1

METODOLOGÍA DE LA PROGRAMACIÓN

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

1.1 REPRESENTACIÓN DE ALGORITMOS

La realización de programas informáticos bajo un lenguaje determinado implica la aplicación de una serie de reglas semánticas y sintácticas y el desarrollo de un proceso que responde a unos sucesos concretos.

A menudo, este proceso se consigue con un análisis previo que resuelve nuestro problema de una manera menos rígida y más compresible con un nivel de detalle más o menos profundo, dependiendo de a donde se desee llegar y, el cual, nos permite comprobar su validez y exactitud antes de pasar al proceso de codificación bajo ese determinado lenguaje.

Para crear estas representaciones más o menos fidedignas de un problema podemos apoyarnos en los ordinogranas, que son una gráfica que muestra de una forma visual la secuenciación de resolución de un problema, en los cursogramas, que son una variación de los ordinogramagramas y/o en los pseudocódigos, que son una forma textual de describir el problema usando un lenguaje más compresible basado en nuestro idioma nativo.

1.1.1 Ordinogramas o diagramas de flujo

Un ordinograma, diagrama de flujo, flujograma o diagrama de actividades puede definirse como la representación gráfica de lo que hace un proceso, programa o parte de ellos.

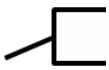
Para realizar estas representaciones gráficas habitualmente se recurre al Lenguaje Unificado de Modelado (UML) ya que representa los flujos de trabajo paso a paso y el cual se caracteriza por disponer de una serie de símbolos que poseen un significado concreto y unas descripciones breves textuales que definen el contexto u operación.

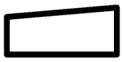
Sin embargo, antes de hacer un ordinograma o diagrama de flujo, es aconsejable definir qué se espera obtener del diagrama de flujo, quién y cómo se empleará, hasta qué nivel de detalle se va a llegar y cuáles serán los límites del proceso a describir.

Una vez realizadas las acciones anteriormente comentadas es momento de construir el diagrama de flujo y, para ello, deberemos seguir los siguientes pasos:

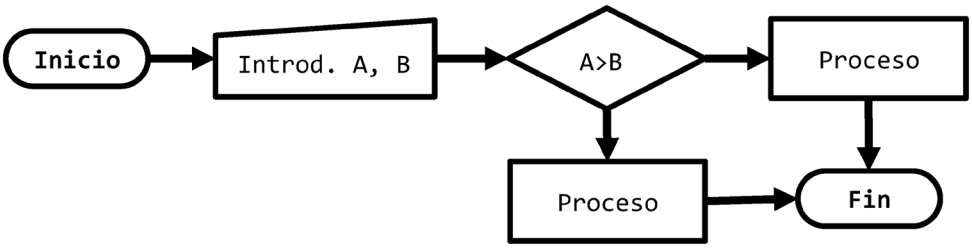
- Establecer el alcance del proceso a describir.
- Identificar y listar las principales actividades/subprocesos que están incluidos en el proceso a describir y su orden cronológico.
- Definir el nivel de detalle incluyendo sus actividades menores.
- Identificar y listar los puntos de decisión.
- Definir alguna forma para realizar una prueba contra errores.

Llegados a este punto, y con el fin de crear nuestro ordinograma, deberemos tener en cuenta los diferentes símbolos y su significado concreto. A continuación, se muestran los más frecuentes:

Símbolo	Representa / Descripción
	**Línea de flujo**. Conecta dos bloques del diagrama y define la direccionalidad o secuencia del proceso.
	**Terminal**. Indica el inicio o fin de un programa o subproceso. Habitualmente suelen contener las palabras “Inicio” o “Fin”.
	**Proceso**. Indica un paso, operación o conjunto de operaciones dentro del proceso.
	**Decisión o alternativa**. Se utiliza con el fin de obtener una decisión a través de una condición dada.
	**Entrada/Salida**. Indica la realización de una entrada o salida de datos externos.
	**Nota o Comentario**. Indica información adicional acerca de un paso o punto concreto del diagrama.
	**Módulo o Subproceso**. Indica el nombre del módulo o subproceso externo, el cual está definido en otro lugar.
	**Conector**. Se utiliza para conectar o enlazar dos o más partes del diagrama de flujo.
	**Conector de página**. Se utiliza para indicar que el objetivo está en otra página.
	**Demora**. Indica que se producirá un período de demora en el proceso.

	**Entrada Manual**. Indica una entrada de datos o información al sistema, habitualmente desde el teclado.
	**Base de Datos**. Indica una operación de Entrada/Salida en un almacenamiento externo como una base de datos o disco.
	**Documento**. Indica que un documento entra, sale, utiliza o se genera dentro del procedimiento.
	**Almacenamiento**. Habitualmente, indica el almacenamiento permanente dentro de un archivo

Veamos un ejemplo:



1.1.2 Cursogramas

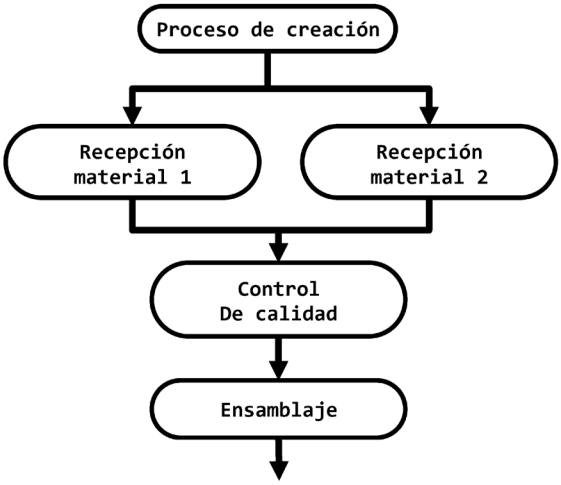
El cursograma es similar, si no idéntico, a un diagrama de flujo, con la diferencia de que se suele utilizar para mostrar la secuencia cronológica de las fases del proceso o programa.

Dentro de los cursogramas podemos distinguir dos tipos, los sinápticos y los analíticos. Mientras que los sinápticos tienen como objetivo realizar visión global y general del proceso antes de hacer el estudio detallado, los analíticos muestran la trayectoria del proceso con una mayor cantidad de información y nivel de detalle.

En lo referente a su simbología disponemos de lo siguiente:

Símbolo	Representa / Descripción
	Línea continua. Flujo de información a través de un formulario o en soporte de papel escrito.
	Línea discontinua. Flujo de información a través de un formulario en formato digital.
	**Elipse**. Inicio del Diagrama y Final del Diagrama
	**Círculo**. Definir una operación
	**Cuadrado**. Proceso de control.
	**Rectángulo**. Formulario o documentación. Se grafica con el doble de largo que su altura.
	**Rectángulo**. Valor o medio de pago. Se grafica con el cuadruple de largo que su altura e idéntico en altura a los formularios o documentación.
	**Tríangulo base arriba**. Archivo Transitorio.
	**Triángulo base abajo**. Archivo definitivo.
	**Rombo**. Decisión o alternativa.
	**Trapezoide**. Carga de datos al sistema.
	**Pentágono**. Conector.
	**Hexágono**. Proceso no representado.
	**Cruz**. Destrucción de Formularios, y recreación de nuevas acciones

Veamos unos ejemplos de cursogramas:



Ejemplo de Cursograma sináptico

Descripción:	Cantidad (Kg)	Distancia (m)	Tiempo (min)	Símbolo	Observaciones
Recepción y almacenamiento de PEBD a reciclar	40		ND		
Escogido de material a reciclar	40		30		Manual
Lavado de material a reciclar	40		40		Manual
Secado de material a reciclar	40		60		Natural
Traslado de material a reciclar	40	35	4		Manual
Inspección de material	40		10		Visual
Picado	40		15		Manual
Espera hasta obtener una cantidad determinada	40		15		
Transporte a aglutinadora	40		1		Manual

Cursograma analítico (CursogramaII r1 c1.jpg) extraído de Wikimedia Commons contributors y con Page Version ID: 487981208.

1.1.3 Pseudocódigos

Un pseudocódigo puede definirse como una descripción de alto nivel compacta e informal de un algoritmo mediante el uso de un lenguaje natural y un conjunto de elementos similares a los usados en los lenguajes de programación.

Las ventajas de usar pseudocódigos es que proporcionan una mayor eficiencia, son más fáciles de leer y comprender, presentan una mayor flexibilidad, mejoran la comunicación e intercambio de ideas y no requieren de un sistema de software propio o dedicado puesto que son descripciones textuales que usan el lenguaje nativo de los desarrolladores y/o colaboradores.

Eso sí, antes de escribir un pseudocódigo, primero necesitaremos:

- Determinar el objetivo del proceso o programa.
- Organizarlo en pasos bajo un orden lógico y secuencial.
- Dividir el proceso o programa en partes simples y manejables.
- Poder presentarlo con sangrías para distinguir los diferentes bloques o estructuras y el ámbito de aplicación.
- Ser capaces de probarlo demostrando su simplicidad, claridad de compresión y lógica.

1.1.3.1 SINTAXIS

Es importante aclarar que el pseudocódigo no es algo que obedezca a unas determinadas reglas de sintaxis particulares o predefinidas ni forma parte de ningún estándar, a pesar de que múltiples autores se empeñen en intentar establecer el suyo propio o aquel que se ajusta más a su modo de pensar.

Eso sí, hay ciertas libertades que están permitidas como la omisión de declaración de variables o que las llamadas a funciones, bloques de código y/o el código contenido dentro de un bucle o estructura iterativa se remplacen por una sentencia de una línea en lenguaje natural.

A continuación, se muestra el ejemplo de un pseudocódigo sencillo:

```
Inicio
   Escribir "Introducir un número A: "
   Leer numA
   Escribir "Introducir un número b: "
   Leer numB
   Res ← numA * numB / 2
   Escribir "La media es: ", Res
Fin
```

Este pequeño pseudocódigo nos muestra cómo calcular la media entre dos números. Evidententemente, las necesidades reales hacen que esto no sea considerado como un programa completo, sino más bien una parte de un todo mucho mayor. Sin embargo, cabe destacar que la forma de espresarlo es o sería básicamente la misma.

1.1.3.2 MANIPULACIÓN DE DATOS

Como se ha dicho antes, cuando estamos definiendo pseudocódigos, no se suele atender a unas normas concretas, sino que se expresa a través de una forma más legible y natural. Esto es lo que pasa, por ejemplo, con las variables y los operadores.

```
Suma ← 0
Suma := 0;
Suma = 0
```

Las tres formas de declarar la variable Suma son válidas, no obstante, están basadas en uno u otro lenguaje ya conocido para el programador.

Ahora, si lo que deseamos es operar, podríamos hacer algo como:

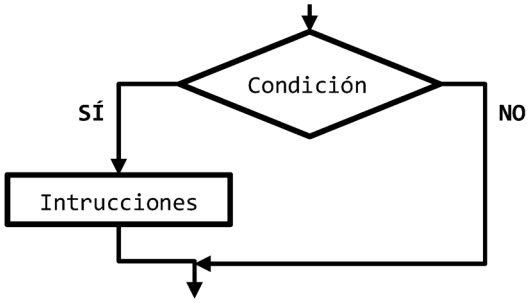
```
hiptenusa ← √(a² + b²)
resultado := sin(a);
volumen = πr²h
operacion1 = (a + b) / 2;
operacion2 = operacion1 mod 3;
```

En general, los operadores y operaciones son las que, de manera natural, usamos en nuestra vida cotidiana. Esto es, la suma, multiplicación, división, módulo, raíces, etcétera.

1.1.3.3 ESTRUCTURAS DE CONTROL

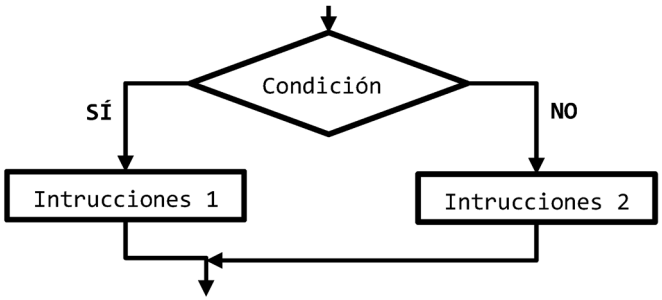
En este tipo de situaciones podemos recurrir a estructuras de una única condición, de dos condiciones o múltiples condiciones.

Para una única condición se podría usar:



```
Si condición Entonces
   instrucciones...
Fin Si
```

Para dos condiciones contrapuestas se podría usar:




```
Si condición Entonces
   instrucciones...
Si no Entonces
   intrucciones...
Fin Si
```

Para múltiples condiciones se podría usar o un condicionante múltiple:

```
Si condición_1 Entonces
   instrucciones...
Si no Si condición_2 Entonces
   intrucciones...
Si no Si condición_3 Entonces
   intrucciones...
...

Si no Entonces
   intrucciones...
Fin Si
```

O una estructura "casos" o "según sea":

```
Según variable Hacer
   Caso: valor_1:
      instrucciones...
   Caso: valor_2:
      instrucciones...
   Caso: valor_3:
```

```
      instrucciones...
...

De Otro Modo
   intrucciones...
Fin Según
```

1.1.3.4 ESTRUCTURAS ITERATIVAS

En este tipo de situaciones podemos recurrir a estructuras con diferentes nombres, aunque todas hacen referencia a lo mismo. En este caso, y por una cuestión de mejor entendimiento yo prefiero usar las estructuras de Mientras, Repetir y Desde Hasta.

La estructura Mientras se repetirá mientras la condición sea cierta, teniendo en cuenta que, podrá no ejecutarse ninguna vez si la evaluación de la condición es falsa.

```
Mientras condición Hacer
   instrucciones...
Fin Mientras
```

Sin embargo, hay una segunda "versión" que permite hacer que la estructura iterativa Mientras se ejecute, al menos, una vez:

```
Hacer
   instrucciones...
Mientras condición
```

La estructura Repetir se repetirá mientras la condición sea cierta, teniendo en cuenta que, las instrucciones se ejecutarán, al menos, una vez.

```
Repetir
   instrucciones...
Hasta Que condición
```

La estructura Hacer Para Cada se repetirá un número determinado de veces en función de los valores inicial, final y el tamaño del paso. Aquí, el paso suele ser 1, pero puede ser 2, 3 o el valor que se desee.

```
Desde x = 0 Hasta n Con Paso 1 Hacer
   instrucciones...
Fin Desde
```

1.1.3.5 FUNCIONES Y PROCEDIMIENTOS

Muchas personas se empeñan en no hacer ninguna diferenciación entre lo que es una función y lo que es un procedimiento. Cierto es que, en muchas ocasiones pueden considerarse lo mismo, sin embargo, hay una diferencia clara... Mientras que las funciones devuelven un valor, los procedimientos no.

Por entenderlo mejor, una función, al igual que una función matemática, puede recibir uno o varios parámetros de entrada y, tras realizar las operaciones pertinentes, devolver un valor de salida. Esto no pasa con los procedimientos, pues, aunque pueden recibir uno o varios parámetros de entrada, no devuelven ningún valor de salida.

```
Función suma(a, b)
   Devolver a + b
Fin Función

Procedimiento suma(a, b)
   Escribir a + b
Fin Procedimiento
```

1.1.3.6 UN EJEMPLO COMPLETO

```
Inicio
   Suma ← 0
   Cantidad ← 0

   Escribir "Introduzca un número positivo: "
   Leer Num

   Mientras Num != -1
   Suma ← Suma + Num
   Cantidad ← Cantidad + 1
   Escribir "Introduzca otro número positivo: "
   Leer Num
Fin Mientras
   Escribir mostrarMedia(suma, cantidad);
Fin

Función mostrarMedia(suma, cantidad)
   Si Cantidad != 0 Entonces
      Media ← Suma / Cantidad
      Devolver "La media es: ", Media

   Sino
      Devolver "No es posible hallar la media" Fin Si

Fin Función
```

1.1.4 Herramientas de desarrollo

Las herramientas de desarrollo, denominadas más formalmente entornos de programación o entornos de desarrollo integrado (IDE), son unos programas o aplicaciones de software que proporcionan soluciones integrales a los programadores para desarrollar software, sea del tipo que sea.

Todos los IDE normalmente poseen un editor de código fuente, herramientas varias para la automatización de tareas, validadores de código como linters, opciones para la compilación y un depurador, entre otras características.

Actualmente, uno de los IDE más extendidos para usar en múltiples arquitecturas como puedan ser PHP, Angular, React, JavaScript, HTML y CSS es Visual Studio Code. Sin embargo, no podemos dejar atrás a los IDE de Eclipse y Netbeans, entre otros.

Visual Code es un IDE (Entorno de Desarrollo Integrado) desarrollado por Microsoft que permite editar los archivos en modo texto y presenta multi-tud de extensiones entre las que se incluyen soporte para la depuración, control integrado de Git y el resaltado de sintaxis.

El Visual Studio Code es descargable desde:

https://code.visualstudio.com/download

Sin embargo, en muchas ocasiones no hace falta recurrir a un IDE para realizar nuestros programas o páginas web. Por ejemplo, una buena herramienta de desarrollo que nos permite programa una página web es Notepad++.

Notepad++ es un editor de código fuente gratuito que admite varios idiomas y lenguajes de programación y fue creado con la intención de reemplazar al Notepad de Windows.

La elección de una u otra herramienta es un poco elección del usuario y, aunque yo personalmente prefiero Visual Studio Code, hay usuarios que prefieren Notepad++ por la sintaxis para todos los idiomas que usa, el modo de presentación de la interfaz de usuario y facilidad de adaptación.

Los que suelen tomar como elección Visual Studio Code lo suelen hacer porque proporciona un entorno multilenguaje muy potente, rápido y con las capacidades de desarrollo frontal listas para usar.

El Notepad++ es descargable desde:

https://notepad-plus-plus.org/downloads/

2

EL LENGUAJE DESCRIPT

Cuando hablamos de lenguajes de script, en realidad, estamos haciendo referencia a una secuencia de comandos que tienen como función manipular, personalizar y automatizar las diversas funcionalidades de un sistema.

Lo que sucede es que, hoy por hoy, hay varios lenguajes de script, lo que provoca que sea difícil definir qué es o cómo usarlo.

2.1 LENGUAJE DE CLIENTE WEB

Por si alguno no lo sabe aún, en Internet existen dos tipos de lenguajes de programación, de cliente y de servidor.

Los lenguajes de cliente son aquellos que se ejecutan en el navegador una vez que la página se ha cargado. Evidentemente, esto implica que el código script ya viene definido desde el servidor, pero, hasta que no llega al navegador, no deja de ser nada más que un trozo de texto plano sin mucha o ninguna relevancia. Los lenguajes de script como JavaScript son de este tipo.

Por su parte, los lenguajes de servidor se ejecutan en el servidor y generan todo lo que le llega al navegador, es decir, todo el código HTML, CSS y JavaScript que se debe correr en la máquina del cliente. Como ejemplos se podría mencionar PHP, NodeJS, Java o .NET.

2.2 LENGUAJE INTERPRETADO

Otra característica fundamental de los lenguajes de script como JavaScript es que son interpretados. Esto es, aunque nosotros recibamos el código completo, el sistema irá ejecutando de forma secuencial instrucción a instrucción. Este tipo de ejecución conllevará una serie de posibles inconvenientes como que cuando se produzca un error, el programa se parará y no ejecutará nada más a partir de ese punto e instante de tiempo.

Además, al ser enviado desde el servidor, el código fuente de script podrá ser accedido fácilmente, lo que podrá conllevar algunos problemas de seguridad y/o protección ya que podrán ser incluso copiados y/o modificados.

Para ver el código fuente de una página web, en la mayoría de los navegadores, bastará con pulsar la tecla F12 una vez que estamos dentro de la página o pulsar el botón derecho del ratón y hacer clic en la opción de "Ver código fuente" o "Inspeccionar elemento".

Si ninguna de estas opciones funciona, se puede probar a usar el atajo CTRL+U, ir al menú de "Desarrollo" y elegir "Mostrar código fuente de la página" pulsar la Opción (⌥) + Comando (⌘) + U en sistema MacOS.

2.3 LENGUAJE ORIENTADO A EVENTOS

Otra característica adicional de los lenguajes de script es que suelen estar orientados a eventos.

Como ya se ha comentado, JavaScript es un lenguaje de programación interpretado, sin embargo, lo que no hemos mencionado aún es que está basado en el estándar ECMAScript (European Computer Manufacturer's Association Script) y que se caracteriza por ser un lenguaje de programación orientado a eventos dinámico que está basado en prototipos sin demasiado tipado.

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

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

¿Y por qué es necesaria esta característica? La razón es bien sencilla.

Si lo pensamos detenidamente, la interactividad de HTML es mínima ya que sólo nos permite navegar entre páginas mediante enlaces, hipervínculos o acciones de formulario.

Por el contrario, JavaScript nos permite controlar desde el estado de un elemento, hasta manejar eventos de ratón, teclado, arrastre, carga y descarga, cambios de tamaño en la ventana y contenedores, actualizar dinámicamente el contenido y un sin fin de cosas más.

2.4 RELACIÓN ENTRE HTML Y JAVASCRIPT

El lenguaje HTML (HyperText Markup Language o lenguaje de marcado de hipertexto) es un lenguaje de marcado dedicado a la elaboración de páginas web. Fue definido por primera vez en 1991 y, en aquel entonces, se caracterizaba por tener algo más de una docena de etiquetas. Más tarde, en 1995 se publicó el primer estándar oficial de HTML al que denominaron HTML 2.0.

En 1997 entró en juego la W3C y desarrolló tres estándares más hasta llegar a lo que hoy conocemos como HTML5 en 2014.

Si bien HTML es un lenguaje formado por entidades que ayudan a estructurar y proporcionar significado a las diferentes partes del documento, cada una de estas entidades, usualmente denominadas elementos o etiquetas, están formadas por un contenido y cero, uno o varios atributos.

```
<p>Esto es un párrafo</p>
<div class="layer">Esto es una capa</div>
```

Cada uno de los atributos tiene una función y puede estar o no asociado a un comportamiento o definición específica. Por ejemplo, el atributo ID habitualmente es utilizado para poder manipular el elemento a través de un nombre corto, sin embargo, también puede ser declarado para vincularse con otro elemento generando una entidad mayor, como es el caso del siguiente código.

```
<label for="nombre">Nombre</label>
<input id="nombre" placeholder="Inserte el nombre completo" />
```

El atributo FOR, utiliza el atributo ID para vincular el LABEL con el INPUT y generar un elemento combinado o pequeño componente.

Cabe destacar que, aunque puede haber etiquetas sin cierre, como es el caso del elemento INPUT, lo normal es que todas las etiquetas o marcas tengan un principio y un final, como es el caso de la etiqueta LABEL.

En lo referente a las novedades de HTML5, como muchos sabrán, una de las más significativas es el valor semántico. La semántica es una característica que dota a los documentos web de mayor significado porque, entre otras cosas, proporciona una mayor estructuración y ayuda a la compresión gracias a lo que se denomina identificador semántico.

El identificador semántico es un término que hace referencia a lo que contiene o representa la etiqueta, es decir, cada etiqueta o elemento tiene un nombre asociado que representa o indica su objetivo. Por ejemplo, en general, la etiqueta SECTION siempre contendrá un conjunto de elementos agrupados que tendrán o guardarán una relación.

Dicho esto, y visto la poca interactividad que presenta HTML, si lo combinamos con un lenguaje de guion o script como es JavaScript, podremos aportar dinamismo a las páginas web. Esto es, podremos mejorar la interacción con los usuarios mediante eventos, realizar operaciones de forma local, validaciones de formulario sin que tenga que intervenir la parte de servidor, mejorar la experiencia visual y una mejor experiencia de usuario.

2.5 EJECUCIÓN Y UBICACIÓN DE CÓDIGO JAVASCRIPT

Aunque HTML puede ser de gran ayuda cuando se trata de describir el contenido, la inmensa mayoría de las veces se suele requerir de una funcionalidad que no nos proporciona el lenguaje. Sirva como ejemplo que, si lo que se desea es saber si el valor de un campo de entrada es válido, se debe recurrir a algún fragmento de código externo en lenguaje script para poder realizar las verificaciones pertinentes.

Dicho esto, el código JavaScript se puede insertar y ejecutar de diversas maneras, pero la más frecuente es a través de la etiqueta SCRIPT de HTML. El elemento SCRIPT permite insertar un código, o fragmento de código, ejecutable dentro de un documento HTML.

Las posibles formas son a través del atributo SRC:

```
<script src="./scripts.js"></script>
```

O mediante la inserción de código en línea:

```
<script>
   function cargar(){
      // instrucciones
   }

   // instrucciones
</script>
```

El lugar dónde colocar esta etiqueta puede ser una discusión diferente en función de quién lo diga y el uso que se le desee dar. Sin embargo, lo más frecuente es encontrarla dentro del elemento HEAD del documento HTML o justo antes de cerrar el elemento BODY.

Si se desea que el código script esté disponible lo antes posible, lo normal es ubicarlo dentro del elemento HEAD, sin embargo, esto conlleva el problema del bloqueo de contenidos, lo que, entre otras cosas, suele perjudicar la velocidad de carga y su posicionamiento en buscadores.

Si se desea que el código script se ejecute sin perjudicar al resto de contenidos y se optimice el proceso de carga, lo recomendable es ubicarlo justo antes de cerrar el elemento BODY. Esto no sólo provocará un aumento de la velocidad de carga desde el punto de usuario, sino que también lo hará desde un punto de vista interno que entiende e interpretan los buscadores como Google.

2.6 LA EJECUCIÓN DE SCRIPTS

Por último, dentro de este capítulo, es importante destacar que la ejecución de código script o de guion es secuencial y paralizante, es decir, si durante la ejecución de un script se produce un error, el resto de las instrucciones que están por debajo no se ejecutarán, a no ser que se utilice algún control de errores que permita continuar con la ejecución.

Además, el código script puede venir precargado o generado por un framework o librería como jQuery, Mootools, Angular, React, ... Estos frameworks o librerías puede optimizar el tiempo de desarrollo, pero suelen perjudicar en mucho el tiempo de ejecución ya que el código de terceros, a menudo, no está optimizado para un mejor rendimiento, sino para un proceso de creación más rápido o eficiente.

Veamos un ejemplo:

```
// ----------------------------
// Ejecución con JavaScript puro
// ----------------------------
console.time();
document.querySelectorAll("*").forEach(function(el){
   console.log(el);
});
console.timeEnd()
Mínimo: 8.37 ms, Máximo: 17.44 ms, Media: 11.63 ms

// ----------------------------
// Ejecución con jQuery
// ----------------------------
console.time();
$("*").each(function(idx, el){
   console.log(el);
});
console.timeEnd()
Mínimo: 9.83 ms, Máximo: 18.52 ms, Media: 13.82 ms
```

Como se puede apreciar, el uso de frameworks, librerías y programas de terceros puede perjudicar en mucho el rendimiento de las páginas. Sirva como ejemplo este código que acabamos de mostrar en donde, sólo por usar jQuery, hemos tardado un 15% de tiempo más, sin contar el tiempo que ha tardado en cargarse e incorporarse la librería al DOM.

3

DESARROLLO DE SCRIPTS Y ELEMENTOS BÁSICOS DE JAVASCRIPT

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

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

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

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

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

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

3.1 VARIABLES Y ÁMBITOS

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

3.1.1 Declaración de variables

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

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

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

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

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

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

Mientras que el uso de VAR permite la redefinición o sobreescritura de variables, este tipo de declaración no. Una vez que se haya realizado la primera definición, no se permitirá que el nombre de la variable pueda volver a ser definida dentro del mismo contexto o bloque, no obstante, esta limitación puede ayudar a evitar errores debidos a la sobreescritura accidental.

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

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

En este caso, la principal diferencia es que, mientras que VAR y LET permiten la reasignación de valores, CONST define el identificador como una declaración de constante y prohíbe su reasignación.

3.1.2 Ámbito de las variables

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

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

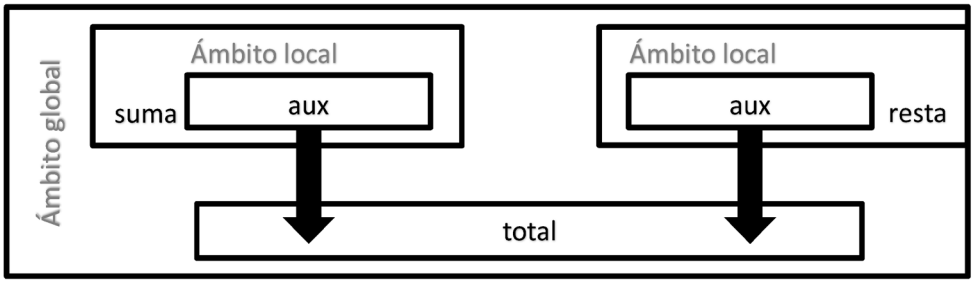
Aunque es un poco más complejo, podríamos decir que, cuando se accede o utiliza una variable, primeramente, se busca en la parte del código o bloque que está delimitado por las llaves (el ámbito local). Más tarde, si se no encuentra la declaración de esa variable en ese ámbito, se busca en los ámbitos locales de sus bloques padre hasta llegar al ámbito global, que, como ya veremos, es el objeto global (WINDOW).

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

```
let total = 0;

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

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

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

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



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

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

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

3.2 TIPOS DE DATOS

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

Los **tipos de datos primitivos** son los que representan un único dato, son inmutables y no tienen métodos. Los **tipos de datos objeto** son los que representan una o varias colecciones de datos primitivos y permiten su manipulación a través de propiedades y/o métodos.

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

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

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

3.2.1 Tipo String

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

```
String(4);                // Devuelve "4"
String(true);             // Devuelve "true"
String(String(5));        // Devuelve "5"
String(var);              // Devuelve error de sintaxis
```

3.2.1.1 PROPIEDADES

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

Propiedad	Descripción y ejemplo
length	Devuelve la longitud de la cadena, en unidades de código UTF16. `"Hola".length;  // devuelve 4`

3.2.1.2 MÉTODOS

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

Método	Descripción y ejemplo
charAt	Devuelve el carácter correspondiente a la posición proporcionada por parámetro. Por defecto, la posición es 0. `"Hola".charAt(0); // devuelve "H"`
charCodeAt	Devuelve el código Unicode del carácter que corresponda a la posición proporcionada por parámetro. Por defecto, la posición es 0. `"Hola".charCodeAt(0); // devuelve 72`
endsWith	Devuelve un booleano que indica si la cadena termina con la subcadena proporcionada por parámetro. `"Hola".endsWith("do"); // devuelve false`
indexOf	Devuelve la primera posición en la que aparezca la subcadena proporcionada por parámetro. Si el resultado de la búsqueda fue infructuoso, el resultado será -1. Tiene un segundo parámetro opcional que indica desde qué posición se debe empezar a buscar y que, por defecto, es 0. Es sensible a mayúsculas y minúsculas. `"aa".indexOf("A",0); // devuelve -1`
lastIndexOf	Este método presenta los mismos argumentos al anterior, con la salvedad de que busca desde el final hasta el principio. `"aa".lastIndexOf("a"); // devuelve 1`

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

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

search	Devuelve la posición de la primera aparición de la cadena proporcionada por parámetro. Aunque este método acepta cadenas como parámetro, si esto se produce, será transformado de forma automática a una expresión regular, las cuales se verán más adelante. `"Hola mundo".search("mundo"); // devuelve 5`

slice	Devuelve el fragmento de la cadena que esté comprendido entre las posiciones proporcionadas por parámetro. Aunque puede resultar similar al método SUBSTRING, sus resultados pueden ser muy diferentes. `"Hola mundo".slice(0, 4);  // devuelve "Hola"`
split	Devuelve un array con todos los fragmentos de cadena que resulten de dividir la cadena origen a través otra cadena o expresión regular proporcionada por parámetro. `"Hola mundo".split(" ");  // devuelve ["Hola", "mundo"]`
starsWith	Devuelve un booleano que indica si la cadena empieza por el valor proporcionado por parámetro. Acepta un segundo parámetro que indica dónde se debe empezar a realizar la búsqueda. Por defecto es 0. `"Hola mundo".startsWith("mundo", 5);  // devuelve true`
substr	Devuelve el fragmento de cadena que empieza por la posición indicada en el primer parámetro y cuya longitud es el valor proporcionado por el segundo. `"Hola mundo".substr(1,6);  // devuelve "ola mu"`
substring	Devuelve el fragmento de cadena que se encuentre entre las posiciones proporcionadas por los parámetros. Aunque puede resultar similar al método SLICE, sus resultados pueden ser muy diferentes. `"Hola mundo".substring(1,6);  // devuelve "ola m"`
toLowerCase	Devuelve la cadena convertida a minúsculas. `"hOla".toLowerCase("");  // devuelve "hola"`
toUpperCase	Devuelve la cadena convertida a mayúsculas. `"mUndo".toUpperCase("");  // devuelve "MUNDO"`

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

3.2.1.3 CONVERSIÓN DE STRINGS

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

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

```
parseFloat("1.5")           // Devuelve 1.5
parseFloat("1,5")           // Devuelve 1
String(new Date())          // Devuelve la fecha actual en formato GMT
```

3.2.1.4 FORMATEADO DE STRINGS

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

```
/* Literales de cadena */
'Esto es un literal de cadena'
"Esto es otro literal de cadena"

/* Secuencia escapada en hexadecimal */
"\x41"      // Devuelve "A"

/* Secuencia escapada en Unicode */
"\u0041"    // Devuelve "A"
```

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

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

```
console.log('\u{1F440}', "\uD83D\uDC40");
```

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

Todos los ejemplos anteriores representan valores en una única línea, no obstante, también existe la posibilidad de trabajar en modo multilínea. El modo multilínea se puede realizar de dos formas, con ayuda del símbolo de barra invertida, o a través de literales de plantilla.

```
/* Literales de cadena multilínea */
console.log('Nombre: Pablo\n\
Apellidos: Fernández');
```

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

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

```
let nombre = 'Pablo';

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

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

```
let nombre = 'Pablo';

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

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

3.2.2 Tipo Number

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

```
Number("4");            // Devolverá 4
Number("Hola")          // Devolverá NaN porque no es un número
Number("21 calles")     // Devolverá NaN porque no es un número
Number(1e3)             // Devolverá 1000
Number(true);           // Devolverá 1
Number(false);          // Devolverá 0
```

3.2.2.1 PROPIEDADES

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

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

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

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

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

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

3.2.2.2 MÉTODOS

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

Propiedad	Descripción y ejemplo

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

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

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

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

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

3.2.2.2.1 CONVERSIÓN DE NÚMEROS

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

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

3.2.2.3 FORMATEADO DE NÚMEROS

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

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

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

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

Propiedad	Descripción y ejemplo

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

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

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

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

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

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

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

3.2.2.4 OPERACIONES CON NÚMEROS

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

3.2.2.4.1 EL OBJETO MATH

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

Como constantes disponemos de:

Constante	Descripción
E	Devuelve un valor aproximado de la constante de Euler. `Math.E // Devuelve 2.718281828459045`
LN2	Devuelve un valor aproximado al logaritmo neperiano de 2. `Math.LN2 // Devuelve 0.6931471805599453`
LN10	Devuelve un valor aproximado del logaritmo neperiano de 10. `Math.LN10 // Devuelve 2.302585092994046`

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

Y como métodos disponemos de:

Método	Descripción
abs	Devuelve el valor absoluto del valor indicado. `Math.abs('-1'); // Devuelve 1`
acos	Devuelve el arcocoseno del valor indicado. `Math.acos('0.999'); // Devuelve 0.044725087168733454`
acosh	Devuelve el arcocoseno hiperbólico del valor indicado. `Math.acosh('2'); // Devuelve 1.3169578969248166`
asin	Devuelve el arcoseno del valor indicado. `Math.asin('0.999'); // Devuelve 1.526071239626163`
asinh	Devuelve el arcoseno hiperbólico del valor indicado. `Math.asinh('2'); // Devuelve 1.4436354751788103`

atan	Devuelve la arcotangente del valor indicado. `Math.asin('0.999');  // Devuelve 0.784897913314115`
atanh	Devuelve la arcotangente hiperbólica del valor indicado. `Math.asinh('0.999');  // Devuelve 3.8002011672501994`
cbrt	Devuelve la raíz cúbica del valor indicado. `Math.cbrt('27');  // Devuelve 3`

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

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

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

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

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

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

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

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

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

tanh	Devuelve la tangente hiperbólica del valor indicado. `Math.tanh('1');  // Devuelve 0.7615941559557649`

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

3.2.3 Tipo BigInt

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

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

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

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

3.2.3.1 PROPIEDADES

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

3.2.3.2 MÉTODOS

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

Método	Descripción y ejemplo

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

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

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

3.2.4 Tipo Boolean

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

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

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

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

3.2.4.1 PROPIEDADES

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

3.2.4.2 MÉTODOS

El tipo BOOLEAN sólo presenta un único método.

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

3.2.5 Tipo Symbol

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

Para crear un símbolo sólo hay que hacer:

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

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

3.2.6 Literal null

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

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

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

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

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

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

3.2.7 Literales undefined y typeof

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

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

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

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

3.3 OPERADORES Y EXPRESIONES

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

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

```
3 + 5 * 2;                          // Devolverá 13
(3 + 5) * 2;                        // Devolverá 16
```

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

3.3.1 Operadores generales

Operador	Descripción y ejemplos
Tipo	**ID**

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

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

Aritmético	**%**	Permite que el operador izquierdo sea dividido por el derecho y devuelva el resto de su división entera. `"20" % "3";  // Devolverá 2`
Aritmético	**	Permite que el operador izquierdo sea elevado al valor indicado por el derecho y devuelva su resultado. `2 ** 3;  // Devolverá 8`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.2 Operadores bit a bit

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

3.3.2.1 OPERADOR &

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

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

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

3.3.2.2 OPERADOR |

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

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

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

3.3.2.3 OPERADOR ^

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

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

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

3.3.2.4 OPERADOR ~

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

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

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

3.3.2.5 OPERADORES DE DESPLAZAMIENTO

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

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

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

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

3.4 CONTROL DE FLUJO Y GESTIÓN DE ERRORES

3.4.1 Estructura if

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

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

3.4.2 Estructura if...else

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

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

3.4.3 Estructura switch

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

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

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

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

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

3.4.4 Control de errores por tipo de dato

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

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

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

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

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

3.4.5 Control de errores por presencia

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

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

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

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

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

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

3.4.6 Manejo de excepciones

3.4.6.1 SENTENCIA TRY...CATCH

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

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

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

Imaginemos el siguiente caso:

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

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

Ahora probemos con el siguiente código:

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

console.log("¡Todo OK!");
```

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

3.4.6.2 SENTENCIA FINALLY

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

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

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

3.4.6.3 SENTENCIA THROW

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

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

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

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

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

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

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

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

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

La función excepcionPersonalizada define el nuevo tipo de excepción y, más tarde, con prototype, le asignamos el prototipo del objeto Error. De esta manera,

cuando se lance la excepción con throw, se mostrará el mensaje requerido indicando en qué objeto se produjo y la línea donde se produjo la excepción.

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

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

3.5 BUCLES Y LA ITERACIÓN

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

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

3.5.1 Estructura for

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

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

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

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

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

Un ejemplo de uso básico podría ser:

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

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

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

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

En este ejemplo, podemos observar que se definen varias variables y que, en la consola del navegador, se muestra el valor de **i** y **j** que, en este caso, irán desde 0 a

5 y desde 10 hasta 5 respectivamente. Por lo tanto, se ejecutará mientras **i** y **j** sean distintos, es decir, se ejecutará 6 veces.

3.5.2 Estructura for...in

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

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

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

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

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

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

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

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

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

3.6 ESTRUCTURA FOR...OF

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

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

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

Para ver mejor hasta qué punto puede afectar esta estructura al rendimiento, veámoslo con un ejemplo. Ya sabemos lo que tarda el bucle for en copiar el array "chars" utilizado en el primer ejemplo (unos 11ms) y también, lo que tarda en copiar

ese array mediante la estructura for...in (unos 23ms). Por ello, vamos a conocer cuánto tarda en realizar la copia a través del bucle for...of:

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

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

3.7 ESTRUCTURA FOREACH

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
function calculadora() {
    this.suma = 0;
    this.producto = 1;
}

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

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

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

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

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

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

3.8 ESTRUCTURA DO...WHILE

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

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

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

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

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

ⓘ NOTA

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

3.9 ESTRUCTURA WHILE

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

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

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

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

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

ⓘ NOTA

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

3.10 SENTENCIA BREAK

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

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

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

3.11 SENTENCIA CONTINUE

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

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

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

3.12 TEST ONLINE

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

4

GESTIÓN DE OBJETOS EN JAVASCRIPT

En JavaScript existen varios tipos de objetos, desde objetos de tipo lista de alto nivel hasta objetos complejos formados por propiedades y métodos, pero en general, la inmensa mayoría de los objetos heredan del objeto **Object**.

Una de las peculiaridades que presenta JavaScript es que todos los objetos tienen una propiedad **prototype** que mantiene un vínculo al objeto que le prototipó que, a su vez, tiene su propio prototipo y así sucesivamente. A esta idea o concepto se la suele denominar cadena de prototipos o modelo de prototipos.

Muchos programadores consideran que el modelo de prototipos es una de sus principales debilidades, sin embargo, este modelo es mucho más potente de lo que, a simple vista parece.

4.1 TIPOS DE OBJETO

- **Predefinidos**: Son los que proporciona el lenguaje. Ejemplos de ello pueden ser el objeto **Date** para la gestión de fechas, **Math** para realizar operaciones matemáticas o **RegExp** para trabajar con expresiones regulares.
- **Personalizables**: Son aquellos que permiten la declaración de **funciones**, **clases**, **objetos** o, también, la adición de otros objetos para implementar nuevas características.
- **Arrays**: Son aquellos que permiten crear conjuntos de elementos a modo de matriz o lista de alto nivel.

- **JSON:** Son aquellos que permiten crear conjuntos de elementos jerarquizables y que, en ocasiones, pueden proveer de métodos para trabajar en diferentes ámbitos o contextos.
- **Especiales:** Son aquellos que han sido diseñados para tener una funcionalidad específica. Entre los más utilizados están **this** y **prototype**.

4.2 PROPIEDADES

Un objeto tiene, esencialmente, tres propiedades:

Propiedad	Descripción
constructor	Devuelve la función constructora nativa.
length	Devuelve la longitud del objeto, normalmente 1.
prototype	Permite añadir nuevas propiedades y métodos al objeto.

4.3 MÉTODOS

El número de métodos disponibles para los objetos es elevado y cambia en función de quién herede, sin embargo, existen algunos que son comunes a todo objeto. A continuación, se muestran los más frecuentes:

Método	Descripción y ejemplo

assign	Copia el objeto referenciado por el segundo parámetro en el objeto referenciado por el primer parámetro y lo devuelve. Si el objeto es un elemento HTML, este método sólo copia el objeto o elemento en sí, no copia sus eventos asociados.
	`let b = {};`
	`Object.assign(b, {a:1, b:2});  // Devuelve {a:1, b:2}`

create	Este método es otra forma de llamar al constructor de la clase. `Object.create({});  // Devolverá {}`

entries	Devuelve un array multidimensional que contiene los pares de clave valor en cada array subyacente. Esto es útil para poder iterar un objeto que, a primera vista, no parece iterable.
	`let json = {"name": "Pablo", "surname": "Fernández",`
	`"age": 18};`
	`Object.entries(json);`
	`// Devuelve lo siguiente: [`
	`0: ["name", "Pablo"]`
	`1: ["surname", "Fernández"]`
	`2: ["age", 18]`
	`length: 3`
	`► __proto : Array(0)`
	`]`

getOwn Property Descriptor	Devuelve un objeto con toda la descripción de la propiedad que se envía como segundo parámetro. Las propiedades que se muestran son: el valor, si es enumerable, si es escribible y si es configurable.
	`Object.getOwnPropertyDescriptor(json, "surname");`
	`// Devuelve un JSON con lo siguiente:`
	`{`
	`value: "Fernández",`
	`writable: true,`
	`enumerable: true,`
	`configurable: true,`
	`}`

getOwn Property Descriptors	Devuelve un objeto con la descripción de todas las propiedades que posee el objeto. Las propiedades que se muestran son las mismas que las devueltas por getOwnPropertyDescriptor.

getOwn Property Names	Devuelve un array con los nombres de las claves enumerables y no enumerables del objeto proporcionado por parámetro.
	`Object.getOwnPropertyNames(json);`
	`// Devuelve un array como ["name", "surname", "age"]`

hasOwn Property	Devuelve un booleano si el objeto contiene la propiedad proporcionada por parámetro.
	`console.log(json.hasOwnProperty("name"));`
	`// Devuelve true`

keys	Devuelve un array con los nombres de las claves enumerables del objeto proporcionado por parámetro.
	`Object.keys(json);`
	`// Devuelve ["name", "surname", "age"]`

values	Devuelve un array con los valores de las claves enumerables del objeto proporcionado por parámetro.
	
	`Object.values(json);`
	`// Devuelve ["Pablo", "Fernández", 18]`

4.4 ARRAYS

Los arrays, comúnmente, son un conjunto de elementos que se guardan en memoria de forma secuencial y que pueden ser accedidos a través de valores enteros o Strings denominados índices.

En JavaScript, sin embargo, un array no es precisamente esto, sino que, más bien, es un tipo objeto que tiene propiedades que hace que se asemeje a un array, y al que se le ha dotado de algunas características para que se maneje como si lo fuese.

A modo de curiosidad, aunque en la consola del navegador veamos índices que parecen números enteros, en realidad, internamente, se están convirtiendo a **String** y utilizados a modo de identificador de propiedad.

4.4.1 Creación de arrays

Una forma de crear o definir una matriz (o array) es, o a través de su constructor**,** o a través de unos corchetes:

```
let arr = new Array();
let arr = [];
```

4.4.2 Acceso a elementos de un array

Para acceder a sus elementos podemos hacerlo a través del índice entre corchetes:

```
let arr = [1, 1, 2, 3, 5, 8];
console.log(arr[0]);                                    // Devuelve 1
```

El objeto **Array** es una estructura que empieza con el índice CERO, por tanto, para acceder a su último elemento deberemos acceder a su longitud menos uno.

El tipo de datos **String**, internamente, también es considerado como un array, por lo que podemos acceder a una posición concreta de una cadena como si fuese un array y recuperar el carácter que se encuentra en esa posición.

4.4.3 Inserción y almacenamiento de elementos en un array

Para almacenar un nuevo elemento podemos hacerlo a través del índice entre corchetes o utilizar el método **push**:

```
let arr = [1, 1, 2, 3, 5, 8];
arr[6] = 13;
arr.push(21);
console.log(arr[7]);                                    // Devuelve 21
```

Sin embargo, no es posible realizar inserciones directas en los arrays de un array. Esto es porque no se pueden insertar valores en los objetos que no hayan sido, previamente, definidos. Por ejemplo, si intentásemos insertar un valor en un elemento de un array no definido, pero que se supone que está dentro de otro array ya definido, nos saltaría un error de propiedad no definida. Para solucionarlo se debe definir el array antes de insertar el valor:

```
let arr = new Array();
arr[0][0] = 1;                                  // Devuelve un error de tipo

let arr[0] = new Array();
arr[0][0] = 1;                                  // Ahora sí lo inserta
```

4.4.4 Eliminación de elementos de un array

Para eliminar un elemento de un objeto array podemos utilizar dos sentencias o instrucciones.

La sentencia **delete** no elimina realmente el dato, sino que establece su valor a vacío, pero conserva el espacio reservado en memoria.

```
let arr = [1, 1, 2, 3, 5, 8];
delete(arr[4]);

console.log(arr);                          // Devolverá [1, 1, 2, 3, empty, 8]
```

La sentencia **splice**, sin embargo, sí que elimina y remplaza el elemento del array.

El método splice se alimenta de dos parámetros. El primer parámetro, indica la posición dónde empezar a eliminar. El segundo, indica cuantos elementos se deben eliminar desde la posición indicada por el primer parámetro.

```
let arr = [1, 1, 2, 3, 5, 8];
arr.splice(4, 1);

console.log(arr);                                    // Devolverá [1, 1, 2, 3, 8]
```

4.4.5 Propiedades

Como buen objeto de JavaScript, el objeto Array tiene las típicas propiedades

CONSTRUCTOR, **LENGTH** y **PROTOTYPE**, anteriormente comentadas.

4.4.6 Métodos

El número de métodos disponibles para los arrays es elevado. Por ello, a continuación, se muestran los más frecuentes:

Método	Descripción y ejemplo

filter	Devuelve un nuevo array que contiene los resultados que cumplen con la función indicada que se pasa como parámetro.
	
	`let frutas = [Manzana", "Kiwi", "Platano", "Pera"]; frutas.filter(function(x){ return x.length > 4});`
	`// Devuelve el array [Manzana", "Platano"]`

join	Devuelve una cadena que es el resultado de unir todos los elementos por el separador pasado por parámetro. Por defecto, el separador es el símbolo de coma. `[1, 2, 3, 4, 5].join("|");  // Devuelve "1|2|3|4|5"`

indexOf	Devuelve la primera posición en la que aparezca el valor proporcionado por parámetro teniendo en cuenta que es sensible a mayúsculas y minúsculas. Si el resultado de la búsqueda fue infructuoso, el resultado será -1. Tiene un segundo parámetro opcional que indica desde qué posición empieza a buscar y que, por defecto, es 0.
	
	`["Pablo", "Elena", "Pablo"].indexOf("Pablo");`
	`// Devuelve 0`

lastIndexOf	Devuelve la última posición en la que aparezca el valor proporcionado por parámetro teniendo en cuenta que es sensible a mayúsculas y minúsculas. Si el resultado de la búsqueda fue infructuoso, el resultado será -1. Tiene un segundo parámetro opcional que indica desde qué posición empieza a buscar y que, por defecto, es la longitud del array.
	`["Pablo", "Elena", "Pablo"].lastIndexOf("Pablo");`
	`// Devuelve 2`

map	Devuelve un nuevo array que contiene los resultados tras haber sido tratados por la función que se proporciona por parámetro.
	`[1, 2, 3, 4, 5].map(function(x){ return x * x });`
	`// Devuelve el array [1, 4, 9, 16, 25]`

pop	Elimina y devuelve el último elemento del array.
	`[1, 2, 3, 4, 5].pop();`
	`// Devuelve 5 y deja el arr con los valores [1, 2, 3, 4]`

push	Añade el elemento al final del objeto y devuelve el array resultante.
	`[1, 2, 3, 4, 5].push(6);`
	`// Devuelve el array [1, 2, 3, 4, 5, 6]`

reverse	Devuelve el array en orden inverso.
	`[1, 2, 3, 4, 5].reverse();`
	`// Devuelve el array [5, 4, 3, 2, 1]`

reduce	Permite realizar operaciones con arrays. Normalmente recibe dos parámetros (acumulador y elemento actual) y opera con el elemento sobre el acumulador. El resultado de la operación se devuelve en el acumulador.
	`[1, 2, 3, 4].reduce(function(acumulador, elemento){`
	`  return acumulador + elemento;`
	`});  // Devuelve 10`

shift	Elimina y devuelve el primer elemento del array.
	`let aux = [1, 2, 3];`
	`aux.shift();`
	`// Devuelve 1 y deja aux como [2, 3]`

sort	Devuelve un array unidimensional ordenado por valor. Antes de hacer la ordenación se hace una conversión a String, por lo que se ordena en formato de códigos Unicode, es decir, que el valor 80 está antes que el 9 y Manzana está antes que plátano.
	`["Pablo", "Elena", "Adrian", "ana"].sort();`
	`// Devuelve el array ['Adrian', 'Elena', 'Pablo', 'ana']`

slice	Devuelve un nuevo array que contiene el número de elementos proporcionado como segundo parámetro, empezando por la posición pasada como primer parámetro.
	
	`let aux = [3, 40, 200, 5, 1];`
	`aux.slice(0,3);`
	`// Devuelve [3, 40, 200] y aux mantiene todos sus valores`

splice	Devuelve un nuevo array que contiene el número de elementos proporcionado como segundo parámetro, empezando por la posición pasada como primer parámetro. La diferencia con el método slice es que, splice, elimina del array original los elementos contenidos del array devuelto.
	
	`let aux = [3, 40, 200, 5, 1];`
	`aux.slice(0,3);`
	`// Devuelve [3, 40, 200] y aux se queda con [5, 1]`

unshift	Añade, al principio del array, los elementos proporcionados por parámetro.
	
	`let aux = [1, 2, 3];`
	`aux.unshift(4, 5);`
	`// Devuelve 5 y aux se queda como [4, 5, 1, 2, 3]`

4.5 JSON

Según Wikipedia, JSON es un acrónimo de **JavaScript Object Notation** y resulta un formato de texto sencillo y ligero para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplio uso como alternativa al XML, se ha considerado un formato independiente del lenguaje.

Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos es que resulta mucho más sencillo desarrollar un analizador sintáctico, lo que se suele conocer como **parser**. En JavaScript, un objeto JSON puede ser analizado fácilmente usando la función **eval**, algo que (debido a la ubicuidad de JavaScript en casi cualquier navegador web) ha sido fundamental para que haya sido aceptado por parte de la comunidad de desarrolladores AJAX.

Resulta muy frecuente utilizar JSON en entornos donde el tamaño del flujo de datos entre cliente y servidor es de vital importancia (de aquí que sea utilizado por grandes compañías como Yahoo!, Google o Mozilla cuando la fuente de datos es de confianza y donde no es importante el hecho de no disponer de procesamiento XSLT para manipular los datos en el cliente).

A modo de apunte introductorio final diremos que, si bien se tiende a considerar JSON como una alternativa a XML, lo cierto es que es frecuente el uso combinado de JSON y XML en algunas aplicaciones, como es el caso del servicio de Google Maps**.**

4.5.1 Sintaxis

Los tipos de datos disponibles en JSON puede ser valores numéricos, con el punto como separador de decimales, cadenas de texto entrecomilladas, valores booleanos true o false, valores nulos o arrays que suelen contener otros JSON.

```
[
    { "abbreviation": "Ene", "name": "Enero", "days": 31 },
    { "abbreviation": "Feb", "name": "Febrero", "days": 28 },
    { "abbreviation": "Mar", "name": "Marzo", "days": 31 },
    { "abbreviation": "Abr", "name": "Abril", "days": 30 },
    { "abbreviation": "May", "name": "Mayo", "days": 31 },
    { "abbreviation": "Jun", "name": "Junio", "days": 30 },
    { "abbreviation": "Jul", "name": "Julio", "days": 31 },
    { "abbreviation": "Ago", "name": "Agosto", "days": 31 },
    { "abbreviation": "Sep", "name": "Septiembre", "days": 30 },
    { "abbreviation": "Oct", "name": "Octubre", "days": 31 },
    { "abbreviation": "Nov", "name": "Noviembre", "days": 30 },
    { "abbreviation": "Dic", "name": "Diciembre", "days": 31 },
]
```

4.5.2 Creación de JSON

La forma más frecuente de crear o definir un objeto JSON es a través de dos llaves:

```
// Declaración de un JSON vacío
let persona = {};

// Declaración de un JSON con datos de una persona
let persona = {
    nombre: 'Pablo',
    apellido: 'Fernández',
    estatura: 1.60,
    edad: 18,
    trabaja: true
}
```

Sin embargo, también es posible crearlo a través de su constructor:

```
let persona = JSON.constructor();
persona.nombre = 'Pablo';
persona.apellido = 'Fernández';
persona.estatura = 1.60;
persona.edad = 18;
```

4.5.3 Acceso a elementos de un JSON

Para acceder a sus elementos podemos hacerlo a través de sus propiedades en formato objeto o en formato array:

```
console.log(persona['nombre']);                    // Devuelve 'Pablo'
console.log(persona.edad);                         // Devuelve 18
```

Al igual que pasa con el objeto **Array**, si el JSON está formado por un array de JSON su índice inicial será CERO, por tanto, para acceder a su último elemento deberemos acceder a su longitud menos uno.

4.5.4 Inserción y almacenamiento de elementos en un JSON

Para almacenar una nueva propiedad a un JSON, también podemos hacerlo en formato objeto o en formato array:

```
persona.talla = "S";
persona['nombre'] = "Elena";
```

4.5.5 Eliminación de elementos de un JSON

Para eliminar un elemento de un objeto array podemos utilizar la instrucción **delete** que elimina y reemplaza el elemento en el objeto donde se aplica.

```
delete(persona.talla);
console.log(persona);

// Devuelve lo siguiente:
{
    apellido: "Fernández"
    edad: 18
    estatura: 1.6
    nombre: "Elena"
```

```
    trabaja: true
    ► __proto__: Object
}
```

4.5.6 Envío y recepción de JSON

Los JSON son unos objetos con los que trabajamos de manera muy frecuente. Tanto es así, que incluso se les utiliza para enviar y recibir información al servidor o comunicarse con APIs, entre otras muchas posibilidades.

Dado que un JSON es también un objeto en JavaScript, este, hereda las propiedades y métodos propios del objeto como son el **contructor**, **hasOwnProperty** o **toString**.

Lo más normal es que, cuando se desea enviar información al servidor, el objeto que contiene esa información, sea transformado a un tipo String para que luego, en el servidor, pueda ser reconstruida y manipulada.

En lo referente a la recepción, lo habitual, es que el objeto que contiene esa información, venga en formato String. Sin embargo, puede ser convertido a un tipo concreto de objeto nada más ser recibido y, así, poder manipular dicha información.

Cuando se trata de JSON, en el proceso de envío y recepción, lo que se suele hacer es recurrir a los métodos **stringify** y **parse**.

4.5.6.1 MÉTODO STRINGIFY

La sentencia **stringify** convierte un objeto analizable en una cadena de texto de tipo JSON.

Por ejemplo, uno de los usos más frecuentes de esta instrucción es utilizarla para almacenar datos en la memoria intermedia o para transferir datos entre el cliente y el servidor.

```
let objeto = {
    texto: 'valor',
    digito: 1
}

JSON.stringify(objeto);              // Devolverá '{"texto":"valor","digito":1}'
```

Si el JSON está mal formado, en el método **stringify**, se provocará un error al intentar convertirlo a formato texto de JSON y devolverá un error de "Error de sintaxis".

4.5.6.2 MÉTODO PARSE

La sentencia **parse** convierte una cadena de texto de tipo JSON en un objeto analizable por JavaScript.

Por ejemplo, uno de los usos más frecuentes de esta instrucción es utilizarla para recuperar datos de la memoria intermedia o del servidor y, así, poder validar alguna propiedad.

```
let cadena = '{"texto":"valor","digito":1}';

let objeto = JSON.parse(cadena);
// Devuelve un String como:
{
    texto: "valor"
    digito: 1
    ► proto__: Object
}

console.log(objeto.texto); // Devolverá "valor"
```

Si el JSON está malformado, en el método **parse**, se provocará un error al intentar convertirlo a formato analizable y devolverá un error de "Error de sintaxis".

4.6 ESPECIALES

En JavaScript hay objetos que se consideran especiales, bien porque tienen una funcionalidad muy concreta, bien porque son un enlace o vínculo con otros. Aquí vamos a describir algunos de ellos y que se utilizan durante el recorrido del libro.

4.6.1 El objeto window

El objeto window suele denominársele el objeto global ya que es la forma de referenciar a la ventana del navegador.

Cada vez que se accede a una página, el navegador crea un objeto window que, entre otras propiedades y métodos, contiene un objeto document con la información de la página solicitada.

El objeto window es único para cada pestaña del navegador, es decir, que los cambios que se puedan producir sobre el objeto window no se ven reflejados entre pestañas. En lo referente a marcos o frames el comportamiento del objeto window es idéntico al comportamiento entre pestañas, es decir, que también son independientes.

4.6.1.1 MÉTODOS

Muchos de los métodos que utilizamos normalmente pertenecen al objeto window, sin embargo, aunque pertenecen a este, la mayoría de las veces no requieren que se ejecuten o llamen a través de él. Véase por ejemplo el caso de **console**.

4.6.1.2 PROPIEDADES

Si nos ponemos a hablar sobre sus propiedades, la lista podría ser interminable. Por ello, aquí sólo describiremos las más frecuentes.

Propiedad	Descripción

console	Proporciona acceso a la consola del navegador. Este objeto, entre otras cosas, nos permite lanzar mensajes a la consola del navegador como, por ejemplo:
	`console.log(objeto); // Mensaje sin estado console.`
	`warn(objeto); // Mensaje de advertencia`
	`console.error(objeto); // Mensaje de error`

innerHeight	Devuelve el alto en píxeles de la ventana.
innerWidth	Devuelve el ancho en píxeles de la ventana.
length	Devuelve el número total de marcos (**frames** o **iframes**) que tiene la ventana.
localStorage	Provee acceso para gestionar el almacenamiento de datos permanentes, sin fecha de caducidad, aunque se cierre la pestaña o el navegador.
opener	Proporciona acceso a la ventana que fue abierta desde la ventana actual. Sólo es accesible cuando se realiza a través de **window.open**, en caso contrario, devuelve **null**.
outerHeight	Devuelve el alto en píxeles de la ventana incluyendo la barra de notificaciones y los bordes, si los hubiese.
outerWidth	Devuelve el ancho en píxeles de la ventana incluyendo la barra de notificaciones y los bordes, si los hubiese.

pageXOffset, scrollX	Devuelven la posición en píxeles del scroll horizontal, es decir, el valor del desplazamiento en píxeles de la barra de desplazamiento horizontal.
pageYOffset, scrollY	Devuelven la posición en píxeles del scroll horizontal, es decir, el valor del desplazamiento en píxeles de la barra de desplazamiento horizontal.
screen	Proporciona acceso a la interfaz **Screen** que provee toda la información disponible sobre el dispositivo dónde se muestra el documento. Entre sus propiedades podemos encontrar el alto y ancho en píxeles de la pantalla, la profundidad en bits de color y la orientación.
screenX	Devuelve la posición horizontal de la ventana en relación con el ancho de la pantalla.
screenY	Devuelve la posición vertical de la ventana en relación con el ancho de la pantalla.
sessionStorage	Provee acceso para gestionar el almacenamiento de datos temporales, que serán eliminados cuando se cierre la pestaña o el navegador.
top	Proporciona acceso al objeto **window**, marco o iframe superior de la ventana. Si no hay ningún nivel superior devolverá el objeto **window** actual.

4.6.2 El objeto document

El objeto **document** es quién representa a la página actualmente cargada. Dicho de otra manera, es quién proporciona acceso al DOM y describe los métodos y propiedades para poder manejar cualquier tipo de documento, sea del formato que sea (HTML, XHML, SVG, ...).

Desde aquí, podemos acceder y manipular todo el DOM, como se verá más adelante.

4.6.2.1 MÉTODOS

El objeto document dispone de muchos métodos. Debido a ello, es mejor que se conozcan poco a poco según vayan surgiendo las necesidades. En este libro, veremos varios de ellos, pero si surgen dudas, lo mejor siempre seguirá siendo consultar la documentación oficial desde alguna fuente fidedigna.

4.6.2.2 PROPIEDADES

Al igual que pasa con el objeto window, si nos ponemos a hablar sobre sus propiedades, la lista podría ser interminable. Por ello, aquí sólo describiremos las más frecuentes.

Propiedad	Descripción
all	Devuelve un objeto **HTMLCollection** similar a un array que proporciona acceso a todos los elementos HTML que conforman el documento.
activeElement	Devuelve el elemento que actualmente está activo o tiene el foco.
body	Devuelve el elemento que representa y contiene el cuerpo del documento, en otras palabras, el elemento que se corresponde con la etiqueta **BODY** de la página actual.
characterSet	Devuelve el juego de caracteres que se utiliza en el documento. Lo habitual será que contenga el valor UTF-8.
cookie	Devuelve una lista con los nombres de las cookies que están asignadas o utiliza el documento. Van separadas por el símbolo punto y coma.
contentType	Devuelve el tipo de contenido MIME que se utiliza en el documento. Lo habitual será que contenga el valor **text/html** o **multipart/form-data**.
defaultView	Devuelve el objeto **window** al que pertenece.
designMode	Devuelve o establece la capacidad de editar todo el documento. Lo habitual es que su valor esté establecido a **off**. Si se cambia a **on**, todo lo que en principio era de sólo lectura (como un **LABEL** o un **H1**), ahora será editable.
docType	Devuelve el DTD o Definición del Tipo de Documento del documento actual. Lo habitual será que contenga el valor **<!DOCTYPE html>**.
documentElement	Devuelve el elemento que representa y contiene el documento, en otras palabras, el elemento que se corresponde con la etiqueta **HTML** de la página actual.
documentURI	Devuelve la URL del documento actual.
forms	Devuelve un objeto **HTMLCollection** que provee acceso a todos los formularios que están definidos en el documento.
head	Devuelve el elemento que representa y contiene la cabecera del documento, en otras palabras, el elemento que se corresponde con la etiqueta **HEAD** de la página actual.
height	Devuelve el alto en píxeles del documento.
images	Devuelve un objeto **HTMLCollection** que provee acceso a todas las imágenes que están definidas en el documento.
links	Devuelve un objeto **HTMLCollection** que provee acceso a todos los hipervínculos que están definidos en el documento.
onreadystatechange	Es el evento que nos permite controlar el **readyState**, es decir, los diferentes estados por los que pasa la página durante su carga. Esta parte se verá más adelante en detalle.
styleSheets	Devuelve un objeto StyleSheetList que representa y contiene todas las hojas de estilo y bloques **STYLE** incluidos en el documento.
referrer	Devuelve la URL de la página desde donde se entró. Si hacemos una búsqueda en Google y pinchamos en uno de los resultados, lo normal es que, esta propiedad tenga su valor establecido a https://www.google.com/.

readyState	Devuelve el estado actual de carga del documento. Los posibles valores por los que puede pasar son **loading**, que indica que está cargándose todavía, **Interactive**, que indica que se ha terminado de cargar y el DOM está accesible, pero todavía faltan imágenes, estilos o iframes que no se han cargado y **complete**, que indica que el documento está totalmente cargado. Cuando readyState entra en el estado Interactive, el evento **DOMContentLoaded** se dispara automáticamente y cuando el documento entra en este estado, el evento **onload** se dispara automáticamente. El modo de conseguir los estados por los que pasa el documento es establecer un oyente o **listener** sobre el objeto **onreadystatechange**.
scripts	Devuelve un objeto **HTMLCollection** que provee acceso a todos los scripts que están definidos en el documento.
width	Devuelve el ancho en píxeles del documento.

4.6.3 El objeto Screen

El objeto **Screen** proporciona información sobre la pantalla del dispositivo en el que nos encontramos actualmente.

Las propiedades más importantes son:

Propiedad	Descripción
availHeight	Devuelve el alto disponible de la pantalla. Lo habitual es que no suela coincidir con la altura de la pantalla porque, esta propiedad, le resta la altura asignada a la barra de tareas de la interfaz gráfica del sistema (lo que viene siendo la barra de tareas de Windows, por ejemplo).
availWidth	Devuelve el ancho disponible de la pantalla. Lo habitual es que este valor coincida con el ancho de la pantalla.
colorDeep	Devuelve la profundidad de color en bits de la pantalla. Lo habitual es que contenga el valor 24.
height	Devuelve la altura en píxeles de la pantalla.
orientation	Devuelve un objeto **ScreenOrientation** que contiene la información acerca de la orientación de la pantalla. Entre las diferentes propiedades que se nos ofrece podemos encontrar la propiedad **angle**, que devuelve el ángulo de giro de la pantalla y la propiedad **type**, que muestra una serie de posibles valores que son **landscape-primary**, que indica que la pantalla está en posición horizontal natural, **landscape-secondary**, que indica que la pantalla está en posición horizontal natural girada 180 grados, **portrait-primary**, que indica que la pantalla está en posición vertical natural y **portrait-secondary**, que indica que la pantalla está en posición vertical natural girada 180 grados.
width	Devuelve el ancho en píxeles de la pantalla.

4.6.4 La interfaz Navigator

La interfaz **Navigator** es un objeto que contiene la información sobre el agente de usuario, es decir, la información sobre la identidad del usuario.

La interfaz **Navigator** permite consultar, entre otras muchas más cosas, el lenguaje seleccionado por el usuario, la plataforma que está utilizando, el sistema operativo y si admite cookies.

Las propiedades más importantes son:

Propiedad	Descripción
appCodeName	Esta propiedad devuelve el nombre interno del navegador. Si se comprueba en diferentes navegadores, se puede observar que muchos, por no decir todos, tienen esta propiedad establecida a "Mozilla".
appName	Esta propiedad devuelve el nombre oficial del navegador. Si se comprueba en diferentes navegadores, se puede observar que muchos, por no decir todos, tienen esta propiedad establecida a "Netscape".
appVersion	Esta propiedad devuelve la versión del navegador. En varios navegadores, esta propiedad puede contener información adicional, como el sistema operativo.
connection	Esta propiedad devuelve un objeto **NetworkInformation** que contiene información acerca de la conexión de red del dispositivo.
cookieEnabled	Esta propiedad devuelve un booleano que indica si el navegador tiene habilitadas las cookies o no.
hardwareConcurrency	Si está disponible, esta propiedad devuelve el número de núcleos lógicos del procesador.
language	Esta propiedad devuelve el lenguaje que tiene seleccionado el navegador. El valor representado debe cumplir el estándar ISO 639-1.
online	Esta propiedad devuelve un booleano que indica si el navegador tiene acceso a la red, sea local o Internet.
platform	Esta propiedad devuelve la plataforma dónde se está ejecutando el navegador. Lo más frecuente es encontrar valores del tipo "Win32", "Win64", "MacIntel", ...
productSub	Si está disponible, esta propiedad devuelve el número de compilación del navegador.
userAgent	Esta propiedad devuelve un String que contiene la cadena que se corresponde con el agente de usuario. Lo habitual es que contenga mucha información, por lo que puede ser tratada para otros fines no "puramente identificativos".
vendor	Si está disponible, esta propiedad devuelve el fabricante del navegador.

4.6.5 La interfaz Location

La interfaz **Location** es un objeto que guarda toda la información referente a la URL actual. Los objetos **document** y **window** tienen, ambos, una propiedad que contiene los datos recuperados por la interfaz Location.

Las propiedades y métodos más importantes son:

Propiedad	Descripción
href	Contiene la URL completa.
protocol	Contiene el protocolo utilizado.
hostname	Contiene el dominio de la URL
port	Contiene el puerto utilizado.
pathname	Contiene todo lo que no es el dominio.
search	Contiene los parámetros de la URL. Normalmente, le añade el símbolo de cierre de interrogación delante.
reload	Provoca la recarga de la página. `location.reload();`
toString	Devuelve la URL completa en formato String.

4.6.6 La interfaz HTMLElement

HTMLElement es una interfaz de JavaScript que representa a todos los elementos HTML del DOM. Hereda todas las propiedades del objeto Element e implementa las propiedades de los manejadores de eventos globales y táctiles. Además, es una interfaz muy útil cuando se desean añadir funcionalidades y/o sobrecargar métodos ya existentes.

A través de esta interfaz, se pueden leer y establecer la mayoría de los atributos que componen los elementos de HTML, así, por ejemplo, HTMLElement nos da acceso a atributos como los estilos del elemento (style), si está oculto (hidden), si es arrastrable (draggable), su orden de tabulación (tabindex), su combinación de teclado para acceder a él (accesskey), si su contenido es editable (contentEditable)... y así, casi una infinidad de propiedades.

Sirva como ejemplo que, gracias a esta interfaz, podríamos proveer a todos los elementos del DOM de un método que pueda eliminar nodos y elementos sin importar el navegador en el que nos encontramos, como se verá, más adelante, en el capítulo de “El DOM”.

4.6.6.1 PROPIEDADES

A continuación, se muestran las propiedades más importantes o más frecuentemente utilizadas.

Propiedad	Descripción
accesskey	Devuelve o establece la combinación de teclado asignada al elemento. Dependiendo del navegador, el establecimiento de esta propiedad puede variar.
accesskeyLabel	Devuelve el contenido de la combinación de teclado asignada al elemento.
attributes	Devuelve todos los atributos que presenta un elemento.
childNodes	Devuelve todos los nodos hijos que tiene un elemento.
children	Devuelve todos los elementos hijos que tiene un elemento.
className	Permite manipular las clases de un elemento.
clientHeight	Devuelve la altura interior en píxeles del elemento.
clientWidth	Devuelve el ancho interior en píxeles del elemento.
contentEditable	Permite que los elementos de sólo lectura que, en principio no deberían ser editables, puedan serlo. Sus posibles valores son **true** o **false**. Si, por ejemplo, establecemos esta propiedad a true en un DIV, todo su contenido será editable.
dataset	Devuelve un objeto DOMStringMap que da acceso a la creación y manipulación de los atributos personalizados de un elemento.
draggable	Devuelve un booleano que indica si el elemento es arrastrable o no.
hidden	Devuelve un booleano que indica si el elemento es está oculto o no.
id	Devuelve o establece el identificador del elemento.
innerHTML	Devuelve el contenido de todos los nodos o elementos hijos del elemento seleccionado en formato HTML con codificación de caracteres. **Nota:** La codificación de caracteres significa que, el código, se devuelve en formato de entidad HTML, es decir, que convertirá los caracteres especiales como “<” en “<” y “>” en “>”, entre otros.
innerText	Devuelve el contenido de todos los nodos o elementos hijos del elemento seleccionado en formato HTML sin codificación de caracteres. Es equivalente a utilizar **childNodes[0].nodeValue**. **Nota:** esta propiedad no devolverá los elementos que estén ocultos por CSS.
isContentEditable	Devuelve si el elemento es editable o no.
lang	Devuelve o establece el idioma asignado al elemento.

localname	Devuelve el nombre local del elemento. Se considera nombre local de un elemento a la parte de la etiqueta que está detrás del símbolo dos puntos, es decir, la que está detrás del prefijo. Si no tiene prefijo, es equivalente al valor que devuelve la propiedad **tagName**.
name	Devuelve o establece el nombre del elemento.
namespaceURI	Devuelve el espacio de nombres del elemento. Habitualmente devolverá *http://www.w3.org/1999/xhtml* si estamos trabajando con HTML5. Si no tiene ningún valor asignado, su valor será null.
nextSibling	Devuelve el nodo inmediatamente posterior. Si no tiene, su valor será **null**.
nextElementSibling	Devuelve el elemento inmediatamente posterior. Si no tiene, su valor será **null**.
nodeName	Devuelve el nombre del nodo. Si el nodo es un elemento HTML, el nombre coincidirá con al valor devuelto por la propiedad **tagName**.
nodeType	Devuelve un número entero que representa el tipo de nodo. En este caso, como todo son elementos, esta propiedad siempre contendrá el valor 1.
nodeValue	Devuelve o establece el valor del nodo. Esta propiedad será null para el propio elemento. Para los nodos de tipo texto, comentario o **CDATA**, devolverá el contenido del nodo. Si el nodo es un atributo, devolverá el valor del atributo.
offsetHeight	Devuelve la altura en píxeles del elemento.
offsetLeft	Devuelve la distancia en píxeles que hay entre el borde izquierdo del elemento padre y el borde izquierdo del elemento actual.
offsetParent	Devuelve el elemento padre desde dónde se calculan los valores de offset. Normalmente se corresponde con el elemento que tiene el último posicionamiento relativo ascendente más cercano.
offsetTop	Devuelve la distancia en píxeles que hay entre el borde superior del elemento padre y el borde superior del elemento actual.
offsetWidth	Devuelve la anchura en píxeles del elemento.
outerHTML	Devuelve el contenido del elemento seleccionado (incluyéndose a sí mismo, no sólo sus hijos) en formato HTML con codificación de caracteres. **Nota:** La codificación de caracteres significa que, el código, se devuelve en formato de entidad HTML, es decir, que convertirá los caracteres especiales como “<” en “<” y “>” en “>”, entre otros.
ownerDocument	Devuelve un objeto document al que está asociado el elemento. Normalmente, será el documento actual, aunque puede que esté establecido a null.
parentNode	Devuelve el nodo padre (o contenedor) del elemento. Si no tiene padre, su valor será **null**.
parentElement	Devuelve el elemento padre (o contenedor) del elemento. Si no tiene padre, su valor será **null**.

previousSibling	Devuelve el nodo inmediatamente anterior. Si no tiene, su valor será **null**.
previousElementSibling	Devuelve el elemento inmediatamente anterior. Si no tiene, su valor será **null**.
scrollHeight	Devuelve la altura en píxeles de la barra de desplazamiento vertical del elemento.
scrollLeft	Devuelve la posición actual de la barra de desplazamiento horizontal de un elemento, con respecto a la izquierda.
scrollTop	Devuelve la posición actual de la barra de desplazamiento vertical de un elemento, con respecto a la parte superior.
scrollWidth	Devuelve el ancho en píxeles de la barra de desplazamiento horizontal del elemento.
selectedIndex	Devuelve o establece el índice seleccionado de un elemento de tipo desplegable (select). Si su valor es -1 indica que no hay ninguno seleccionado.
style	Devuelve el objeto CSSStyleDeclaration que contiene todos los estilos asociados al elemento.
tabIndex	Devuelve o establece un valor que representa el orden de enfoque del elemento cuando se accede a través del tabulador. El valor 0 indica orden secuencial por definición o aparición en el código. El valor -1 indica que no puede tomar el foco.
tagName	Devuelve el nombre de la etiqueta del elemento. Ejemplos de ello pueden ser DIV, LABEL, FORM, INPUT, BUTTON, ...
textContent	Devuelve o establece el contenido del elemento y todos sus descendientes en formato de texto plano. Si se establece esta propiedad, los nodos hijos serán eliminados y se convertirá en un nodo de tipo texto. **Nota:** esta propiedad sí devolverá los elementos que estén ocultos por CSS.
title	Devuelve o establece el texto que se mostrará cuando el cursor del ratón o puntero se sitúe encima del elemento.
value	Devuelve o establece el valor en un elemento de formulario. Estos pueden ser INPUT, BUTTON, SELECT y DATALIST.

4.6.6.2 MÉTODOS MÁS IMPORTANTES

A continuación, se muestran los métodos más importantes o más frecuentemente utilizados.

Método	Descripción y ejemplo
addEventListener	Añade o registra un manejador de evento a un elemento. No es posible utilizarlo con una colección de elementos directamente. Para ello habrá que recorrerla, elemento a elemento, con una estructura iterativa y asignar el listener de forma independiente, es decir, a cada elemento. `$0.addEventListener("click", nombreFn, false);`
appendChild	Inserta un nodo como último hijo del elemento. El nodo a insertar puede ser también un elemento. `document.body.appendChild(el);`
cloneNode	Realiza la copia del elemento, incluyendo su contenido si está establecido a **true**. `el.cloneNode(true)`
closest	Realiza una búsqueda ascencente desde el elemento hasta el último de sus padres y llegar a la raiz del documento. Si encuentra el nodo que coincida con el CSS selector especificado lo devuelve, en caso contrario, devuelve null. `$0.closest("body");`

dispatchEvent	Dispara el evento indicado por parámetro al elemento que se indique.
	
	`let evt = new Event('click');`
	`$0.dispatchEvent(event);`

getAttribute	Devuelve el valor del atributo proporcionado por parámetro del elemento indicado. `$0.getAttribute("class");`
getElementsBy ClassName	Devuelve un objeto **HTMLCollection** con todos los elementos que tengan el nombre de clase proporcionada por parámetro. `$0.getElementsByClassName("nombre_clase");`
getElementsBy TagName	Devuelve un objeto **HTMLCollection** con todos los elementos que estén definidos con la etiqueta proporcionada por parámetro. `$0.getElementsByTagName("nombre_etiqueta");`
querySelector	Devuelve el primer elemento descendiente del elemento que coincide con el grupo de selectores CSS especificado. `document.body.querySelector("div");`

querySelectorAll	Devuelve el conjunto de elemento descendientes del elemento que coinciden con el grupo de selectores CSS especificados. `document.body.querySelectorAll("div");`
hasAttribute	Devuelve si el elemento tiene establecido o no el atributo proporcionado por parámetro. `$0.hasAttribute("class");`
hasChildNodes	Devuelve un booleano que indica si el elemento tiene hijos o no. `$0.hasChildNodes();`
insertAdjacent HTML	Inserta el código HTML proporcionado como segundo parámetro en la posición especificada por el primer parámetro. Los posibles valores del primer parámetro son **afterbegin**, que inserta el código HTML como primer elemento hijo, **beforebegin**, que inserta el HTML antes del elemento, **afterend**, que inserta el HTML después del elemento, **beforeend**, que inserta el HTML como último elemento hijo. `$0.insertAdjacentHTML('afterend','<i>Texto</i>');`

insertAdjacent Element	Inserta el elemento HTML proporcionado como segundo parámetro en la posición especificada por el primer parámetro.
	Los posibles valores del primer parámetro son **afterbegin**, que inserta el elemento como primer elemento hijo, **beforebegin**, que inserta el elemento antes del elemento, **afterend**, que inserta el elemento después del elemento y **beforeend**, que inserta el elemento como último hijo.
	
	`let p = document.createElement("p");`
	`$0.insertAdjacentElement('afterend', p);`

insertBefore	Permite insertar un elemento inmediatamente antes que el elemento indicado. Tiene una variación de comportamiento que permite realizar inserciones inmediatamente detrás. Esto se consigue si elemento adyacente tienen establecida la propiedad **nextSibling**. `nodoPadre.insertBefore(nuevoNodo, NodoIndicado);`
removeAttribute	Elimina el atributo solicitado del elemento indicado. `$0.removeAttribute("class");`
remove	Elimina el elemento indicado. **Nota:** No es válido para Internet Explorer 11 o inferiores. `$0.remove();`

removeEventListener	Elimina el manejador de evento del elemento indicado. No es posible utilizarlo con una colección de elementos directamente. Para ello habrá que recorrerla, elemento a elemento, con una estructura iterativa y eliminar el listener de forma independiente, es decir, a cada elemento. `$0.removeEventListener("click", nombreFn, false);`
setAttribute	Establece el valor del atributo proporcionado por parámetro al elemento indicado. `$0.setAttribute("data-row", "1");`

4.6.7 El objeto History

Este objeto proporciona una serie de métodos y propiedades que permiten controlar el historial del navegador.

Entre las propiedades que nos ofrece este objeto podemos encontrar:

Propiedad	Descripción
constructor	Devuelve la función constructora nativa.
length	Devuelve la longitud del array.
prototype	Permite añadir nuevas propiedades y métodos al objeto.
scrollRestoration	Permite establecer o configurar cómo serán los cambios de desplazamiento sobre la navegación del historial. Sus posibles propiedades son auto y manual. Su valor predeterminado es auto.
state	Permite recuperar el valor de estado que posee la parte superior de la pila del historial. En general, siempre será null hasta que se produzca una llamada a **pushState** o **replaceState**.

En lo referente a las acciones que se pueden realizar con este objeto, tenemos cinco métodos y un evento para controlar en avance o retroceso del historial.

4.6.7.1 MÉTODO BACK

Permite retroceder un paso atrás en la navegación como si le pulsásemos el botón de “atrás” del navegador.

```
history.back();
```

4.6.7.2 MÉTODO FORWARD

Permite avanzar un paso hacia adelante en la navegación como si le pulsásemos el botón de "adelante" del navegador.

```
history.forward();
```

4.6.7.3 MÉTODO GO

Permite saltar un número determinado de pasos en ambas direcciones, es decir, tanto retroceder como avanzar un número determinado de pasos en la navegación.

El número de pasos se indica como parámetro. Si su valor de este parámetro es negativo, retrocederá el número indicado de pasos en el historial. Si el valor resulta ser positivo, se avanzará el número indicado de pasos en el historial.

```
history.go(-2);
```

4.6.7.4 MÉTODO PUSHSTATE

Permite añadir registros o entradas en el historial del navegador.

Para poder configurarlo se deben suministrar tres parámetros.

Parámetro	Descripción
datos	Es un objeto que está asociado a la entrada y puede ser recuperable a través de la propiedad state desde dentro del evento denominado **PopStateEvent**. En este objeto sólo se puede almacenar hasta 640KB de texto y puede ser cualquier cosa que pueda ser convertida con el método **stringify** del objeto JSON.
Título	Es el nuevo título de la página que se va a insertar.
URL	Es la nueva URL de la página que se va a insertar en el historial.

```
history.pushState(null, 'Listado de Productos', "./catalogo.html");
```

Este método puede ser muy interesante si lo que queremos es controlar los botones de avance y retroceso del navegador, sin embargo, todas las entradas que aquí se manejen deben seguir la política del mismo origen. El intento de introducir una entrada que no pertenezca al mismo dominio provocará una excepción del DOM.

```
history.pushState(null, 'mosquis!', "http://google.es");
```

Otra cosa que cabe destacar de este método es que no provocan la navegación ni la recarga de la entrada, es decir, aunque insertemos una entrada que no exista,

el navegador no informará de ello hasta que forcemos la navegación a través del método **reload** del objeto **location** o, a través de la pulsación de F5.

ⓘ NOTA

Es posible que algunos piensen que, en el fondo, el resultado de ejecutar este método es similar a ejecutar window.location = "#...", sin embargo, pushState es una opción mejor porque, por ejemplo, permite cambiar la URL entera si se desea.

4.6.7.5 MÉTODO REPLACESTATE

Permite reemplazar la entrada actual del historial del navegador.

Para poder configurarlo se deben suministrar tres parámetros.

Parámetro	Descripción
datos	Es un objeto que está asociado a la entrada y puede ser recuperable mediante la propiedad state desde dentro del evento denominado **PopStateEvent**. En este objeto sólo se puede almacenar hasta 640KB de texto y puede ser cualquier cosa que pueda ser convertida con el método **stringify** del objeto JSON.
Título	Es el nuevo título de la página que se va a reemplazar.
URL	Es la nueva URL de la página que se va a reemplazar en el historial.

Imaginemos el supuesto caso de estar en la URL indicada anteriormente por el método pushState, "catalogo.html". Si quisiéramos que apareciese "Catalogo" en vez de eso, podríamos realizar la siguiente acción:

```
history.replaceState(null, 'Mi Web', "/Productos/Catalogo");
```

Al igual que pasa con el método pushState, replaceState no provoca la navegación ni la recarga de la entrada, es decir, aunque insertemos una entrada que no exista, el navegador no informará de ello hasta que forcemos la navegación a través del método **reload** del objeto **location** o, a través de la pulsación de F5.

Si echamos una mirada atrás en el tiempo, igual alguno descubre que este tipo de acciones se realizaban a través del objeto **location** y su propiedad **hash**, que modificaban la parte del anclaje de una URL.

4.6.7.6 EVENTO ONPOPSTATE

Aunque este evento está asociado al historial, en realidad pertenece al objeto window.

Cada vez que el usuario pulsa en los botones de avance o retroceso del historial de navegación, se realiza una llamada al evento onpopstate.

El evento que recibe es un objeto PopStateEvent que contiene varias propiedades, no obstante, como se ha comentado antes, la propiedad que nos interesa es state. La propiedad state guarda el objeto con los datos que definimos a través de los métodos pushState y replaceState.

Si quisiéramos ver el valor se state cuando pulsamos en los botones de volver o avanzar en el historial podríamos verlo a través del siguiente código:

```
window.onpopstate = function(e){
    console.log(e.state)
}
```

NOTA

Los métodos de pushState y replaceState, trabajan de forma coordinada con este evento y el objeto History.

4.6.7.7 EJEMPLO DE ANULACIÓN DEL BOTÓN VOLVER DEL NAVEGADOR

Como decíamos antes, en el pasado, la anulación del botón volver se realizaba a través del objeto location y su propiedad hash. Desde hace ya algunos años, una nueva forma de hacer esto es mediante el siguiente script:

```
let title = document.head.querySelector("title").innerHTML;

history.pushState(null, title, location.href);

window.onpopstate = function(e){
    history.forward();
}
```

4.6.8 El objeto this

El objeto **this** es un objeto genérico que provee acceso al objeto actual, ya sea una función u otro objeto. Si este objeto es llamado desde un contexto global, hará referencia al objeto window, mientras que, si se llama desde una función o evento, hará referencia al propio objeto destino.

En lo referente a su comportamiento, el objeto **this** puede provocar cierta confusión cuando se trata de manipular sus propiedades o métodos. Por ejemplo, puede ocurrir que un objeto trate de actualizar una de sus propiedades a través de this y el resultado sea que, aparentemente, no hace nada.

Supongamos el caso de un objeto que tiene que actualizar una de sus propiedades:

```
function producto(a, b){
let p = a * b;
this.p = p;
}
producto.p = 0;

// Ejecutamos la función y mostramos su propiedad "p"
producto(2,3);
console.log(producto.p);                                    // Devolverá 0
```

La razón de porqué la ejecución de este código da como resultado su valor inicial es que, la asignación de **this**, espera un objeto instanciado. Si en vez de ejecutar la función directamente, la ejecutamos instanciando primero el objeto, veremos que la propiedad **p** sí se ha actualizado.

```
let prod = new producto(2,3);
console.log(prod.p);                                    // Devolverá 6
```

Ahora bien, si lo que queríamos era utilizarlo sin tener que instanciarlo, entonces la respuesta es únicamente cambiar this por el nombre del objeto, en este caso producto.

```
function producto(a, b){
    let p = a * b;Ç
    producto.p = p;
} producto.p = 0;

// Ejecutamos la función y mostramos su propiedad "p"
producto(2,3);
console.log(producto.p);                                    // Devolverá 6
```

Otra de las dudas que surgen cuando se realizan nuevos componentes en JavaScript es porqué, en ocasiones, **this** tiene un valor **undefined** o **null**. La respuesta a esa pregunta suele ser que se encuentra en modo estricto.

En modo no estricto, la ejecución del siguiente código devolvería verdadero, porque la función fue declarada en el contexto de window.

```
function valorThis(){
     console.log(this == window);
}
valorThis();                                          // Devolverá true
```

Sin embargo, en modo estricto, la ejecución del siguiente código devolvería falso, porque el objeto window es reemplazado por null.

```
function valorThis(){
    'use strict'
    console.log(this == window)
}
valorThis();                                           // Devolverá false
```

4.6.9 El objeto globalThis

Si el objeto this nos permite manejar el contexto actual, **globalThis** nos permite acceder al contexto global, que es dónde se encuentra el objeto this. Dicho de otra forma, si estamos dentro de una función de un objeto que fue declarado bajo el contexto de window, la propiedad this será la declaración de la función y globalThis será un alias de window.

```
let it = {name: 'IT', version: '1.0'}
it.imprimirContextos = function(){
    console.log("this: ", this);
    console.log("globalThis: ", globalThis);
}
```

Si ejecutamos el código, podremos comprobar que **this** representa al objeto **it** (que contiene la definición de imprimirContextos, name y versión), y que, **globalThis**, representa al objeto **window**.

Veamos otro ejemplo:

```
let it = function(){ console.log('IT inicializado'); }
it.imprimirContextos = function(){
    console.log("this: ", this);
    console.log("globalThis: ", globalThis);
}
```

Si ahora ejecutamos este código, podremos comprobar que **this** representa la declaración del **contenido de la función**, es decir, la función que contiene el método console mientras que, **globalThis**, representa al objeto **window**.

4.6.10 El objeto prototype

Todos los objetos en JavaScript provienen del objeto Object, por lo que, todos los objetos heredan sus métodos y propiedades. El problema surge cuando se desean incorporar nuevas funcionalidades o métodos a los objetos prototipados. Para eso JavaScript provee de la propiedad **prototype**, la cual permite precisamente esto.

El objeto **prototype** tiene bastantes propiedades obsoletas o no estandarizadas, pero hay 2 propiedades que no podemos ignorar:

Propiedad	Descripción
constructor	Especifica la función que creó el prototipo del objeto.
_proto___	Especifica la llamada de acceso provee acceso al interior del prototipo a través del cual se accede a ella.

En lo referente a sus métodos, hay algunos que cabe destacar:

Método	Descripción

hasOwnProperty	Devuelve un booleano que indica si la propiedad proporcionada por parámetro está presente en el objeto.
	`let data = { id: 1, code: 0 };`
	`data.hasOwnProperty("idd")  // Devuelve false;`

isPrototypeOf	Devuelve un booleano que indica si el objeto pertenece a la cadena de prototipos del objeto especificado.
	`let data = { id: 1, code: 0 };`
	`data.isPrototypeOf(JSON)  // Devuelve false;`

propertyIsEnumerable	Devuelve un booleano que indica si la propiedad tiene el atributo enumerable establecido.
	`let data = { id: 1, code: 0 };`
	`data.propertyIsEnumerable("id")  // Devuelve true;`

toString	Devuelve la definición del objeto convertido en formato cadena de texto.
	`let data = { id: 1, code: 0 };`
	`data.toString()  // Devuelve '[object Object]';`

A continuación, se muestra cómo se puede añadir la funcionalidad de fecha actual en formato español (Little Endian) al objeto Date de JavaScript:

```
Date.prototype.littleEndianFormat = function () {
    const local = new Date(this);

    // Se calcula el diferencial GMT
    local.setMinutes(this.getMinutes()-this.getTimezoneOffset());

    let aux = local.toJSON().slice(0, 10);
    aux = aux.split('-')[2] + "-" +
          aux.split('-')[1] + "-" +
          aux.split('-')[0];

    return aux;
};

new Date().littleEndianFormat();
```

La función **setMinutes** nos sirve para calcular el diferencial GMT.

Seguidamente, se transforma una cadena con la fecha y la hora en formato estándar de JavaScript a formato compatible con JSON y extraemos la subcadena del resultado desde la posición 0 hasta la posición 10.

Finalmente, y con ayuda de la función **split** convertimos la fecha, de formato inglés (Big Endian) a formato español (Little Endian) y, el resultado es lo que se devuelve.

4.7 OTRAS COSAS QUE SABER SOBRE LOS OBJETOS DE JAVASCRIPT

4.7.1 La herencia

Cuando se empieza a programar en un lenguaje como JavaScript, su sintaxis liberal puede causar muchos problemas de adaptación e, incluso, puede hacer que los desarrolladores rechacen el lenguaje.

El paradigma de JavaScript es, en varios aspectos, muy diferente a muchos lenguajes orientados a objetos. Sin embargo, no olvidemos que, en JavaScript, todo son objetos, incluyendo las entidades que, por definición, no deberían serlo.

Como decíamos en un capítulo anterior, en JavaScript todo hijo hereda de su padre y, casi todos, heredan de **Object**. El problema surge, cuando, se desean incorporar nuevas funcionalidades o métodos a los objetos prototipados, pero, para eso, JavaScript provee de algunas "técnicas" para ayudar durante el proceso.

Veámoslo con un ejemplo:

Si queremos crear un nuevo objeto desde cero, lo primero que debemos hacer es crear su constructor.

```
function Persona(nombre, apellidos, edad) {
    this.nombre = nombre + " " + apellidos; t
    his.edad = edad;
};
```

Con esto hemos definido un objeto que hemos llamado **Persona** y al que le hemos dotado de unas pocas propiedades, no obstante, como no tiene ningún método, le creamos uno.

```
// Método para recuperar la edad de la persona
Persona.prototype.getEdad = function() {
    alert(this.nombre + ' tiene ' + this.edad + ' años!');
};

// Método para recuperar el nombre de la persona
Persona.prototype.getNombre = function() {
    alert('Mi nombre es ' + this.nombre);
};
```

Ahora lo que queremos hacer es, definir a esas personas como alumnos o profesores, por lo que tendremos que crear los objetos **Alumno** y **Profesor**.

```
function Alumno(persona, curso, asignaturas) {
    for(let key in persona){
        this[key] = persona[key];
    }
    this.curso = curso;
    this.asignaturas = asignaturas;
};

function Profesor(curso) {
    for(let key in persona){
        this[key] = persona[key];
    }
    this.curso = curso;
};
```

Ambos son Persona, por lo que tendrán que heredar de la clase que hemos definido antes. Para ello definimos cuál es el prototipo del que heredan y sobrescribimos su constructor para que las propiedades y métodos pertenezcan a ese objeto.

```
Alumno.prototype = new Persona();
Alumno.prototype.constructor = Alumno;

Profesor.prototype = new Persona();
Profesor.prototype.constructor = Alumno;
```

Ahora los objetos Alumno y Profesor son también Personas y, por tanto, ya hemos provocado la herencia de sus propiedades y métodos.

```
let pablo = new Persona('Pablo', 'Fernández',18);
let alumno = new Alumno(pablo, '1º FP', '...');
```

Si ahora consultamos lo que contiene la variable alumno, veremos que tiene todas las propiedades del objeto Alumno y todas las propiedades y métodos del objeto Persona.

```
console.log(pablo);
// devuelve lo siguiente:
{
    edad: 18
    nombre: "Pablo Fernández"
    ► proto__: Object
}

console.log(alumno);

// Devuelve lo siguiente:
{
    asignaturas: "..."
    curso: "1º FP"
    edad: 18
    ► getEdad: f{}
    ► getNombre: f{}
    nombre: "Pablo Fernández"
    ► proto__: Persona
}
```

4.7.2 Sentencias get y set

Permite definir un método que realizará una funcionalidad concreta cuando se acceda a una determinada propiedad.

```
get ultimo() {
    if (this.log.length > 0) {
        return this.log[this.log.length 1];
    } else {
        return "";
    }
}
```

Si ejecutásemos el ejemplo anterior, podríamos observar que cuando se llama a la propiedad **ultimo**, el objeto nos devuelve el último elemento del array denominado log.

Y con la sentencia set pasa un poco lo mismo, podemos hacer que, cada vez que se actualice una propiedad, se ejecute una acción asociada de forma interna.

```
set mensaje(mensaje) {
    this.log.mensaje(mensaje);
}
```

Si ejecutásemos el ejemplo anterior, podríamos observar que, cuando se actualiza la propiedad mensaje, dicho mensaje se inserta en un array contenedor.

4.7.2.1 EJEMPLO COMPLETO DE GET Y SET

```
let Historico = {
    get ultimo() {
        if (this.log.length > 0) {
            return this.log[this.log.length - 1];
        } else {
            return "";
        }
    },
    set mensaje(mensaje) {
       this.log.push(mensaje);
    }, log: [] }

// Imprimimos el ultimo valor
console.log(Historico.ultimo);                          // Devuelve ""

// Añadimos un mensaje
```

```
Historico.mensaje = "hola que tal";                 // Devuelve "Hola que tal"

// Imprimimos el ultimo valor
console.log(Historico.ultimo);                      // Devuelve "Hola que tal"
```

4.8 TEST ONLINE

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

5

DECLARACIÓN DE FUNCIONES EN JAVASCRIPT

Uno de los propósitos de las funciones en JavaScript es el de ser utilizadas como constructores para los objetos.

Por lo general, la definición de una función normal suele estar caracterizada por tener un nombre o identificador, una lista de argumentos incluidos entre paréntesis y separados por comas y un conjunto de instrucciones que se encuentran incluidas entre llaves que realizan unas acciones.

Las funciones en JavaScript no tienen por qué devolver nada, no obstante, si se necesita devolver algo, se puede hacer a través de la palabra reservada **return**.

5.1 CREACIÓN DE FUNCIONES

En JavaScript existen dos modos de definir una función, como expresión o como declaración. Cuando nos referimos a una función en **modo de expresión**, lo que se desea expresar es:

```
let fn = function(){
    console.log("Función definida como expresión")
}
```

Cuando nos referimos a una función en **modo de declaración**, lo que se desea expresar es:

```
function fn(){
    console.log("Función definida como expresión")
}
```

5.1.1 Diferencia entre modo estricto o modo no estricto

En JavaScript existen dos modos de programar funciones, el modo estricto y el modo no estricto. Si una función está diseñada en **modo no estricto**, su rendimiento será menor y no habrá errores silenciosos. Además, los valores de **null** y **undefined** serán reemplazados por el objeto global y los valores primitivos serán transformados a objetos. Veámoslo con un ejemplo:

En ocasiones los desarrollos requieren heredar de otros objetos o contextos que no son el actual. Por ejemplo, cuando se ejecuta una función o método desde la consola del navegador, el objeto **this** es equivalente al objeto desde donde se llama la función, en este caso, el objeto **window**.

```
let mostrarContenidoThis = function(){
    console.log(this)
}

mostrarContenidoThis();             //          Devolverá        ►     Window
{postMessage:...}
```

Si ejecutamos el código veremos que el valor del objeto **this** ha sido remplazado por el objeto **window**. Esto es porque, como decíamos, la función está diseñada en **modo no estricto** y dado que la función mostrarContenidoThis está definida bajo el contexto del objeto window, **this**, se vuelve un alias de **window**.

Sin embargo, si la función mostrarContenidoThis la construimos en **modo estricto**, el resultado de su ejecución será completamente diferente.

```
let mostrarContenidoThis = function(){
'use strict'
console.log(this)
}

mostrarContenidoThis();                                    // Devolverá undefined
```

Si ahora ejecutamos el código, veremos que el valor del objeto **this** ha sido remplazado por **undefined**. Esto es porque, ahora sí, la función está diseñada en **modo estricto** y JavaScript no hace ningún remplazo automático.

5.2 PASO DE PARÁMETROS

Las funciones de JavaScript tienen un comportamiento, diríamos, que dinámico.

Cuando los parámetros enviados a una función son **de tipo primitivo** (como pueda ser un tipo Number o un tipo String), **son pasados por valor**, es decir que, si el valor del parámetro es modificado, el cambio no se verá reflejado fuera del contexto de la función.

Sin embargo, cuando los parámetros enviados a la función **no son de tipo primitivo** (como pueda ser un Array o un JSON), **son pasados por referencia**, es decir que, si el valor del parámetro es modificado, el cambio sí se verá reflejado fuera del contexto de la función y puede que afecte, incluso, a nivel global.

Una vez que tenemos claro cómo se envían los parámetros a una función, sólo queda conocer cómo se reciben. Pues bien, se puede recibir por asignación directa o a través del objeto arguments.

5.2.1 Por asignación directa

En muchas ocasiones, cuando se declara una función se indican los nombres de los argumentos, es decir, si tenemos la función alert(msg) y realizamos una llamada con un valor concreto, el argumento "msg" contendrá el valor establecido o enviado.

```
alert("Página cargada");  // "Página cargada" es el argumento
```

5.2.2 El objeto arguments

El objeto **arguments** es una variable local autogenerada que se "crea" cada vez que se invoca a un método o función. Este objeto se asemeja a un array (aunque no lo es puesto que la única propiedad que tienen en común es **length**) y contiene todos los argumentos que recibe una función, accesibles a través de su índice o posición.

El objeto **arguments** tiene tres propiedades importantes.

5.2.2.1 PROPIEDAD CALLEE

Esta propiedad nos permite averiguar el número de argumentos y el nombre de la función que se está ejecutando en el momento actual.

5.2.2.2 PROPIEDAD CALLER

Dentro de la propiedad **callee**, se encuentra esta propiedad, que lo que nos permite es conocer el nombre de la función que llamó a la función que se está ejecutando en momento actual. Por ejemplo, si una función getPersona llamase a otra función getApellidos, la propiedad caller debería tener como valor getPersona y la propiedad callee getApellidos.

5.2.2.3 PROPIEDAD NAME

También, dentro de **callee**, encontramos una propiedad nos permite recuperar el nombre de la función que se está ejecutando en el momento actual.

5.2.2.4 PROPIEDAD LENGTH

Esta propiedad está disponible tanto en la propiedad **callee**, como en la propiedad **caller**. Dependiendo de dónde sea leída, lo que nos indicará es cuántos argumentos tiene la función llamadora o cuántos argumentos tienen la función que se está ejecutando en el momento actual.

```
let square = function(a){
    return a ** 2;
}

let op = function(op, v1, v2){
    console.log(arguments);
    if(op == "Square") return square(v1);
}

op("Square", 2);
```

Si ejecutamos el código anterior, lo que devuelve es algo como lo siguiente:

```
Arguments(2) ["Square", 2, callee: ƒ, Symbol(Symbol.iterator): ƒ]
0: "Square"
1: 2
► callee: f (op, v1, v2)
    arguments: null
    caller: null
    length: 3
    name: "op"
    prototype: {constructor: ƒ}
    __proto__: f ()
length: 2
```

```
Symbol(Symbol.iterator): f values()
 __proto__: Object
```

Como se puede apreciar en el desglose del objeto Arguments, cuando se llama a la función square, la propiedad callee tiene como valor el nombre de la función que la llamó. Además, se puede ver que el número de argumentos de la función square es de 2 y que, el número de argumentos de la función op (dentro de la propiedad callee) es de 3.

5.3 FUNCIONES ANÓNIMAS

En JavaScript hay dos formas de definir funciones. La más frecuente y conocida es la que se le asigna un nombre detrás de la palabra reservada **function**. Esta forma de declarar funciones es lo que se suele denominar función nominal.

Las funciones anónimas son exactamente idénticas a las nominales, con la diferencia de que no tienen un nombre o identificador asociado.

A continuación, se muestra un ejemplo de una función anónima.

```
function(){
    console.log("Página cargada");
}
```

Si intentamos ejecutar esta declaración de función, probablemente, nos aparecerá un mensaje de error de sintaxis puesto que no es posible asociarse a ningún objeto ni a ningún nombre de función.

Sin embargo, si esta definición, la encerramos entre paréntesis y colocamos otros dos al final de la declaración, lo que se conseguirá es que se defina y ejecute a la vez.

```
(function(){
    console.log("Página cargada");
})();
```

Todo sea dicho de paso, este tipo de declaraciones no es nada frecuente porque es como si ejecutásemos el contenido sin más. No obstante, las funciones anónimas sí que se utilizan mucho para el paso de parámetros, temporizadores, declaración de eventos y en otras muchas situaciones.

5.3.1 Ventajas e inconvenientes

Es muy frecuente utilizar funciones anónimas en listeners por su facilidad de declaración o porque pueden decrementar el tiempo de desarrollo considerablemente. En estos casos, en los que la función anónima no puede ser referenciada, aunque se permite, no se recomienda su uso, sobre todo, porque no puede ser alcanzada por el recolector de basura de JavaScript y, como consecuencia, no puede ser eliminada de la memoria.

Sin embargo, las ventajas de usar este tipo de funciones pueden ser muchas, puesto que se pueden pasar como parámetro a otra función y conseguir que todo lo que se defina o ejecute dentro de ella no interfiera con la aplicación.

Un claro ejemplo de esto y que se da bastante a menudo es en el uso de los métodos **setTimeout** o **setInterval**:

```
setTimeout(function(){
    console.log('Este mensaje está dentro de una función anónima')
}, 1000);
```

Si ejecutamos el código anterior veremos que, el método **setTimeout**, ahora muestra el mensaje pasados 1000 milisegundos.

```
let x = 0;
let interval = setInterval (function(){
    console.log('Este mensaje está dentro de una función anónima')

    x++;

    if(x == 10) clearInterval(interval)
}, 1000);
```

Si, ahora, ejecutamos el código anterior veremos que, el método **setInterval**, producirá la salida del mensaje en 10 ocasiones, una vez cada 1000 milisegundos. Pasados 10 segundos, se cumplirá la condición y se ejecutará la sentencia **clearInterval** que parará el temporizador.

5.4 FUNCIONES CLAUSURA

Las funciones clausura son funciones que se declaran dentro del contexto de otra función. Se les llama así porque siguen la técnica de implementación de contextos (también llamada ámbitos léxicos) aplicable a lenguajes de programación

con funciones, como es el caso de JavaScript y porque, al tener una función definida dentro de otra, se produce un efecto de clausura y ocultación.

```
let Persona = function() {
    var _fname = new Array();

    function setName(val) {
    _fname['name'] = val;
}

    function setSurname(val) {
    _fname['surname'] = val;
    }

    return {
        asignarApellidos: function(t) {
            setSurname(t);
        },
        asignarNombre: function(t) {
            setName(t);
        },
        mostrar: function() {
           return _fname['name'] + " " + _fname['surname'];
        }
    }
}

// Para utilizar esta función
let p = Persona();
    p.asignarNombre("Pablo");
    p.asignarApellidos("Fernández");

p.mostrar();                                  // Devuelve "Pablo Fernández"
```

5.4.1 Ventajas e inconvenientes

Utilizar funciones de clausura son muy útiles para emular la declaración de propiedades y métodos privados, es decir, funciones que ni son visibles ni pueden ser llamadas desde fuera de su contexto (en nuestro ejemplo, **Persona**).

De hecho, si ahora quisiéramos ver el contenido del objeto **Persona** comprobaríamos que tanto el nombre de la propiedad como los métodos de asignación no están disponibles.

```
console.log(Persona())
console.log(p)

// Ambos devolverán lo mismo:

{asignarApellidos: ƒ, asignarNombre: ƒ, mostrar: ƒ}
    ► asignarApellidos: f (t)
    ► asignarNombre: f (t)
    ► mostrar: f (t)
    ► __proto__: Object
```

Sin embargo, las funciones de clausura tienen un gran inconveniente y es que bajan el rendimiento del sistema porque provocan un aumento innecesario de la memoria y, a veces, del procesamiento.

5.5 FUNCIONES FLECHA

Las funciones flecha (o funciones arrow) no son nada más que una forma de reducción de código, es decir, una forma abreviada de codificar funciones. Se las llama flecha porque se escriben con una simbología que recuerda a una flecha y pueden resultar muy cómodas a la hora de desarrollar, sin embargo, no son compatibles con varios navegadores, incluyendo Internet Explorer 11 ni Microsoft Edge 13.

Por ejemplo, una función flecha podría ser:

```
let getName = () => "Pablo Fernández";
```

Esto sería equivalente a:

```
let getName = function(){
    return "Pablo Fernández"
}
```

Todas las funciones flecha son anónimas y, todas ellas, definen sus argumentos en la parte izquierda de la "flecha" y el contenido a la derecha de la "flecha", pero, aunque puedan resultar muy similares a las funciones convencionales, se comportan de forma muy diferente como ver verá más adelante.

```
let a = new Array(3, 40, 200, 5, 1);
a.sort((a, b) => {
    return a b;
});
```

Como se puede apreciar, la función de ordenación ascendente resulta ser un código corto y sencillo. Sólo a modo de aclaración, si cambiásemos la expresión "a b" por "b a", el resultado de la ordenación sería descendente.

Ahora veamos el ejemplo un poco más complejo con funciones de este tipo.

```
let ArrayOperations = function() {
    return Calculadora
}

ArrayOperations.sort = (arr) => arr.sort((a, b) => a - b );

ArrayOperations.max = (arr) => {
    return arr.reduce((a,b) => {
       return a >= b ? a : b;
    });
}

ArrayOperations.min = (arr) => {
    return arr.reduce((a,b) => {
        return a <= b ? a : b;
    });
}

ArrayOperations.addIVA = (arr, iva) => {
    return arr.map((a) => {
       return a * (1 + iva);
    });
}

let a = new Array(3, 40, 200, 5, 1);
console.log(ArrayOperations.sort(a));
console.log(ArrayOperations.max(a));
console.log(ArrayOperations.min(a));
console.log(ArrayOperations.addIVA(a, 0.21));
```

Si observamos el código anterior, una de las cosas que podremos ver que, por ejemplo, el método **sort** está declarado sin llaves y sin la sentencia **return**, pero, sin embargo, funciona perfectamente. Cuando se declaran funciones flecha en una única línea se pueden obviar las llaves y, por tanto, también la instrucción return.

Y otra cosa que se puede observar es que, las funciones **max** y **min**, podrían haberse desarrollado utilizando la librería Math de JavaScript, no obstante, aquí se han diseñado de la forma tradicional, como si no la tuviésemos o conociésemos. En ellas, sí que se utilizan las llaves y la sentencia return.

Por último, indicar que, las funciones flecha, se suelen ver mucho en declaraciones de promesas (que se verán más adelante) porque se vuelven, si cabe, más legibles y ayudan a su comprensión. Un ejemplo de ello podría ser:

```
leerFichero().then(() => analizarFichero()).then(() =>
devolverResumen());
```

5.5.1 Ventajas e inconvenientes

Las funciones flecha pueden ser un gran aliado a la hora de desarrollar clases u objetos, sin embargo, tienen un gran inconveniente, no pueden asociarse a un

constructor. Si intentásemos realizar un new con una función flecha, lo menos que puede pasar es se produzca un error de "typeError".

Otra cosa que resulta llamativa es que las funciones flecha no tienen vinculado ningún objeto de binding como pueda ser this, super o arguments. De hecho, si llamásemos a this desde dentro de una función flecha, lo que se nos devolvería es el objeto global (en modo no estricto) o undefined (en modo estricto), a no ser que esté contendida dentro de otra función.

5.6 FUNCIONES ESPECIALES

En JavaScript existen unas funciones especiales que sirven para un mismo objetivo.

5.6.1 Función de prototipo bind

El método **bind** es una característica de JavaScript que permite crear una nueva función ligada en tiempo de ejecución a la que se le asigna un objeto **this** concreto, además de poder enviarle otros parámetros. Es decir, el método bind crea una copia de la función, pero bajo un contexto (this) que le indicamos como primer parámetro.

No obstante, this puede ser ignorado si la invocación viene precedida de la palabra reservada **new**.

Cabe destacar que, esta "copia de función" o "función ligada" **no se ejecuta**, sólo se realiza su definición para, más tarde, ser llamada.

Cuando se utiliza bind, el primer argumento es el objeto que se utilizará como this, y el resto, serán utilizados como parámetros comunes.

```
funcion.bind(nuevoThis, arg1, arg2, ..., argN);
```

5.6.1.1 EJEMPLO

```
// Definimos la función saludar
let saludar = function(){
    alert("Hola " + this.nombre)
}

// Definimos el objeto Persona
function Persona(nombre, apellidos, edad) {
    this.nombre = nombre + " " + apellidos;
    this.edad = edad;
};

// Llamaaos a la función
saludar.call(new Persona("Pablo", "Fernández", 18));
```

Si ejecutamos el anterior código, podremos observar que no hace nada, o al menos eso parece. Efectivamente, como hemos dicho antes, el método bind no ejecuta la función, por lo que no hace nada porque no se ha llamado nada.

Para hacer que se ejecute debemos añadir unos paréntesis al final.

```
saludar.bind(new Persona("Pablo", "Fernández", 18))();
```

5.6.2 Función de prototipo call

El método **call** es una característica de JavaScript que permite enviar a una función el objeto que actuará como **this**.

El método **call** no sólo admite un argumento. Si se necesitan enviar más argumentos a la función, además del que actuará como this, se pueden ir añadiendo detrás de él separándolos por comas.

```
funcion.call(nuevoThis, arg1, arg2, ..., argN);
```

5.6.2.1 EJEMPLO

```
// Definimos la función saludar
let saludar = function(){
    alert("Hola " + this.nombre)
}

// Definimos el objeto Persona
function Persona(nombre, apellidos, edad) {
    this.nombre = nombre + " " + apellidos;
    this.edad = edad;
};

// Llamaaos a la función
saludar.call(new Persona("Pablo", "Fernández", 18));
```

Si observamos el resultado de la ejecución del anterior código, veremos que el navegador nos muestra una alerta que dice "Hola Pablo Fernández" porque, cuando se ejecuta la función de saludar, el objeto **this** toma como valor el objeto que se ha enviado como parámetro, en este caso el objeto Persona.

5.6.3 Función de prototipo apply

El método **apply** es similar al método **call**. La diferencia más notable es que apply invoca a las funciones asignando explícitamente el objeto this y que, en vez de aceptar una lista de argumentos, lo que acepta es un segundo argumento que contiene un array de valores.

Puede parecer absurdo tener dos funcionalidades tan similares, sin embargo, apply puede resultar interesante cuando no tenemos un número fijo de argumentos o cuando un argumento puede tener un número impredecible de valores.

Cierto es, además, que este array de valores podría ser sustituido por el objeto arguments, sin embargo, el código quedará más limpio si lo utilizamos en su forma predefinida.

5.6.3.1 EJEMPLO

Tomemos como ejemplo, el mismo que para el método call.

```
let saludar = function(){
    alert("Hola " + this.nombre)
}
```

```
function Persona(nombre, apellidos, edad) {
    this.nombre = nombre + " " + apellidos;
    this.edad = edad;
};

saludar.apply(new Persona("Pablo", "Fernández", 18));
```

Si observamos el resultado de la ejecución del anterior código, comprobaremos que es exactamente el mismo que si lo ejecutásemos con el método call. También comprobaremos que el objeto **this** toma su mismo valor, es decir, el objeto que se envió como parámetro que, en este caso, vuelve a ser el objeto Persona.

5.6.4 Diferencias entre call y apply

Ahora supongamos un caso en el que una empresa quiere dar la bienvenida a los usuarios cuando entran, sin embargo, su nombre de identificación puede ser una única palabra o dos (véase 'Pablo' o 'Pablo Fernández').

```
function Empresa(nombre) {
    this.nombre = nombre || 'una empresa desconocida';
};

let saludar = function(nombre, apellidos){
    if(typeof apellidos == "undefined") apellidos = "";

    const msg1 = "Acabas de acceder a " + this.nombre;
    const msg2 = "Bienvenid@ " + nombre + ' ' + apellidos;

    alert(msg1 + '.<br/>' + msg2);
}
```

Si observamos la función Empresa, vemos que, si el parámetro nombre no está definido o está determinado a **null**, se establece al valor "una empresa desconocida" por defecto.

En la función saludar pasa un poco lo mismo, pero se realiza de otra manera. Si el tipo de dato de apellidos es **undefined** (no está definido) se establece a cadena vacía.

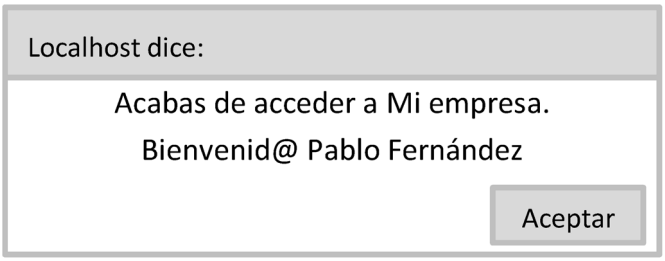
Si ahora deseamos ejecutar función con **call**, lo que deberemos hacer es:

```
let empresa = new Empresa("Mi empresa");
saludar.call(empresa, 'Pablo', 'Fernández');
```

Mientras que, si queremos ejecutar la función con **apply**, lo que deberemos hacer es:

```
let empresa = new Empresa("Mi empresa");
saludar.apply(empresa, ['Pablo', 'Fernández']);
```

El resultado, en ambos casos, será algo parecido a:



5.7 CONTEXTOS Y ENCAPSULAMIENTO

Según se va ejecutando un script, se van generando contextos.

Un **contexto** podría definirse como un fragmento o sección de memoria que contiene todas las variables y objetos a las que la función actual tiene acceso. Mientras la función actual esté ejecutándose, su contexto permanecerá. Cuando esta termine, su contexto terminará con ella, es decir, se eliminará.

Cualquier objeto o función que esté dentro de un contexto tendrá acceso a todas las variables y objetos que se encuentran a su mismo nivel y a todas las variables y objetos que pertenezcan a los contextos de niveles superiores.

Imaginemos que estamos declarando una nueva función. Durante ese proceso de "diseño" podemos distinguir dos contextos, el contexto dónde se define la función y el contexto que crea la función. Veamos un ejemplo:

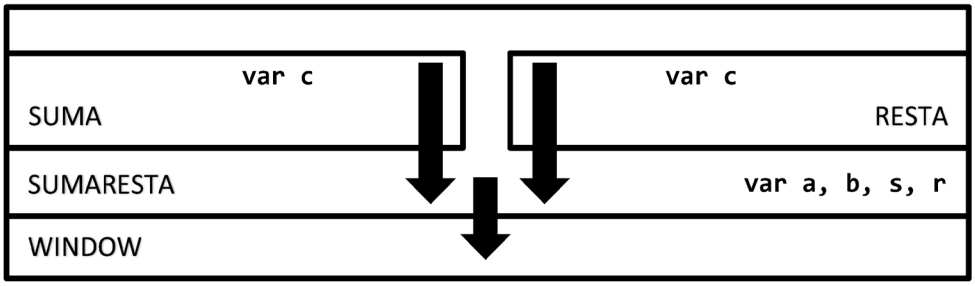
```
function sumaResta(a, b){
    let s = suma();
    let r = resta();

    function suma(){ let c = a + b; return c }
    function resta(){ let c = a b; return c }
```

```
    return [s, r];
}

console.log(sumaResta(3, 2));
```

Según el ejemplo expuesto, desde cualquiera de las funciones **suma** o **resta** podemos acceder a la variable **c**, a la función **sumaResta** y a sus variables **a**, **b**, **s** y **r**.

Sin embargo, desde la función **sumaResta** podemos acceder a las funciones de **suma** y **resta**, a sus variables **a**, **b**, **s** y **r**, pero no a la variable **c**.

Si tuviésemos que representarlo gráficamente, sería algo así:



Si observamos detenidamente el gráfico, podremos ver que todo se gestiona como si fuese una especie de pila, en dónde los elementos más internos tienen acceso a las variables y objetos de los elementos más externos.

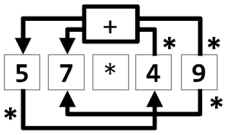
También podemos observar que, las funciones **suma** y **resta** están dentro de la función **sumaResta**, por lo que no podrán ser llamadas desde fuera de su contexto. Esto es lo que se suele denominar como encapsulación o encapsulamiento.

Sé que el **encapsulamiento** es un concepto que, en ocasiones, se confunde, por ello intentaremos resumirlo de una forma sencilla. Si los contextos permiten el acceso a las variables y objetos que están en niveles superiores y ya definidos, el encapsulamiento impide el acceso a las variables u objetos desde fuera de su propio contexto.

En nuestro ejemplo de **sumaResta**, las funciones de **suma** y **resta** se encuentran encapsuladas, por lo tanto, si intentamos acceder a alguna de ellas desde fuera de la susodicha función **sumaResta**, se producirá un error indicándonos que no existe.

5.8 EJERCICIOS RESUELTOS

Cuenta atrás	Código QR
Crear una función que genere una cuenta atrás a partir de un valor dado en segundos y otra que provoque la parada y puesta a cero. **https://codepen.io/pefc/full/bGxoyQX**	

Regla Producto Rápido	Código QR
Crear dos funciones que realicen el producto de dos números de 1 o 2 dígitos (p.e. 57*49) de forma normal y a través de la siguiente regla: 	

6

DECLARACIÓN DECLASES EN JAVASCRIPT

La construcción de clases es una característica que se introdujo en el estándar ECMAScript 2015 y que pretende ser una mejora sintáctica de la herencia basada en prototipos.

Como no podía ser de otra manera, las clases no son otra cosa que un tipo especial de objeto, donde la sintaxis cambia para intentar ser un poco más legible y asemejarse más a cómo se codifica en otros lenguajes de programación.

De hecho, una característica común que comparten las clases y las funciones en JavaScript es que se pueden codificar como una expresión o como una declaración.

6.1 CREACIÓN DE CLASES

Como decíamos existen dos formas de definir clases. Si lo que se desea es realizar una declaración de clase, la forma de crearla en JavaScript es comenzando con la palabra reservada **class**.

```
class Persona {
    constructor(nombre, apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

Si lo que se desea es realizar una definición de clase a modo de expresión, la forma de hacerlo es declararla como una variable y escribir la clase, igual que si fuese una declaración, pero sin el nombre.

```
let Persona = class {
    constructor(nombre, apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

Cabe destacar que las clases de JavaScript funcionan de manera diferente a como lo hacen las funciones.

- Las clases **siempre se ejecutan en modo estricto**, lo que hace que mejore su rendimiento.
- Las clases pueden tener **la declaración de variables en cualquier posición** del código ya que son asignadas durante el proceso de compilación. No como pasa en las funciones que requieren tener la declaración de variables al principio de la definición.

6.2 INSERCIÓN DE MÉTODOS

Como se ha visto, cuando se empieza a escribir una clase, lo primero que se hace es definir su **constructor**. El método **constructor** es el encargado de crear e inicializar la clase y sólo puede ser definido una única vez, puesto que es un método considerado especial. Si por casualidad, durante la definición del código, apareciese otro método constructor, el sistema lanzaría una excepción de error de sintaxis.

Pero, evidentemente, una clase no sólo tiene un método constructor, también necesita de otros métodos para poder llegar a ser funcional.

Para definir un método en una clase, lo que se debe hacer es declararlo como si fuese una función, pero sin la palabra reservada **function**.

```
getNombre(){
    return this.nombre + " " + this.apellidos
}
```

Este tipo de métodos sólo pueden ser llamados cuando la clase está instanciada. Probablemente, esta afirmación no le resulte extraña a nadie, si tenemos

en cuenta que es una característica normalizada en la definición de clases de cualquier lenguaje de programación.

Y posiblemente tampoco resultará raro, como pasa en otros lenguajes, la necesidad de poder definir métodos estáticos, es decir, métodos que no requieren que se instancie la clase.

Si queremos crear un método para que sea accesible sin tener que instanciar la clase, lo que se debe hacer es definirlo con la palabra reservada **static,** la cual permite definir un método para que pueda ser utilizado precisamente así, sin tener que instanciar su clase.

```
static getList(){
    let personasList = globalThis.personas;
    for(let x in personasList){
        console.log(personasList [x]);
    }
}
```

Si observamos el ejemplo, veremos que se utiliza el objeto **globalThis**. En lo referente a clases, este objeto, permite acceder al objeto global al que pertenece el método, es decir, a la clase constructora y, esto es así, porque el objeto this hace referencia al método, no a la clase.

6.2.1 Sentencias get y set

Las clases, como los objetos, pueden utilizar la sentencia **get** y la sentencia **set**, sin embargo, en las clases, no deben utilizarse directamente si lo que se está gestionando es una propiedad porque puede provocar desbordamientos de pila. Veamos un ejemplo:

```
class Persona {
    get age(){
        return this.age;
    }

    set age(n){
        this.age = n;
    }
}

let p1 = new Persona("Pablo", "Fernández");
p1.age = 18;
```

Cuando definimos un getter y un setter como los mostrados arriba, y ejecutemos la instrucción **p1.age = 18**, la actualización de la propiedad provocará que se vuelva a ejecutar el setter porque la propiedad y el método tienen el mismo identificador, por lo que entrará en una recursión infinita y causará un desbordamiento de pila.

Para evitar esto, una posible solución es establecer unos nombres de propiedad diferentes a los de la función para manipular la propiedad. Veamos un ejemplo:

```
class Persona {
    get age(){
        return this._age;
    }

    set age(n){
        this._age = n;
    }
}

let p1 = new Persona("Pablo", "Fernández");
p1.age = 18;
```

Como se puede apreciar, las propiedades que se manipulan llevan delante un guion bajo, lo que permite que podamos actualizar y recuperar el valor a través de sus métodos, pero como si fuese una propiedad.

Ahora, sólo por cimentar lo aprendido, veamos un código de ejemplo completo:

```
class Persona {
    constructor(nombre, apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

    nombreCompleto(){ return this.nombre + " " + this.apellidos; }
    get edad(){ return this._age; }
    set edad(n){ this._age = n; }
}

// Creamos una nueva instancia
let p1 = new Persona("Pablo", "Fernández");

// Le asignamos la edad
p1.edad = 18;

// Recuperamos el nombre completo como propiedad
p1.nombreCompleto();
```

Cuando creamos la nueva instancia de Persona, se asignan el nombre y los apellidos a través del constructor. Seguidamente, se le asigna una edad a modo de propiedad y, como última instrucción, se solicita el nombre completo a modo de método normal.

6.3 EXTENSIÓN DE CLASES

La extensión de clases es también una característica muy habitual en los lenguajes de programación y son muy útiles cuando se desean crear objetos que tienen rasgos o herencias comunes.

De hecho, en JavaScript, es frecuente ver extensiones de clases sobre objetos nativos, como el objeto Date, para obtener funcionalidades personalizadas y manejarlas como si fuesen funciones nativas.

Además, las extensiones permiten sobrescribir los métodos, incluso lo constructores por defecto, lo que proporciona mucha potencia y facilidad de uso.

La forma de crear este tipo de clases es a través de la palabra reservada **extends**.

```
class Persona {
    constructor(nombre, apellidos) {
        this.nombre = nombre;
       this.apellidos = apellidos;
    }

    saluda(){
       return "Hola soy " + this.nombre + " " + this.apellidos;
    }
}

class Estudiante extends Persona {
    saluda(){
        return "Hola soy " + this.nombre + " " + this.apellidos +
        " y soy estudiante";
    }
}

// Creamos una nueva instancia
let e1 = new Estudiante("Pablo", "Fernández");

// Le pedimos que salude
e1.saluda();
```

Como se aprecia en el ejemplo, para instanciar un nuevo estudiante, lo que se hace internamente es crear una instancia de la clase Persona y luego se sobrescribe el método saluda para que haga referencia al propio de la clase Estudiante.

6.3.1 Extensión a través de species

La ex tensión a través de species (perteneciente al tipo de datos Symbol) nos permite trabajar con otro tipo de objeto en vez de con el constructor propio de la clase.

```
class String2 extends String {
    static get [Symbol.species]() { return String; }
}

let s = new String2("Esto es una prueba");
let r = s.toString();

console.log(s);                                    // Devuelve String2
console.log(r);                                    // Devuelve un String
console.log(s instanceof String);                  // Devuelve true
```

Lo que, en realidad, está sucediendo es que, la propiedad **species** nos devuelve el constructor predeterminado para el objeto solicitado, en este caso String y, eso, hace que podamos devolver objetos de la clase String desde los métodos de la clase String2.

6.3.2 Extensión a través de super

Si se desea, también es posible llamar a un método de una clase padre desde la extensión. La forma de hacerlo es utilizando la palabra reservada **super**.

Si la palabra **super** la ponemos en el constructor, tendremos acceso a todas las propiedades y métodos y se instanciará como un objeto de la clase padre, además de la suya.

```
class CustomArray extends Array{
    constructor(){
        super();
    }

    static get [Symbol.species]() { return Array; }

    sumarTodo(){
       return this.reduce(function(a, b){ return a + b });
```

```
    }
}

let ca = new CustomArray();
ca.push(1,1,2,3,5,8);
ca.sumarTodo();
```

Si ejecutamos el código anterior, veremos que nos devuelve 20, que es la suma de todos los elementos que hay en el array, en este caso.

Cabe destacar que la variable **ca** es una instanciación tanto del objeto CustomArray como al objeto Array.

```
console.log(a instanceof MyArray);                    // Devolverá true
console.log(a instanceof Array);                      // Devolverá true
```

Sin embargo, si la variable la hubiésemos instanciado como Array, en vez de CustomArray, la única diferencia perceptible para nosotros, sería que no tendríamos disponible el método sumarTodo.

Pero la palabra **super** no es sólo aprovechable para la construcción de clases, también se puede utilizar para llamar a los métodos del padre desde uno de los métodos de la extensión. Veámoslo con un ejemplo.

```
class Persona {
    constructor(nombre, apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

    saluda(){
        return "Hola soy " + this.nombre + " " + this.apellidos;
    }
}

class Estudiante extends Persona {
    saluda(){
        return super.saluda();
    }
}

// Creamos una nueva instancia
let e1 = new Estudiante("Pablo", "Fernández");

// Le pedimos que salude
e1.saluda();
```

Si ejecutamos el código, podremos comprobar que, ahora, el saludo es "Hola soy Pablo Fernández", sin la coletilla de estudiante. Esto sucede porque, en realidad, desde el método **saluda** de la clase extendida, se está llamando al método **saluda** de la clase padre.

6.4 CLASES ABSTRACTAS Y MIXINS

Según la definición del estándar de ECMAScript sólo puede haber una clase padre. En términos de herencia, la clase padre se puede considerar una superclase y la clase hija una subclase (lo que hasta ahora hacíamos con una extensión). Pues bien, las clases abstractas son aquellas que aplican múltiples superclases para crear grupos relacionados de clases modificadas.

Los mixins, por su parte, son unas subclases que, normalmente, no están ideadas para funcionar de forma autónoma, sino que están pensadas para proporcionar otras capacidades.

En JavaScript, hoy por hoy, ni las clases abstractas ni los mixins son una opción a tener en cuenta, puesto que muchos navegadores no tienen esta funcionalidad soportada.

6.5 EJERCICIOS RESUELTOS

Regla Producto Rápido con clase	Código QR
Convertir el código de la práctica anterior referente a la regla de multiplicación rápida para que funcione mediante una clase de JavaScript. **https://codepen.io/pefc/full/PodEvmO**	

7

LOS EVENTOS EN JAVASCRIPT

En la programación del lado del servidor las aplicaciones se ejecutan en orden de principio a fin y, normalmente, bajo el paradigma de la programación orientada a objetos. Sin embargo, en los lenguajes de script como es JavaScript, aunque la ejecución sigue los mismos principios, el paradigma que se utiliza es la programación orientada a eventos.

Un evento podría definirse como la interacción que se produce cuando un usuario realiza alguna acción sobre una aplicación o sistema.

Por ejemplo, cuando un usuario realiza una acción como escribir en un elemento de formulario o realizar un clic en un botón, el sistema, provoca la ejecución de un fragmento de código específico. Ese fragmento de código, es seleccionado mediante un oyente o listener que es quién establece un vínculo entre la acción del usuario y el objeto con el que interactúa.

Sólo como aclaración. Cuando decimos interacción con el usuario, no nos referimos sólo a la interacción entre máquina y persona, también puede referirse a la interacción con el sistema u otra entidad. Sirva como ejemplo que hay eventos que pueden ser programados por los desarrolladores para ejecutarse de forma automática.

El DOM (Document Object Model), del cual se hablará más adelante, tiene una serie de eventos que permite a los lenguajes como JavaScript registrar esos listeners, que hablábamos antes, en cualquiera de sus elementos o nodos.

7.1 PRINCIPIO FUNDAMENTAL DE PROPAGACIÓN

El principio fundamental de propagación o "burbujeo" en JavaScript afirma: Después de que un evento se desencadene en el elemento más profundo posible, se dispararán los mismos eventos en sus ancestros por orden de anidamiento.

Dicho de otro modo, supongamos que se tienen tres elementos anidados, el primero en disparar el evento será el que esté más abajo en la jerarquía, es decir, el nieto. Después, se disparará su ancestro inmediatamente superior, es decir, su padre y, finalmente, se disparará el ancestro de su ancestro inmediatamente superior, es decir, su abuelo.

Este proceso de propagación, normalmente, se realiza en 2 fases. Primero se captura y luego se propaga. La **captura** se realiza en orden descendente, es decir, va bajando desde el abuelo hasta el nieto y es aquí donde lanza el primer evento. La **propagación**, sin embargo, se realiza al revés, es decir, va subiendo en la jerarquía y va lanzando el evento en cada elemento hasta llegar al último en la escala.

Todo sea dicho de paso, este principio proporciona muchas ventajas e inconvenientes a los desarrolladores puesto que puede ayudar a controlar todo lo que sucede en la página, pero también puede provocar pérdidas de rendimiento significativas. Por esta razón, existe un método que permite detener la propagación de eventos.

```
event.stopPropagation();
```

7.2 EL OBJETO EVENT

Cuando el usuario realiza una interacción con la aplicación, un objeto Event es creado y enviado a los listeners para que pueda ser accedido desde los manejadores de eventos a través de un parámetro de entrada.

Todos los eventos en JavaScript tienen muchas propiedades y métodos, algunos específicos de cada tipo. Por esa razón, sólo vamos a destacar los importantes o más comunes.

7.2.1 Propiedades más frecuentes

7.2.1.1 PROPIEDAD TARGET

Contiene la referencia al objeto del elemento que lanzó el evento.

Todos los eventos tienen esta característica en su definición y es habitual utilizarla para acceder o modificar sus propiedades.

```
input.addEventListener('change', function (e){
    // Mostramos por consola el valor del input cuando cambia
    console.log(e.target.value);
});
```

7.2.1.2 PROPIEDAD TYPE

Contiene un String con el nombre o identificador del tipo de evento y, aunque resulte increíble, este parámetro puede no estar disponible en Internet Explorer.

```
input.addEventListener('change', function (e){
console.log(e.type);                // Devuelve 'change'
});
```

7.2.1.3 PROPIEDAD BUBBLES

Contiene un valor booleano que indica si el evento puede propagarse a través del DOM según el principio fundamental de propagación. Por defecto, está establecido a **true**, lo que significa que el efecto de propagación o "burbujeo" está habilitado.

No todos los eventos tienen esta característica en su definición, sin embargo, es habitual encontrarlo en los más frecuentemente utilizados.

```
elemento.addEventListener('keydown', function (e){
    if(e.bubbles) console.log("El burbujeo está habilitado!");
});
```

7.2.1.4 PROPIEDAD CANCELABLE

Contiene un valor booleano que indica si el evento puede ser cancelado por el usuario. Por defecto, está establecido a **true**, lo que significa que se puede cancelar.

La cancelación de eventos es una característica que sólo se puede determinarse en el momento en el que son inicializados y la forma de hacerlo es a través del método **preventDefault**.

```
// Mostrar si es cancelable en el momento de hacer click
elemento.onclick = function(e){ console.log(e.cancelable); });
```

7.2.1.5 PROPIEDAD DEFAULTPREVENTED

Contiene un valor booleano que indica si el evento lanzó una llamada al método **preventDefault**.

```
console.log(event.defaultPrevented);
```

7.2.1.6 PROPIEDAD PATH

Contiene una estructura de árbol con la ruta desde el objeto window hasta el elemento que lanzó el evento.

```
console.log(event.path);
```

Viendo el contenido de esta propiedad podemos ver quiénes son los ancestros del elemento que lanzó el evento, deducir el posible burbujeo o acceder a uno de sus ancestros directamente, entre otras cosas.

```
KeyboardEvent {
    ...
    ► path: Array(5)
    0: input#fecha
    1: body.chrome
    2: html
    3: document
    4: window
}
```

7.2.1.7 PROPIEDADES CLIENTX, CLIENTY, PAGEX Y PAGEY

Estas propiedades permiten conocer la posición del puntero del ratón en el momento que el evento ocurrió.

Las propiedades que contienen la letra X hacen referencia la horizontalidad y las propiedades que contienen la letra Y hacen referencia la verticalidad.

Estas propiedades están disponibles para todos los eventos en los que interviene el ratón.

```
// Mostrar la posición del puntero en el momento de hacer click
elemento.onclick = function(event){
    console.log(event.clientX, event.clientY);
});
```

7.2.1.8 PROPIEDADES KEYCODE Y WHICH

Estas propiedades permiten conocer el código de la tecla que se presionó. Si por cualquier razón no se pudiese recuperar el código de tecla, estos valores serían establecidos 0.

Estas propiedades están disponibles para los eventos keydown, keyup y keypress.

```
input.addEventListener('keydown', function (e){
    var code = e.keyCode | e.which;
    console.log(code); // Si pulsamos la tecla 'a' devuelve 65
});
```

En el ejemplo podemos observar que ser está realizando una operación tipo OR entre las dos propiedades por si una de las dos no existiese, estuviese vacía o fuese nula.

7.2.1.9 PROPIEDADES ALTKEY, CTRLKEY Y SHIFTKEY

Estas propiedades sólo están disponibles para los eventos **keydown** y permiten conocer si la tecla pulsada es una combinación de dos como, por ejemplo, Ctrl+F5.

Todas ellas contienen un valor booleano que indica si están o no presionadas, no obstante, sólo tienen valor cuando se están presionando, como mínimo, dos teclas a la vez.

Si únicamente pulsamos la tecla **Shift** el evento devolverá el código de tecla 16 y la propiedad **shiftKey** estará a false o no tendrá valor.

Si únicamente pulsamos la tecla **Ctrl** el evento devolverá el código de tecla 17 y la propiedad **ctrlKey** estará a false o no tendrá valor.

Si únicamente pulsamos la tecla **Alt** o **Alt Gr** el evento devolverá el código de tecla 18 y la propiedad **altKey** estará a false o no tendrá valor.

```
input.addEventListener('keydown', function (e){
    var code = e.keyCode | e.which;
    console.log('Ctrl presionada:', e.ctrlKey, ', Código:', code);
});
```

Si ejecutásemos el código de este evento en un elemento de formulario podríamos comprobar que, si pulsamos la tecla 'a', lo que se mostrará por consola es "Ctrl presionada: false, Código: 49" y si presionamos la tecla Ctrl + F1, lo que se mostrará es "Ctrl presionada: true, Código: 112".

7.3 LA INTERFAZ TOUCHEVENT

La interfaz TouchEvent es un objeto que puede ser utilizado cuando se está trabajando con dispositivos sensibles al tacto como son las pantallas táctiles o trackpads. Además, tiene la peculiaridad de poder gestionar más de un punto de contacto a la vez o detectar el desplazamiento.

El evento TouchEvent utiliza el objeto Touch que describe desde la posición y el tamaño hasta el elemento que lo percibe.

Los manejadores de eventos disponibles para esta interfaz son: touchstart, touchend, touchleave, touchmove y touchcancel.

Entre sus propiedades más importantes podemos encontrar:

Propiedad	Descripción
changedTouches	Devuelve un array con la información de cada toque cuando se añade o elimina un toque, o cuando se produce un cambio de estado en los toques.
targetTouches	Devuelve un array con la información de cada toque con respecto al elemento de origen.
touches	Devuelve un array con la información de cada toque que encuentra en el dispositivo.

7.3.1 El objeto Touch

Este objeto es el responsable de describir el punto de contacto cuando se realiza un toque en un dispositivo sensible al tacto.

Al ser un objeto que representa una entidad estática, no dispone de método alguno y sólo proporciona una definición de propiedades descriptivas.

7.3.1.1 PROPIEDADES MÁS IMPORTANTES

Propiedad	Descripción
create	Devuelve un identificador único para cada toque. Mientras el toque se siga produciendo, es decir, mientras no se levante el puntero o dedo, el código identificador se mantendrá. Esto puede servir para controlar el desplazamiento, por ejemplo. `Object.create({});  // Devolverá {}`
identifier	
clientX	Devuelve la posición en píxeles del toque respecto al eje horizontal desde la parte la izquierda de la ventana.
create	Este método es otra forma de llamar al constructor de la clase. `Object.create({});  // Devolverá {}`
clientY	Devuelve la posición en píxeles del toque respecto al eje vertical desde la parte superior de la ventana.
pageX	Devuelve la posición en píxeles del toque respecto al eje horizontal desde la parte la izquierda del documento.
pageY	Devuelve la posición en píxeles del toque respecto al eje vertical desde la parte superior del documento.
radiusX	Devuelve el radio del eje horizontal (en píxeles) que define la elipse que será utilizada para manejar el área de contacto.
radiusY	Devuelve el radio del eje vertical (en píxeles) que define la elipse que será utilizada para manejar el área de contacto.
radiusAngle	Devuelve el valor del ángulo de rotación. Este ángulo sólo puede adquirir valores entre 0 y 90. El 0 indica que está en posición vertical y el 90 indica que está en posición horizontal.
force	Devuelve un valor numérico decimal que indica la presión del puntero o dedo. Va desde 0.0 a 1.0 y sólo lo soportan algunos dispositivos.
screenX	Devuelve la posición en píxeles del toque respecto al eje horizontal desde la parte la izquierda de la pantalla.
screenY	Devuelve la posición en píxeles del toque respecto al eje vertical desde la parte superior de la pantalla.
target	Devuelve el elemento que recibió el toque.

7.4 LA INTERFAZ KEYBOARDEVENT

La interfaz KeyboardEvent es un objeto que puede ser utilizado cuando se está trabajando con elementos que pueden interactuar con el teclado. Sólo cuenta con tres métodos, pero permiten un gran control sobre el elemento que se maneja.

Los manejadores de eventos disponibles para esta interfaz son: keydown, keypress y keyup.

7.4.1 Propiedades más importantes

Propiedad	Descripción
altKey	Es un valor booleano que indica si la tecla ALT fue presionada cuando el evento de **keydown** se lanzó.
bubbles	Es un valor booleano que indica si el evento puede propagarse a través del DOM según el principio fundamental de propagación. Por defecto está establecido a **true**, lo que significa que el efecto de propagación o "burbujeo" está habilitado.
cancelable	Es un valor booleano que indica si el evento puede ser cancelado o no. Por defecto está establecido a **true**, lo que significa que se puede cancelar. Esta característica sólo se puede determinarse en el momento en que es inicializado el evento.
charCode	Es el valor del carácter Unicode que disparó o lanzó el evento. En muchas ocasiones suele estar establecido a 0.
code	Devuelve un String con el nombre en clave de la tecla que se pulsó. Por ejemplo, si pulsamos la tecla "1" devuelve "Digit1".
ctrlKey	Es un valor booleano que indica si la tecla CTRL fue presionada cuando el evento de **keydown** se lanzó.
key	Es el carácter Unicode de la tecla que lanzó el evento. Por ejemplo, si pulsamos la tecla "A" devuelve "A".
keycode	Es el código Unicode de la tecla que lanzó el evento. Por ejemplo, si pulsamos la tecla "1" devuelve 49. También se corresponde con la tabla de códigos ASCII.
location	Devuelve la ubicación o zona física dónde se encuentra la tecla que disparó en evento. En muchas ocasiones suele estar establecido a 0.
metaKey	Es un valor booleano que indica si la tecla META fue presionada cuando el evento de **keydown** se lanzó. Si estamos en OSX este valor se corresponde con la tecla COMMAND.
shiftKey	Es un valor booleano que indica si la tecla SHIFT (mayúsculas) fue presionada cuando el evento de **keydown** se lanzó.
target	Es el elemento que originó el evento.
type	Es el nombre del evento que se lanzó.
which	Es el código Unicode de la tecla que lanzó el evento. Por ejemplo, si pulsamos la tecla "1" devuelve 49. También se corresponde con la tabla de códigos ASCII y, por lo general, suele ser el mismo que el valor de la propiedad **keyCode**.

7.5 LA INTERFAZ MOUSEEVENT

La interfaz MouseEvent es un objeto que puede ser utilizado cuando se está trabajando con dispositivos de tipo puntero como pueda ser un ratón.

La interfaz MouseEvent cuenta con bastantes métodos que permiten un gran control sobre el elemento que se maneja.

Los principales manejadores de eventos disponibles para esta interfaz son: click, dblclick, mouseeenter, mouseleave, mouseover, mouseout, mousedown, mouseup, mousewheel y mousemove.

7.5.1 Propiedades más importantes

Propiedad	Descripción
button	Devuelve un entero que representa el botón que se ha presionado. Normalmente, el 0 indica que es el botón izquierdo.
butons	Devuelve el número de botones del dispositivo o ratón. Con frecuencia está establecido a 0.
clientX	Devuelve la posición en píxeles del ratón respecto al eje horizontal desde la parte la izquierda de la ventana.
clientY	Devuelve la posición en píxeles del ratón respecto al eje vertical desde la parte superior de la ventana.
movementX	Devuelve la distancia recorrida en píxeles del ratón con respecto a la anterior posición X de la pantalla. Es decir, es la diferencia entre la propiedad screenX del evento actual y la propiedad screenX del evento anterior.
movementX	Devuelve la distancia recorrida en píxeles del ratón con respecto a la anterior posición Y de la pantalla. Es decir, es la diferencia entre la propiedad screenY del evento actual y la propiedad screenY del evento anterior.
offsetX	Sobre el eje horizontal, devuelve la posición en píxeles del puntero con respecto al elemento que provocó el evento.
offsetY	Sobre el eje vertical, devuelve la posición en píxeles del puntero con respecto al elemento que provocó el evento.
screenX	Devuelve la posición en píxeles del ratón respecto al eje horizontal desde la parte la izquierda de la pantalla.
screenY	Devuelve la posición en píxeles del ratón respecto al eje vertical desde la parte superior de la pantalla.
relatedTarget	Devuelve el elemento padre del elemento que recibió el evento
target	Devuelve el elemento que recibió el evento.

7.6 PRINCIPALES MANEJADORES DE EVENTOS

7.6.1 Eventos de ratón

En esta descripción de eventos suponemos que el ratón está configurado para diestros. Por ello, si estuviese configurado para zurdos, los eventos asociados al botón izquierdo serían los del botón derecho.

Todos los eventos aquí descritos pueden ser utilizados por todos los elementos.

Evento	Cuando se produce
click	Cuando pulsa el botón izquierdo del ratón.
dblclick	Cuando se pulsa rápidamente dos veces el botón izquierdo del ratón.
mousedown	Cuando se presiona un botón del ratón, pero todavía no se ha soltado.
mouseup	Cuando se suelta el botón del ratón.
mouseover	Cuando el puntero del ratón está dentro de los límites del elemento. **Nota:** Si se define este evento en elementos que contienen otros elementos, el evento será efectivo para todos y cada uno de los hijos, incluyendo el padre. Es decir, que se lanzará cuando pase por el elemento seleccionado y cuando pase por cualquiera de sus hijos.
mouseout	Cuando el puntero del ratón sale de los límites del elemento. **Nota:** Si se define este evento en elementos que contienen otros elementos, el evento será efectivo para todos y cada uno de los hijos, incluyendo el padre. Es decir, que se lanzará cuando pase por el elemento seleccionado y cuando pase por cualquiera de sus hijos.
mouseenter	Sucede cuando el puntero del ratón entra al elemento.
mouseleave	Sucede cuando el puntero del ratón sale del elemento.
mousemove	Mientras se está moviendo el puntero del ratón dentro de los límites del elemento.
mousewheel	Cuando se manipula la rueda del ratón. Este método está obsoleto.
drag	Cuando se arrastra un elemento.
drop	Cuando se suelta un elemento durante el proceso de arrastre.

7.6.2 Eventos de formulario

Evento	Cuando se produce	Se puede utilizar en
select	Al seleccionar un texto.	input y textarea
change	Cuando el valor del campo cambia y pierde el foco.	input, select y textarea
submit	Cuando se envía el formulario, ya sea por una pulsación en un botón de tipo "submit", ya sea por envío a través de JavaScript.	form
reset	Cuando se inicializa un formulario o se pulsa un botón de tipo "reset".	form
focus	Cuando el elemento de formulario toma el foco.	button, label, input, select, textarea y body
focusin	Cuando el elemento de formulario está a punto de tomar el foco.	button, label, input, select, textarea y body
focusout	Cuando el elemento de formulario está a punto de perder el foco.	button, label, input, select, textarea y body
blur	Cuando el elemento de formulario pierde el foco.	button, label, input, select, textarea y body

7.6.3 Eventos de HTML

Evento	Cuando se produce	Se puede utilizar en
load	Cuando el objeto que lo lanzó terminó la descarga. Cuando se utiliza para controlar la página, indica que todo ha sido descargado, pero no así el DOM.	Todo elemento susceptible de poder descargar algo. Por ejemplo, scripts, hojas de estilo, imágenes, marcos, ...
unload	Cuando se abandona la página por una nueva solicitud de navegación o cualquier otro motivo.	Body
abort	Cuando se aborta la carga de un elemento. Sólo es compatible con Internet Explorer.	img
error	Cuando hay un error de carga de imágenes o de script en la página.	img y por el DOM
resize	Cuando el tamaño de la ventana cambia.	Con el objeto window de JavaScript
scroll	Cuando el valor que indica la posición de una barra de desplazamiento horizontal o vertical cambia.	Todos los elementos que sean susceptibles de tener una barra de desplazamiento
DOMContentLoaded	Cuando el documento ha sido completamente cargado y que el árbol DOM está construido, pero puede que haya recursos externos que no estén cargados aún.	En el Objeto document de JavaScript

7.6.4 Eventos de tratamiento táctil

Todos los eventos aquí descritos pueden ser utilizados por todos los elementos.

Evento	Cuando se produce
touchstart	Cuando se presiona con el dedo o puntero en la pantalla de un dispositivo táctil.
touchend	Cuando se levanta el dedo o puntero de una pantalla de un dispositivo táctil.
touchmove	Cuando se arrastra el dedo o puntero por la pantalla de un dispositivo táctil.
touchenter	Cuando el dedo o puntero entra en contacto con la pantalla de un dispositivo táctil. Este evento no se propaga.
touchleave	Cuando el dedo o puntero abandona el contacto con la pantalla de un dispositivo táctil. Este evento no se propaga.
touchcancel	Cuando se interrumpe la pulsación en un dispositivo móvil.

7.7 OYENTES O LISTENERS

La especificación del DOM define dos métodos para definir y controlar los eventos que suceden en las páginas o aplicaciones. Estos métodos son **addEventListener** y **removeEventListener**.

7.7.1 Método addEventListener

Este método define un manejador de evento para un elemento u objeto específico.

Se puede utilizar para elementos de formulario, en los objetos document o window de JavaScript o incluso en peticiones Ajax.

Los manejadores de eventos suelen ser referidos con el prefijo “on” delante, sin embargo, en este método, se debe obviar y poner sólo el identificador de evento.

A continuación, se muestra cómo definir un evento en la barra de desplazamiento de un documento HTML.

```
document.addEventListener('scroll', function (e){
    console.log(e)
});
```

Cada vez que la barra de desplazamiento asociada al documento en pantalla se mueva se mostrará en consola la descripción completa del evento, habitualmente en formato JSON.

7.7.2 Método removeEventListener

Este método elimina un manejador de evento para un elemento u objeto específico.

Se puede utilizar para elementos de formulario, en los objetos document o window de JavaScript o incluso en peticiones Ajax.

Los manejadores de eventos suelen ser referidos con el prefijo "on" delante, sin embargo, en este método, se debe obviar y poner sólo el identificador de evento.

A continuación, se muestra cómo eliminar un evento en la barra de desplazamiento de un documento HTML.

```
function addEvent(){
...
}
document.removeEventListener('scroll', addEvent);
```

El método **removeEventListener** sólo será efectivo cuando se defina sin funciones anónimas, puesto que las funciones anónimas no guardan referencia en memoria y, por tanto, no pueden ser recuperadas por el recolector de basura. Como consecuencia, no pueden ser eliminados los manejadores de eventos.

7.7.3 Otras formas de establecer listeners

7.7.3.1 A TRAVÉS DE HTML

Se puede habilitar un manejador de eventos, a través de HTML, usando el identificador precedido del prefijo 'on' como propiedad.

```
<input type="number" onclick="console.log(Event)/>
```

7.7.3.2 A TRAVÉS DE UN OBJETO O ELEMENTO

Se puede habilitar un manejador de eventos, a través de JavaScript, usando el identificador precedido del prefijo 'on' como método.

```
input.onclick = function (e){ console.log(e) });
```

7.8 PRINCIPALES EVENTOS DEL DOM

7.8.1 Document DOMContentLoaded

Cuando se trabaja con JavaScript una buena práctica es esperar a que el DOM esté cargado para ejecutar un código concreto.

```
function allReady(){
    console.log("El DOM está cargado!");
}

document.addEventListener("DOMContentLoaded", ready);
```

Esto suele ser más rápido de ejecutar que el evento **onload** porque no espera a que las imágenes, marcos o peticiones Ajax, etcétera hayan terminado de cargar.

7.8.2 Window load

Este evento se ejecuta cuando la página se ha cargado por completo, es decir, que se han cargado todos los elementos de la página, incluyendo peticiones Ajax, marcos e imágenes, etcétera.

```
window.onload = function (e){
    console.log("Página cargada");
});
```

7.8.3 Window resize

Este evento se ejecuta cuando se realiza un cambio en el tamaño del campo de visualización de la ventana del navegador.

```
window.onresize = function (e){
    console.log("El tamaño de la ventana cambió");
});
```

7.8.4 El evento scroll

Permite controlar el scroll de un objeto, incluido el objeto **window**.

```
window.onscroll = function (){
    console.log(document.body.scrollTop);
});
```

Esta declaración de evento nos mostrará por consola el valor de la posición de la barra de desplazamiento vertical. Si, por el contrario, queremos conocer la posición de la barra de desplazamiento horizontal sólo debemos cambiar la propiedad del objeto body **scrollTop** por **scrollLeft**.

7.9 EJERCICIOS RESUELTOS

Crear evento personalizado	Código QR
Se trata de crear una funcionlidad que controle cuándo se ha cargado completamente la página. Para ello, nos valdremos de una variable que previamente estará declarada a false. Cuando esta variable, llamada loaded, se establezca a true, el programa terminará y mostrará "¡Página cargada!". **https://codepen.io/pefc/pen/GRXQqea**	

8

EL DOM DE JAVASCRIPT

El DOM (Document Object Model), es un modelo que indica cómo se deben estructurar las páginas o aplicaciones web. Estás páginas o aplicaciones se construyen y se interpretan de forma secuencial, por lo que puede pasar que se desee acceder a un elemento concreto y todavía no esté disponible.

El objeto inicial del que parten todos los objetos, en el DOM, es el objeto **document**, y de este objeto se van definiendo y estructurando todos los demás. Sólo por aclarar, el objeto **document** de JavaScript representa, de alguna manera, a la etiqueta **html** del lenguaje HTML, no a la etiqueta body.

Por si alguien se lo pregunta, el objeto window es quién está por encima del objeto document en la jerarquía, pero eso es porque el objeto window está asociado al navegador, no al DOM.

8.1 PROCESO DE CARGA

Cada vez que insertamos una librería o framework de JavaScript, la página detiene la carga hasta que recibe el fragmento de código solicitado previamente. Este es el motivo por el que las herramientas como Google PageSpeed Insights recomiendan poner la solicitud de los scripts al final de la página, y no al principio.

Evidentemente, si el uso de librerías es escaso, el tiempo de carga de las páginas no se verá muy afectado, sin embargo, lo más normal en un proyecto actual, es que utilice gran cantidad de librerías, frameworks o personalizaciones.

Algunos pensarán que, gran parte de este problema, es solucionable sólo con establecer el atributo **async** en las etiquetas **script**, no obstante, esta configuración

puede provocar que las cosas no se carguen en el orden adecuado porque muchas de los fragmentos de código que insertamos a través de estas etiquetas tienen dependencias con otros.

Sin ir más lejos, el framework Bootstrap requiere que esté ya insertado el código de jQuery. Si se intentase añadir Bootstrap antes que jQuery, provocaría un error en la página y se pararía el proceso de carga.

Por todos estos motivos, el proceso de carga, hay que mirarlo con rigor sabiendo lo que se hace y optimizarlo para bajar al máximo sus tiempos.

8.2 LOS NODOS Y SUS TIPOS

Como decíamos, el objeto inicial del DOM es **document** y, de él, aparecen otros como el objeto **body** o el objeto **head**. Todos estos objetos se suelen denominar comúnmente como nodos por su similitud jerárquica con la teoría de grafos.

Aunque existen 12 tipos de nodos en JavaScript, en realidad, sólo se suelen utilizar 5 porque son los que se necesitan para realizar todas las acciones u operaciones cuando se trabaja con páginas o aplicaciones web.

Tipo Nodo	Descripción
Document	Nodo raíz del que dependen todos los demás.
Element	Son los elementos representados por las etiquetas HTML.
Attr	Son las propiedades de las etiquetas asociadas a cada Element.
Text	Son los valores o textos contenidos dentro de las etiquetas del HTML.
Comment	Son los que se ha generado a partir de etiquetas doctype y que se transforman en nodos o han sido creados como comentarios.

8.3 SELECCIÓN DE ELEMENTOS

Antiguamente, para acceder a los elementos del DOM debíamos recurrir, básicamente a los métodos de **getElementById** y **getElementsByTagName**. El primero, permite recuperar el objeto referenciado a través de un identificador enviado por parámetro mientras que, el segundo, permite recuperar todos los nodos del DOM que concuerden con el nombre de etiqueta proporcionado como parámetro y los devuelve en un objeto de tipo array.

También se podía recurrir al método **getElementsByClassName** que permite recuperar todos los nodos del DOM que concuerden con el nombre de clase proporcionada por parámetro y los devuelve en un objeto de tipo array.

```
// Encontrar todos los elementos tipo tabla
document.getElementsByTagName("table");

// Encontrar todos los elementos que contengan la clase "card"
document. getElementsByClassName("card");

// Encontrar el elemento con id="cabecera"
document.getElementById("cabecera");
```

Por suerte, hoy día disponemos de dos instrucciones más completas y, a veces eficientes, para recuperar los diferentes elementos o nodos del DOM y que veremos más adelante.

8.3.1 Interfaz NodeList

NodeList es una interfaz de JavaScript que representa un conjunto de nodos. Habitualmente, este objeto, será utilizado por métodos como **querySelectorAll** o **childNodes**.

Un de las curiosidades de **NodeList** es que es un objeto que puede tener un comportamiento estático o dinámico, dependiendo de cómo se utilice. Es decir, según sea el caso, los cambios se reflejarán en DOM de forma automática o no. En el caso que **querySelectorAll** la lista devuelta es estática.

```
document.querySelectorAll("body");
```

Si ejecutamos el código anterior, nos devolverá todos los elementos que sean hijos directos de la etiqueta body de HTML.

8.3.1.1 MÉTODO ITEM

Permite acceder a un elemento del objeto NodeList devuelto a través de su índice. Si el índice no se corresponde con ningún elemento del conjunto, devolverá "undefined".

```
document.querySelectorAll("body").item(0)
```

8.3.1.2 MÉTODO FOREACH

Permite recorrer el objeto NodeList devuelto y parsearlo a través de una función de retorno (callback) proporcionada como parámetro.

```
var list = document.querySelectorAll('body > *');
Array.prototype.forEach.call(list, function (element) {
    console.log(element)
});

// Otra manera, un poco más clara, de hacer lo mismo sería:
var list = document.querySelectorAll('body > *');
list.forEach(function (element) {
    console.log(element)
});
```

8.3.2 Los selectores

La base para acceder a los nodos del DOM son los selectores.

Para los que están familiarizados con CSS, les resultará bastante sencillo ya que usa el mismo sistema de selectores para interactuar con los elementos HTML.

Si no se está familiarizado con los selectores de CSS siempre se puede ir al *apéndice I: "Resumen de Selectores de CSS"* para ver una muestra de los selectores más frecuentemente utilizados.

La elección de buenos selectores es un punto a tener en cuenta cuando se desea mejorar el rendimiento del código. Sirva como ejemplo que, incluir el nombre de la etiqueta HTML cuando se realiza una selección por el nombre de clase, puede acotar la búsqueda y disminuir el tiempo de acceso.

Por otro lado, si intentamos ser demasiado precisos en nuestro selector puede que se vuelva perjudicial y convertirse en una búsqueda ineficiente.

A continuación, se muestran unos cuantos ejemplos de cómo realizar la selección:

```
// Encontrar el primer elemento que contenga la clase button
document.querySelector(".button");

// Encontrar todos los elementos que contengan la clase button
document.querySelectorAll(".button");
```

8.3.3 Métodos para acceder a los nodos y elementos

Como decíamos antes, además de los métodos getElementById, getElementsByTagName y getElementsByClassName, JavaScript dispone de otros dos que, en la actualidad, son los más frecuentemente utilizados en el desarrollo de aplicaciones web.

8.3.3.1 MÉTODO QUERYSELECTOR

Este método devuelve el primer nodo (o elemento) que coincida con el selector proporcionado por parámetro. Si la búsqueda fue infructuosa devolverá **null**.

Para que este método sea efectivo, la sintaxis a utilizar en el parámetro de entrada debe seguir el estándar de CSS, de lo contrario, devolverá un valor nulo.

```
document.querySelector("table");
document.querySelector(".tarjeta");
document.querySelector("#cabecera");
```

8.3.3.2 MÉTODO QUERYSELECTORALL

Este método devuelve todos los nodos (o elementos) que coincidan con el selector proporcionado por parámetro. Si la búsqueda fue infructuosa devolverá un objeto **NodoList** con longitud cero.

Para que este método sea efectivo, la sintaxis a utilizar en el parámetro de entrada debe seguir el estándar de CSS, de lo contrario, devolverá un valor nulo.

```
document.querySelectorAll("input");
document.querySelectorAll(".button");
document.querySelectorAll(":checked");
```

Según el orden presentado de los ejemplos:

- La primera instrucción, devolverá un objeto NodoList con todos los inputs que haya en el DOM.
- La segunda instrucción, devolverá un objeto NodoList con todos los elementos que tengan la clase “button” en el DOM.
- La tercera instrucción, devolverá un objeto NodoList con todos los elementos que estén seleccionados o chequeados, es decir, que devolverá una colección con los elementos **option** (pertenecientes a elementos **select**) que estén seleccionados y los inputs de tipo radio y checkbox que estén chequeados.

8.4 MANUPULACIÓN DE NODOS Y ELEMENTOS

En JavaScript, como si de un partido de futbol se tratase existen un conjunto de métodos que podríamos llamar el "dream-team" de la creación por su utilización.

Si pensamos un poco en los métodos que más se utilizan en los proyectos, seguro que llegaremos a la conclusión de que la mayoría de las necesidades de estos proyectos se podrían resolver con las propiedades y métodos que conforman este "equipo".

8.4.1 Interfaz DOMTokenList

DOMTokenList es una interfaz de JavaScript con propiedades similares a un array y que representa un conjunto de tokens (habitualmente nombres o palabras concretas) separados por espacios.

DOMTokenList tiene como índice inicial el 0, al igual que los arrays, y es sensible a mayúsculas y minúsculas.

Dicho así, es posible que este objeto no llame mucho la atención, sin embargo, resulta muy útil porque es quién nos proporciona unos métodos que, seguramente, usaremos con mucha frecuencia. Estos métodos son contains, add, remove y toggle.

Método	Descripción y ejemplo
add	Permite agregar al final un token concreto dentro de un conjunto, siempre y cuando no exista. Es decir, si el identificador que se está intentando agregar en el DOMTokenList no existe, se añade al final. `document.queySelector("body").classList.add("open");`
contains	Permite averiguar si existe un token concreto dentro de un conjunto. Es decir, devuelve true si identificador buscado está contenido en el DOMTokenList. `document.queySelector("body").classList.contains("open");`
item	Permite acceder a un elemento del DOMTokenList a través de su índice. Si el índice no se corresponde con ningún elemento del conjunto, devolverá "undefined". `document.queySelector("body").classList.item(0);`

remove	Permite eliminar un token concreto de un conjunto, siempre y cuando exista. Es decir, si el identificador que se está intentando eliminar del DOMTokenList existe, se elimina. `document.querySelector("body").classList.remove("open");`
toggle	Permite agregar o quitar un token concreto de un conjunto de tokens. Es decir, si el identificador existe, se llamará a **remove**. Si el identificador no existe, se llamará a **add**. `document.querySelector("body").classList.toggle("open");`

8.4.2 Método createElement

Este método permite la creación de nuevos nodos y elementos en el DOM para, más tarde, insertarlos en el documento.

El método **createElement** sólo requiere de un parámetro que indica que tipo de elemento HTML es el que se va a crear.

```
var label = document.createElement("label");
```

No obstante, cuando se trata de crear elementos nuevos, este método resulta insuficiente, porque necesita de otros métodos y propiedades para poder finalizar el proceso de creación. Todos estos métodos y propiedades de los hablamos, se cuentan en detalle a continuación.

8.4.3 Propiedad id

Esta propiedad permite establecer el nombre de identificador único en el documento. Aunque este identificador debe ser único, no existen restricciones para poder utilizarlo de forma repetida, por lo que, aunque exista un elemento con ese mismo ID, el sistema no advertirá ningún tipo de error.

No se recomienda el uso de caracteres especiales que no sigan las especificaciones de CSS, como es el caso del símbolo de los dos puntos, porque puede dificultar su uso en operaciones posteriores. Pero, si se desea utilizar caracteres especiales como el símbolo de dos puntos, lo que se debe hacer es escaparlo a través de la barra invertida.

```
label.id = "nombre-label";
```

8.4.4 Propiedad innerHTML

Esta propiedad permite devolver o establecer el contenido HTML en un elemento del DOM. Si se intenta establecer un contenido que no está bien formado, es decir, una construcción HTML con errores, JavaScript advertirá un mensaje de error de sintaxis.

La propiedad innerHML está disponible para la mayoría de los elementos del DOM, exceptuando los elementos de formulario que utilizan la propiedad value en su lugar.

```
console.log(document.body.innerHTML);
```

Cuando se establece un contenido HTML a través innerHTML se pierden todos los manejadores eventos, aunque se hayan definido con anterioridad. La única forma de que no se pierdan los manejadores de eventos es si se establecen como atributo en el elemento, como por ejemplo **onclick**.

```
label.innerHTML = '<label for="name">Nombre de Usuario</label>';
```

8.4.5 Propiedad value

Esta propiedad permite devolver o establecer el valor de un elemento de formulario del DOM. Si se intenta establecer un contenido que no sea un tipo primitivo de datos, JavaScript advertirá un mensaje de error de sintaxis.

La propiedad value está disponible sólo para los elementos de formulario.

```
// Establecemos un valor en el elemento con ID "name"
document.querySelector("#nombre").value = "Pablo";

// Recuperamos su valor y lo mostramos por consola
console.log(document.querySelector("#nombre").value);
```

8.4.6 Método setAttribute

Este método permite establecer atributos a los elementos accesibles desde JavaScript. Sirve para cualquier tipo elemento que pueda ser referenciado desde la capa vista de la página o HTML.

El método **setAttribute** requiere de dos parámetros que indican el nombre del atributo que se va a establecer y el valor que se desea asignar a ese atributo.

```
label.setAttribute("class", "clase-de-prueba");
```

8.4.7 Propiedad classList

Esta propiedad devuelve un objeto DOMTokenList con las diferentes clases que tiene asignado el elemento.

La propiedad classList es, posiblemente, la única propiedad de HTML5 de este tipo que se utiliza con frecuencia en las aplicaciones. Quizás sea porque es compatible con todos los navegadores, incluyendo Internet Explorer 10 y superiores.

La propiedad classList, además de los métodos que obtiene por herencia del objeto DOMTokenList, nos proporciona otro adicional que permite realizar reemplazos de forma sencilla.

```
var body = document.body;

// Añadimos la clase "Chrome" al body
body.classList.add("Chrome");

// Añadimos la clase "Firefox" al body, porque no la tiene
body.classList.toggle("Firefox");

// Eliminamos la clase "Chrome" de la etiqueta body
body.classList.remove("Chrome");

// Reemplazamos la clase "Firefox" por "IE" en la etiqueta body
body.classList.replace("Firefox", "IE");

// Si contiene la clase IE, la eliminamos, si no, no hacemos nada
body.classList.contains("IE") ? body.classList.remove("IE") : ''
```

8.4.8 Propiedad previousElementSibling

Esta propiedad devuelve el elemento que se encuentra justo antes del referenciado, es decir, devuelve su hermano anterior.

Si se desea acceder al nodo anterior, en vez del elemento, la propiedad que se debe que utilizar es **previousSibling**, ya que previousElementSibling ignora los nodos de texto y comentario.

```
var elem = document.querySelector("#cabecera");
console.log(elem.previousElementSibling);
```

8.4.9 Propiedad nextElementSibling

Esta propiedad devuelve el elemento que se encuentra justo después del referenciado, es decir, devuelve su hermano posterior.

Si se desea acceder al siguiente nodo, en vez del elemento, la propiedad que se debe que utilizar es **nextSibling**, ya que nextElementSibling ignora los nodos de texto y comentario.

```
var elem = document.querySelector("#cabecera");
console.log(elem.nextElementSibling);
```

8.4.10 Propiedad parentElement

Esta propiedad devuelve el contenedor del elemento referenciado, es decir, devuelve su padre.

Si se desea acceder al nodo padre, en vez del elemento padre, la propiedad que se debe que utilizar es **parent**, ya que **parentElement** ignora los nodos de texto y comentario.

```
var elem = document.querySelector("#cabecera");
console.log(elem.parentElement);
```

8.4.11 Método appendChild

Este método añade un nodo o elemento proporcionado como parámetro a un contenedor o padre.

Si se intenta utilizar esta instrucción con código HTML en formato texto producirá un error de sintaxis.

```
// Creamos un nuevo elemento
var label = document.createElement("label");

// Le asignamos un ID
label.id = "nombre-label";

// Le añadimos el atributo for
body.setAttribute("for", "nombre");

// Le añadimos al documento
document.body.appendChild(label);
```

8.4.12 Método insertBefore

Este método permite insertar un nodo o elemento antes del elemento o nodo que se proporciona como referencia.

Si el contenedor donde se va a insertar el elemento está vacío, lo añade como primer hijo. Si el contenedor donde se va a insertar ya tiene el elemento referenciado, lo inserta justo antes y desplaza los demás una posición hacia abajo.

```
// Imaginemos un documento con la siguiente estructura:
► body
  └ span

var nuevo = document.createElement("label");
var span = document.querySelector("#span");
span.parentElement.insertBefore(nuevo, span);

// la estructura quedaría:
► body
  ├ label
  └ span
```

Si lo que se desea es insertarlo después, en vez de antes, se puede realizar pequeña una modificación en el método insertBefore que provocará que la inserción se realice por detrás. Esa modificación, básicamente es, llamar a la propiedad **nextSibling** o **nextElementSibling** desde el elemento referenciado.

```
var nuevo = document.createElement("label");
var span = document.querySelector("#span");
span.parentElement.insertBefore(nuevo, span.nextElementSibling);
```

Si ejecutamos este último bloque de instrucciones, la situación debería quedar de la siguiente manera:

```
// la estructura quedaría:
► body
  ├ span
  └ label
```

Otra necesidad habitual que se suele cubrir con este método es añadir un elemento dado como el primer hijo del elemento referenciado. Para realizar este efecto, lo que podemos hacer es:

```
var nuevo = document.createElement("label");
var span = document.querySelector("#span");
span.insertBefore(nuevo, span.childNodes[0]);
```

Esta modificación, lo que provocará es la siguiente situación.

```
// la estructura quedaría:
► body
  └ span
    └ label
```

8.5 ELIMINACIÓN DE NODOS Y ELEMENTOS

8.5.1 Método remove

Este método elimina el elemento o nodo referenciado del DOM.

```
label.remove();
```

Este método no es compatible con Internet Explore, pero si queremos que funcione se puede añadir al DOM un polyfill a través del objeto **HTMLElement**.

```
HTMLElement.prototype.remove=function(){
    try{ this.parentElement.removeChild(this); } catch(e){}
}
```

8.5.2 Método removeChild

El método **removeChild** elimina un elemento o nodo hijo proporcionado como parámetro. Funciona de forma muy similar al método **remove**, con la diferencia de que debe ser llamado desde el nodo o elemento padre.

```
// Añadimos un elemento label
var label = document.createElement("label");
document.body.appendChild(label);

// Ahora eliminamos el elemento label
var parentNode = document.body;
parentNode.removeChild(label);
```

Si utilizamos este método sin asignarlo a una variable, como es el caso, el elemento será eliminado del DOM y de la memoria.

```
// Ahora eliminamos el elemento label
var parentNode = document.body;
var oldElement = parentNode.removeChild(label);
```

Sin embargo, si hiciésemos este último caso, el elemento eliminado ya no formaría parte del DOM, pero permanecería en memoria y podría seguir siendo referenciado más tarde en cualquier parte de la aplicación.

Si en el proceso de eliminación se produjese algún error, la razón más frecuente es que no se ha encontrado o que no hay coincidencia de tipos.

8.6 DEFINICIÓN DE ESTILOS

8.6.1 La interfaz CSSStyleDeclaration

CSSStyleDeclaration es una interfaz de JavaScript con propiedades similares a un JSON y que representa un conjunto de pares de propiedades CSS con sus respectivos valores. Habitualmente se accede a ella a través de la propiedad style de un objeto HTML y las propiedades y métodos que proporciona pueden ser útiles cuando se desea crear estilos o reglas en las aplicaciones en tiempo de ejecución.

A partir de aquí, y para entender mejor sus propiedades y métodos supongamos que el elemento $0 que tiene declarado en su atributo STYLE lo siguiente:

```
$0.style = "width: 360px; height: 100vh;";
```

8.6.1.1 PROPIEDADES

Entre las propiedades más frecuentes podemos encontrar:

Propiedad	Descripción y ejemplo
length	Devuelve la longitud o número total de propiedades CSS del elemento referenciado. `document.body.style.length`

cssText	Devuelve o establece una declaración en formato String de estilos del elemento referenciado.
	
	`$0.style.cssText;`
	`// devuelve "width: 382px; height: 100vh;"`

parentRule	Si estamos trabajando con hojas de estilo, devuelve un objeto CSSRule que representa un conjunto de reglas CSS con su selector y declaraciones. `document.styleSheets[0].rules[0].style.parentRule`

8.6.1.2 MÉTODOS

Entre los métodos más frecuentes podemos encontrar:

Método	Descripción y ejemplo
item	Permite acceder a un elemento del NodeList a través de su índice. Si el índice no se corresponde con ningún elemento del conjunto, devolverá "undefined". `$0.style.item(0);  // devuelve "width"`

setProperty	Permite añadir una propiedad CSS al elemento referenciado. Si el segundo argumento es un literal no vacío, se añadirá la propiedad CSS a la declaración en línea. Por el contrario, si el segundo argumento es una cadena vacía, se provocará el efecto contrario y se eliminará de su declaración en línea.
	
	`$0.style.setProperty("height", "100%")`
	`// Cambia el valor de height de 100vh a 100%`

getProperty Value	Permite recuperar la propiedad CSS solicitada del elemento referenciado. Si el argumento enviado no existe en la declaración de estilos en línea del elemento referenciado, se devolverá una cadena vacía.
	
	`$0.style.getPropertyValue("height")`
	`// devuelve "100%"`

getProperty Priority	Permite recuperar la prioridad aplicada a la propiedad CSS solicitada del elemento referenciado. Si el argumento solicitado no tiene asignada la prioridad "important" o no existe en la declaración de estilos en línea del elemento referenciado, se devolverá una cadena vacía.
	
	`$0.style.getPropertyPriority("height")`
	`// devuelve "" porque no tiene !important asociado`

Remove Property	Permite eliminar una propiedad CSS del elemento referenciado y devolver su valor antes de provocar su eliminación. Si el argumento enviado no existe en la declaración de estilos en línea del elemento referenciado, se devolverá una cadena vacía. `$0.style.removeProperty("height")`

8.6.2 La interfaz CSSStyleSheet

Las hojas de estilo están compuestas por reglas CSS que puede definir el aspecto y cambiar el comportamiento de los elementos que se presentan en pantalla.

CSSStyleSheet es una interfaz de JavaScript con propiedades similares a un JSON y que representa un conjunto de esas reglas que están asociadas a los

elementos de la aplicación. Todas y cada una de ellas, pueden ser manipuladas de forma independiente a través del objeto **CSSRule**, la interfaz que representa a una regla CSS.

Para acceder a todas las hojas de estilo de un documento se puede hacer a través de la propiedad **styleSheets** del objeto **document**. Esta propiedad devolverá un array con todas las hojas de estilo definidas en la aplicación.

```
console.log(document.styleSheets);

// Devolverá algo parecido a:
► 0: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, ...}
► 1: CSSStyleSheet {ownerRule: null, type: "text/css", ...}
► 2: CSSStyleSheet {ownerRule: null, type: "text/css", ...}
```

Una vez que recuperemos uno de estos objetos **StyleSheetList**, que representa una de las hojas del documento actual, podremos acceder a sus propiedades y métodos para poder manipular las reglas que deseemos.

8.6.2.1 PROPIEDADES

Entre las propiedades más frecuentes podemos encontrar:

Propiedad	Descripción y ejemplo
cssRules	Devuelve todas las reglas CSS de una hoja de estilo en formato array de objetos. `document.styleSheets[0].cssRules`
disabled	Esta propiedad permite conocer si la hoja de estilo actual se está aplicando actualmente o no. `document.styleSheets[0].disabled`
href	Devuelve la dirección absoluta donde se encuentra localizada la hoja de estilos. Si el valor es null, indica que la hoja de estilos ha sido insertada a través de una etiqueta style de HTML. `document.styleSheets[0].href`
media	Especifica el medio dónde se aplica la hoja de estilos. Sus valores, habitualmente, son screen, print o all. `document.styleSheets[0].media`

ownerNode	Devuelve el nodo (elemento link) que asocia la hoja de estilos con el documento. Si el valor es null, indica que la hoja de estilos ha sido insertada a través de una etiqueta style de HTML. `document.styleSheets[0].ownerNode`
ownerRule	Indica si la hoja de estilo proviene de una llamada interna a través de la cláusula @import de CSS. `document.styleSheets[0].ownerRule`
parentStyle Sheet	Si procede, devuelve la hoja de estilos que incluye la actual hoja de estilos. `document.styleSheets[0].parentStyleSheet`
title	Devuelve el título de la hoja de estilos. Al ser un parámetro opcional, lo normal es que esta propiedad esté establecida a null. `document.styleSheets[0].title`
type	Especifica el lenguaje de programación utilizado por la hoja de estilos. Normalmente devolverá "text/css". `document.styleSheets[0].type`

8.6.2.2 MÉTODOS

Entre los métodos más frecuentes podemos encontrar:

Método	Descripción y ejemplo
insertRule	Añade una regla de la hoja de estilos seleccionada. let styleSheet = document.styleSheets[0]; styleSheet.insertRule("#w { color: white }", 0);
deleteRule	Elimina una regla de la hoja de estilos seleccionada. styleSheet.removeRule(0);

8.6.3 Propiedad style

Esta propiedad es una de esas propiedades que muchas veces no podemos evitar utilizar. Su utilización es equivalente a la definición a través del atributo style declarado desde código HTML.

En realidad, la propiedad style, es un objeto que contiene un montón de propiedades y métodos que tienen como fin, establecer y definir todo lo que se podría definir desde las hojas de estilos.

Cabe destacar que, en general, es mejor utilizar la propiedad style para aplicar estilos a los elementos que realizarlo a través del método **setAttribute** puesto que, este último, reemplaza y/o elimina todos los estilos anteriormente declarados.

Para establecer, modificar o eliminar estilos a un elemento debemos basarnos en la interfaz **CSSStyleDeclaration**. Por ejemplo, si se desea establecer la propiedad display a un elemento deberíamos hacer lo siguiente:

```
document.getElementById("id").style.display = "block";
```

Para recuperar el valor de una propiedad CSS aplicada a un elemento concreto deberíamos hacer:

```
console.log(document.getElementById("id").style.display);
```

Sin embargo, para eliminar el valor de una propiedad CSS aplicada a un elemento concreto, deberíamos establecerlo a una cadena vacía, es decir:

```
document.getElementById("id").style.display = "";
```

8.6.4 Método insertRule

Este método inserta una nueva regla CSS en una hoja de estilos. Para ello, requiere que se le proporcione tanto la declaración de la nueva regla como la posición dónde se insertará. Si el valor de posición es CERO, la regla se insertará al principio de la hoja de estilos, de lo contrario, se insertará en la posición que se le indique.

Para poder realizar la eliminación de la regla, previamente, se deberá recuperar la hoja de estilos a través de la propiedad **document.styleSheets**.

```
// Recuperamos la hoja de estilo
var s = document.styleSheets[0];

s.insertRule("label { color: red; } ", s.cssRules.length);
```

8.6.5 Método deleteRule

Este método elimina una regla CSS de una hoja de estilos. Para ello, se le debe proporcionar un índice o posición como parámetro.

Para poder realizar la eliminación de la regla, previamente, se deberá recuperar la hoja de estilos a través de la propiedad **document.styleSheets**.

```
// Recuperamos la hoja de estilo
var s = document.styleSheets[0];

s.deleteRule(0);
```

8.7 EJERCICIOS RESUELTOS

QuerySelector abreviado	Código QR
Componente que permita realizar consultas al DOM como querySelectorAll, pero mejorado al estilo de jQuery. Para más información se puede consultar el método constructor $ de jQuery. **https://codepen.io/pefc/pen/abaeLvw**	

Recuperar los ancestros de un elemento	Código QR
Añadir a la función $ una función que permita recuperar todos los ancestros que se correspondan con un selector CSS dado. Si no se proporciona nungún selector deberá mostrar todos sus ancestros. **https://codepen.io/pefc/pen/gOBYaaO**	

9

JAVASCRIPT ASÍNCRONO

La programación AJAX (Asynchronous JavaScript And XML) es un modelo de desarrollo que, hoy día, está más que presente en muchos aplicativos y sistemas web.

La razón de que tenga tanto éxito es porque permite actualizar código dinámicamente sin tener que recargar la página o enviar y recibir datos del servidor en segundo plano sin que el usuario pierda la sensación de control ni tenga frustraciones por la falta de compresión de lo que hace la página.

9.1 EL ESTÁNDAR CORS

El estándar CORS (Cross Origin Resource Sharing) es un mecanismo que permite configurar las capacidades y el modo de comunicación con el servidor cuando se realizan peticiones desde un cliente que está en diferente dominio. Esto es lo que se suele denominar como Intercambio de Peticiones de Orígenes Cruzados.

Este mecanismo es muy recurrido, sobre todo, en aplicaciones móviles por su fácil manejo.

Por temas de seguridad, los navegadores restringen las solicitudes de orígenes cruzados que se realizan desde JavaScript. Por esta razón, si se quiere hacer peticiones Ajax a sitios remotos, tanto el servidor como el cliente deberán tener configurada la CORS.

Sin embargo, aunque no lo parezca, este estándar está más presente de lo que los usuarios normales puedan pensar. Esto se debe a que se utiliza, además de en

peticiones Ajax, cuando se realizan peticiones de Web Fonts (como Google Fonts), o cuando se cargan hojas de estilo o scripts remotos.

9.1.1 Encabezados de solicitud HTTP

El estándar CORS dispone de una lista de cabeceras de solicitud que los clientes envían a los servidores y, de este modo, poder utilizar el mecanismo de intercambio de orígenes cruzados.

Encabezado	Descripción y ejemplo
AccessControlRequestMethod	Este encabezado indica el método que se va a realizar en la solicitud. Este encabezado siempre se incluye, aunque el método sea un método HTTP simple como GET, POST o HEAD. `Access-Control-Request-Methos: GET`
AccessControlRequestHeaders	Este encabezado indica qué cabeceras que se va a requerir la solicitud. Los valores de este encabezado deben ir separados por comas. `Access-Control-Request-Headers: X-Custom-Header`

9.1.2 Encabezados de respuesta HTTP

El estándar CORS dispone de una lista de cabeceras de respuesta que los servidores devuelven a los clientes y, de este modo, poder utilizar el mecanismo de intercambio de orígenes cruzados.

Encabezado	Descripción y ejemplo
AccessControlAllowOrigin	Este encabezado indica quién puede tener acceso. Lo más frecuente es que este valor sea *, lo que significa que se admitirá cualquier origen. `Access-Control-Allow-Origin: "*"`
AccessControlAllowCredentials	Es un valor booleano que indica si se deben incluir las cookies en los encabezados. Si esta cabecera está establecida a true, el valor de la propiedad withCredentials también debe ser true. `Access-Control-Allow-Credentials: true`

AccessControlExposeHeaders	Es un String que indica las cabeceras que están expuestas. Los valores de esta cabecera deben ir separados por comas. Por defecto, esta cabecera está configurada para acceder, únicamente, a encabezados de respuesta simple, es decir, los que se definen como Cache-control, Content-Language, Content-Type, Expires, LastModified y Pragma. `Access-Control-Expose-Headers: "X-UA-Compatible"`
AccessControlMax-Age	Es un valor entero que indica el tiempo, en segundos, que pueden estar almacenadas las respuestas en caché. `Access-Control-Max-Age: 900`
AccessControlAllowMethods	Es un String que indica los métodos que se pueden utilizar o están permitidos. Los valores de esta cabecera vendrán separados por comas. `Access-Control-Allow-Methods: "POST, GET, OPTIONS"`
AccessControlAllowHeaders	Es un String que indica los encabezados que se pueden utilizar o están permitidos. Los valores de esta cabecera deben ir separados por comas. `Access-Control-Allow-Headers: "*"`

9.2 CONEXIONES HTTP

Una solicitud de conexión HTTP es una comunicación entre cliente servidor que se realiza a través del protocolo HTTP. El cliente demanda una conexión enviándole un mensaje con la solicitud y, el servidor, le contesta con otro mensaje parecido que, lleva consigo, el estado de la conexión y el resultado de la misma.

Las conexiones HTTP pueden manejar varios métodos para demandar, al servidor, un tipo de acción que se desea realizar sobre un recurso concreto. Esto significa que, dependiendo de cuál sea, se podrán realizar unas determinadas acciones. A continuación, se muestra una lista con los tipos de conexión o métodos más utilizados.

Tipo/Método	Descripción
DELETE	Representa una eliminación de datos. Lo habitual es que, si la conexión tuvo éxito, devuelva un código de estado 200 o 204, en función de si retorna o no algún contenido.
GET	Representa una lectura, recuperación o descarga de datos, aunque se suele utilizar para envíos, con ciertas limitaciones. Lo habitual es que, si la conexión tuvo éxito, devuelva un código de estado 200.
HEAD	Representa una recuperación de datos cabecera HTTP, incluyendo su código de respuesta. Es decir, en la respuesta no se incluye el HTTP Response (cuerpo de la respuesta). Lo habitual es que, si la conexión tuvo éxito, devuelva un código de estado 200.
POST	Representa un envío de datos. Lo habitual es que, si la conexión tuvo éxito, devuelva un código de estado 201.
PUT	Representa una creación o actualización de datos. Lo habitual es que, si la conexión tuvo éxito, devuelva un código de estado 201 o 204, en función de si retorna o no algún contenido.
OPTIONS	Representa una solicitud de información sobre las opciones de comunicación. Lo habitual es que, si la conexión tuvo éxito, devuelva un código de estado 200.

En JavaScript, las conexiones HTTP se realizan a través de un objeto que, por cierto, lo diseñó Microsoft, fue adoptado por Mozilla y que, actualmente se ha convertido en un estándar de la W3C. Ese objeto es XMLHttpRequest.

9.2.1 Objeto XMLHttpRequest

El objeto **XMLHttpRequest** se supone que proporciona una forma sencilla de realizar las conexiones, aunque, en ocasiones, puede volverse una forma de trabajar un poco tediosa.

Aunque se supone que este objeto está pensado para realizar conexiones asíncronas, la realidad, es que permite realizar llamadas síncronas gracias a uno de sus parámetros de configuración.

A continuación, destacamos las propiedades y métodos más utilizados o frecuentes de este objeto.

9.2.1.1 PROPIEDADES

<table>
<tr><th>Propiedad</th><th>Descripción</th></tr>
<tr><td>readyState</td><td>Esta propiedad indica el estado de la petición AJAX.
<table>
<tr><th>Valor</th><th>Estado</th><th>Descripción</th></tr>
<tr><td>0</td><td>Uninitialized</td><td>La petición todavía no está inicializada. En otras palabras, no se ha llamado al método open.</td></tr>
<tr><td>1</td><td>Loading</td><td>La petición está inicializada pero todavía no se ha solicitado su envío, En otras palabras, no se ha llamado al método send.</td></tr>
<tr><td>2</td><td>Loaded</td><td>La petición se ha realizado y ya se tienen los encabezados y el estado de la conexión.</td></tr>
<tr><td>3</td><td>Interactive</td><td>La petición se encuentra en progreso. Se supone que la propiedad responseText ya contiene información de forma parcial, pero sigue recibiendo más datos.</td></tr>
<tr><td>4</td><td>Completed</td><td>La petición se ha completado. Esto no quiere decir que haya tenido éxito, sólo que ha terminado. La propiedad que indica si la operación ha tenido éxito o no es status.</td></tr>
</table></td></tr>
<tr><td>Response Text</td><td>Esta propiedad devuelve un String con el contenido de la respuesta que creó la petición. Si la respuesta de la petición no fue exitosa, el contenido devuelto será null.</td></tr>
<tr><td>responseXML</td><td>Esta propiedad devuelve el contenido de la respuesta que creó la petición en formato DOM Document. Si la respuesta de la petición no fue exitosa, el contenido devuelto será null. Para que esta propiedad sea efectiva, el contentType debe estar establecido a “text/xml”.</td></tr>
<tr><td>status</td><td>Esta propiedad devuelve el código de estado HTTP enviado por el servidor y representa cómo terminó la petición. Por ejemplo, si la petición fue exitosa, el valor devuelto será un 200, pero, si la petición tuvo un error interno, el valor devuelto, normalmente será un 500.</td></tr>
<tr><td>statusText</td><td>Esta propiedad devuelve mensaje de texto asociado al código de status que envió el servidor. Por ejemplo, si la petición fue exitosa, lo más normal sería que este valor fuese “200 OK”, pero, si la petición no fue realizada por falta de comunicación con el servidor, lo más normal sería que este valor fuese “404 Not Found”.</td></tr>
<tr><td>with Credentials</td><td>Esta propiedad contiene un valor booleano que indica cuando una petición debe llevar las credenciales del usuario en los encabezados. El método para enviar dichas credenciales puede ser a través de cookies, encabezados de autorización o certificados de cliente TSL. No obstante, esta propiedad no tiene ningún efecto cuando se realizan peticiones dentro del mismo dominio.</td></tr>
<tr><td>onready statechange</td><td>Esta propiedad establece la forma de controlar los diferentes estados por los que pasa la petición AJAX. Se realiza a través de una función que será llamada, de forma automática, cada vez que la propiedad status cambie.</td></tr>
</table>

9.2.1.2 MÉTODOS

Método open

Este método inicializa la petición. Para abrir la petición se necesitan proveer, al menos, dos parámetros, pero admite cinco, los cuales son **método**, que normalmente será GET o POST, pero admite cualquier método de petición HTTP, **URL**, que es la dirección donde realizar la petición, **modo**, que por defecto está a true y es un booleano que indica si la petición será realizada de forma asíncrona o no y **usuario** y **password**, que son opcionales, están vacíos y representan el nombre y contraseña del usuario.

```
http.open("GET", "http://ejemplo.org/doc");
```

Método setRequestHeader

Este método permite establecer valores de encabezado a través de pares nombre – valor. Aunque se pueden establecer muchísimas posibilidades de configuración, aquí sólo describiremos las más comunes:

Campo	Descripción
Accept	Es un String que indica los tipos de medios o contenido que son aceptables para la respuesta. `Accept: text/html`
Accept-Chartset	Es un String que indica los conjuntos de caracteres que son aceptables para la respuesta. `Accept-Charset: utf-8`
Accept-Encoding	Es un String que indica las posibilidades de codificación permitidas para la respuesta. `Accept-Encoding: gzip, deflate`
Autorization	Es un String que indica las credenciales de autorización. `Autorization: Basic QWx67bjp2YGW12zxYl==`
Cache-Control	Es un String que indica las directivas que deben cumplir los sistemas de almacenamiento en caché. `Control-Cache: no-cache`

Connection	Es un String que indica si la conexión debe permanecer abierta, una vez termine la solicitud. Los valores que suele tener son **close** (para indicar que la conexión se ha cerrado al terminar la solicitud) o **keep-alive** (para indicar que la conexión es persistente y, por tanto, no se cerrará). `Connection: keep-alive`
Content-Language	Es un String que indica el idioma utilizado en el cuerpo de la solicitud. `Content-Language: es, en`

Content-Type	Es un String que indica el tipo de contenido del cuerpo de la solicitud.
	Los más utilizados son:
	`Content-Type: application/json`
	`Content-Type: application/x-www-form-urlencoded`
	`Content-Type: text/html`
	`Content-Type: application/octet-stream`
	`Content-Type: application/pdf.`

Date	Es un String que indica la fecha y hora en que se realizó la solicitud. `Mon, 29 Jul 2019 09:02:45 GMT`
Expires	Es un String que indica la fecha y hora en que se considerará que, la respuesta guardada en caché será caducada. `Tue, 30 Jul 2019 09:02:45 GMT`
Last-Modified	Es un String que indica la fecha y hora de modificación para el contenido solicitado. `Mon, 29 Jul 2019 09:02:45 GMT`
Pragma	Es un String que indica uno o varios campos específicos de implementación que pueden tener diferentes efectos en cualquier parte de la cadena de solicitud-respuesta. `Pragma: no-cache`

```
http.setRequestHeader("Content-type", " text/html; charset=utf-8");
```

Método send

Este método es el que realiza petición. Si la petición se configuró como asíncrona la secuencia de código de JavaScript seguirá su curso normal. Si, por el contrario, se configuró como síncrona, el método se quedará a la espera de la

respuesta del servidor y, como consecuencia, la secuencia del código no continuará hasta que termine.

```
http.send();
```

9.2.2 Eventos

Evento	Descripción
onloadstart	Este evento se lanza cuando la petición empieza a transferir datos.
onload	Este evento se lanza cuando la petición se haya completado correctamente.
onprogress	Este evento se lanza periódicamente cuando la petición se encuentra en progreso, es decir, mientras se está ejecutando.
onloadend	Este evento se lanza cuando la solicitud se completa por cualquier motivo, sea satisfactoriamente o no.
ontimeout	Este evento se lanza cuando la petición supera el número máximo de milisegundos permitido para obtener una respuesta. Su valor predeterminado es 0, lo que significa que no tiene establecido un tiempo máximo para responder. Cabe destacar que este evento no es aconsejable utilizarlo cuando la petición está configurada en modo síncrono.
onerror	Este evento se lanza cuando la petición fracasó, sea por la razón que sea.
onabort	Este evento se lanza cuando la petición es abortada, normalmente, por el usuario.

9.2.3 Ejemplo sencillo de XMLHttpRequest

```
var http = new XMLHttpRequest();
http.open('GET', 'http://ejemplo.org/doc', true);
http.setRequestHeader("Content-type", " text/html; charset=utf-8");

http.onreadystatechange = function (aEvt) {
  if (http.readyState == 4) {
    if(http.status == 200)
    console.log(http.responseText);
    else
    console.log("Error al realizar la petición");
  }
};

http.send(null);
```

9.3 PROMESAS

Una promesa es una entidad que permite gestionar la finalización y/o el fallo de una operación asíncrona.

Se les llama promesas porque devuelven un objeto que garantiza que, la operación, terminará más tarde o más temprano. Además, representan valores que pueden existir en el momento actual, en el futuro o nunca.

Al igual que pasa con muchos métodos de JavaScript, como es el método ForEach, las promesas utilizan dos funciones de retorno o callbacks, sin embargo, su manipulación y sintaxis están algo más elaboradas. El objetivo de esas funciones de retorno, será emplear una, en caso de éxito y la otra, en caso de fracaso o fallo.

No obstante, aunque todo parezca jauja, uno de los grandes problemas que tienen las promesas es que no son compatibles con muchos navegadores, incluyendo Internet Explorer 11, ya que es una funcionalidad bastante nueva y sólo se contempla en la versión de ECMAScript 6.

9.3.1 Objeto Promise

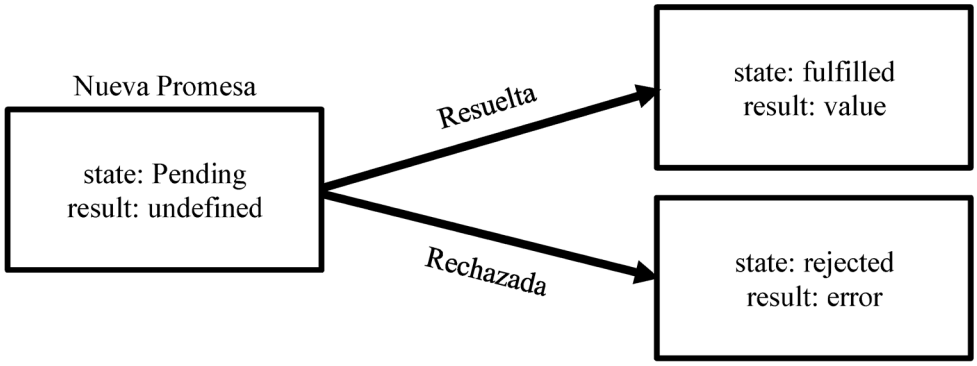
Como se puede deducir, el objeto Promise es quién permite manipular las promesas en JavaScript. Sin embargo, puede no estar disponible de forma nativa. Para detectar si el navegador dispone de promesas de forma nativa podemos utilizar el siguiente script:

```
if(!self.Promise) {
    console.log("El objeto Promise no está disponible");

    let f = document.createElement('script');
    f.src =
"//cdn.jsdelivr.net/npm/bluebird@3.7.2/js/browser/bluebird.min.js";

    f.onload = function() { continuar_la_carga(); };
}
```

Este objeto se alimenta de una función que se suele denominar como ejecutora y que recibe dos funciones como argumentos adjuntos. Estos argumentos, serán los que resuelvan o rechacen la promesa en función de si se provocó o no un error.

Cuando la promesa se ejecuta, la función ejecutora nos devuelve, de manera inmediata, un objeto que tiene dos propiedades, **state**, que es el estado

(que inicialmente está establecida a "pending" y, luego, cambia a "fulfilled", si se tuvo éxito o a "rejected", si se produjo un error) y **result**, que es el resultado (que inicialmente se establece a "undefined" y, más tarde, cambia a "value", si se tuvo éxito o a "error", si se produjo un error.



Dicho esto, una manera sencilla de utilizar las promesas de JavaScript podría ser:

```
var promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve((function(){
        alert("Promesa resuelta");
    })()), 1000);
});
```

Si ejecutamos el código anterior, comprobaremos que se ejecuta el método **resolve** sin problemas y nos saca una alerta diciendo "Promesa resuelta".

Sin embargo, si ejecutamos el siguiente código, lo que comprobaremos es que se ejecuta el método **reject** y nos saca por consola un error que dice "Uncaught (in promise) Error: Promesa rechazada".

```
var promise = new Promise(function(resolve, reject) {
    setTimeout(() => reject(new Error("Promesa Rechazada")), 1000);
});
```

Cabe destacar, que los métodos **resolve** y **reject** de dentro de la promesa, son funciones predefinidas por JavaScript, es decir, que no necesitan ser creadas. Pero, además, deben ser únicos en una misma secuencia, es decir, aunque haya varios resolve o reject declarados uno detrás de otro, sólo el primero será ejecutado, el resto, se ignorarán.

Otra cosa interesante de las promesas es que, si asignamos su definición a una variable, además de ser ejecutada podremos recuperar el objeto descriptivo **Promise** con el resultado de la ejecución, como se puede ver a continuación:

```
▾ Promise {<resolved>}
    ► proto__: Promise
        [[PromiseStatus]]: "resolved"
        [[PromiseValue]]: "Promesa resuelta"

▾ Promise {<rejected>: "Promesa Rechazada"}
    ► proto__: Promise
        [[PromiseStatus]]: "rejected"
        [[PromiseValue]]: "Promesa Rechazada"
```

Visto el funcionamiento de las promesas, el siguiente paso es entender los consumidores then, catch y finally.

9.3.1.1 CONSUMIDOR THEN

Es, quizás, el más importante, porque puede gestionar tanto el éxito como el rechazo.

El consumidor then recibe dos argumentos. El primero, es una función que se ejecuta cuando se resuelve la promesa con éxito. El segundo, es otra función que se ejecuta en cualquier otro caso y que es opcional. Veamos un ejemplo:

```
var nuevaPromesa = new Promise((resolve, reject) => {
    setTimeout(function(){
        try {
            allert("¡Hola mundo!");
            resolve("¡Todo bien!");
        } catch(err) {
            reject(err.message);
        }
    }, 1000);
});

nuevaPromesa.then(
    (result) => { console.log(result); },
    (error) => { console.log(error); },
);
```

Si ejecutamos esta nueva promesa, lo que veremos es que, pasado un segundo, se muestra el mensaje de error “allert is not defined” en la consola. Sin embargo, si eliminamos la doble “l” del bloque try y escribimos correctamente la

sentencia alert, veremos que, se saca la alerta y, en la consola se nos muestra un mensaje de "¡Todo bien!".

9.3.1.2 CONSUMIDOR CATCH

Como se ha visto, el consumidor then, permite gestionar tanto el éxito como el fracaso de las promesas, sin embargo, si sólo nos interesa gestionar el error o queremos gestionarlo por separado, podemos recurrir a este consumidor.

El consumidor catch recibe un único argumento, la función que se ejecuta cuando se produce un error. Veamos el ejemplo anterior, pero con los mensajes gestionados por separado:

```
var nuevaPromesa = new Promise((resolve, reject) => {
    setTimeout(function(){
        try {
            allert("¡Hola mundo!");
            resolve("¡Todo bien!");
        } catch(err) {
            reject(err.message);
        }
    }, 1000);
});

nuevaPromesa.then( (result) => { console.log(result); });
nuevaPromesa.catch((error) => { console.log(error); });
```

A igual que antes, si ejecutamos este código, el objeto Promise ejecutará el método **reject** y desviará la ejecución por el consumidor **nuevaPromesa.catch**.

Sin embargo, si eliminamos la doble "l" del bloque try y escribimos correctamente la sentencia alert, el objeto Promise saldrá por el método **resolve** y seguirá la ejecución por el consumidor **nuevaPromesa.then**.

9.3.1.3 CONSUMIDOR FINALLY

Al igual que tenemos un finally para el manejo de errores (try catch), también lo tenemos para las promesas.

El consumidor finally no tiene argumentos. Su cometido es, únicamente, hacer que el código que tenga asociado, se ejecutará sí o sí, es decir, se ejecute independientemente de que tenga éxito o error.

El uso de finally es una buena práctica para ejecutar código que se ejecuta, tanto en el consumidor then, como en el consumidor catch.

El orden de posicionamiento del consumidor finally es indiferente, es decir, que puede estar antes o después de cualquiera de los demás consumidores.

```
new Promise((resolve, reject) => {
  throw new Error("Se lanza excepción");
})
.finally(() => alert("Se ejecutó la promesa"))
.catch(err => alert(err));
```

Como creo que ya sabremos que hace este último ejemplo de promesa, sólo comentaré lo referente al finally. Cuando se lance esta promesa, se mostrarán dos alertas. La primera e inmediata, será el finally con el mensaje de "Se ejecutó la promesa" y, la segunda, con el mensaje de error.

9.3.2 La API fetch

La API fetch es un nuevo estándar que proporciona una alternativa a XMLHttpRequest, pero con un diseño más actual y basado en promesas.

Dado que esta API es bastante reciente, la compatibilidad entre navegadores es un poco escasa, por ello, si se desea utilizar y no se dispone de ella, se puede recurrir al polyfill de *https://github.com/github/fetch*.

Para detectar si el navegador dispone de la API fetch de forma nativa, podemos utilizar el siguiente script:

```
if(!self.fetch) {
    console.log("La API fetch no está disponible");

    var f = document.createElement('script');
    f.src = "//raw.githubusercontent.com/github/fetch/master/fetch.js";
    f.onload = function() {
        continuar_la_carga();
    };

    document.head.appendChild(f);
}
```

La sentencia fetch puede recibir dos parámetros que pueden ser definidos de forma independiente o a través del objeto Request. Tanto en uno, como en otro caso, el primero de estos parámetros, es un String que representa a la URL de destino.

El segundo, es un JSON con las opciones de configuración y que nos permite personalizar el método, modo, cabeceras, credenciales, entre otras opciones y, es opcional.

```
fetch('http://www.islavisual.com/rss')
.then(function(response) {
    return response.text();
})
.then(function(myRSS) {
    console.log(myRSS);
});
```

Si ejecutamos el código anterior, veremos que se nos muestra el contenido de la url solicitada, que en este caso es de tipo RSS. También se observa que, al igual que las promesas, se gestiona mediante consumidores then. El primero, se ejecuta para recuperar el texto, el segundo, se utiliza para mostrarlo por consola.

No obstante, como decíamos, puede recibir un JSON al que se le deberá indicar qué valor de propiedad pertenece a qué nombre o clave.

Las principales propiedades de este JSON son:

9.3.2.1 PROPIEDAD BODY

Indica el tipo de contenido que se va a recuperar. Sus posibles valores son:

Valor	Descripción
String	Indica que el contenido será de texto plano. Va asociado con una cabecera tipo text/plain;charset=UTF-8.
URLSearchParams	Indica que el contenido está en formato application/x-wwwform-urlencoded;charset=UTF-8.
FormData	Indica que el contenido es un formulario codificado en formato multipart/form-data.
Blob	Indica que el contenido es un fichero de datos planos inmutables.
ArrayBuffer	Indica que el contenido es búfer de datos binarios sin longitud fija.
TypedArray	Indica que el contenido es búfer de datos binarios de longitud fija.

Cuando se reciba la respuesta, se podrán utilizar los siguientes métodos:

Método	Descripción
text	Recupera el texto de respuesta como un String.
json	Recupera el texto de respuesta y lo devuelve parseado de forma automática. Esto es equivalente a realizar un JSON.parse(responseText).
blob	Recupera el contenido como un fichero de datos inmutables.
arrayBuffer	Recupera el contenido como un búfer de datos binarios sin longitud fija.
formData	Recupera el contenido como si fuese un objeto FormData o formulario.

Como dato adicional, diremos que existen otros métodos de respuesta como son clone(), Response.error() o Response.redirect() pero son bastante menos frecuentes.

9.3.2.2 PROPIEDAD CACHE

Define el comportamiento de la solicitud con la caché. Sus posibles valores que puede tomar son:

Valor	Descripción
default	Indica que la solicitud de recuperará de la caché de un modo normal o habitual.
no-store	Indica que la solicitud ignore la caché, es como si no existiese, por lo que nunca se guardará en la caché.
reload	Indica que la solicitud no debe consultarse desde la caché, pero sí será guardada o actualizada.
no-cache	Indica que la solicitud se debe traer desde el servidor remoto, aunque esté ya en caché. Además, la encuentre o no, sea nueva o no, siempre provocará la actualización de la caché.
force-cache	Indica que siempre se utilice la versión que se encuentra en caché, aunque esté obsoleta.

9.3.2.3 PROPIEDAD CREDENTIALS

Define cómo y desde dónde se deben recuperar las credenciales de la solicitud.

Sus posibles valores son:

Valor	Descripción
include	Indica que las credenciales van incluidas. Puede ser cookies.
same-origin	Indica que las credenciales sólo deben ser enviadas cuando la solicitud es del mismo origen.
omit	Indica que las credenciales no deben ser incluidas en la solicitud nunca.

9.3.2.4 PROPIEDAD HEADERS

Definen las cabeceras de la solicitud y respuesta. Tiene las mismas opciones que las utilizadas por el objeto XMLHttpRequest.

Un posible ejemplo muy utilizado:

```
headers:{ 'Content-Type': 'application/json' }
```

9.3.2.5 PROPIEDAD MODE

Define el modo de recuperación. Esto permite que sólo se resuelvan ciertas solicitudes. Sus posibles valores son:

Valor	Descripción
same-origin	Únicamente permitirá solicitudes que provengan del mismo origen. En otras palabras, todas las solicitudes que se realicen desde fuera serán rechazadas.
cors	Permitirá solicitudes que provengan del mismo origen y de otros orígenes, siempre y cuando devuelvan los encabezados CORS correspondientes.
cors-withforced-preflight	Obligará a realizar una verificación previa antes de realizar la solicitud deseada.
no-cors	Permite realizar solicitudes a otros orígenes que no disponen de encabezados CORS, pero no podremos recuperar los datos devueltos ni ver el estado de la solicitud, lo que significa que no podremos verificar si la solicitud fue exitosa o no.

9.3.2.6 PROPIEDAD METHOD

Define el método para realizar la solicitud, normalmente, GET o POST.

9.3.2.7 EJEMPLO COMPLETO DE SOLICITUD EN MODO TEXTO PLANO

```
var request = new Request("https://www.islavisual.com/rss", {
    method: 'POST',
    mode: 'cors',
    credentials: 'omit',
    cache: 'no-cache',
    referrerPolicy: 'no-referrer'
});

fetch(request)
.then(function(response) { return response.text(); })
.then(function(data) { console.log(data); })
.catch(function(err) { console.error(err); });
```

El código anterior, solicita a una web su RSS y, cuando recibe la respuesta, lo trata como texto y lo muestra por la consola. Si hubiese algún problema, se desviaría por el catch, mostrando el consecuente mensaje de error.

9.3.2.8 EJEMPLO COMPLETO DE SOLICITUD CON JSON

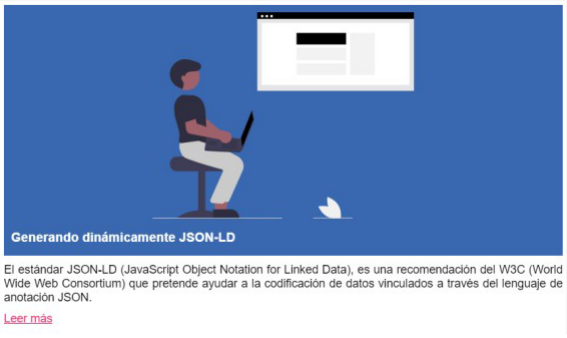
```
var request = new Request("http://ejemplo.com/municipios.json", {
    method: 'GET',
    mode: 'cors',
    credentials: 'include',
    cache: 'default',
    referrerPolicy: 'no-referrer'
});

fetch(request)
.then(function(response) {
    console.log('response.body =', response.body);
    console.log('response.bodyUsed =', response.bodyUsed);
    console.log('response.headers =', response.headers);
    console.log('response.ok =', response.ok);
    console.log('response.redirect =', response.redirect);
    console.log('response.status =', response.status);
    console.log('response.statusText =', response.statusText);
    console.log('response.type =', response.type); c
    onsole.log('response.url =', response.url);
    return response.json();
});
.then(function(json) { console.log('Municipio', json[0].name); })
.catch(function(err) { console.error(err); });
```

Este último código, realiza una solicitud a un servidor para traerse un JSON con todos los municipios de España. Después, una vez que lo ha recibido, muestra los datos de la respuesta y, finalmente, muestra el registro 4362 del array de JSONs. Si se produjese algún problema, se desviaría por el catch, mostrando el consecuente mensaje de error.

El resultado podría ser similar a lo siguiente:

```
response.body = ReadableStream {locked: false}
response.bodyUsed = false
response.headers = Headers {}
response.ok = true
response.redirect = undefined
response.status = 200
response.statusText = OK
response.type = cors
response.url = http://datosejemplo.com/municipios.json
Municipio "Amurrio"
```

9.4 EJERCICIOS RESUELTOS

RSS Feed	Código QR
Extraer las últimas noticias del feed RSS de la URL *https://www.islavisual.com/rss* mediante la API Fetch.  **https://codepen.io/pefc/full/JjaLjRE**	

10

VALIDACIÓN DE DATOS EN PÁGINAS WEB

Dicen que una de las razones por las que se diseñó el lenguaje de JavaScript fue la necesidad de poder validar la introducción de datos de usuario. Independientemente de que esto sea o no una leyenda, los formularios son una de esas cosas que más se utilizan en el desarrollo de aplicaciones web. Ya sea por estructurar, recopilar y enviar información o, simplemente, por ayudar a la introducción de datos, la realidad es que, el uso de elementos de formulario en las páginas web, es un recurso absolutamente necesario.

10.1 PROPIEDADES DE LOS FORMULARIOS

Cada vez que se produce la carga o actualización de una página, JavaScript rellena la propiedad **forms** del objeto **document** con una referencia a todos los formularios que encuentra de forma automática. Esta propiedad, básicamente, es un array de objetos que puede ser accedido a través de su índice.

Por ejemplo, si quisiésemos ver todos los formularios de la página podríamos iterar esta propiedad de la siguiente forma:

```
for(var i = 0; i < document.forms.length; i++){
    console.log(document.forms[i]);
}
```

A su vez, cada vez que el sistema añade un formulario, este es analizado y se le añaden todos los elementos que lo componen. Todos estos elementos de formulario están disponibles a través de la propiedad **elements**, que también puede ser manipulada como si fuese un array.

```
document.forms[0].elements[0];
```

El acceso a los elementos de formulario a través de estos arrays es un método francamente rápido, sin embargo, en entornos dinámicos no suele ser posible utilizarlo porque la posición de los elementos o, incluso, de los formularios puede cambiar en función de cómo se creen los objetos.

Cuando los entornos son dinámicos y no podemos predecir la posición donde se encontrarán los formularios, lo frecuente es acceder a ellos a través de su propiedad **name** o propiedad **id**.

```
var formName = document.forms.nombreFormulario;
var formID = document.forms.idFormulario;
```

Sin embargo, un formulario no solo dispone de los atributos de nombre y de identificación. También tiene atributos para definir la codificación, el método de envío o la dirección de destino.

A continuación, se muestran las principales propiedades a gestionar:

Propiedad	Descripción
acceptCharset	Establece o devuelve el conjunto de caracteres que serán utilizados para la recepción o envío del formulario. Si el atributo no está establecido o está vacío, el conjunto de caracteres será el que defina la codificación de la página en la que se encuentre.
action	Indica dónde se van a enviar los datos del formulario cuando se envíe el formulario. Si el atributo no está establecido o está vacío, la acción del formulario será llamarse a sí mismo.
autocomplete	Indica al navegador si el formulario puede o no recuperar los valores anteriormente introducidos por el usuario y, así, ayudarle en la introducción de los mismos ahorrando pulsos de teclado y tiempo. Por defecto, está activado y establecido a **"on"**.
enctype	Indica cómo se deben codificar los datos antes de ser enviados al servidor. Por defecto, su valor es **application/x-www-formurlencoded**, lo que significa que todos los datos del formulario se codifican como si fueran URLs, sin embargo, admite los valores de **multipart/form-data** (que envía los datos sin codificar) y **plain/text** (que sólo convierte los espacios a +).
length	Devuelve el número de elementos que tiene el formulario.
method	Establece o devuelve el método por el que se realizará el envío del formulario. Sus posibles valores son **GET** y **POST**.
name	Indica el nombre del formulario.
noValidate	Indica si el formulario debe validarse antes de ser enviado. Sus posibles valores son **true** o **false** y, por defecto, está establecido a **true**.
target	Indica dónde debe mostrarse la respuesta tras el envío del formulario al servidor. Habitualmente. sus posibles valores son **_blank** (para indicar que se abra en una nueva ventana) y **_self** (para indicar que se abra en la misma ventana). Por defecto, su valor es **_self**.

Como se puede deducir, los formularios y sus elementos también son accesibles a través de las funciones que proporciona el DOM. Puede que no sean tan eficaces, pero sí que lo hacen, en ocasiones, muy cómodo.

```
var formName = document.querySelector("[name='nombreFormulario']");
var formID = document.getElementById("idFormulario");
```

10.2 PROPIEDADES DE LOS ELEMENTOS DE FORMULARIO

Sea cual sea el tipo de elemento de formulario y su método para acceder a él, cada uno de ellos tiene unas propiedades o atributos que le define y configura. Dado que el número de propiedades o atributos que pueden tener es muy grande, a continuación, se muestran las más frecuentes.

Propiedad	Descripción
autofocus	Indica que el elemento debe tomar el foco cuando se carga o reinicia la página.
disabled	Indica que el elemento se encuentra desactivado. Esta propiedad es válida para todos los elementos de formulario.
form	Indica que el elemento pertenece al formulario que tiene como nombre el valor indicado por este parámetro. Esto permite que el elemento de formulario pueda encontrarse fuera del ámbito de declaración del mismo. Esta propiedad es válida para todos los elementos de formulario.
id	Indica el ID, se supone que único, del elemento. Esta propiedad es válida para para todos los elementos de formulario y HTML.
list	Indica que el elemento está vinculado a una lista predefinida suministrada por el elemento datalist. Esta propiedad es válida sólo para input.
max	Indica el valor máximo que puede tomar el elemento. Esta propiedad es válida para inputs de tipo range, number, date, datetime-local, month, time y week.
min	Indica el valor mínimo que puede tomar el elemento. Esta propiedad es válida para inputs de tipo range, number, date, datetime-local, month, time y week.
maxlength	Indica que el elemento tiene establecida una longitud máxima de caracteres. Esta propiedad es válida para input y textarea.
minLength	Indica que el elemento tiene establecida una longitud máxima de caracteres. Esta propiedad es válida para input y textarea.
multiple	Indica que el elemento puede admitir varios valores en su selección. Esta propiedad es válida para los inputs de tipo email y file y, para el elemento select.
name	Indica el nombre, se supone que único, del elemento. Esta propiedad es válida para para todos los elementos de formulario.

pattern	Indica una expresión regular que será utilizada como validación de entrada. Esta propiedad es válida para los inputs de tipo text, search, url, tel, email, y password.
placeholder	Indica el texto de ayuda o una sugerencia de valor que se muestra cuando el elemento no contiene ningún valor. Esta propiedad es válida para los inputs de tipo text, search, url, tel, email, y password.
readonly	Indica que el elemento se encuentra en modo de sólo lectura. Esta propiedad es válida para todos los elementos de formulario.
required	Indica que el elemento es un parámetro requerido. Esta propiedad es válida para todos los elementos de formulario.
selectionStart	Indica la posición de inicio del texto seleccionado. Esta propiedad es válida para todos los elementos de formulario que permitan la edición de texto.
selectionEnd	Indica la posición de finalización del texto seleccionado. Esta propiedad es válida para todos los elementos de formulario que permitan la edición de texto.
size	Indica el tamaño en caracteres del elemento. Esta propiedad es válida para todos los tipos de input.
step	Indica el intervalo de incremento o decremento del elemento cuando se pulsan sus indicadores o se pulsan los cursores de arriba y abajo. Esta propiedad es válida para los inputs de tipo number, range, date, datetime-local, month, time y week.
type	Indica el tipo del elemento (Consultar el apéndice II al final del libro). Esta propiedad es válida para todos los tipos de input.
value	Indica el valor inicial del elemento. Esta propiedad es válida para los todos los elementos de formulario.

10.3 CREACIÓN Y ENVÍO DE FORMULARIOS

Primero definimos el elemento que actuará como formulario y sus propiedades.

```
// Creamos el elemento de formulario
var f = document.createElement("form");

// Establecemos el método de envío y su URL destino
f.setAttribute('method',"post");
f.setAttribute('action',"cambiarcontraseña.html");
```

Ahora, procedemos a crear sus elementos. Primero definimos los campos de entrada.

```
// Añadimos un elemento de formulario
var i = document.createElement("input");
i.setAttribute('type',"email");
i.setAttribute('name',"email");
```

Seguidamente, definimos el botón que actuará como control de envío.

```
// Añadimos el botón para envío
var b = document.createElement("button");
b.setAttribute('type',"submit");
b.setAttribute('value',"Enviar");
```

Finalmente, añadimos los elementos al formulario de forma ordenada y, si lo deseamos, lo añadimos al body.

```
// Añadimos el input y el button al formulario
f.appendChild(i);
f.appendChild(b);

// Finalmente, lo añadimos al body
document.body.appendChild(f);
```

Efectivamente, lo más frecuente es añadir una declaración de formulario en JavaScript al elemento body, pero, no lo hiciésemos y le añadiésemos un valor a través de su propiedad value, podríamos llamar a su método submit para realizar un envío automático directamente desde memoria.

```
// Enviamos el formulario ahora
f.submit();
```

10.4 VALIDACIÓN DE FORMULARIOS

La validación de formularios, hasta no hace tanto, no era casi personalizable ni eficiente. Ahora, sin embargo, gracias a la amplia gama de propiedades que poseen los elementos de formulario, junto con los métodos de notificación y manipulación que nos provee HTML5, podemos realizar validaciones de forma bastante rápida y sencilla.

10.4.1 La interfaz ValidityState

La interfaz **ValidityState** es un objeto que representa todos los posibles estados por los que puede pasar un elemento de formulario. Además, suele indicar la razón o el motivo por el que se encuentra en ese estado. Entre sus propiedades más frecuentes podemos encontrar:

Propiedad	Descripción
badInput	Esta propiedad devuelve true si ha habido algún problema con la conversión del dato introducido.
customError	Esta propiedad devuelve true si el elemento tiene asignado un error personalizado establecido a través del método **setCustomValidity**.
patternMismatch	Esta propiedad devuelve true si el elemento no cumple el patrón definido. Si su valor es true, la pseudo-clase :invalid de CSS también se activará.
rangeOverflow	Esta propiedad devuelve true si el elemento contiene un valor mayor al provisto por la propiedad **max**. Si su valor es true, las pseudo-clases :invalid y :out-of-range de CSS también se activarán.
rangeUnderflow	Esta propiedad devuelve true si el elemento contiene un valor menor al provisto por la propiedad **min**. Si su valor es true, las pseudo-clases :invalid y :out-of-range de CSS también se activarán.
stepMismatch	Esta propiedad devuelve true si el elemento contiene un valor que no concuerde con paso provisto por la propiedad **step**. Si su valor es true, las pseudo-clases :invalid y :out-ofrange de CSS también se activarán.
tooLong	Esta propiedad devuelve true si el elemento tiene una longitud mayor a la provista por el atributo **maxlength**. Si su valor es true, las pseudo-clases :invalid y :out-of-range de CSS también se activarán.
tooShort	Esta propiedad devuelve true si el elemento tiene una longitud menor a la provista por el atributo **minlength**. Si su valor es true, las pseudo-clases :invalid y :out-of-range de CSS también se activarán.
typeMismatch	Esta propiedad devuelve true si el elemento contiene una sintaxis incorrecta y sólo es válido para los tipos de input **email** y **url**. Si su valor es true, la pseudo-clase :invalid de CSS también se activará.
valid	Esta propiedad devuelve true si el elemento cumple todas las restricciones requeridas. Si su valor es true, la pseudo-clase :valid de CSS también se activará.
valueMissing	Esta propiedad devuelve true si el elemento es requerido y se encuentra vacío. Si su valor es true, la pseudo-clase :invalid de CSS también se activará.

10.4.2 Propiedades y métodos

A continuación, se muestran los principales métodos y propiedades que pueden ser utilizados en la validación de formularios.

Propiedad / Método	Descripción
validity	Esta propiedad es un objeto que devuelve un conjunto de datos de tipo **ValidityState** y permite conocer el resultado de todos los posibles problemas que se han producido tras realizar una comprobación de validación.
setCustomValidity	El método **setCustomValidity** permite definir mensajes de error personalizados en los elementos de formulario. El mensaje, proporcionado como parámetro, es guardado en la propiedad **validationMessage**.
validationMessage	Esta propiedad contiene el mensaje generado tras el proceso de validación. Si el elemento no ha sido validado aún o ha pasado con éxito todo el proceso de validación, el contenido de esta propiedad estará establecida a cadena vacía. Si, por el contrario, existe algún error o problema con la validación, esta propiedad mostrará el mensaje de error o información sobre el problema.
checkValidity	El método **checkValidity** comprueba si se cumplen las restricciones que tiene definido el elemento de formulario. Si todo es correcto, es decir, que pasa la validación, devolverá true, en cualquier otro caso, devolverá false.

10.4.3 Eventos

A continuación, se muestran los principales eventos que pueden ser utilizados en la validación de formularios.

Evento	Descripción
invalid	Cada vez que se solicita la acción de enviar un formulario, se realiza una acción de validación previamente. Si el proceso de validación no tuvo éxito, el navegador lanza una especie de excepción que marca al elemento como inválido y le asigna la pseudo-clase de CSS :invalid. Pues bien, si además de controlar la validación interna queremos o necesitamos gestionarla de forma externa, para esto, tenemos el evento **invalid**. El evento **invalid** es lanzado cuando el elemento realiza el proceso de validación y no cumple alguna de sus restricciones. Una vez, dentro de este evento podemos, por ejemplo, mostrar los mensajes personalizados que hemos creado para nuestra aplicación en el lugar de los predefinidos.
onsubmit	Este evento ocurre cada vez que se envía un formulario y permite realizar una validación justo antes de ser enviado. Si el valor devuelto por la función que realiza la validación es false, el formulario no se enviará, provocando la sensación de que no hace nada. Por ello es recomendable que, cada vez que se use esta técnica se muestre un mensaje de aviso o alerta para que el usuario sepa por qué no se ha realizado el envío del formulario.

10.4.4 Ejemplo de validación

Imaginemos que deseamos comprobar que el valor introducido en un input denominado status concuerde con uno de los tres posibles valores "Asignado", "En Progreso" o "Finalizado".

Para ello, primero necesitaremos declarar un código HTML que contenga un label con un campo de entrada y dos elementos externos para poder establecer los posibles mensajes de error.

Cuando se introduzca un estado inválido se mostrará, en el elemento validateMsg, el mensaje de invalidez y, en el elemento validity, el listado de las restricciones incumplidas.

Cuando sea válido, se eliminarán el mensaje de error y su estado de invalidez.

```
<h1>Prueba de validación</h2>
<form name="frm" enctype="multipart/form-data" method="get">
    <label>
        Estado:
        <input id="status" type="text" oninput="check(this)" />
    </label>

    <h2>Mensaje tras validación</h2>
    <div id="validateMsg"></div>

    <h2>Listado de errores de validity</h2>
    <div id="validity"></div>
</form>
```

Una vez insertado el HTML, necesitamos declarar la funcionalidad de JavaScript que realizará la validación:

```
function check(input) {
    var val = input.value;
    var vm = document.getElementById("validateMsg");
    var va = document.getElementById("validity");

    if (input.value != "Asignado" &&
        input.value != "En Progreso" &&
        input.value != "Finalizado") {
        var msg = '"' + val + '" no es un estado.';
       input.setCustomValidity(msg);
       vm.innerHTML = input.validationMessage;
       va.innerHTML = '';
       for(var key in input.validity){
           var status = input.validity[key];
           if(status){
```

```
                va.innerHTML += key + ": " + status.toString();
                va.innerHTML += "<br/>";
                }
            }

        } else {
            // Todo correcto.
            // Eliminamos el mensaje y el listado.
            input.setCustomValidity('');
            vm.innerHTML = "";
            va.innerHTML = "";
        }
}

var s = document.getElementById("status");
s.addEventListener("invalid", check, true);
```

10.5 EJERCICIOS RESUELTOS

Validación de formulario	Código QR
Crear todo lo necesario para realizar una validación de un formulario como el siguiente  **https://codepen.io/pefc/pen/VwGXwXX**	

11

EFECTOS ESPECIALES EN PÁGINAS WEB

11.1 INTRODUCCIÓN A LOS COMPONENTES

Un componente en JavaScript no deja de ser otro objeto. Se podría considerar que un componente en JavaScript es un conjunto de instrucciones que están pensadas para proveer una funcionalidad concreta. Normalmente, este componente proporciona una forma sencilla que utilizar ciertas características que, de otro modo, serían un trabajo laborioso o tedioso.

Una de las cualidades importantes que debe tener un componente es que sea fácil de utilizar y de integrar. Además, debe estar programado para que sea reutilizable porque, no olvidemos una de las premisas más importantes de la programación y de la usabilidad, "Para que algo sea reutilizable, primero debe ser utilizable".

Desde un punto de vista más funcional, los componentes se componen de la definición del constructor, unos parámetros de entrada y un conjunto de métodos que proporcionan la funcionalidad deseada.

11.2 DEFINICIÓN POR DECLARACIÓN

Dado que JavaScript es un lenguaje que no está muy tipificado, al final, los componentes se pueden hacer de mil formas diferentes.

Mientras que unos hacen los componentes a través de un JavaScript básico y compatible con todos los navegadores, otros, los hacen con definición de clases, JSON y/o sin retrocompatibilidad. Todos ellos pueden ser una buena opción, siempre y cuando el rendimiento de la página no se vea muy afectado y sea reutilizable.

Por lo tanto, crear un componente depende un poco de uno mismo. Seguir una estructura lógica, facilitar la entrada y salida de datos, una interfaz gráfica personalizable y una compatibilidad progresiva evolutiva.

11.2.1 Formas básicas de crear componentes

Existen varias formas de crear un componente, sin embargo, las más utilizadas son a través de un JSON o a través de la definición de objetos.

11.2.1.1 CREACIÓN A TRAVÉS DE JSON

Una forma frecuente de crear componentes es mediante la definición de un objeto tipo JSON que está provisto de una serie de atributos que pueden actuar como propiedad o como método.

```
var Plugin = {
    name: "Componente JSON",
    version: 1.0,
    init: function(texto){
        var elemento = document.createElement("label");
        elemento.innerHTML = texto;

        document.body.appendChild(elemento);

        console.log("Componente insertado en el body");
    },
};
```

Este tipo de definición se asemeja más con el concepto de clases con métodos estáticos. Es decir, para llamar a uno de sus métodos necesitaremos hacerlo como si fuese una propiedad. En el ejemplo, el método **init** debe llamarse de forma explícita para poder ejecutarse y, en el proceso, crea un elemento label, le asigna un texto que se le pasa por parámetro y lo añade al body.

```
// Para llamarlo o ejecutarlo
Plugin.init("Escriba nombre del plugin:");
```

11.2.1.2 CREACIÓN A TRAVÉS DE OBJETOS

La otra forma frecuente de crear componentes es mediante la definición de un objeto tipo función. Este objeto tiene una declaración inicial y, más tarde si procede, se le añaden una serie de atributos que pueden actuar como propiedad o método.

```
var Plugin = function(texto){
    var elemento = document.createElement("label");
    elemento.innerHTML = texto;
    document.body.appendChild(elemento);
    console.log("Componente insertado en el body");
};

Plugin.name = "Componente Objeto";
Plugin.version = 1.0;
```

A diferencia de una declaración en formato JSON, este tipo de definición no necesita llamar a ningún método para poder ejecutarse porque, su nombre es la función de inicialización, que es quién hace el proceso de crear un elemento label, asignarle un texto que se le pasa por parámetro y añadirlo al body.

```
// Para llamarlo o ejecutarlo
Plugin("Escriba nombre del plugin:");
```

Cabe destacar que, los componentes declarados en forma de objeto son normalmente instanciados puesto que, por lo general, devuelven algún tipo de información o se utilizan para referenciar a los elementos afectados.

11.2.2 El paso de parámetros

El paso de parámetros puede ser muy diferente dependiendo de cómo diseñemos el componente.

Lo habitual, cuando se diseña bajo el formato de JSON es que los parámetros vayan todos en un argumento, también de tipo JSON.

```
/* ************************************ */
/* Buscar y devolver un valor en un JSON.   */
/* Parámetros:                              */
/* arr: JSON con los datos.                 */
/* field: Clave o campo dónde buscar        */
/* val: Valor a buscar                      */
/* ************************************ */
```

```
function getItemFromJSON(arr, field, val){
    for (var i=0 ; i < arr.length ; i++){
        if (arr[i][field] == val) {
            return arr[i];
        }
    }
}

// La forma de usarla:
getItemFromJSON ({...}, "nombre", "Pablo");
```

Lo habitual, cuando se diseña en forma de objetos es que, los parámetros, o vayan todos en argumento separados o vayan todos en un único argumento de tipo JSON.

```
/* ************************************ */
/* Buscar y devolver un valor en un JSON.   */
/* Parámetros:                          */
/* arr: JSON con los datos.             */
/* field: Clave o campo dónde buscar    */
/* val: Valor a buscar                  */
/* ************************************ */
function getItemFromJSON(arr, field, val){
    for (var i=0 ; i < arr.length ; i++){
        if (arr[i][field] == val) {
            return arr[i];
        }
    }
}

// La forma de usarla:
getItemFromJSON ({...}, "nombre", "Pablo");
```

11.2.3 La interfaz de usuario

La interfaz de usuario es el medio a través del cual un usuario se comunica con una aplicación, web o sistema. En este sentido, creo que todos estaremos de acuerdo en que, dicha interfaz, debe ser lo más fácil e intuitiva posible.

Dicho esto, si observamos un poco veremos que hay un inmenso número de componentes que requieren de una interfaz de usuario y que, normalmente, suele estar construida a partir de elementos HTML y estilos CSS.

El HTML nos aporta la semántica y la estructuración de la información que maneja el componente mientras que, el CSS nos aporta el aspecto y forma de presentar dicha información.

11.2.3.1 INTERFAZ DE USUARIO ENCAPSULADA

Que un componente incruste o añada elementos y estilos es más que habitual de lo que uno, a primera vista, pueda pensar. Por ejemplo, pensemos en un componente web de tipo vista de árbol (treeview). Este componente es frecuente que esté vinculado con una etiqueta de HTML de tipo lista (como un UL) y que, más tarde durante el proceso de carga, el componente web añada elementos de tipo nodo (como un LI) que, a su vez, pueden añadir otros elementos de tipo lista y así sucesivamente.

```
<ul id="treeview" class="treeview">
    <li><i class="icon"></i><i class="toggler">
        ▼</i><span>Padre</span>
    <ul>
            <li><i class="icon"></i><i class="toggler">
                ▼</i><span>Elemento 1</span>
                <ul>
                   <li>
                     <i class="icon"></i>
                      <a href="#">
                          <span>Hijo 1 de Elemento 1</span>
                      </a>
                    </li>
                    <li>
                      <i class="icon"></i>
                      <a href="#">
                          <span>Hijo 2 de Elemento 1</span>
                          </a>
                       </li>
                    </ul>
                </li>
            ...
        </ul>
    </li>
</ul>
```

Con los estilos pasa lo mismo, el componente puede añadir una etiqueta de estilos <style> en la que inserta las reglas y estilos para las diferentes partes que lo definen.

```
<style>
    ul.treeview{
        list-style: none;
    }
    ul.treeview li.collapsed ul{
        height: 0;
        overflow: hidden;
    }
    ul.treeview li span{
        padding: 2px 5px;
        display: inline-block;
    }
    ul.treeview li .active{
        background: #000;
        color: #fff;
    }

    /* ... */
</style>
```

Para hacer todo esto, se suele recurrir a la manipulación del DOM a través de JavaScript, pero también es frecuente incrustar algunas partes a través de la propiedad innerHTML.

11.2.3.2 INTERFAZ DE USUARIO INDEPENDIENTE

A veces, no interesa crear una interfaz de usuario a través de JavaScript porque puede ser una tarea tediosa, poco agradecida y puede que, incluso, difícil de personalizar.

Cuando esto ocurre, lo mejor es dejar claro cómo debe ser la estructura que utiliza el componente y dejar, a los desarrolladores o diseñadores, que sean ellos los que decidan qué y cómo usarlo.

Sirva como ejemplo un buscador predictivo o autocomplete. El componente proporciona la funcionalidad necesaria y, fuera de él, el desarrollador sólo debe saber cómo funciona, es decir, sólo necesita conocer cómo hay que definirlo y cómo configurarlo. Un ejemplo podría ser el siguiente código, dónde el desarrollador, únicamente, pone el HTML que necesita.

```
<input
    class="autocomplete"
    id="poblacion"
    name="poblacion"
```

```
      type="text"
      maxlength="255"
      autocomplete="off"
      placeholder="Ej: Tres Cantos " />
```

Luego, durante el proceso de ejecución, la aplicación ejecutará un script que convertirá, este único input, en el buscador predictivo o autocomplete que el usuario necesita. Por ejemplo, podríamos que tener que ejecutar algo como:

```
document.querySelector("#poblacion").Autocomplete(arrayList);
```

En este caso, el parámetro arrayList, sería el listado de las poblaciones que tenemos disponibles en el autocomplete.

11.2.4 La personalización

Por último, los componentes deben ser personalizables. Cuantas más opciones y métodos estén disponibles, más reutilizable se volverá y menos tiempo costará adaptarlo a nuestras necesidades.

En este tema, no hay nada habitual, es decir, cada uno lo hace como quiere o como puede, pero todos intentan hacerlo lo menos doloroso posible.

Aquí, también, entra en juego la posibilidad de que los componentes puedan ser accesibles, es decir, que contemplen estándares como el WAI-ARIA o la WCAG.

A lo largo de todo este capítulo veremos varios ejemplos de personalización, por lo que, no nos detendremos más en este tema.

11.3 EXTENSIÓN DE ELEMENTOS NATIVOS

Hasta ahora, los componentes se han considerado como cualquier otro objeto, y a consecuencia de ello, siempre se han definido con la palabra reservada **función** o como si se fuese a definir un objeto JSON. Sin embargo, existen otras técnicas de implementación de componentes menos conocidas, pero puede que, incluso, más elegantes y eficientes.

Una de estas técnicas es mediante el uso de extensiones sobre elementos nativos. Esto es posible gracias al método registerElement del objeto document.

11.3.1 Método registerElement

Este método permite la adición de elementos personalizados al HTML, asociarlos al DOM con un objeto constructor.

Para poder configurarlo se pueden suministrar uno o dos parámetros.

Parámetro	Descripción
nombreEtiqueta	Es el nombre personalizado de la etiqueta y debe contener, al menos, un guion medio en su definición.
opciones	Es un objeto con las diferentes opciones de configuración, herencia, propiedades y métodos. Este parámetro es opcional.

```
var inputLabel = document.registerElement('input-label', {
    prototype: Object.create(HTMLLabelElement.prototype),
    extends: 'label'
});
```

Si prestamos atención al código presentado, podremos observar que, el método de registrado utiliza como prototipo el propio del elemento label de HTML.

Esta forma de definir componentes personalizados tiene una ventaja añadida y es que puede realizarse la declaración de la etiqueta antes de que se registre el elemento mediante JavaScript. Cuando una etiqueta es añadida al HTML y no está registrada se le denomina **elemento no resuelto**. Más tarde, cuando se registra en el DOM, se convierte en un **elemento personalizado**.

11.3.2 Adición de propiedades y métodos

Pero la extensión de métodos nativos no sólo permite el registrado, también permite la adición de propiedades y métodos.

Para crear una propiedad de sólo lectura podemos recurrir al método **defineProperty** del objeto Object.

```
var inputLabelProto = Object.create(HTMLElement.prototype);
Object.defineProperty(inputLabelProto,
    "description", {value: "Custom Element"}
);

var inputLabel = document.registerElement('input-label',
    {prototype: inputLabelProto}
);
```

O también a través de formato JSON

```
var inputLabel = document.registerElement('input-label', {
    prototype: Object.create(HTMLElement.prototype, {
    description: {
        get: function() { return "Custom Element"; }
    }
  })
});
```

Para crear métodos hacer un poco lo mismo. Si lo hacemos en formato de prototipo:

```
inputLabelProto.console = function(msg) {
    console.log(msg);
};
```

O, si lo hacemos en formato JSON:

```
console: {
    value: function() {
        console.log(msg);
    }
}
```

11.3.3 Ciclo de vida de un elemento personalizado

Los elementos personalizados, al igual que los predefinidos, tienen un ciclo de vida prefijado. Durante ese ciclo, se realizan llamadas automáticas a unos métodos especiales que tienen un propósito específico.

11.3.3.1 MÉTODO CREATEDCALLBACK

Este método se ejecuta automáticamente en el momento en el que, el elemento, es creado.

```
var inputLabelProto = Object.create(HTMLElement.prototype);

inputLabelProto.createdCallback = function() {
    console.log("Creado");
};
```

11.3.3.2 MÉTODO ATTACHEDCALLBACK

Este método se ejecuta automáticamente en el momento en el que, el elemento, es insertado en el documento.

```
var inputLabelProto = Object.create(HTMLElement.prototype);

inputLabelProto.attachedCallback = function() {
    console.log("Añadido");
};
```

11.3.3.3 MÉTODO DETACHEDCALLBACK

Este método se ejecuta automáticamente en el momento en el que, el elemento, es eliminado del documento.

```
var inputLabelProto = Object.create(HTMLElement.prototype);

inputLabelProto.detachedCallback = function() {
    console.log("Añadido");
};
```

11.3.3.4 MÉTODO ATTRIBUTECHANGEDCALLBACK

Este método se ejecuta automáticamente en el momento en el que, el elemento, actualiza, añade o borra una o varias propiedades.

Para poder configurarlo se pueden suministrar uno o dos parámetros.

Parámetro	Descripción
nombreAtributo	Es el nombre del atributo que ha sido añadido, actualizado o eliminado.
valorAntiguo	Es el valor que tenía antes del cambio.
ValorNuevo	Es el valor que tiene ahora, después del cambio.

```
var inputLabelProto = Object.create(HTMLElement.prototype);

inputLabelProto.attributeChangedCallback =
    function(attrName, oldVal, newVal) {
        console.log("El atributo", attrName, "ha cambiado de:");
        console.log(oldVal, "al valor", newVal);
    };
```

11.3.4 Adición del Shadow DOM

El Shadow DOM es una herramienta para encapsular ciertos contenidos dentro de los componentes. Entre otras cosas, permite ocultar la implementación del componente y la definición de estilos sin que afecten temas como la herencia.

```
var inputLabelProto = Object.create(HTMLElement.prototype);

inputLabelProto.createdCallback = function() {
    var shadowRoot = this.createShadowRoot();
    shadowRoot.innerHTML = "<label>__HEADLINE__<input /></label>"
    console.log("Creado");
};

inputLabelProto.attributeChangedCallback =
    function(attrName, oldVal, newVal) {
    if(attrName == "id" || attrName == "for"){
        var lbl = this.shadowRoot.querySelector("label");
        var inp = this.shadowRoot.querySelector("input");

        inp.setAttribute("id", newVal);
        lbl.setAttribute("for", newVal);
    } else if(attrName == "headline"){
        var lbl = this.shadowRoot.querySelector("label");
        lbl.replace("__HEADLINE__", newVal);
    } else {
    var inp = this.shadowRoot.querySelector("input");

    inp.setAttribute("id", newVal);
    }
  };

Object.defineProperty(inputLabelProto, "description", {value: "Custom Element"});

var inputLabel = document.registerElement('input-label', {prototype: inputLabel-
Proto});
```

Si ejecutásemos este código e insertásemos algún elemento, podríamos ver que se genera una estructura como la siguiente:

```
<input-label>
    #shadow-root (open)
    <label for="XXX">
```

```
            " HEADLINE "
            <input id="XXX" />
        </label>
    </input-label>
```

Los identificadores HEADLINE y XXX representan los valores que serán reemplazados cuando se establezcan los atributos correspondientes.

Si introdujésemos una etiqueta **style** dentro del innerHTML de la variable shadowRoot, veríamos que se pueden añadir reglas CSS que sólo afectan al componente, pero permite la herencia de las reglas CSS desde fuera.

11.4 WEB COMPONENTS

El W3C ha propuesto un estándar para el desarrollo de componentes y que, se supone, ya está implementado en los navegadores para su utilización.

El estándar se llama "Web Components" y está pensado para proporcionar a los desarrolladores una forma de crear nuevas etiquetas personalizadas que puedan ser encapsulables y reutilizables. En él, se dice que todo componente web debe basarse en las especificaciones siguientes:

11.4.1 La interfaz Custom Elements

Esta especificación indica cómo se deben definir los nuevos tipos de elementos en el DOM para que tengan todas sus funcionalidades, es decir, se nos indica cómo diseñar los componentes para que puedan ser utilizados tanto desde el HTML como desde JavaScript o CSS.

Para diseñar estos componentes y poder utilizarlos, tendremos que recurrir a la interfaz CustomElementRegistry, que es la herramienta que nos proporciona un medio para poder registrar y consultar elementos personalizados.

La interfaz CustomElementRegistry distingue entre dos tipos de elementos personalizados:

- **Autónomo**: Son elementos independientes que no heredan nada de los elementos HTML ya incorporados.
- **Integrado**: Son elementos que son considerados como extensiones de otros elementos HTML ya incorporados.

Para nosotros, la diferencia más notable será que, uno utilizará la cláusula **extends** de JavaScript y el otro no.

Pero, la cosa no termina aquí, para poder crear un elemento personalizado también tendremos que recurrir al objeto **customElements**, que es quién nos proveerá de los métodos necesarios para poder definir y utilizar nuestro nuevo componente.

11.4.1.1 MÉTODO DEFINE

Este método nos permite añadir etiquetas personalizadas HTML al DOM para que sean utilizadas como cualquier otro elemento.

El método define se alimenta de tres parámetros:

- **Nombre**: Indica el nombre de la etiqueta que hace referencia al nuevo elemento. Este nombre de elemento debe **empezar por una letra** y debe **contener un guion medio** para garantizar que estos elementos personalizados no coincidan con los futuros elementos de HTML5 que se puedan definir.
- **Constructor:** La clase u objeto que se utilizará como constructor.
- **Opciones:** Indica el objeto que controla cómo está definido el elemento personalizado y, actualmente sólo admite el valor **extends**. Este parámetro es opcional.

Otras características que se deben conocer antes de crear un elemento personalizado es que:

- No se puede registrar el mismo nombre de etiqueta varias veces.
- No se puede definir los elementos personalizados en modo de cierre automático, como puedan ser el elemento **img** o el elemento **br** de HTML.

```
customElements.define("nuevo-elemento", NuevoElemento);
```

El código anterior crearía un nuevo componente que se llamaría "nuevo-elemento" y podría ser llamado desde HTML como si de cualquier otra etiqueta se tratase.

```
<div>
    <nuevo-elemento> ... </nuevo-elemento>
</div>
```

Pero, también es posible que al ejecutar la instrucción anterior se presenten errores, como, por ejemplo:

Error	Descripción
… is not a valid custom element name	Este error suele darse porque el identificador que se ha utilizado como nombre o ya existe o no contiene un guion medio.
… is not defined	Este error suele darse porque el constructor pasado como parámetro no existe.
… parameter 3 ('options') is not an object.	Este error suele darse porque se ha enviado como parámetro algo que no es un objeto.

11.4.1.2 MÉTODO GET

Este método devuelve el constructor del elemento personalizado una vez esté definido, en cualquier otro caso, devolverá undefined.

Requiere de un único parámetro y que se corresponde con el nombre de la etiqueta.

```
customElements.get("nuevo-elemento");
```

11.4.1.3 MÉTODO WHENDEFINED

Este método devuelve una promesa (Promise) que se resolverá una vez que se defina el elemento personalizado. Sin embargo, esto pasará sólo si, el nombre de la etiqueta se corresponde con el nombre del elemento personalizado.

Requiere de un único parámetro y que se corresponde con el nombre de la etiqueta.

```
customElements.whenDefined("nuevo-elemento").then(_ => {
    console.log("El nuevo element está definido");
});
```

11.4.2 Shadow DOM

Esta especificación habla de cómo los navegadores deben tener la capacidad de incluir un subárbol de elementos en los documentos web. De esta parte, por ejemplo, salen las directrices para poder definir algunas etiquetas HTML5 de componente complejos como son **audio** o **video**.

El Shadow DOM está compuesto por un contenido oculto de componentes web que tienen etiquetado y estilos CSS y que no presentan problemas de herencia porque están fuera del alcance del contento visible del documento.

No es de extrañar que el Shadow DOM y el Custom Elements vayan, casi siempre, de la mano, porque, CustomElements es quién encapsula la lógica, pero el Shadow DOM, es quién proporciona un entorno controlado y aislado para esa lógica.

Para adjuntar un Shadow DOM a nuestro componente deberemos recurrir al método attachShadow, el cual añadirá un nodo raíz dentro del DOM que se considerará como otro DOM independiente. A este DOM independiente se le suele denominar ShadowRoot, porque es el árbol de elementos oculto dentro del elemento que se está definiendo.

11.4.2.1 MÉTODO ATTACHSHADOW

Este método permite añadir una colección de elementos al Shadow DOM del elemento y asigna su referencia al **ShadowRoot**.

Requiere de un único parámetro, en forma de JSON, que contiene un atributo llamado **mode**. Si este atributo se establece a **open**, se podrá acceder a los elementos que conforman el Shadow DOM desde fuera del mismo. Si este atributo se establece a **closed**, se negará todo acceso desde fuera.

Este JSON tiene otros atributos, aunque no es frecuente utilizarlos.

```
var nuevoElemento = document.querySelector('nuevo-elemento');

var shadowRoot = this.attachShadow({mode: 'open'});
shadowRoot.appendChild(nuevoElemento.content.cloneNode(true));
```

Es posible que al ejecutar una instrucción como la anterior se presenten errores como que se está intentando adjuntar a algo que ya es un Shadow DOM o que no se puede adjuntar por conflictos en los espacios de nombres.

11.4.3 HTML Templates

Esta especificación concreta algunos mecanismos para mantener el HTML que no es incluido en el DOM cuando se produce la carga de la página pero que, pueden instanciarse más tarde a través de JavaScript.

El uso de templates, esencialmente, se basa en el manejo de la etiqueta template de HTML5, que está soportada por todos los navegadores, a excepción de Internet Explorer 12 e inferiores.

Por si alguno no lo sabe, la etiqueta template, se utiliza para mantener oculto un contenido que, más tarde, puede ser tratado por JavaScript a modo de plantilla. Es decir, que carga un fragmento de código en la página, pero no se renderiza hasta que se le dé la orden mediante JavaScript.

Al igual que pasa con el resto de las especificaciones de la W3C, esta etiqueta permite su uso con o sin Custom Elements, lo que hace mucho más fácil su interacción con el DOM, sin embargo, no es una práctica muy recurrida porque el HTML Import se ha quedado en desuso y ahora se recomiendan los módulos ES.

Esto módulos ES, no son más que una especificación para incluir y reutilizar el código JavaScript basándose en estándares e intentando conseguir modularidad y una mejor eficiencia.

11.4.4 Métodos utilizados para la definición de clases

11.4.4.1 CONNECTEDCALLBACK

Este método se ejecuta automáticamente en el momento en el que, el componente, es añadido al DOM.

Es frecuente encontrar, en esta parte de la declaración, la definición del Shadow DOM, es decir, capas y estilos propios e independientes.

También es frecuente utilizarlo para aplazar algunas funciones, como la recuperación de un Ajax u otro recurso.

11.4.4.2 MÉTODO DISCONNECTEDCALLBACK

Este método se ejecuta automáticamente en el momento en el que, el componente, es eliminado del DOM.

Es frecuente utilizarlo para liberar recursos, sin embargo, puede ser la causa de muchos problemas a posteriori porque nunca se llama cuando se sale de la página si se cierra la pestaña del navegador o el navegador en sí.

11.4.4.3 MÉTODO ATTRIBUTECHANGEDCALLBACK

Este método se llama automáticamente al instanciarse para obtener los valores iniciales. Básicamente, controla los atributos del componente, es decir, que se ejecuta cuando se añade, actualiza, o elimina un atributo de la clase u objeto.

Es relativamente frecuente encontrar problemas al manejar este método porque, desde aquí, sólo se pueden manipular los atributos que estén contemplados en la propiedad estática **observedAttributes**.

11.4.4.4 MÉTODO ADDOPTEDCALLBACK

Este método sólo es ejecutado cuando el elemento es trasladado de un documento a otro, por lo que no es frecuente utilizarlo mucho.

11.4.5 Ejemplo de componente Web

Para realizar este cometido, lo primero que debemos conocer es la creación y manipulación de clases para poder realizar una extensión del objeto **HTMLElement** de JavaScript.

```
class Autor extends HTMLElement {
    constructor() {
        super();
        this._name = null;
    }

    connectedCallback () {
        var shadowRoot = this.attachShadow({mode: 'open'});
        shadowRoot.innerHTML = `<style>
    :host {
       --fg: #ffffff;
       --bg: #384a74;
    }
    div {
        width: 128px;
        height: 32px;
        line-height:32px;
        display:block;
        background:var(--bg);
        color:var(--fg);
```

```
        }
    </style>
    <div>` + this._name + `</div>`;

            this._updateRendering();
        }

        static get observedAttributes() { return ["name"]; }
        attributeChangedCallback(attrName, oldValue, newValue) {
            this._name = newValue;
            this._updateRendering();
        }

        disconnectedCallback() {
            console.log(this, "fue elemento eliminado!");
        }

        get name() {
            return this._name;
        }

        set name(v) {
           this.setAttribute("name", v);
        }

        _updateRendering() {
            console.log(this)
        }
    }
```

Una vez definida la clase constructora, lo que hacemos es definir el nuevo elemento personalizado que vinculará el HTML con el DOM.

```
customElements.define("custom-autor", Autor);
```

Ahora mismo, si todo ha ido bien, ya podríamos añadir a nuestro código HTML el nuevo elemento y manipularlo desde JavaScript. Por si alguien se lo pregunta, también puede añadirse CSS.

```
<custom-autor name="Pablo Fernández"></custom-autor>
```

Y dado que ambas partes están vinculadas, también podríamos definir el componente directamente desde JavaScript de la siguiente manera:

```
var autor = document.createElement("custom-autor");
autor.name = "Pablo Fernández";
document.body.appendChild(autor);
```

11.4.6 Compatibilidad con los navegadores

Aunque la propuesta de la W3C es muy buena, todavía está en fase de desarrollo y no está disponible para todos los navegadores, a no ser que se utilice un componente externo como un polyfill.

Por si alguien no lo sabe aún, se podría decir que, un polyfill es un componente de JavaScript que provee de una o varias herramientas concretas que los desarrolladores esperarían encontrar de forma nativa.

Uno de los polyfills más conocidos es HTML5Shiv, que nos provee de todo lo necesario para que todas las características de HTML5 estén disponibles en cualquier navegador.

En lo referente a creación de elementos personalizados, uno de los polyfills más utilizados es WebComponents.js descargable desde la URL:

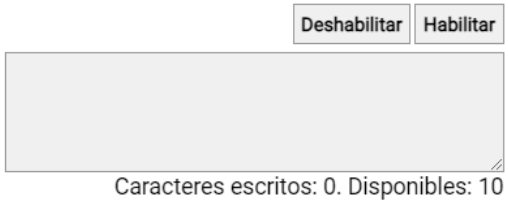
https://github.com/webcomponents/polyfills.

11.5 EJERCICIOS RESUELTOS

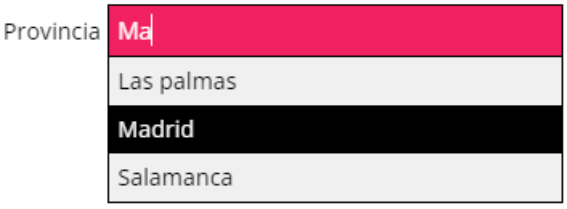
Detectar tipo de dispositivo	Código QR
Averiguar si el dispositivo dónde se carga la página es de tipo móvil o de escritorio, es decir, si es un móvil o, un ordenador portátil o de sobremesa. El análisis se realizará tanto en el proceso de carga, como en el momento en que se detecte un cambio de tamaño en la ventana. **https://codepen.io/pefc/pen/OJoYRxG**	

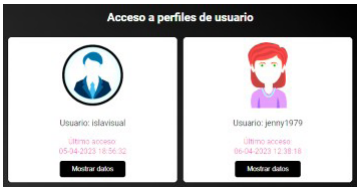
Algoritmo de sustitución simple	Código QR
La idea de este componente es realizar un algoritmo sencillo que sea capaz de codificar y descodificar, de alguna manera, un texto de cualquier clase. **https://codepen.io/pefc/pen/gOdJLMb**	

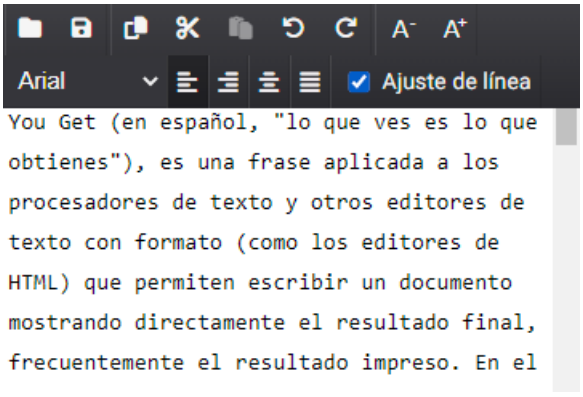
Botón de tipo interruptor	Código QR
Crear una forma fácil de integrar botones con aspecto de interruptor (switch) en las aplicaciones a partir de un elemento de formulario de verificación (checkbox).  **https://codepen.io/pefc/pen/oNPRWMK**	

Contador de carácteres	Código QR
Crear una forma sencilla para integrar la posibilidad de ver cuántos caracteres contiene un elemento de formulario y, si procede, cuántos quedan disponibles.  **https://codepen.io/pefc/pen/ExezmKL**	

Diálogo emergente	Código QR
Crear una forma sencilla de gestionar la visibilidad de los diálogos emergentes, también conocidos como popup's.  **https://codepen.io/pefc/pen/LYJoLPY**	

Autocomplete sencillo	Código QR
Crear una forma sencilla de gestionar la visibilidad de los diálogos emergentes, también conocidos como popup's.  **https://codepen.io/pefc/pen/RwYmZwQ**	

Destectar si un elemento está visible	Código QR
Crear un componente web que muestre el avatar, nombre de usuario, fecha de último acceso y un botón para acceder a todo su perfil completo.  **https://codepen.io/pefc/pen/MWPgZQx**	

Web Component: Editor de textos	Código QR
Crear un editor de texto similar al bloc de notas con las opciones de abrir, guardar, copiar, cortar, deshacer, rehacer, incrementar y decrementar el tamaño de fuente, cambiar el tipo de letra, poder cambiar la alineación y establecer o eliminar el ajuste de línea.  **https://codepen.io/pefc/pen/MWqNqod**	

12

PRUEBAS Y VERIFICACIÓN EN PÁGINAS WEB

La prueba y verificación de un producto de software consiste en responder a dos preguntas. Esto es, ¿es correcto? y ¿se está construyendo de forma correcta? Es decir, por una parte, se debe cumplir que los requisitos que precisa el usuario son los adecuados y, por otra, que el funcionamiento es el correcto y que cabe esperar.

Para garantizar estas premisas lo primero que debemos hacer es establecer un entorno de trabajo en el que podamos realizar varios test de pruebas a todas y cada una de las páginas que componen nuestra aplicación o sitio web y que nos permitan verificar su correcto funcionamiento.

Para ello podemos valernos de los test A/B, pruebas unitarias, pruebas alfa, pruebas beta y algunas herramientas que garanticen la calidad del código.

12.1 TEST A/B

Un test A/B es un experimento en el que se analizan dos versiones de un mismo producto para identificar y maximizar un determinado resultado sin que los usuarios finales lo noten. Consta de dos partes, la primera se denomina "de control" y a la segunda "variante". Si el número de variantes es mayor que uno, se produce una modificación llamada "split test" que genera una versión para cada una de las variaciones que aplican sobre la versión de control.

Normalmente, se utilizan en el ámbito de la analítica web y marketing digital, sin embargo, en el campo de la usabilidad y diseño web se utilizan también para la preparación de prototipos y mejora de las funcionalidades. Un ejemplo de uso

típico es realizarlo en el carrito de la compra de un comercio electrónico ya que se puede conseguir bajar la tasa de abandono e incrementar los beneficios económicos al optimizar el proceso de compra y su funcionalidad.

Existen muchas razones por las que se deben realizar test A/B, pero cabe destacar que una de las más importantes es la de conocer y entender a nuestros usuarios. De esta forma se mejora la rentabilidad, la imagen de la marca, incrementa la fidelización y se innova de acuerdo a un pensamiento colectivo.

Prácticamente todo se puede medir si se le puede asignar un ratio de éxito. La calidad de los contenidos, la facilidad de lectura, los colores, la estructura visual, la legibilidad, las campañas publicitarias, clics en un banner. Para decidir qué medir, lo primero es conocer cuál es el objetivo y que beneficio puede aportar. Los datos obtenidos permitirán evaluar las diferentes versiones y conseguir el objetivo codiciado.

12.1.1 Etapas de un test A/B

12.1.1.1 ANÁLISIS DEL PROCESO

Lo primero que hay que hacer es dividir el proceso en pasos más pequeños para conocer el proceso y averiguar cómo funciona intentando encontrar los fallos y mejoras.

12.1.1.2 ESTABLECIMIENTO DE HIPÓTESIS

Una vez se sabe cómo funciona y cuáles son los errores susceptibles de tener mejora, se lanzan hipótesis para realizar los futuros cambios y probarlos en las diferentes versiones.

12.1.1.3 ELABORACIÓN DE LAS VARIANTES

Con las ideas claras se realizan las modificaciones en la interfaz o sistema y se establecen los medios para poder ponerla en marcha. La creación de variantes se puede realizar a través de herramientas ya creadas como CrazyEgg o Google Analytics o a través de modificaciones en el código.

12.1.1.4 ACTIVACIÓN Y TESTING

Lo habitual es establecer unas fechas de inicio y finalización de test y, si durante el proceso de testing se prevé que no se van a recoger suficientes datos, se mantiene abierto hasta que se obtenga un volumen suficientemente elevado como para extraer las conclusiones y averiguar si las hipótesis eran o no correctas.

En el caso de los resultados no sean suficientemente significativos u objetivos se deberá volver a empezar el proceso desarrollando nuevas hipótesis y variantes.

12.2 PRUEBAS UNITARIAS

En entornos dónde se realiza desarrollo de software es muy habitual oír hablar de pruebas unitarias. Una prueba unitaria es una forma de comprobar el correcto funcionamiento de una unidad de código y asegurar que funciona de forma eficiente tanto en conjunto como por separado.

Básicamente, lo que hace es verificar que el código fuente hace lo que debe hacer, que los nombres y tipos de parámetros son los adecuados y que el tipo de valor devuelto es correcto.

12.2.1 Creación de una prueba unitaria

La forma de realizar este tipo de pruebas es a través de códigos externos, generados habitualmente, en el mismo lenguaje que el utilizado por el sistema o interfaz. Si la interfaz utiliza PHP, las pruebas unitarias suelen realizarse a través de PHP.

Por ejemplo, si se quisiera comprobar el funcionamiento de un método que tiene como resultado una suma, se debería crear una función que tuviese un valor fijo (que representa el valor esperado) y otro variable (que representa el valor devuelto tras ejecutar la función suma). Si el valor esperado fuese idéntico al valor devuelto y los tipos de datos de E/S también, entonces la prueba habría tenido éxito y se podría asegurar que el método es correcto y que hace lo que se esperaba.

Para que una prueba unitaria sea considerada como efectiva debe cumplir los siguientes requerimientos:

- **Automatizable:** Debe de poder automatizarse sin intervención manual del desarrollador o usuario.
- **Completas:** Deben cubrir la mayor cantidad de código fuente.
- **Reutilizables:** Deben ser pruebas que se puedan utilizar de manera repetida en el tiempo.
- **Independientes:** Deben ser independientes unas de otras sin llegar a afectarse.

Cuando se realizan un conjunto de pruebas unitarias que componen un proceso, se las suele denominar pruebas integrales o de integración.

12.3 PRUEBAS ALFA

Un test de prueba alfa es una de las estrategias de prueba de software más utilizadas en proceso de diseño o desarrollo de aplicaciones. Son un tipo de pruebas finales realizadas antes de que el software se lance al mercado o se ponga en producción.

Son muy útiles, por ejemplo, cuando se desea realizar el prototipo de una interfaz o sistema y no se dispone de los requerimientos necesarios parcial o totalmente. Si los resultados obtenidos fuesen positivos se podría ir por esa vía, de lo contrario, se deberían replantear los requerimientos propuestos.

Los test de pruebas alfa tienen dos fases:

- En la primera fase de las pruebas alfa, el software se prueba por los desarrolladores internos utilizando herramientas complementarias, como depuradores, para detectar errores rápidamente.
- En la segunda fase de las pruebas alfa, el software se entrega al personal de control de calidad del software dónde realizan pruebas adicionales en un entorno similar al real.

Las pruebas alfa, a menudo, se emplean para software comercial como una forma de prueba de aceptación interna, antes de que el software pase al test de pruebas beta.

12.4 PRUEBAS BETA

Los test de pruebas beta son la siguiente fase a los test de prueba alfa y son una estrategia de prueba de software que se realiza cuando el sistema o interfaz está finalizado y se pasa a utilizar en un entorno de real o de producción.

Este tipo de pruebas se vuelven necesarias porque, no importa la opinión que tiene una empresa sobre su producto a lanzar, importa lo que piensan los usuarios, para qué lo utilizan y cómo lo utilizan. Es en estas pruebas dónde suelen salir la mayoría de los errores que no han sido detectados por los desarrolladores ni por el personal de calidad y que se corrigen antes de liberar la versión beta, de ahí su nombre.

Dado que la realización de estas pruebas suele ser un trabajo arduo, tedioso y frustrante y se debe realizar con usuarios finales, por lo general, se consiguen a través de incentivos como descuentos, regalos y, en ocasiones, a través de pequeños contratos temporales. Este tipo de usuarios finales que prueban el producto suelen recibir el nombre de betatester.

12.5 HERRAMIENTAS PARA LA VALIDACIÓN DE CÓDIGO

Existen muchos complementos para ayudar a validar la calidad del código de los sitios web, no obstante, en muchas ocasiones estos complementos no son compatibles con todos los navegadores y hay que recurrir a componentes específicos de cada uno de ellos. Aún así, aquí exponemos las más frecuentemente utilizadas.

12.5.1 Inspector / Firebug / Lighthouse

Dependiendo del navegador que usemos se denominará de una manera u otra, pero todas hacen referencia a un conjunto de herramientas con las que se puede revisar el CSS, HTML y JavaScript, comprobar la velocidad de carga, consultar la estructura del DOM, monitorizar y depurar el código fuente, entre otras funcionalidades.

12.5.2 HTML Validator

HTML Validator es una herramienta que agrega un monitor que valida el HTML y muestra el número de errores encontrados en forma de icono.

12.5.3 Markup Validation Service

URL: https://validator.w3.org/

Markup Validation Service es servicio gratuito creada por la W3C que permite comprobar la validez del marcado en códigos HTML, XHTML, SMIL y MathML, entre otros, en contextos de Web Semántica, Accesibilidad y Usabilidad Web.

12.5.4 CSS Validation Service

URL: http://jigsaw.w3.org/css-validator/

CSS Validation Service es un software creado por la W3C y pensado para diseñadores y desarrolladores web que permite revisar las hojas de estilo en cascada (CSS) y los documentos XHTML con hojas de estilo. Puede utilizarse mediante su servicio gratuito vía web, o puede descargarse para ser usado bien como un programa Java, o como un servlet Java en un servidor Web.

12.5.5 Link Checker

URL: https://validator.w3.org/checklink

Link Checker es una herramienta web que busca problemas en vínculos, anclajes y otros objetos referenciados en una página web, hojas de estilo CSS o, recursivamente, en todo el sitio Web completo.

REFERENCIAS

Casado, P. E. (2020). *Diseño y Construcción de Páginas Web.* Ra-Ma.

Casado, P. E. (2020). *Domine JavaScript 4ª Edición.* RA-MA.

Mozilla.org. (Marzo de 2023). *https://developer.mozilla.org/es/docs/Web/*. Obtenido de https://developer.mozilla.org/es/docs/Web/.

W3C Schools. (2023, Febrero). *HTML The language for building web pages*. Retrieved from HTML The language for building web pages: https://www.w3schools.com/

Wikipedia. (Marzo de 2023). *Wikipedia*. Obtenido de https://es.wikipedia.org/

Wikipedia, c. d. (Febrero de 2023). *Pseudocódigo*. Obtenido de https://es.wikipedia.org/w/index.php?title=Pseudoc%C3%B3digo&oldid=149404690

World Wide Web Consortium. (2023, Febrero). *W3C*. Retrieved from https://www.w3.org