¿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 qué 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 cómo «trabaja un ordenador». Programar no es más que decirle a una máquina qué cosas debe hacer y cómo 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.

Dicho esto, es necesario tener presente que **aprender a programar** es una tarea que no ocurre de un día para otro. Requiere tiempo, esfuerzo, acostumbrarse a cambiar la forma de pensar y practicar mucho.

- Puedes **copiar un programa** en segundos, pero eso no significa que lo entiendas.
- Puedes **comprender un programa** en minutos, pero eso no significa que lo puedas crear.
- Puedes **crear un programa** en horas, pero eso no significa que sepas programar.
- Puedes **aprender a programar** en semanas, pero eso no significa que no cometas errores.
- Puedes **aprender a programar bien y sin demasiados errores** en meses.

Pero **dominar la programación** es una tarea que requiere años.

Fuente: https://lenguajejs.com/javascript/introduccion/que-es-javascript/

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
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/ 201 5	ECMAScript 2015 (ES6)	Clases, módulos, generadores, hashmaps, sets, for of, proxies...
7	JUN/ 201 6	ECMAScript 2016	Array includes(), Exponenciación **
8	JUN/ 201 7	ECMAScript 2017	Async/await

9	JUN/ 201 8	ECMAScript 2018	Rest/Spread operator, Promise.finally()...
10	JUN/ 201 9	ECMAScript 2019	Flat functions, trimStart(), errores opcionales en catch...
11	JUN/ 202 0	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** https://kangax.github.io/compat-table/es6/. 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 como 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, empresas que desarrollan aplicaciones **SPA** 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 soportan 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** https://babeljs.io/ 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 implementa.

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.

La consola

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, separándolas por comas:

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

De momento nos puede parecer algo inútil, pero cuando empecemos a trabajar con variables y objetos, será muy necesario.

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 Codo a Codo!",

"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.

¿Cómo funciona Javascript?

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 descargara 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 https://caniuse.com/#search=defer*) que no soportan **defer**.

Tienes más información sobre **etiquetas <script>** https://lenguajehtml.com/html/scripting/etiquetas-html-scripts en la página de **LenguajeHTML**.

Conceptos Básicos

Si no has programado hasta ahora, debes conocer una serie de conceptos básicos que tendrás que trabajar y dominar dentro del campo de la programación.

Glosario general

Programa: En programación se suele llamar «programa» a el conjunto total de código que desarrollamos. En Javascript, quizás el término más utilizado es **aplicación web** (*cuando es un desarrollo con mucha cantidad de Javascript*). También se suelen generalizar utilizando términos como «script» o «código Javascript».

Algoritmo: Un algoritmo es un conjunto de pasos conocidos, en un determinado orden, para conseguir realizar una tarea satisfactoriamente y lograr un objetivo.

Comentarios: Los comentarios en nuestro código son fragmentos de texto o anotaciones que el navegador ignora y no repercuten en el programa. Sirven para dejar por escrito detalles importantes para el programador. De esta forma cuando volvamos al código, nos será más rápido comprenderlo. Es una buena costumbre comentar en la medida de lo posible nuestro código.

Indentación: Se llama **indentar** a la acción de colocar espacios o tabuladores antes del código, para indicar si nos encontramos dentro de un **if**, de un **bucle**, etc... Esta práctica es muy importante y necesaria, y más adelante profundizaremos en ella.

Variables: Es el nombre genérico que se le da a pequeños espacios de memoria donde guardás una información determinada, de forma muy similar a las incógnitas en matemáticas. Un programa puede tener muchas variables, y cada una de ellas tendrá un **nombre**, un **valor** y un **tipo de dato**. El nombre se utiliza para diferenciarlas unas de otras y hacer referencia a ellas, el valor es la información que contienen y el tipo de dato es la naturaleza de ese valor. Se llaman variables porque podemos cambiar su valor a lo largo del programa, según necesitemos.

x = 5; // nombre: x, valor: 5, tipo de dato: número

y = "Hola"; // nombre: y, valor: Hola, tipo de dato: texto

Manu = "me"; // nombre: Manu, valor: me, tipo de dato: texto

Constantes: Es el mismo concepto de una variable, salvo que en este caso, la información que contiene es siempre la misma (*no puede variar*).

Funciones: Cuando comenzamos a programar, nuestro código se va haciendo cada vez más y más grande, por lo que hay que buscar formas de organizarlo y mantenerlo lo más simple posible. Las funciones son **agrupaciones** de código que, entre otras cosas, evitan que tengamos que escribir varias veces lo mismo en nuestro código. Una función contendrá una o más acciones a realizar y cada vez que ejecutemos una función, se realizarán todas ellas.

Parámetros: Es el nombre que reciben las variables que se le pasan a las funciones. Muchas veces también se les denomina **argumentos**.

Bucles: Cuando estamos programando, muchas veces necesitaremos realizar tareas repetitivas. Una de las ventajas de la programación es que permite automatizar acciones y no es necesario hacerlas varias veces. Los bucles permiten indicar el número de veces que se repetirá una acción. De esta forma, sólo la escribimos una vez en nuestro código, y simplemente indicamos el número de veces que queremos que se repita.

Iteración: Cuando el programa está en un bucle repitiendo varias veces la misma tarea, cada una de esas repeticiones se denomina **iteración**.

Librería: Muchas veces, desarrollamos código que resuelve tareas o problemas que, posteriormente, queremos reutilizar en otros programas. Cuando eso ocurre, en Javascript se suele empaquetar el código en lo que se llaman **librerías**, que no es más que código listo para que otros programadores puedan utilizarlo fácilmente en sus programas y beneficiarse de las tareas que resuelven de forma muy sencilla.

Comentarios

Cuando comenzamos a programar, por lo general, se nos suele decir que es una buena práctica mantener **comentado** nuestro código con anotaciones que faciliten la comprensión de las tareas que realizamos y los problemas que pretendemos solucionar, ya que el código que creamos no suele ser muy bueno, ni mucho menos descriptivo, ya que estamos en fase de aprendizaje.

A medida que conseguimos destreza programando, notaremos que los comentarios son cada vez más prescindibles, sin embargo, conviene no dejar de comentar, sino en su lugar, **aprender a comentar mejor**.

Una serie de consejos a tener presentes a la hora de dejar comentarios en nuestro código:

No comentes detalles **redundantes**. No escribas lo que haces, escribe **por qué** lo haces.

Mejor nombres de variables/funciones/clases descriptivas que comentarios descriptivos.

Sé **conciso y concreto**. Resume. No escribas párrafos si no es absolutamente necesario.

Intenta usar siempre el mismo **idioma** y **estilo** de comentarios.

Con el tiempo, los comentarios no se suelen mantener (*modificar*), el código sí.

Tipos de comentarios

En Javascript existen dos tipos de comentarios: los **comentarios de una sola línea**

y los comentarios de múltiples líneas.

El primero de ellos se caracteriza porque comienza con // y sólo comenta la **línea actual** desde donde se escribe.

El segundo tipo se utiliza para hacer comentarios extensos que ocupan **varias líneas**. Comienza por /* y comentará todo el texto que escribamos hasta que cerremos el comentario con un */.

Veamos un ejemplo:

// Comentarios cortos de una sola línea. Suelen explicar la línea siguiente. var a = 1;

var x = 45; // También se utilizan al final de una línea.

/* Por otro lado, existen los comentarios múltiples de varias líneas consecutivas.

Suelen utilizarse para explicaciones largas que requieren bastante espacio porque

se mencionan gran cantidad de cosas :-) */ **Ejemplos**

Comentar código también es un arte que debe ser aprendido, ya que al principio es muy fácil **cometer errores** y comentar en exceso o no ser concreto al comentar. No suele ser grave porque los comentarios no afectan al funcionamiento del programa, pero en equipos de trabajo donde hay varios programadores suele ser molesto para los programadores con más experiencia.

Un ejemplo de comentario que suele ser contraproducente es aquel que se limita a decir lo que hacemos en la línea siguiente:

// Declaramos una variable llamada x var x = 50;

// La mostramos por consola

console.log(x);

// Cambiamos su valor multiplicando por 0,5 x = x * 0.5;

Estos comentarios pueden ser útiles para el **programador novato** que comienza a programar y necesita recordar lo que hace porque aún no conoce bien la sintaxis de programación, de hecho muchos de los comentarios del tema de introducción son así (***para ayudar al programador que recién empieza a programar***), pero el objetivo real de un comentario **no debe ser recordar que hace una línea de código**, sino conocer porque lo estamos realizando o que representa lo que estamos haciendo:

var x = 50; // Establecemos el precio del producto console.log(x);

x = x * 0.5; // Lo rebajamos al 50%

Sin embargo, hay una opción todavía mejor que conecta con uno de los temas que veremos más adelante. Poner **nombres descriptivos** a las variables debería ser algo obligatorio a lo que acostumbrarnos, puesto que puede ahorrarnos muchos comentarios y tiempo, simplificar el código considerablemente y hacerlo mucho más legible y con menos ambigüedades:

var precio = 50; console.log(precio);

var oferta = precio * 0.5;

En este fragmento de código, no utilizamos comentarios porque el nombre de las variables ya ayuda a entender el código y lo hace **autoexplicativo**. De esta forma, generamos menos código (*e incluso comentarios*) y se entiende igualmente. En los siguientes temas, veremos una serie de consejos a la hora de nombrar variables, funciones u otros elementos dentro de la programación.

Tipo de Datos

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 qué 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 https://www.typescriptlang.org/, 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, débilmente 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 identificamos que tipo de dato tiene cada variable:

var s = "Hola, me llamo Manu"; // s, de string var n = 42;

// n, de número

var b = true; // b, de booleano var 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 qué 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 qué 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 debemos de preocuparnos 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.

Variables y Constantes

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:

var a; // Declaramos una variable "a", pero no le asociamos ningún contenido. var 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 **var** 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 var a = 3;

var c = 1; var d = 2;

// Método 2: Declaración masiva de variables con el mismo var var a = 3,

c = 1,

d = 2;

// Método 3: Igual al anterior, pero mejorando la legibilidad del código var 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 **var**:

```
var 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 qué 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 dónde y cómo accedemos 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 = "Manu";

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 qué 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 volver a declararse. No confundir con valores inmutables, ya que como veremos posteriormente, los objetos sí pueden ser modificados aún siendo constantes.

Objetos Básicos

Uno de los aspectos más importantes del lenguaje Javascript es el concepto de **objeto**, puesto que prácticamente todo lo que utilizamos en Javascript, son objetos. Sin embargo, tiene ligeras diferencias con los objetos de otros lenguajes de programación, así que vamos a comenzar con una explicación sencilla y más adelante ampliaremos este tema en profundidad.

¿Qué son los objetos?

En Javascript, existe un tipo de dato llamado **objeto**. No es más que una variable especial que puede contener más variables en su interior. De esta forma, tenemos la posibilidad de organizar múltiples variables de la misma temática dentro de un objeto. Veamos algunos ejemplos:

En muchos lenguajes de programación, para crear un objeto se utiliza la palabra clave **new**. En Javascript también se puede hacer:

```
const objeto = new Object(); // Esto es un objeto «genérico» vacío
```

Sin embargo, siempre que podamos, en Javascript se prefiere utilizar lo que se llaman los **literales**, un método abreviado para crear objetos directamente, sin necesidad de utilizar la palabra **new**.

Declaración de un objeto

Los literales de los objetos en Javascript son las llaves **{}**. Este ejemplo es equivalente al anterior, pero es más corto, rápido y cómodo, por lo que se aconseja declararlos así:

```
const objeto = {}; // Esto es un objeto vacío
```

Pero hasta ahora, solo hemos creado un objeto vacío. Vamos a crear un nuevo objeto, que contenga variables con información en su interior:

```
// Declaración del objeto const

player = {

name: "Manu", life: 99,

strength: 10,
```

};

Estas variables dentro de los objetos se suelen denominar **propiedades**. Como se puede ver, un objeto en Javascript nos permite encapsular en su interior información relacionada, para posteriormente poder acceder a ella de forma más sencilla e intuitiva.

Acceso a sus propiedades

Una vez tengamos un objeto, podemos acceder a sus propiedades de dos formas diferentes: a través de la notación con **puntos** o a través de la notación con **corchetes**.

// Notación con puntos console.log(player.name); //

Muestra "Manu" console.log(player.life); // Muestra 99

// Notación con corchetes console.log(player["name"]); // Muestra

"Manu" console.log(player["life"]); // Muestra 99

El programador puede utilizar la notación que más le guste. La más utilizada en Javascript suele ser la **notación con puntos**, mientras que la notación con corchetes se suele conocer en otros lenguajes como «arrays asociativos».

A algunos programadores puede resultar confuso utilizar objetos con la notación de corchetes, ya que en otros lenguajes de programación los objetos y los arrays asociativos son cosas diferentes, y en Javascript ambos conceptos se mezclan.

Hay ciertos casos en los que sólo se puede utilizar la **notación con corchetes**, como por ejemplo cuando se utilizan espacios en el nombre de la propiedad. Es imposible hacerlo con la notación con puntos.

Añadir propiedades

También podemos añadir **propiedades** al **objeto** después de haberlo creado, aunque la sintaxis cambia ligeramente. Veamos un ejemplo equivalente al anterior:

```
// Declaración del objeto const

player = {};


// Añadimos mediante notación con puntos player.name =

"Manu";

player.life = 99;

player.strength = 10;


// Añadimos mediante notación con corchetes player["name"] =

"Manu";

player["life"] = 99;

player["strength"] = 10;
```

Las **propiedades** del objeto pueden ser utilizadas como variables. De hecho, utilizar los objetos como elementos para organizar múltiples variables suele ser una buena práctica en Javascript.

Tipos de objetos

Hasta ahora, solo hemos visto los objetos «genéricos», en Javascript conocidos como tipo, declarándolos con un **new Object()** o con un literal **{}**, dos formas equivalentes de hacer lo mismo. Al generar una variable de tipo , esa variable

«hereda» una serie de métodos (*del objeto **Object** en este caso*).

```
const o = {};

o.toString(); // Devuelve '[object Object]' (Un objeto de tipo Object)
```

En este ejemplo, **toString()** es uno de esos métodos que tienen todas las variables de tipo . Sin embargo, hasta ahora y sin saberlo, cuando creamos una variable de un determinado tipo de dato (*sea primitivo o no*), es también de tipo , ya que todas las variables heredan de este tipo de dato. Por lo tanto, nuestra variable tendrá no sólo los métodos de su tipo de dato, sino también los métodos heredados de :

```
const s = "hola";

s.toString(); // Devuelve 'hola'
```

Más adelante, veremos los métodos que heredan las variables de tipo y comprobaremos que los objetos tienen detrás de sí muchos más conceptos que los que hemos visto hasta ahora y que su definición es mucho más amplia.

Objeto Number

En Javascript crear variables numéricas es muy sencillo, pero hay que conocer bien cómo trabajar con ellas y los diferentes métodos de los que dispone.

¿Qué es una variable numérica?

En Javascript, los **números** son uno de los tipos de datos básicos (*tipos primitivos*) que para crearlos, simplemente basta con escribirlos. No obstante, en Javascript todo son objetos, como veremos más adelante, y también se pueden declarar como si fueran un objeto:

Constructor	Descripción
new Number(n)	Crea un objeto numérico a partir del número n pasado por parámetro.
n	Simplemente, el número en cuestión. Notación preferida.

Sin embargo, aunque existan varias formas de declararlos, no se suele utilizar la notación **new** con objetos primitivos ya que es bastante más tedioso y complicado que utilizar la notación de literales:

```
// Literales const n1 = 4;

const n2 = 15.8;



// Objetos

const n1 = new Number(4); const n2 =

new Number(15.8);
```

Cualquier parámetro pasado al **new Number()** que no sea un número, dará como resultado un valor **NaN** (*ver más adelante*).

Constantes numéricas

Existe una serie de constantes definidas en relación a las variables numéricas. La mayoría de ellas establecen límites máximos y mínimos, veamos su significado:

Constante	Valor en Javascript	Descripción
Number.POSITIVE_INFINITY	Infinity	Infinito positivo: +∞
Number.NEGATIVE_INFINITY	-Infinity	Infinito negativo: -∞
Number.MAX_VALUE	1.7976931348623157e+308	Valor más grande
Number.MIN_VALUE	5e-324	Valor más pequeño
Number.MAX_SAFE_INTEGER	9007199254740991	Valor seguro más grande
Number.MIN_SAFE_INTEGER	-9007199254740991	Valor seguro más pequeño
Number.EPSILON	2^{-52}	Número muy pequeño: ε
Number.NaN	NaN	Not A Number

La diferencia entre **Number.MAX_VALUE** y **Number.MAX_SAFE_INTEGER** es que, el primero es el **valor máximo** que es posible representar en Javascript. Por otro lado, el segundo es el **valor máximo** para realizar cálculos con seguridad en Javascript.

Los lenguajes de programación están sujetos a la precisión numérica debido a la forma interna en la que guardan valores numéricos. Si necesitamos realizar operaciones con muy alta precisión numérica en Javascript, se recomienda utilizar librerías como decimal.js o bigNumber.js.

NaN (Not A Number)

El acrónimo **NaN** es un valor especial de Javascript que significa **Not A Number** (*No es un número*). Este valor se usa para representar valores imposibles o indeterminados, como por ejemplo, resultados matemáticos de operaciones como:

0 / 0 (Indeterminaciones)

4 - 'a' (Valores imposibles)

NaN + 4 (Operaciones con **NaN** como operando)

Este valor se utiliza habitualmente para detectar si una operación ha fallado o ha dado un valor no representable. Sin embargo, no podemos compararlo literalmente con **NaN**, sino que tenemos que usar la función **Number.isNaN()**:

let num = NaN;

// La siguiente operación, contra toda lógica, es falsa num == NaN;

// false

// Se debe usar Number.isNaN() para comprobar si el valor es NaN Number.isNaN(num); // true

// Si comprobamos el tipo de dato de NaN, nos dirá que es numérico typeof num; //

number

Como se puede ver en la última línea del ejemplo anterior, mencionar que en Javascript, si comprobamos el tipo de dato de **NaN** con **typeof** nos dirá que es un número. Puede parecer ilógico que **Not A Number** sea un número, esto ocurre porque **NaN** está en un contexto numérico.

En otras palabras, dentro de los tipos de datos numéricos, **NaN** es un conjunto de números que no se pueden representar.

Comprobaciones numéricas

En Javascript tenemos varias funciones para conocer la naturaleza de una variable numérica (número finito, número entero, número seguro o si no es representable como un número). Las podemos ver a continuación en la siguiente tabla:

Método	Descripción
Number.isFinite(n)	Comprueba si n es un número finito.
Number.isInteger(n)	Comprueba si n es un número entero.
Number.isSafeInteger(n)	Comprueba si n es un número seguro.
Number.isNaN(n)	Comprueba si n no es un número.

Ten en cuenta que estas funciones devuelven un booleano (*valor verdadero o falso*), lo que lo hace ideales para usarlas como condiciones en bucles o condicionales. A continuación veamos dos ejemplos para cada una de estas funciones:

```
// ¿Número finito? Number.isFinite(42);

// true

Number.isFinite(Infinity); // false, es infinito
```

```
// ¿Número entero?
Number.isInteger(5); // true
Number.isInteger(4.6); // false, es decimal

// ¿Número seguro? Number.isSafeInteger(1e15);  //
true
Number.isSafeInteger(1e16); // false, es un valor no seguro

// ¿No es un número? Number.isNaN(NaN); // true
Number.isNaN(5); // false, es un número
```

Recuerda no hacer comprobaciones directas con NaN, sino utilizar la función

Number.isNaN().

Conversión numérica

En muchos casos tendremos variables de texto que nos interesa convertir a número, para realizar operaciones posteriormente con ellas. Para ello, lo ideal es utilizar las funciones de parseo numérico, **parseInt()** y **parseFloat()**. Veamos cuales son y cómo se pueden utilizar:

Método	Descripción
Number.parseInt(s)	Convierte una cadena de texto s en un número entero.
Number.parseInt(s, radix)	Idem al anterior, pero desde una base radix.
Number.parseFloat(s)	Convierte una cadena de texto s en un número decimal.
Number.parseFloat(s, radix)	Idem al anterior, pero desde una base radix.

Para ilustrar esto, veamos un ejemplo con **parseInt()** cuando solo le pasamos un parámetro (*un texto*) que queremos convertir a número:

```
Number.parseInt("42"); // 42

Number.parseInt("42€"); // 42

Number.parseInt("Núm. 42"); // NaN

Number.parseInt("A"); // NaN
```

Nota que la función **parseInt()** funciona perfectamente para variables de texto que contienen números o que empiezan por números. Esto es muy útil para eliminar unidades de variables de texto. Sin embargo, si la variable de texto comienza por un valor que no es numérico, **parseInt()** devolverá un **NaN**.

Si lo que queremos es quedarnos con el número que aparece más adelante en la variable de texto, habrá que manipular ese texto con alguna de las funciones que veremos en el apartado de variables de texto.

Veamos ahora que ocurre si utilizamos **parseInt()** con dos parámetros, donde el primero es el **texto con el número** y el segundo es la **base numérica** del número:

Number.parseInt("11101", 2); // 29 en binario

Number.parseInt("31", 8); // 25 en octal

Number.parseInt("FF", 16); // 255 en hexadecimal

Esta modalidad de **parseInt()** se suele utilizar cuando queremos pasar a base decimal un número que se encuentra en otra base (*binaria, octal, hexadecimal...*).

Al igual que con **parseInt()** tenemos otra función llamada **parseFloat()**. Funciona exactamente igual a la primera, sólo que la primera está específicamente diseñada para utilizar con números enteros y la segunda para números decimales. Si utilizamos **parseInt()** con un número decimal, nos quedaremos sólo con la parte entera, mientras que **parseFloat()** la conservará.

Representación numérica

Por último, en el caso de querer cambiar el tipo de representación numérica, podemos utilizar las siguientes funciones para alternar entre **exponencial** y punto fijo:

Método	Descripción
.toExponential(n)	Convierte el número a notación exponencial con n decimales.
.toFixed(n)	Convierte el número a notación de punto fijo con n decimales.
.toPrecision(p)	Utiliza p dígitos de precisión en el número.

Observemos el siguiente ejemplo aplicando las funciones anteriores al número decimal **1.5**:

(1.5).toExponential(2); // "1.50e+0" en exponencial (1.5).toFixed(2);

// "1.50" en punto fijo (1.5).toPrecision(1); // "2"

Objeto 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

// Ejemplo de clz32 (count leading zeros) const x = 1;

"0".repeat(Math.clz32(x)) + x.toString(2);

// Devuelve "00000000000000000000000000000001"

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);



// Equivalente al anterior
const x = ~~(Math.random() * 5);
```

Como se puede ver en el segundo ejemplo anterior, utilizamos el operador a nivel de bits **~~** (*doble negación*) como reemplazo rápido de **Math.floor()**, una función que realiza un redondeo inferior, y que veremos al final de este tema.

Si lo deseas, puedes utilizar librerías específicas para generar números aleatorios como random.js o chance.js, esta última permitiendo incluso generar otros tipos de datos aleatorios como textos, GUIDs o colores hexadecimales.

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 $a^2 + b^2 + ...$

Otras librerías matemáticas

Si de forma nativa no encuentras una forma sencilla de resolver el problema matemático que tienes entre manos, no olvides que existen una serie de **librerías de terceros** que pueden hacernos la vida más fácil a la hora de trabajar con otros valores matemáticos.

A continuación, detallamos algunas de ellas:

Librería	Descripción	GitHub
Math.js	Librería matemática de propósito general.	josdejong/mathjs
Fraction.js	Librería matemática para trabajar con fracciones.	infusion/Fraction.js
Polynomial. js	Librería matemática para trabajar con polinomios.	infusion/Polynomial.js
Complex.js	Librería matemática para trabajar con números complejos.	infusion/Complex.js
Angles.js	Librería matemática para trabajar con ángulos.	infusion/Angles.js
BitSet.js	Librería matemática para trabajar con vectores de bits.	infusion/BitSet.js

Habrás comprobado que, al contrario que muchos otros objetos de Javascript, en estas ocasiones hemos indicado explícitamente el objeto, por ejemplo **Math.round(numero)**, en lugar de hacerlo

sobre la variable: **numero.round()**. Esto ocurre porque **Math** es un objeto con métodos y constantes **estáticas**, algo que veremos en profundidad en futuros temas.

Operadores en JavaScript

Operador de Asignación

El operador de asignación (=) asigna un valor a una variable. var x = 10;

Operadores Aritméticos

Los operadores aritméticos se utilizan para realizar operaciones aritméticas en números:

Operator	Description
+	Suma
-	Resta
*	Multiplicación
**	Exponenciación
/	División
%	Módulo: resto de dividir
++	Incremento

--	Decremento

Operadores de cadena

El operador + también se puede usar para agregar (concatenar) cadenas.

Ejemplo

```
var txt1 = "John";
var txt2 = "Doe";

var txt3 = txt1 + " " + txt2; El resultado
```

de txt3 será:

John Doe

El +=operador de asignación también se puede usar para agregar (concatenar) cadenas:

Ejemplo

```
var txt1 = "What a very "; txt1 +=
"nice day";
```

El resultado de txt1 será:

What a very nice day

Cuando se usa en cadenas, el operador + se denomina operador de concatenación.

Agregar cadenas y números

Agregar dos números devolverá la suma, pero agregar un número y una cadena devolverá una cadena:

Ejemplo

```
var x = 5 + 5;

var y = "5" + 5;

var z = "Hello" + 5;
```

El resultado de x , y y z será:

10

55

Hello5

Si agrega un número y una cadena, ¡el resultado será una cadena!

Operadores de comparación.

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

Operadores lógicos

Operator	Description
&&	logical and
||	logical or
!	logical not

Operadores de tipo

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Operadores de bit a bit

Los operadores de bits funcionan con números de 32 bits.

Cualquier operando numérico de la operación se convierte en un número de 32 bits. El resultado se convierte de nuevo a un número de JavaScript.

Operator	Description	Example	Same as	Result	Decimal
&	AND	5 & 1	0101 & 0001	0001	1
|	OR	5 | 1	0101 | 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	Zero fill left shift	5 << 1	0101 << 1	1010	10
>>	Signed right shift	5 >> 1	0101 >> 1	0010	2
>>>	Zero fill right shift	5 >>> 1	0101 >>> 1	0010	2

Estructuras de Control

Cuando escribimos código Javascript, por defecto, el navegador leerá el script de forma secuencial, es decir, una línea detrás de otra, desde arriba hacia abajo. Por lo tanto, una acción que realicemos en la línea 5 nunca ocurrirá antes que una que aparece en la línea 3. Ya veremos que más adelante esto se complica, pero en principio partimos de esa base.

Condicionales

Al hacer un programa necesitaremos establecer condiciones o decisiones, donde buscamos que el navegador realice una acción A si se cumple una condición o una acción B si no se cumple. Este es el

primer tipo de estructuras de control que encontraremos. Para ello existen varias estructuras de control:

Estructura de control	Descripción
If	Condición simple: Si ocurre algo, haz lo siguiente...
If/else	Condición con alternativa: Si ocurre algo, haz esto, sino, haz lo esto otro...
?:	Operador ternario: Equivalente a If/else, método abreviado.
Switch	Estructura para casos específicos: Similar a varios If/else anidados.

Condicional If

Quizás, el más conocido de estos mecanismos de estructura de control es el if (*condicional*). Con él podemos indicar en el programa que se tome un camino sólo si se cumple la condición que establezcamos:

```
var nota = 7;

console.log("He realizado mi examen.");



// Condición (si nota es mayor o igual a 5) if (nota >=

5) {

console.log("¡Estoy aprobado!");
```

}

En este caso, como el valor de nota es superior a 5, nos aparecerá en la consola el mensaje «¡Estoy aprobado!». Sin embargo, si modificamos en la primera línea el valor de nota a un valor inferior a 5, no nos aparecerá ese mensaje.

Cuando dentro de las llaves (*{ }*) sólo tenemos una línea, se pueden omitir dichas llaves. Aún así, es recomendable ponerlas siempre si tenemos dudas o no estamos seguros.

Condicional If / else

Pero se puede dar el caso que queramos establecer una alternativa a una condición. Para eso utilizamos el if seguido de un else. Con esto podemos establecer una acción A si se cumple la condición, y una acción B si no se cumple.

Vamos a modificar el ejemplo anterior para mostrar también un mensaje cuando estamos suspendidos, pero en este caso, en lugar de mostrar el mensaje directamente con un console.log vamos a guardar ese texto en una nueva variable calificacion:

```
var nota = 7;

console.log("He realizado mi examen. Mi resultado es el siguiente:");



// Condición if (nota <

5) {
```

// Acción A (nota es menor que 5)

calificacion = "suspendido";

} else {

// Acción B: Cualquier otro caso a A (nota es mayor o igual que 5) calificacion =

"aprobado";

}

console.log("Estoy", calificacion);

Nuevamente, en este ejemplo comprobaremos que podemos conseguir que se muestre el mensaje Estoy aprobado o Estoy suspendido dependiendo del valor que tenga la variable nota. La diferencia con el ejemplo anterior es que creamos una nueva variable que contendrá un valor determinado dependiendo de la condición del If.

Por último, el console.log del final, muestra el contenido de la variable calificacion, independientemente de que sea el primer caso o el segundo.

var nota = 7;

console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Condición if (nota <

5) {

```
// Acción A (nota es menor que 5)

calificacion = "suspendido";

}

if (nota >= 5) {


//Acción B (nota es mayor o igual que 5) calificacion =

"aprobado";

}


console.log("Estoy", calificacion);
```

Este nuevo ejemplo, es equivalente al ejemplo anterior. Si nos fijamos bien, la única diferencia respecto al anterior es que estamos realizando dos if independientes: uno para comprobar si está suspendido y otro para comprobar si está aprobado.

Pero aunque son equivalentes, no son exactamente iguales, ya que en el ejemplo que vimos anteriormente sólo existe un if, y por lo tanto, sólo se realiza una comprobación. En este ejemplo que vemos ahora, se realizan dos if, y por lo tanto, dos comprobaciones.

En este caso se trata de algo insignificante, pero es importante darse cuenta de que el primer ejemplo estaría realizando menos tareas para conseguir un mismo resultado, ergo, el primer ejemplo sería más eficiente.

Operador ternario

El operador ternario es una alternativa de condicional if/else de una forma mucho más corta y, en muchos casos, más legible. Vamos a reescribir el ejemplo anterior utilizando este operador:

```
var nota = 7;

console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Operador ternario: (condición ? verdadero : falso) var calificacion
= nota < 5 ? "suspendido":"aprobado"; console.log("Estoy",
calificacion);
```

Este ejemplo hace exactamente lo mismo que el ejemplo anterior. La idea del operador ternario es que podemos condensar mucho código y tener un if en una sola línea. Obviamente, es una opción que sólo se recomienda utilizar cuando son if muy pequeños.

Condicional If múltiple

Es posible que necesitemos crear un condicional múltiple con más de 2 condiciones, por ejemplo, para establecer la calificación específica. Para ello, podemos anidar varios if/else uno dentro de otro, de la siguiente forma:

```
var nota = 7;

console.log("He realizado mi examen.");

// Condición if (nota <
5) {
```

```
calificacion = "Insuficiente";

} else if (nota < 6) { calificación =
"Suficiente";

} else if (nota < 8) { calificacion
= "Bien";

} else if (nota <= 9) { calificacion =
"Notable";

} else {

calificacion = "Sobresaliente";

}

console.log("He obtenido un", calificacion);
```

Sin embargo, anidar de esta forma varios if suele ser muy poco legible y produce un código algo feo. En algunos casos se podría utilizar otra estructura de control llamada switch, que puede ser útil en ciertos casos.

Condicional Switch

La estructura de control switch permite definir casos específicos a realizar en el caso de que la variable expuesta como condición sea igual a los valores que se especifican a continuación mediante los case. No obstante, hay varias puntualizaciones que aclarar sobre este ejemplo:

var nota = 7;

console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Nota: Este ejemplo NO es equivalente al ejemplo anterior (leer abajo) switch (nota) {

case 10:

calificacion = "Insuficiente"; break;

case 9:

case 8:

calificacion = "Notable"; break;

case 7:

case 6:

calificacion = "Bien";

break; case 5:

calificacion = "Suficiente"; break;

case 4:

case 3:

case 2:

case 1:

case 0:

calificacion = "Insuficiente"; break;

default:

// Cualquier otro caso calificacion =

"Nota errónea"; break;

}

console.log("He obtenido un", calificacion);

*En primer lugar, el ejemplo anterior no es exactamente equivalente al anterior. Este ejemplo funcionaría si sólo permitimos notas que sean números enteros, es decir, números del 0 al 10, sin decimales. En el caso de que nota tuviera por ejemplo, el valor 7.5, mostraría Nota errónea.

El ejemplo de los if múltiples si controla casos de números decimales porque establecemos comparaciones de rangos con mayor o menor, cosa que con el switch no se puede hacer. El switch está indicado para utilizar sólo con casos con valores concretos y específicos.

En segundo lugar, observa que al final de cada caso es necesario indicar un break para salir del switch. En el caso que no sea haga, el programa saltará al siguiente caso, aunque no se cumpla la condición específica.

Bucles e iteraciones

Una de las principales ventajas de la programación es la posibilidad de crear bucles y repeticiones para tareas específicas, y que no tengamos que realizarlas varias veces de forma manual. Existen muchas formas de realizar bucles, vamos a ver los más básicos, similares en otros lenguajes de programación:

Tipo de bucle	Descripción
while	Bucles simples.
for	Bucles clásicos por excelencia.
do..while	Bucles simples que se realizan siempre como mínimo una vez.
for..in	Bucles sobre posiciones de un array. Los veremos más adelante.
for..of	Bucles sobre elementos de un array. Los veremos más adelante.
Array functions	Bucles específicos sobre arrays. Los veremos más adelante. https://lenguajejs.com/javascript/caracteristicas/array-functions/

Antes de comenzar a ver que tipos de bucles existen en Javascript, es necesario conocer algunos conceptos básicos de los bucles:

Condición: Al igual que en los if, en los bucles se va a evaluar una condición para saber si se debe repetir el bucle o finalizarlo. Generalmente, si la condición es verdadera, se repite. Si es falsa, se finaliza.

Iteración: Cada repetición de un bucle se denomina iteración. Por ejemplo, si un bucle repite una acción 10 veces, se dice que tiene 10 iteraciones.

Contador: Muchas veces, los bucles tienen una variable que se denomina contador, porque cuenta el número de repeticiones que ha hecho, para finalizar desde que llegue a un número concreto. Dicha variable hay que inicializarla (*crearla y darle un valor*) antes de comenzar el bucle.

Incremento: Cada vez que terminemos un bucle se suele realizar el incremento (*o decremento*) de una variable, generalmente la denominada variable contador.

Bucle infinito: Es lo que ocurre si en un bucle se nos olvida incrementar la variable contador o escribimos una condición que nunca se puede dar. El bucle se queda eternamente repitiéndose y el programa se queda «colgado».

Bucle while

El bucle while es uno de los bucles más simples que podemos crear. Vamos a repasar el siguiente ejemplo y todas sus partes, para luego repasar que ocurre en cada iteración del bucle:

```
i = 0; // Inicialización de la variable contador

// Condición: Mientras la variable contador sea menor de 5 while (i < 5) {

console.log("Valor de i:", i);

i = i + 1; // Incrementamos el valor de i

}
```

Veamos qué es lo que ocurre a la hora de hacer funcionar ese código: Antes de entrar en el bucle while, se inicializa la variable i a 0.

Antes de realizar la primera iteración del bucle, comprobamos la condición. Si la condición es verdadera, hacemos lo que está dentro del bucle.

Mostramos por pantalla el valor de i y luego incrementamos el valor actual de i en 1.

Volvemos al inicio del bucle para hacer una nueva iteración. Comprobamos de nuevo la condición del bucle.

Iteración del bucle	Valor de i	Descripción	Incremento
Antes del bucle	i = undefined	Antes de comenzar el programa.	
Iteración #1	i = 0	¿(0 < 5)? Verdadero. Mostramos 0 por pantalla.	i = 0 + 1
Iteración #2	i = 1	¿(1 < 5)? Verdadero. Mostramos 1 por pantalla.	i = 1 + 1
Iteración #3	i = 2	¿(2 < 5)? Verdadero. Mostramos 2 por pantalla.	i = 2 + 1
Iteración #4	i = 3	¿(3 < 5)? Verdadero. Mostramos 3 por pantalla.	i = 3 + 1
Iteración #5	i = 4	¿(4 < 5)? Verdadero. Mostramos 4 por pantalla.	i = 4 + 1
Iteración #6	i = 5	¿(5 < 5)? Falso. Salimos del bucle.	

El bucle while es muy simple, pero requiere no olvidarse accidentalmente de la inicialización y el incremento (*además de la condición*), por lo que el bucle for resulta más interesante, ya que para

hacer un bucle de este tipo hay que escribir previamente siempre estos tres factores.

La operación i = i + 1 es lo que se suele llamar un incremento de una variable. Es muy común simplificarla como i++, que hace exactamente lo mismo: aumenta en 1 su valor.

Bucle for

El bucle for es quizás uno de los más utilizados en el mundo de la programación. En Javascript se utiliza exactamente igual que en otros lenguajes como Java o C/C++. Veamos el ejemplo anterior utilizando un bucle for:

```
// for (inicialización; condición; incremento) for (i = 0; i <

5; i++) {

console.log("Valor de i:", i);

}
```

Como vemos, la sintaxis de un bucle for es mucho más compacta y rápida de escribir que la de un bucle while. La primera vez puede parecer algo confusa, pero es mucho más práctica porque te obliga a escribir la inicialización, la condición y el incremento antes del propio bucle, y eso hace que no te olvides de estos tres puntos fundamentales.

En programación es muy habitual empezar a contar desde cero. Mientras que en la vida real se contaría desde 1 hasta 10, en programación se contaría desde 0 hasta 9.

Incremento múltiple

Aunque no suele ser habitual, es posible añadir varias inicializaciones o incrementos en un bucle for separando por comas. En el siguiente ejemplo además de aumentar el valor de una variable i, inicializamos una variable con el valor 5 y lo vamos decrementando:

```
for (i = 0, j = 5; i < 5; i++, j--) { console.log("Valor de i y

j:", i, j);

}
```

Si i++ aumenta en 1 el valor de i en cada iteración, lo que hace j-- es disminuir en 1 el valor de j en cada iteración.

Bucles sobre arrays

Más adelante hablaremos de los arrays https://lenguajejs.com/javascript/fundamentos/arrays y veremos varias formas que tiene Javascript de realizar bucles adaptados a ese tipo de variables.