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.

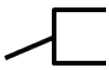
Para realizar estas representaciones gráficas habitualmente se recurre al Lenguaje Unificado de Modelado (UML) ya que representa los flujos de trabajo paso a paso y el cual se caracteriza por disponer de una serie de símbolos que poseen un significado concreto y unas descripciones breves textuales que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

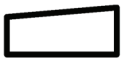
Sin embargo, antes de hacer un ordinograma o diagrama de flujo, es aconsejable definir qué se espera obtener del diagrama de flujo, quién y cómo se empleará, hasta qué nivel de detalle se va a llegar y cuáles serán los límites del proceso a describir.

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

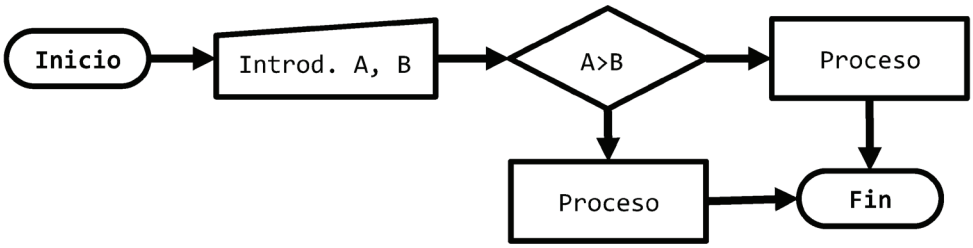
- Establecer el alcance del proceso a describir.
- Identificar y listar las principales actividades/subprocesos que están incluidos en el proceso a describir y su orden cronológico.
- Definir el nivel de detalle incluyendo sus actividades menores.
- Identificar y listar los puntos de decisión.
- Definir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

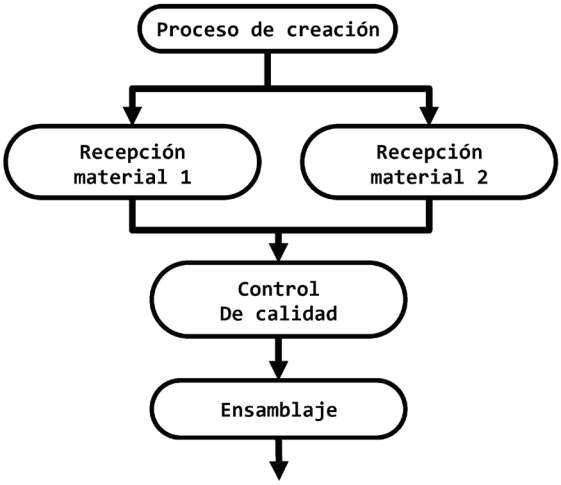
El cursograma es similar, si no idéntico, a un diagrama de flujo, con la diferencia de que se suele utilizar para mostrar la secuencia cronológica de las fases del proceso o programa.

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:

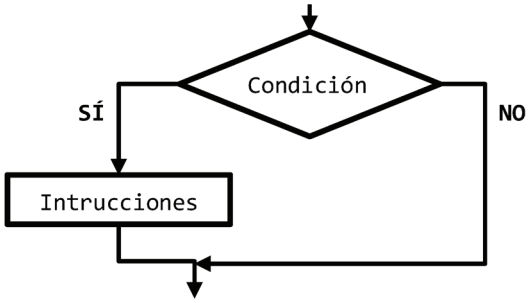
```
hiptenusa ← √(a² + b²)
resultado := sin(a);
volumen = πr²h
operacion1 = (a + b) / 2;
operacion2 = operacion1 mod 3;
```

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

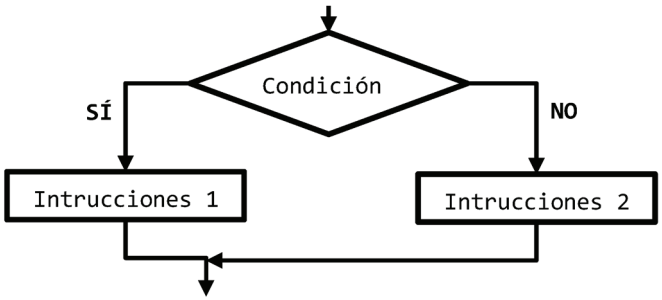
En este tipo de situaciones podemos recurrir a estructuras de una única condición, de dos condiciones o múltiples condiciones.

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.

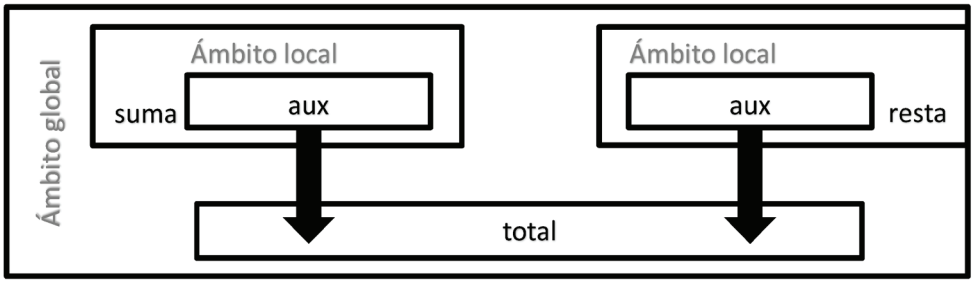
```
let total = 0;

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
```