explorer soporta*). Este enfoque permite asegurarse de que el código funcionará sin problemas en cualquier navegador, pero por otro lado, implica que para muchas tareas deberá escribir mucho código, código extra o no podrá disfrutar de las últimas novedades de Javascript.

> Uno de los principales motivos por los que se suele elegir esta estrategia es porque se necesita compatibilidad con navegadores, sistemas antiguos y/o Internet Explorer. También se suele elegir porque es más sencilla o porque funciona nativamente sin necesidad de herramientas externas.

Enfoque delegador

El programador decide delegar la responsabilidad «crossbrowser» a un framework o librería que se encargará de ello. Este enfoque tiene como ventaja que es mucho más cómodo para el programador y ahorra mucho tiempo de desarrollo. Hay que tener en cuenta que se heredan todas las ventajas y desventajas de dicho framework/librería, así como que se adopta como **dependencia** (*sin dicho framework/librería, nuestro código no funcionará*). Además, también se suele perder algo de rendimiento y control sobre el código, aunque en la mayoría de los casos es prácticamente inapreciable.

> Hoy en día, salvo para proyectos pequeños, es muy común escoger un framework Javascript para trabajar. Un framework te ayuda a organizar tu código, a escribir menos código y a ser más productivo a la larga. Como desventaja, genera dependencia al framework.

Enfoque evergreen

El programador decide no preocuparse de la compatibilidad con navegadores antiguos, sino dar soporte sólo a las últimas versiones de los navegadores (*evergreen browsers*), o incluso sólo a determinados navegadores como **Google Chrome** o **Mozilla Firefox**. Este enfoque suele ser habitual en programadores novatos o proyectos que van dirigidos a un público muy concreto y no están abiertas a un público mayoritario.

Enfoque transpilador

El programador decide crear código de la última versión de **ECMAScript**. Para asegurarse de que funcione en todos los navegadores, utiliza un **transpilador**, que no es más que un

sistema que revisa el código y lo traduce de la versión actual de ECMAScript a **ECMAScript 5**, que es la que leerá el navegador.

La ventaja de este método es que se puede escribir código Javascript moderno y actualizado (*con sus ventajas y novedades*) y cuando los navegadores soporten completamente esa versión de ECMAScript, sólo tendremos que retirar el **transpilador** (*porque no lo necesitaremos*). La desventaja es que hay que preprocesar el código (*cada vez que cambie*) para hacer la traducción.

> Quizás, el enfoque más moderno de los mencionados es utilizar **transpiladores**. Sistemas como Babel son muy utilizados y se encargan de traducir de ECMAScript 6 a ECMAScript 5.

En estos primeros temas, tomaremos un **enfoque conservador** para hacer más fácil el inicio con Javascript. A medida que avancemos, iremos migrando a un enfoque **transpilador**.

Independientemente del enfoque que se decida utilizar, el programador también puede utilizar **polyfills** o **fallbacks** para asegurarse de que ciertas características funcionarán en navegadores antiguos. También puede utilizar enfoques mixtos.

- Un **polyfill** no es más que una librería o código Javascript que actúa de «parche» o «relleno» para dotar de una característica que el navegador aún no posee, hasta que una actualización del navegador la implemente.
- Un **fallback** es algo también muy similar: un fragmento de código que el programador prepara para que en el caso de que algo no entre en funcionamiento, se ofrezca una alternativa.

Consola de JavaScript

Para acceder a la consola Javascript del navegador, podemos pulsar CTRL+SHIFT+I sobre la pestaña de la página web en cuestión, lo que nos llevará al **Inspector de elementos** del navegador. Este inspector es un panel de control general donde podemos ver varios aspectos de la página en la que nos encontramos: su etiquetado HTML, sus estilos CSS, etc...

Concretamente, a nosotros nos interesa una sección particular del inspector de elementos. Para ello, nos moveremos a la pestaña **Console** y ya nos encontraremos en la **consola Javascript** de la página.

> También se puede utilizar directamente el atajo de teclado CTRL+SHIFT+J, que en algunos navegadores nos lleva directamente a la consola.

En esta consola, podemos escribir funciones o sentencias de Javascript que estarán actuando en la página que se encuentra en la pestaña actual del navegador. De esta forma podremos observar los resultados que nos devuelve en la consola al realizar diferentes acciones. Para ello, vamos a ver algunas bases:

La consola

El clásico primer ejemplo cuando se comienza a programar, es crear un programa que muestre por pantalla un texto, generalmente el texto «**Hola Mundo**». También podemos realizar, por ejemplo, **operaciones numéricas**. En la consola Javascript podemos hacer esto de forma muy sencilla:

```
console.log("Hola Mundo");
console.log(2 + 2);
```

En la primera línea, veremos que al pulsar enter nos muestra el texto «**Hola Mundo**». En la segunda línea, sin embargo, procesa la operación y nos devuelve **4**. Para mostrar estos textos en la consola Javascript hemos utilizado la función **console.log**, pero existen varias más:

Función	Descripción
console.log()	Muestra la información proporcionada en la consola Javascript.
console.info()	Equivalente al anterior. Se utiliza para mensajes de información.

console.warn()	Muestra información de advertencia. Aparece en amarillo.
console.error()	Muestra información de error. Aparece en rojo.
console.clear()	Limpia la consola. Equivalente a pulsar CTRL+L o escribir **clear()**.

La idea es utilizar en nuestro código la función que más se adapte a nuestra situación en cada caso (*errores graves con console.error(), errores leves con console.warn(), etc...*).

Aplicar varios datos

En el ejemplo anterior, solo hemos aportado un dato por cada línea (*un texto o una operación numérica*), pero **console.log()** y sus funciones hermanas permiten añadir varios datos en una misma línea, separándolo por comas:

```
console.log("¡Hola a todos! Observen este número: ", 5 + 18);
```

Aplicar estilos en la consola

Aunque no es muy práctico y sólo se trata de puro divertimento, se pueden aplicar estilos CSS en la consola Javascript haciendo uso de **%c**, que se reemplazará por los estilos indicados:

```
console.log("%c¡Hola Manz!",
            "background:linear-gradient(#000, #555); color:#fff; padding: 5px 10px;");
```

> Es importante recalcar que cuando escribimos en la consola podemos obviar el **console.log()** y escribir directamente la información, pero si queremos mostrar algo por consola desde nuestra página web o aplicación Javascript, es absolutamente necesario escribir **console.log()** (*o cualquiera de las funciones de su familia*) en nuestro código.

El **esquema general** de una página web es un **documento HTML** donde están todas las etiquetas HTML de la página. A lo largo de ese documento, pueden existir **referencias o relaciones** a otros documentos, como archivos CSS o archivos Javascript.

Por ejemplo, si dentro del documento HTML se encuentra una referencia a un **archivo CSS**, el navegador lo descarga y lo aplica al documento HTML, cambiando su apariencia visual. De la misma forma, si encuentra una referencia a un **archivo Javascript**, el navegador lo descarga y ejecuta las órdenes o acciones que allí se indican.

Ejemplo de script en línea

En este primer y sencillo ejemplo, sólo tenemos un documento: el **archivo HTML**. En él, existe una etiqueta **<script>** que contiene las órdenes o líneas de Javascript que le indican al navegador que tiene que hacer (*en este caso, mostrar un "¡Hola!" en la consola*):

```
<html>
  <head>
    <title>Título de la página</title>
    <script>
      console.log("¡Hola!");
    </script>
  </head>
  <body>
    <p>Ejemplo de texto.</p>
  </body>
</html>
```

Este método de escribir scripts se denomina **Javascript en línea** (*inline*), y significa que el Javascript está escrito directamente en el código HTML. Nos puede servir como ejemplo inicial, pero no es la forma recomendable de escribirlo, ya que lo ideal es separar el código HTML del código Javascript (*en archivos diferentes*) para organizarnos mejor.

Ejemplo de script externo

Esta otra forma de incluir Javascript en una página tiene la ventaja de, en el caso de necesitar incluir el código Javascript desde varios documentos HTML, no tendremos que volver a escribir dicho código, sino simplemente referenciar el nombre del mismo archivo Javascript a incluir en todas las páginas HTML.

Para relacionar un **documento Javascript** desde una página web, igual que antes, utilizaremos la etiqueta **<script>**, sólo que en este caso, haremos referencia al archivo **Javascript** con un atributo **src** (*source*), como se ve en el siguiente ejemplo:

```
<html>
  <head>
    <title>Título de la página</title>
    <script src="js/index.js"></script>
  </head>
  <body>
    <p>Ejemplo de texto.</p>
  </body>
</html>
```

El texto **js/index.js** no es más que una referencia a un archivo **index.js** que se encuentra dentro de una carpeta **js**, situada en la misma carpeta que el documento HTML del ejemplo. Si en este archivo Javascript, incluímos el **console.log()** de mensaje de bienvenida, ese mensaje debería aparecer en la consola Javascript al cargar esta página.

Ubicación de la etiqueta script

Si te fijas, en el ejemplo anterior, la etiqueta **<script>** está situada dentro de la etiqueta **<head>** de la página, es decir, en la cabecera de metadatos. Esto significa que la página web descargará el archivo Javascript antes de empezar a dibujar el contenido de la página (*etiqueta* ***<body>***).

Es posible que te hayas encontrado ejemplos donde dicha etiqueta esté ubicada en otra parte del documento HTML. Veamos las posibilidades:

Ubicación	¿Cómo descarga el archivo Javascript?	Estado de la página
En **<head>**	**ANTES** de empezar a dibujar la página.	Página aún no dibujada.
En **<body>**	**DURANTE** el dibujado de la página.	Dibujada hasta donde está la etiqueta **<script>**.
Antes de **</body>**	**DESPUÉS** de dibujar la página.	Dibujada al 100%.

Ten en cuenta que el navegador puede descargar un documento Javascript en cualquier momento de la carga de la página y necesitamos saber cuál es el más oportuno para nosotros.

- Si queremos que un documento Javascript actúe antes que se muestre la página, la opción de colocarlo en el **<head>** es la más adecuada.
- Si por el contrario, queremos que actúe una vez se haya terminado de cargar la página, la opción de colocarlo justo antes del **</body>** es la más adecuada. Esta opción es equivalente a usar el atributo **defer** en la etiqueta **<script>**, sin embargo, esta opción es además compatible con navegadores muy antiguos (*IE9 o anteriores*) que no soportan **defer**.

En Javascript, al igual que en la mayoría de los lenguajes de programación, al declarar una variable y guardar su contenido, también le estamos asignando un **tipo de dato**, ya sea de forma implícita o explícita. El **tipo de dato** no es más que la naturaleza de su contenido: contenido numérico, contenido de texto, etc...

¿Qué tipos de lenguajes existen?

A grandes rasgos, nos podemos encontrar con dos tipos de lenguajes de programación:

- **Lenguajes estáticos**: Cuando creamos una variable, debemos indicar el **tipo de dato** del valor que va a contener. En consecuencia, el valor asignado finalmente, siempre deberá ser del tipo de dato que hemos indicado (*si definimos que es un número debe ser un número, si definimos que es un texto debe ser un texto, etc...*).
- **Lenguajes dinámicos**: Cuando creamos una variable, no es necesario indicarle el tipo de dato que va a contener. El lenguaje de programación se encargará de deducir el tipo de dato (*dependiendo del valor que le hayamos asignado*).

> En el caso de los **lenguajes dinámicos**, realmente el tipo de dato se asocia al valor (*en lugar de a la variable*). De esta forma, es mucho más fácil entender que a lo largo del programa, dicha variable puede «cambiar» a tipos de datos diferentes, ya que la restricción del tipo de dato está asociada al valor y no a la variable en sí. No obstante, para simplificar, en los primeros temas siempre hablaremos de variables y sus tipos de datos respectivos.

Javascript pertenece a los **lenguajes dinámicos**, ya que automáticamente detecta de que tipo de dato se trata en cada caso, dependiendo del contenido que le hemos asignado a la variable.

Para algunos desarrolladores —*sobre todo, noveles*— esto les resulta una ventaja, ya que es mucho más sencillo declarar variables sin tener que preocuparte del tipo de dato que necesitan. Sin embargo, para muchos otros desarrolladores —*generalmente, avanzados*— es una desventaja, ya que pierdes el control de la información almacenada y esto en muchas ocasiones puede desembocar en problemas o situaciones inesperadas.

En Javascript existen mecanismos para convertir o forzar los tipos de datos de las variables, sin embargo, muchos programadores prefieren declarar explícitamente los tipos de datos, ya que les aporta cierta confianza y seguridad. Este grupo de desarrolladores suelen optar por utilizar lenguajes como Typescript, que no es más que «varias capas de características añadidas» a Javascript.

> En muchas ocasiones (*y de manera informal*) también se suele hacer referencia a **lenguajes tipados** (*tipado fuerte, o fuertemente tipado*) o **lenguajes no tipados** (*tipado débil, debilmente tipado*), para indicar si el lenguaje requiere indicar manualmente el tipo de dato de las variables o no, respectivamente.

¿Qué son los tipos de datos?

En Javascript disponemos de los siguientes tipos de datos:

Tipo de dato	Descripción	Ejemplo básico
number	Valor numérico (enteros, decimales, etc...)	**42**
string	Valor de texto (cadenas de texto, carácteres, etc...)	**'MZ'**
boolean	Valor booleano (valores verdadero o falso)	**true**
undefined	Valor sin definir (variable sin inicializar)	**undefined**
function	Función (función guardada en una variable)	**function() {}**
object	Objeto (estructura más compleja)	**{}**

Para empezar, nos centraremos en los tres primeros, denominados **tipos de datos primitivos**, y en los temas siguientes veremos detalles sobre los siguientes.

Para saber que tipo de dato tiene una variable, debemos observar que valor le hemos dado. Si es un valor numérico, será de tipo **number**. Si es un valor de texto, será de tipo **string**, si es verdadero o falso, será de tipo **booleano**. Veamos un ejemplo en el que identificaremos que tipo de dato tiene cada variable:

```
let s = "Hola, me llamo Manz"; // s, de string
let n = 42; // n, de número
let b = true; // b, de booleano
let u; // u, de undefined
```

Como se puede ver, en este ejemplo, es muy sencillo saber qué tipos de datos tienen cada variable.

¿Qué tipo de dato tiene una variable?

Nos encontraremos que muchas veces no resulta tan sencillo saber que tipo de dato tiene una variable, o simplemente viene oculto porque el valor lo devuelve una función o alguna otra razón similar. Hay varias formas de saber que tipo de dato tiene una variable en Javascript:

Utilizando typeof()

Si tenemos dudas, podemos utilizar la función **typeof**, que nos devuelve el tipo de dato de la variable que le pasemos por parámetro. Veamos que nos devuelve **typeof()** sobre las variables del ejemplo anterior:

```
console.log(typeof s); // "string"
console.log(typeof n); // "number"
console.log(typeof b); // "boolean"
console.log(typeof u); // "undefined"
```

Como se puede ver, mediante la función **typeof** podremos determinar que tipo de dato se esconde en una variable. Observa también que la variable **u**, al haber sido declarada sin valor, Javascript le da un tipo de dato especial: **undefined** (*sin definir*).

> La función **typeof()** solo sirve para variables con **tipos de datos** básicos o primitivos.

Utilizando constructor.name

Más adelante, nos encontraremos que en muchos casos, **typeof()** resulta insuficiente porque en tipos de datos más avanzados simplemente nos indica que son **objetos**. Con **constructor.name** podemos obtener el tipo de constructor que se utiliza, un concepto que veremos más adelante dentro del tema de clases. De momento, si lo necesitamos, podemos comprobarlo así:

```
console.log(s.constructor.name); // String
console.log(n.constructor.name); // Number
console.log(b.constructor.name); // Boolean
console.log(u.constructor.name); // ERROR, sólo funciona con variables definidas
```

> **OJO**: Sólo funciona en variables definidas (*no undefined*) y sólo en ECMAScript 6.

Que Javascript determine los **tipos de datos automáticamente** no quiere decir que debamos despreocuparnos por ello. En muchos casos, debemos conocer el tipo de dato de una variable e incluso necesitaremos convertirla a otros tipos de datos antes de usarla. Más adelante veremos formas de convertir entre tipos de datos.

En javascript es muy sencillo declarar y utilizar variables, pero aunque sea un procedimiento simple, hay que tener una serie de conceptos previos muy claros antes de continuar para evitar futuras confusiones, sobre todo si estamos acostumbrados a otros lenguajes más tradicionales.

Variables

En programación, las **variables** son espacios donde se puede guardar información y asociarla a un determinado nombre. De esta forma, cada vez que se consulte ese nombre posteriormente, te devolverá la información que contiene. La primera vez que se realiza este paso se suele llamar **inicializar una variable**.

En Javascript, si una variable no está inicializada, contendrá un valor especial: **undefined**, que significa que su valor no está definido aún, o lo que es lo mismo, que no contiene información:

```
let a; // Declaramos una variable "a", pero no le asociamos ningún contenido.
let b = 0; // Declaramos una variable de nombre "b", y le asociamos el número 0.

console.log(b); // Muestra 0 (el valor guardado en la variable "b")
console.log(a); // Muestra "undefined" (no hay valor guardado en la variable "a")
```

Como se puede observar, hemos utilizado **console.log()** para consultar la información que contienen las variables indicadas.

> **OJO**: Las mayúsculas y minúsculas en los nombres de las variables de Javascript **importan**. No es lo mismo una variable llamada **precio** que una variable llamada **Precio**, pueden contener valores diferentes.

Si tenemos que declarar muchas variables consecutivas, una buena práctica suele ser escribir sólo el primer **let** y separar por comas las diferentes variables con sus respectivos contenidos (*método 3*). Aunque se podría escribir todo en una misma línea (*método 2*), con el último método el código es mucho más fácil de leer:

```
// Método 1: Declaración de variables de forma independiente
let a = 3;
let c = 1;
let d = 2;

// Método 2: Declaración masiva de variables con el mismo var
let a = 3,
 c = 1,
 d = 2;

// Método 3: Igual al anterior, pero mejorando la legibilidad del código
let a = 3,
 c = 1,
 d = 2;
```

Como su propio nombre indica, una **variable** puede variar su contenido, ya que aunque contenga una cierta información, se puede volver a cambiar. A esta acción ya no se le llama inicializar una variable, sino **declarar una variable** (*o más concretamente, redeclarar*). En el código se puede diferenciar porque se omite el **let**:

```
let a = 40; // Inicializamos la variable "a" al valor 40.
a = 50; // Ahora, hemos declarado que pasa a contener 50 en lugar de 40.
```

Ámbitos de variables: var

Cuando inicializamos una variable al principio de nuestro programa y le asignamos un valor, ese valor generalmente está disponible a lo largo de todo el programa. Sin embargo, esto puede variar dependiendo de múltiples factores. Se conoce como **ámbito de una variable** a la zona donde esa variable sigue existiendo.

Por ejemplo, si consultamos el valor de una variable antes de inicializarla, no existe:

```
console.log(e); // Muestra "undefined", en este punto la variable "e" no existe
var e = 40;
console.log(e); // Aquí muestra 40, existe porque ya se ha inicializado anteriormente
```

En el ejemplo anterior, el ámbito de la variable **e** comienza a partir de su inicialización y "vive" hasta el final del programa. A esto se le llama **ámbito global** y es el ejemplo más sencillo. Más adelante veremos que se va complicando y a veces no resulta tan obvio saber en que ámbito se encuentra.

En el enfoque tradicional de Javascript, es decir, cuando se utiliza la palabra clave **var** para declarar variables, existen dos ámbitos principales: **ámbito global** y **ámbito a nivel de función**.

Observemos el siguiente ejemplo:

```
var a = 1;
console.log(a); // Aquí accedemos a la "a" global, que vale 1

function x() {
  console.log(a); // En esta línea el valor de "a" es undefined
  var a = 5; // Aquí creamos una variable "a" a nivel de función

  console.log(a); // Aquí el valor de "a" es 5 (a nivel de función)
  console.log(window.a); // Aquí el valor de "a" es 1 (ámbito global)
}
```

```
x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 1
```

En el ejemplo anterior vemos que el valor de **a** dentro de una función no es el **1** inicial, sino que estamos en otro ámbito diferente donde la variable **a** anterior no existe: un **ámbito a nivel de función**. Mientras estemos dentro de una función, las variables inicializadas en ella estarán en el **ámbito** de la propia función.

> **OJO**: Podemos utilizar el objeto especial **window** para acceder directamente al ámbito global independientemente de donde nos encontremos. Esto ocurre así porque las variables globales se almacenan dentro del objeto **window** (*la pestaña actual del navegador web*).

```
var a = 1;
console.log(a); // Aquí accedemos a la "a" global, que vale 1

function x() {
  console.log(a); // En esta línea el valor de "a" es 1
  a = 5; // Aquí creamos una variable "a" en el ámbito anterior

  console.log(a); // Aquí el valor de "a" es 5 (a nivel de función)
  console.log(window.a); // Aquí el valor de "a" es 5 (ámbito global)
}

x(); // Aquí se ejecuta el código de la función x()
console.log(a); // En esta línea el valor de "a" es 5
```

En este ejemplo se omite el **var** dentro de la función, y vemos que en lugar de crear una variable en el ámbito de la función, se modifica el valor de la variable **a** a nivel global. Dependiendo de donde y como accedamos a la **variable a**, obtendremos un valor u otro.

> Siempre que sea posible se debería utilizar **let** y **const** (*ver a continuación*), en lugar de **var**. Declarar variables mediante **var** se recomienda en fases de aprendizaje o en el caso de que se quiera mantener compatibilidad con navegadores muy antiguos utilizando ECMAScript 5, sin embargo, hay estrategias mejores a seguir que utilizar **var** en la actualidad.

Ámbitos de variables: let

En las versiones modernas de Javascript (*ES6 o ECMAScript 2015*) o posteriores, se introduce la palabra clave **let** en sustitución de **var**. Con ella, en lugar de utilizar los **ámbitos globales y a nivel de función** (*var*), utilizamos los ámbitos clásicos de programación: **ámbito global y ámbito local**.

La diferencia se puede ver claramente en el uso de un **bucle for** con **var** y con **let**:

```
/** Opción 1: Bucle con let **/

console.log("Antes: ", p);    // Antes: undefined
for (let p = 0; p < 3; p++)
  console.log("- ", p);       // Durante: 0, 1, 2
console.log("Después: ", p);  // Después: undefined

/** Opción 2: Bucle con var **/

console.log("Antes: ", p);    // Antes: undefined
for (var p = 0; p < 3; p++)
  console.log("- ", p);       // Durante: 0, 1, 2
console.log("Después: ", p);  // Después: 3 (WTF!)
```

Vemos que utilizando **let** la variable **p** sólo existe dentro del bucle, ámbito local, mientras que utilizando **var** la variable **p** sigue existiendo fuera del bucle, ya que debe tener un ámbito global o a nivel de función.

Constantes

De forma tradicional, Javascript no incorporaba constantes. Sin embargo, en ECMAScript 2015 (*ES6*) se añade la palabra clave **const**, que inicializada con un valor concreto, permite crear variables con valores que no pueden ser cambiados.

```
const NAME = "Manz";
console.log(NAME);
```

En el ejemplo anterior vemos un ejemplo de **const**, que funciona de forma parecida a **let**. Una buena práctica es escribir el nombre de la constante en mayúsculas, para identificar rápidamente que se trata de una constante y no una variable, cuando leemos código ajeno.

> Realmente, las **constantes** de Javascript son variables inicializadas a un valor específico y que no pueden redeclararse. No confundir con valores inmutables, ya que como veremos posteriormente, los objetos si pueden ser modificados aún siendo constantes.

Math

Cuando trabajamos con Javascript, es posible realizar gran cantidad de **operaciones matemáticas** de forma nativa, sin necesidad de librerías externas. Para ello, haremos uso del objeto **Math**, un objeto interno de Javascript que tiene incorporadas ciertas constantes y métodos (*funciones*) para trabajar matemáticamente.

Constantes de Math

El objeto **Math** de Javascript incorpora varias constantes que podemos necesitar en algunas operaciones matemáticas. Veamos su significado y valor aproximado:

Constante	Descripción	Valor
Math.E	Número de Euler	**2.718281828459045**
Math.LN2	Logaritmo natural en base 2	**0.6931471805599453**
Math.LN10	Logaritmo decimal	**2.302585092994046**

Math.LOG2E	Logaritmo base 2 de E	**1.4426950408889634**
Math.LOG10E	Logaritmo base 10 de E	**0.4342944819032518**
Math.PI	Número PI o **Π**	**3.141592653589793**
Math.SQRT1_2	Raíz cuadrada de 1/2	**0.7071067811865476**
Math.SQRT2	Raíz cuadrada de 2	**1.4142135623730951**

Además de estas constantes, el objeto **Math** también nos proporciona gran cantidad de métodos o funciones para trabajar con números. Vamos a analizarlos.

Métodos matemáticos

Los siguientes métodos matemáticos están disponibles en Javascript a través del objeto **Math**. Observa que algunos de ellos sólo están disponibles en **ECMAScript 6**:

Método	Descripción	Ejemplo

Math.abs(x)	Devuelve el valor absoluto de **x**.	**|x|**
Math.sign(x)	Devuelve el signo del número: **1** positivo, **-1** negativo	
Math.exp(x)	Exponenciación. Devuelve el número **e** elevado a **x**.	e^x
Math.expm1(x)	Equivalente a **Math.exp(x) - 1**.	e^x-1
Math.max(a, b, c...)	Devuelve el número más grande de los indicados por parámetro.	
Math.min(a, b, c...)	Devuelve el número más pequeño de los indicados por parámetro.	
Math.pow(base, exp)	Potenciación. Devuelve el número **base** elevado a **exp**.	$base^{exp}$
Math.sqrt(x)	Devuelve la raíz cuadrada de **x**.	$\sqrt{x}$
Math.cbrt(x)	Devuelve la raíz cúbica de **x**.	$\sqrt[3]{x}$

Math.imul(a, b)	Equivalente a **a * b**, pero a nivel de bits.	
Math.clz32(x)	Devuelve el número de ceros a la izquierda de **x** en binario (32 bits).	

Veamos algunos ejemplos aplicados a las mencionadas funciones anteriormente:

```
Math.abs(-5); // 5
Math.sign(-5); // -1
Math.exp(1); // e, o sea, 2.718281828459045
Math.expm1(1); // 1.718281828459045
Math.max(1, 40, 5, 15); // 40
Math.min(5, 10, -2, 0); // -2
Math.pow(2, 10); // 1024
Math.sqrt(2); // 1.4142135623730951
Math.cbrt(2); // 1.2599210498948732
Math.imul(0xffffffff, 7); // -7
```

Existe uno más, **Math.random()** que merece una explicación más detallada, por lo que lo explicamos en el apartado siguiente.

Método Math.random()

Uno de los métodos más útiles e interesantes del objeto **Math** es **Math.random()**.

Método	Descripción	Ejemplo

Math.random()	Devuelve un número al azar entre **0** y **1** con 16 decimales.	

Este método nos da un número al azar entre los valores **0** y **1**, con 16 decimales. Normalmente, cuando queremos trabajar con números aleatorios, lo que buscamos es obtener un número entero al azar entre **a** y **b**. Para ello, se suele hacer lo siguiente:

```
// Obtenemos un número al azar entre [0, 1) con 16 decimales
let x = Math.random();

// Multiplicamos dicho número por el valor máximo que buscamos (5)
x = x * 5;

// Redondeamos inferiormente, quedándonos sólo con la parte entera
x = Math.floor(x);
```

Este ejemplo nos dará en **x** un valor al azar entre **0** y **5** (*5 no incluido*). Lo hemos realizado por pasos para entenderlo mejor, pero podemos realizarlo directamente como se ve en el siguiente ejemplo:

```
// Número al azar entre 0 y 5 (no incluido)
const x = Math.floor(Math.random() * 5);
```

Métodos de logaritmos

Javascript incorpora varios métodos en el objeto **Math** para trabajar con logaritmos. Desde **logaritmos neperianos** hasta **logaritmos binarios** a través de las siguientes funciones:

Método	Descripción y Ejemplo

Math.log(x)	Devuelve el logaritmo natural en base **e** de **x**. **Ej: $\log_e x$** o **ln x**
Math.log10(x)	Devuelve el logaritmo decimal (en base 10) de **x**. **Ej: $\log_{10} x$** ó **log x**
Math.log2(x)	Devuelve el logaritmo binario (en base 2) de **x**. **Ej: $\log_2 x$**
Math.log1p(x)	Devuelve el logaritmo natural de **(1+x)**. **Ej: $\log_e (1+x)$** o **ln (1+x)**

A continuación, unos ejemplos de estas funciones aplicadas:

```
Math.log(2); // 0.6931471805599453
Math.log10(2); // 0.3010299956639812
Math.log2(2); // 1
Math.log1p(2); // 1.0986122886681096
```

Métodos de redondeo

Como hemos visto anteriormente, es muy común necesitar métodos para **redondear números** y reducir el número de decimales o aproximar a una cifra concreta. Para ello, de forma nativa, Javascript proporciona los siguientes métodos de redondeo:

Método	Descripción

Math.round(x)	Devuelve el redondeo de **x** (*el entero más cercano*)
Math.ceil(x)	Devuelve el redondeo superior de **x**. (*el entero más alto*)
Math.floor(x)	Devuelve el redondeo inferior de **x**. (*el entero más bajo*)
Math.fround(x)	Devuelve el redondeo de **x** (*flotante con precisión simple*)
Math.trunc(x)	Trunca el número **x** (*devuelve sólo la parte entera*)

Veamos las diferencias de utilizar los diferentes **métodos** anteriores para redondear un número decimal y los resultados obtenidos:

```
// Redondeo natural, el más cercano
Math.round(3.75); // 4
Math.round(3.25); // 3

// Redondeo superior (el más alto)
Math.ceil(3.75); // 4
Math.ceil(3.25); // 4

// Redondeo inferior (el más bajo)
Math.floor(3.75); // 3
Math.floor(3.25); // 3

// Redondeo con precisión
Math.round(3.123456789); // 3
Math.fround(3.123456789); // 3.1234567165374756

// Truncado (sólo parte entera)
```

```
Math.trunc(3.75); // 3
Math.round(-3.75); // -4
Math.trunc(-3.75); // -3
```

Métodos trigonométricos

Por último, y no por ello menos importante, el objeto **Math** nos proporciona de forma nativa una serie de métodos trigonométricos, que nos permiten hacer cálculos con operaciones como **seno**, **coseno**, **tangente** y relacionados:

Método	Descripción
Math.sin(x)	Seno de **x**
Math.asin(x)	Arcoseno de **x**
Math.sinh(x)	Seno hiperbólico de **x**
Math.asinh(x)	Arcoseno hiperbólico de **x**
Math.cos(x)	Coseno de **x**
Math.acos(x)	Arcocoseno de **x**

Math.cosh(x)	Coseno hiperbólico de **x**
Math.acosh(x)	Arcocoseno hiperbólico de **x**
Math.tan(x)	Tangente de **x**
Math.atan(x)	Arcotangente de **x**
Math.tanh(x)	Tangente hiperbólica de **x**
Math.atanh(x)	Arcotangente hiperbólica de **x**
Math.atan2(x, y)	Arcotangente del conciente de **x/y**
Math.hypot(a, b..)	Devuelve la raíz cuadrada de a^2 + b^2 + ...